Efficient Algorithms for Some Connectivity Problems, in Static and Dynamic Graphs

A Dissertation

submitted to the designated
by the Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

Evangelos Kosinas

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Ioannina
School of Engineering
Ioannina 2024

Advisory Committee:

- **Loukas Georgiadis**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina (supervisor)
- **Christos Nomikos**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
- **Leonidas Palios**, Professor, Department of Computer Science and Engineering, University of Ioannina

Examining Committee:

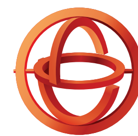
- **Loukas Georgiadis**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
- **Monika Henzinger**, Professor, Institute of Science and Technology, Austria
- **Spyridon Kontogiannis**, Associate Professor, Department of Computer Science and Engineering, University of Patras
- **Christos Nomikos**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
- **Leonidas Palios**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Charis Papadopoulos**, Associate Professor, Department of Mathematics, University of Ioannina
- **Nikos Parotsidis**, Research Scientist, Google Research, Switzerland

Acknowledgements

First and foremost, I would like to thank my advisor, Loukas Georgiadis, who guided me through the process of pursuing my PhD, by suggesting very nice topics for research, and for being very agreeable and encouraging in our conversations. It is fair to say that, without his trust, patience, and understanding, this work would not have been possible. I would like to thank Giuseppe Italiano for his collaboration in various projects, and for inviting me at LUISS as a visiting scholar during the fall semester of 2022. I enjoyed very much my stay in Rome, and it has been very exciting throughout to be in contact with the perspective of a highly experienced and distinguished researcher. I would also like to thank Debasish Pattanayak for his collaboration in a part of this work. I am very grateful to Christos Nomikos and Leonidas Palios for being part of the advisory and examining committee, and I would also like to thank Monika Henzinger, Spyridon Kontogiannis, Charis Papadopoulos, and Nikos Parotsidis, who have accepted to be members of the examining committee. Finally, I would like to thank my parents for their support all these years, and for their unconditional belief in the value of my academic pursuits.

Funding Acknowledgements

The research work was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the 3rd Call for HFRI PhD Fellowships (Fellowship Number: 6547).



ΕΛ.Ι.Δ.Ε.Κ.
Ελληνικό Ίδρυμα Έρευνας & Καινοτομίας
H.F.R.I.
Hellenic Foundation for
Research & Innovation

Table of Contents

List of Figures **vii**

List of Tables **ix**

List of Algorithms **x**

Abstract **xiii**

Εκτεταμένη Περίληψη **xv**

1 Introduction **1**

1.1 Objective . . . 1

1.2 Overview of our results . . . 2

1.2.1 Computing the 4-edge-connected components . . . 2

1.2.2 Computing the 5-edge-connected components . . . 3

1.2.3 Connectivity queries under four edge failures . . . 4

1.2.4 Connectivity queries under vertex failures . . . 5

1.2.5 On computing the maximal k-edge-connected subgraphs . . . 7

1.3 Organization . . . 8

2 Preliminaries **9**

2.1 Basic graph terminology . . . 9

2.2 Partitions and atoms . . . 11

2.3 Edge-connectivity and k-edge-connected components . . . 12

2.4 Maximal k-edge-connected subgraphs . . . 13

2.5 Notation . . . 14

3 Concepts defined on a DFS Tree **16**

3.1 Basic definitions . . . 17

3.1.1 *low* and *high* edges 19
3.1.2 Maximum points, leftmost and rightmost edges 20
3.2 Properties of the DFS parameters 21
3.3 Computing the *low*-edges 29
3.4 Computing the *high*-edges 34
3.5 Computing the leftmost and the rightmost edges 40
3.6 Computing the maximum points 48
3.7 Pointer-machine algorithms for some DFS parameters 50
3.7.1 Computing all $M(v)$ 51
3.7.2 Computing all $\widetilde{M}(v)$, $M_{low1}(v)$ and $M_{low2}(v)$ 54
3.7.3 Computing all $low_M(v)$ and $low_M D(v)$ 60
3.7.4 Computing all $L_1(v)$, $L_2(v)$, $R_1(v)$ and $R_2(v)$ 63
3.8 Two lemmata concerning paths 65
3.9 An oracle for back-edge queries 66
3.10 Segments of vertices that have the same *high* point 69

4 Computing the 4-Edge-Connected Components in Linear Time 74
4.1 Introduction 74
4.2 3-cuts on a DFS tree 75
4.2.1 Type-1 3-cuts 77
4.2.2 Type-2 3-cuts 77
4.2.3 Type-3 3-cuts 79
4.3 Computing all 3-cuts of a 3-edge-connected graph 81
4.3.1 Computing Type-1 3-cuts 81
4.3.2 Computing Type-2 3-cuts 82
4.3.2.1 The upper case 82
4.3.2.2 The lower case 89
4.3.3 Computing Type-3 3-cuts 95
4.4 Computing the 4-edge-connected components 96
4.4.1 Reducing the computation to 3-edge-connected graphs 96
4.4.2 Splitting a 3-edge-connected graph according to its 3-cuts 97
4.5 Testing 4-edge connectivity 99
4.5.1 The upper case 100
4.5.2 The lower case 101

5 Computing the 5-Edge-Connected Components **104**
5.1 Introduction 104
5.1.1 Problem definition 104
5.1.2 Related work 105
5.1.3 Our contribution 107
5.1.4 Technical overview 109
5.1.4.1 Reduction to 3-edge-connected graphs 110
5.1.4.2 Computing enough 4-cuts of a 3-edge-connected graph 111
5.1.4.3 Unpacking the implicating sequences of a complete collection of 4-cuts 113
5.1.4.4 Cyclic families of 4-cuts, and minimal 4-cuts 114
5.1.4.5 Isolated and quasi-isolated 4-cuts 117
5.1.4.6 The full algorithm 118
5.1.5 Organization of this chapter 120
5.2 Properties of 4-cuts in 3-edge-connected graphs 120
5.2.1 The structure of crossing 4-cuts of a 3-edge-connected graph . . 121
5.2.2 Implied 4-cuts, and cyclic families of 4-cuts 125
5.2.3 Properties of cyclic families of 4-cuts 139
5.2.4 Generating the implied 4-cuts 140
5.2.5 Isolated and quasi-isolated 4-cuts 146
5.2.6 Some additional properties satisfied by the output of Algorithm 16 150
5.3 Using a DFS-tree for some problems concerning 4-cuts 159
5.3.1 Computing the r-size of 4-cuts 161
5.3.2 Checking the essentiality of 4-cuts 163
5.3.3 Computing the atoms of a parallel family of 4-cuts 164
5.4 Computing the 5-edge-connected components 177
5.4.1 Overview 177
5.4.2 Computing the minimal 4-cuts 180
5.4.3 Computing the essential isolated 4-cuts 183
5.4.4 Computing enough 4-cuts in order to derive the 5-edge-connected components 190
5.4.5 The algorithm 192
5.5 Computing a complete collection of 4-cuts 194

5.5.1 A typology of 4-cuts on a DFS-tree 194
5.5.2 Type-2 4-cuts 200
5.5.3 Type-3 4-cuts 202
5.5.3.1 Type-3α 4-cuts 203
5.5.3.2 Type-3β 4-cuts 209
5.5.4 Min-max vertex queries 219
5.6 Computing Type-2 4-cuts 222
5.6.1 The case $B(v) = B(u) \sqcup \{e_1, e_2\}$ 224
5.6.2 The case $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$ 229
5.6.3 The case $B(u) = B(v) \sqcup \{e_1, e_2\}$ 239
5.7 Computing Type-3α 4-cuts 242
5.7.1 Type-3αi 4-cuts 246
5.7.1.1 The case where $M(B(u) \setminus \{e_{high}(u)\}) = M(u)$ 249
5.7.1.2 The case where $M(B(u) \setminus \{e_{high}(u)\}) \neq M(u)$ 254
5.7.2 Type-3αii 4-cuts 259
5.8 Computing Type-3β 4-cuts 282
5.8.1 Type-3βi 4-cuts 289
5.8.1.1 Case (1) of Lemma 5.57 289
5.8.1.2 Case (2) of Lemma 5.57 296
5.8.1.3 Case (3) of Lemma 5.57 305
5.8.1.4 Case (4) of Lemma 5.57 315
5.8.2 Type-3βii 4-cuts 333
5.8.2.1 Type-3βii-1 4-cuts 333
5.8.2.2 Type-3βii-2 4-cuts 344
5.8.2.3 Type-3βii-3 4-cuts 372
5.8.2.4 Type-3βii-4 4-cuts 383

6 Connectivity Queries under 4 Edge Failures **441**
6.1 Introduction 441
6.2 E' contains zero tree-edges 450
6.3 E' contains one tree-edge 450
6.4 E' contains two tree-edges 451
6.5 E' contains three tree-edges 452
6.5.1 u and v are not related as ancestor and descendant 452

6.5.2 v is an ancestor of u 453
6.6 E' contains four tree-edges 457
6.6.1 No two vertices in $\{u, v, w\}$ are related as ancestor and descendant 458
6.6.2 w and v are not related as ancestor and descendant, and v is an ancestor of u 459
6.6.3 w is an ancestor of both u and v, and $\{u, v\}$ are not related as ancestor and descendant 461
6.6.4 w is an ancestor of v, and v is an ancestor of u 463
6.7 The data structure 471

7 Connectivity Queries under Vertex Failures 472
7.1 Introduction 472
7.1.1 Previous work 473
7.1.2 Our contribution 473
7.2 Preliminaries 475
7.2.1 DFS-based concepts 476
7.3 The algorithm for vertex failures 479
7.3.1 Initializing the data structure 479
7.3.2 The general idea 481
7.3.3 The structure of the internal components 483
7.3.4 Handling the updates: construction of a connectivity graph for the internal components of $T \setminus F$ 489
7.3.5 Answering the queries 498

8 On Maximal k-Edge-Connected Subgraphs of Undirected Graphs 501
8.1 Introduction 501
8.1.1 Overview of our results 503
8.1.2 Organization 507
8.2 Preliminaries 507
8.3 The decomposition tree of the maximal k-edge-connected subgraphs . . 509
8.3.1 A general framework for maintaining the k-edge-connected subgraphs 510
8.3.2 Maintaining the decomposition tree 512
8.3.3 The decomposition tree of the maximal 3-edge-connected subgraphs 514

8.3.3.1 N is the root or a 3-ecc node 519
8.3.3.2 N is a 1-ecc node 519
8.3.3.3 N is a 2-ecc node 521
8.4 Maintaining the decomposition tree after insertions 524
8.4.1 An $O(n^2 \log^2 n + m\alpha(m, n))$-time algorithm for the incremental maintenance of $\mathcal{T}$ 525
8.4.2 An $O(n^2\alpha(n, n) + m\alpha(m, n))$-time algorithm for the incremental maintenance of $\mathcal{T}$ 529
8.5 Data structures for trees and cactuses 532
8.5.1 An implementation for trees 533
8.5.2 An implementation for cactuses 535
8.6 Improved data structures for trees and cactuses 541
8.6.1 Fractionally rooted trees 541
8.6.2 An implementation for trees 542
8.6.3 An implementation for cactuses 546
8.7 Sparse certificates for the maximal k-edge-connected subgraphs 555
8.8 Computing the maximal k-edge-connected subgraphs 558
8.9 A fully dynamic algorithm for maximal k-edge-connectivity 561
8.10 Conclusions 564

Bibliography **566**

List of Figures

2.1 The maximal 3-edge-connected subgraphs constitute a refinement of the 3-edge-connected components. 14
2.2 It is not straightforward how to efficiently derive the maximal 3-edge-connected subgraphs from the 3-edge-connected components. 14

4.1 The types of 3-cuts with respect to a DFS tree. 76
4.2 The three cases of Type-2 3-cuts (upper case). 84
4.3 The three cases of Type-2 3-cuts (lower case). 90
4.4 Splitting a graph according to a 3-cut. 98

5.1 All possible crossings of two 4-cuts. 122
5.2 The crossing square of two essential 4-cuts. 125
5.3 The possible arrangements of two distinct 4-cuts that share a pair of edges. 127
5.4 The relation of implication for collections of 4-cuts is not transitive. . . 130
5.5 A cyclic family of 4-cuts generated by a collection of five pairs of edges. 130
5.6 A depiction of the situation analyzed in Lemma 5.17. 149
5.7 The 4-cuts with distance 1 are not necessarily implied straightforwardly from the collection of pairs of edges that generates them. 152
5.8 The 4-cuts with distance 2 are not necessarily implied straightforwardly from the collection of pairs of edges that generates them. 153
5.9 The 4-cuts with distance at least 3 are implied straightforwardly from the collection of pairs of edges that generates them. 154
5.10 Companion figures to Lemma 5.20. 157
5.11 Expanding a square family of 4-cuts into a hexagonal family, under some restrictions. 158

5.12 The condition of essentiality of both 4-cuts in the statement of Lemma 5.20 cannot be removed. 160
5.13 A depiction of the situation analyzed in Lemma 5.21. 161
5.14 Splitting a graph according to a 4-cut. 166
5.15 Two essential quasi-isolated 4-cuts may cross. 179
5.16 Illustration of Type-1 4-cuts. 200
5.17 The three different cases for Type-2 4-cuts. 201
5.18 The number of Type-2ii 4-cuts can be $\Omega(n^2)$. 203
5.19 The two different cases of Type-3α 4-cuts. 204
5.20 The number of Type-3αi 4-cuts can be $\Omega(n^2)$. 205
5.21 Cases (1)-(3) of Lemma 5.50 for Type-3αii 4-cuts. 207
5.22 Cases (4.1) and (4.2) of Lemma 5.50 for Type-3αii 4-cuts. 208
5.23 The four different cases of Type-3β 4-cuts. 211
5.24 The number of Type-3βi 4-cuts that satisfy (2) of Lemma 5.57 can be $\Omega(n^2)$. 212
5.25 The number of Type-3βi 4-cuts that satisfy (3) of Lemma 5.57 can be $\Omega(n^2)$. 213
5.26 The last case of Type-3βii-4 4-cuts. 433
5.27 Splitting a vertex with the introduction of five parallel edges. 434
5.28 Splitting a vertex v on a DFS-tree, so that the number of back-edges with lower endpoint v is reduced by one. 439

7.1 A set of failed vertices on a DFS tree. 482

8.1 A sequence of insertions can force $\Omega(n)$ 3-interconnection edges to increase their level by one $\Omega(n)$ times. 515
8.2 The decomposition tree $\mathcal{T}$ of the graph G of Figure 2.1. 516
8.3 Changes to the decomposition tree upon insertion of an edge (x, y) (I). 520
8.4 Changes to the decomposition tree upon insertion of an edge (x, y) (II).522
8.5 Changes to the decomposition tree upon insertion of an edge (x, y) (III).523
8.6 An example of a tree-of-cactuses representation of a cactus. 547

List of Tables

7.1 Comparison of the best-known deterministic bounds for an oracle for connectivity queries upon vertex failures. 474
7.2 Comparison of the best-known deterministic bounds for an oracle for connectivity queries upon vertex failures, when $d_\star$ is a fixed constant. . 475

8.1 Previous best time bounds for computing the maximal k-edge-connected subgraphs. 561
8.2 Improved time bounds for computing the maximal k-edge-connected subgraphs. 561

List of Algorithms

1 Compute the low_i-edges . . . 32

2 Compute the $high_i$-edges . . . 36

3 Compute the sets $L(v_1, c_1, t_1), \ldots, L(v_N, c_N, t_N)$ . . . 44

4 Compute all $M(v)$ . . . 52

5 Compute all $\widetilde{M}(v)$ . . . 55

6 Compute all $M_{low1}(v)$ and $M_{low2}(v)$ . . . 56

7 Compute all $(low_M D(v), low_M(v))$ . . . 61

8 Compute all $L_1(v)$ and $L_2(v)$ . . . 64

9 Determine whether there is a back-edge $(x, y) \in B(v) \setminus B(u)$ such that $y \leq w$, where u is a proper descendant of v, and v is a proper descendant of w . . . 69

10 Compute the collection $\mathcal{S}$ of the segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant . 71

11 Find all Type-2 3-cuts in the upper case . . . 83

12 Find all Type-2 3-cuts in the lower case . . . 91

13 Compute the 4-edge-connected components of a 3-edge-connected graph 100

14 Check whether there exists a Type-2 3-cut in the upper case . . . 102

15 Check whether there exists a Type-2 3-cut in the lower case . . . 103

16 Return a set of collections of pairs of edges that generate in total all the 4-cuts that are implied by a collection of 4-cuts . . . 142

17 Compute the *atoms* of a parallel family of 4-cuts . . . 174

18 Compute the minimal 4-cuts . . . 182

19 Compute the essential isolated 4-cuts . . . 186

20 Generate enough 4-cuts in order to separate the 4-edge-connected vertices190

21 Compute the 5-edge-connected components of a 3-edge-connected graph 192

22 Answering a collection of min-max vertex queries 221
23 Compute all Type-2i 4-cuts 227
24 Compute a collection of Type-2ii 4-cuts, which implies all Type-2ii 4-cuts 236
25 Compute all Type-2iii 4-cuts 241
26 Compute all Type-3αi 4-cuts (for a special case) 253
27 Compute a collection of Type-3αi 4-cuts, which implies all Type-3αi 4-cuts 258
28 Compute all Type-3αii 4-cuts (I) 267
29 Compute all Type-3αii 4-cuts (II) 271
30 Compute all Type-3αii 4-cuts (III) 277
31 Compute all Type-3αii 4-cuts (IV) 281
32 Compute all Type-3βi 4-cuts that satisfy (1) of Lemma 5.57 294
33 Compute a collection of Type-3βi 4-cuts that satisfy (2) of Lemma 5.57, which implies all such 4-cuts 303
34 Compute a collection of Type-3βi 4-cuts that satisfy (3) of Lemma 5.57, which implies all such 4-cuts 313
35 Compute all Type-3βi 4-cuts that satisfy (4) of Lemma 5.57 (I) 329
36 Compute all Type-3βi 4-cuts that satisfy (4) of Lemma 5.57 (II) 331
37 Compute the sets $U_1(v)$ 341
38 Compute all Type-3βii-1 4-cuts 343
39 Compute the values *firstW*(v) and *lastW*(v) 350
40 Compute the sets $U_2(v)$ 357
41 Compute a collection of Type-3βii-2 4-cuts of a special type, which implies all such 4-cuts 359
42 Compute the sets $\widetilde{W}(v)$ 366
43 Compute all Type-3βii-2 4-cuts of a special case 371
44 Compute the sets $U_3(v)$ 380
45 Compute a collection of Type-3βii-3 4-cuts, which implies all such 4-cuts 382
46 Compute the sets $U_4^1(v)$ 395
47 Compute a collection of Type-3βii-4 4-cuts of a special type, which implies all such 4-cuts 399
48 Compute the sets $U_4^2(v)$ 406
49 Compute all Type-3βii-4 4-cuts of a special type (I) 411
50 Compute the sets $U_4^3(v)$ 418
51 Compute all Type-3βii-4 4-cuts of a special type (II) 421

52 Compute the sets $U_4^4(v)$ . . . 426
53 Compute all Type-3βii-4 4-cuts of a special type (III) . . . 431

54 Compute all Type-1 edges to construct a connectivity graph for the internal components of $T \setminus F$ . . . 491
55 Compute enough Type-2 edges to construct a connectivity graph for the internal components of $T \setminus F$ . . . 495
56 Answer a connectivity query . . . 499

57 Update the decomposition tree after inserting a new edge to the graph . 518
58 Merge the children $D_1, \ldots, D_k$ of a 2-ecc node X . . . 519
59 Update the decomposition tree after inserting a new vertex to the graph 529
60 *compressPath*(T, x, y) . . . 545
61 *joinTrees*$(T_1, T_2, (x, y))$ . . . 546
62 `updateCactus`(D, z_1, z_2) . . . 553
63 `compressCyclePath`$(\mathsf{S}, \mathsf{x}, \mathsf{y})$ . . . 554
64 `joinCactuses`$(\mathsf{S}_1, \ldots, \mathsf{S}_\mathsf{k}, (\mathsf{x}_1, \mathsf{x}_2), \ldots, (\mathsf{x}_\mathsf{k}, \mathsf{x}_1))$ . . . 556
65 `initialize_cycle`$(\mathsf{C}, \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{k}, (\mathsf{x}_1, \mathsf{x}_2), \ldots, (\mathsf{x}_\mathsf{k}, \mathsf{x}_1))$ . . . 556
66 Compute a certificate for the maximal k-edge-connected subgraphs of G 557

Abstract

Evangelos Kosinas, Ph.D., Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, 2024.
Efficient Algorithms for Some Connectivity Problems, in Static and Dynamic Graphs.
Advisor: Loukas Georgiadis, Associate Professor.

Graphs are some of the most fundamental and widely used objects in computer science, and they appear naturally in a variety of applications. The notion of connectivity in graphs introduces itself immediately as a very basic and intuitive concept, and as such it is very important in the analysis of networks. Despite its rudimentary nature, it poses highly challenging computational problems, with both theoretical and practical interest. Many such problems are still unresolved, and demand a deeper understanding of the structure of graphs. Furthermore, the sheer size of the graphs that appear in real-world applications, and the fact that they change dynamically over time, makes those problems even more challenging.

In this thesis, we provide efficient algorithms for some connectivity problems in undirected graphs, in the static, dynamic, and sensitivity setting. Our contributions can be summarized as follows.

- We provide the first linear-time algorithms for computing the 4- and 5-edge-connected components in undirected multigraphs. This result answers a theoretical question, and sheds light on the possibility that a linear-time solution may exist for general k. Furthermore, the algorithms that we provide can have a very efficient implementation with the use of elementary data structures. Especially for the case $k = 5$, we provide a novel analysis of the structure of 4-edge cuts in 3-edge-connected graphs, that can guide us into a proper selection of them for our purposes. We believe that this analysis may provide a clue for a

general solution for the k-edge-connected components, or other related graph connectivity problems.

A key component in our algorithm for the case $k = 5$ is an oracle for answering connectivity queries for pairs of vertices in the presence of at most four edge-failures. Specifically, the oracle has size $O(n)$, it can be constructed in linear time, and it answers connectivity queries in the presence of at most four edge-failures in constant time, where n denotes the number of vertices of the graph. We note that this is a result of independent interest.

- We provide an oracle for efficiently answering connectivity queries in the presence of vertex failures. Specifically, we design a data structure that can handle an arbitrary but fixed number of vertex failures, so that it can efficiently answer connectivity queries between vertices in the remaining graph. This very basic connectivity problem has received the attention of researchers for more than a decade now, but the solutions that have been provided are highly complicated and very difficult to be implemented efficiently. On the other hand, our solution is arguably the simplest that has been proposed for this problem; it is relatively easy to describe and analyze, and it uses only standard textbook data structures. Furthermore, it even provides some trade-offs that improve on the state of the art in some respects.

- Finally, we deal with the computation of the maximal k-edge-connected subgraphs in incremental graphs. We provide a general framework that reduces this computation to the incremental maintenance of the k-edge-connected components. As a concrete application of this framework, we provide an algorithm for the incremental maintenance of the maximal 3-edge-connected subgraphs, by relying on algorithms and data structures for the incremental maintenance of the 3-edge-connected components. This provides a significant improvement over the state of the art, which is given by applying the best known static algorithm after every insertion. Furthermore, we provide fast constructions of sparse spanning subgraphs that have the same maximal k-edge-connected subgraphs as the original graph. These can be used in order to speed up computations that involve the maximal k-edge-connected subgraphs.

Εκτεταμενη Περιληψη

Ευάγγελος Κοσίνας, Δ.Δ., Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, 2024.
Αποδοτικοί Αλγόριθμοι για Ορισμένα Προβλήματα Συνεκτικότητας, σε Στατικά και Δυναμικά Γραφήματα.
Επιβλέπων: Λουκάς Γεωργιάδης, Αναπληρωτής Καθηγητής.

Τα γραφήματα εμφανίζονται με φυσικό τρόπο σε πολλά προβλήματα πληροφορικής ως ένας ιδανικός τρόπος για την οργάνωση των δεδομένων, καθώς αποτελούν τα μαθηματικά μοντέλα των σχέσεων. Για παράδειγμα, κάθε κοινωνικό δίκτυο που υποστηρίζει σχέσεις "φιλίας" ή "ακολούθου" μπορεί να αναπαρασταθεί ως ένα γράφημα, όπου οι άνθρωποι που το αποτελούν αναπαρίστανται ως κόμβοι, και δύο κόμβοι συνδέονται με μια ακμή όταν οι άνθρωποι στους οποίους αντιστοιχούν συνδέονται με την αντίστοιχη πραγματική σχέση.

Μία από τις πιο βασικές έννοιες που ορίζονται σε ένα γράφημα είναι η έννοια της συνεκτικότητας, δηλαδή το αν μπορεί ένας κόμβος να φτάσει κάποιον άλλον ακολουθώντας τις συνδέσεις του γραφήματος. Είναι εύκολο να διαπιστώσει κανείς ότι η έννοια της συνεκτικότητας έχει τεράστιο θεωρητικό και πρακτικό ενδιαφέρον. Για παράδειγμα, τίθενται διάφορα ερωτήματα αλγοριθμικής φύσεως, όπως π.χ. το να υπολογίσουμε ομάδες κόμβων που είναι "πολύ καλά" συνδεδεμένοι μεταξύ τους (με βάση διάφορες μετρικές συνεκτικότητας). Τέτοια προβλήματα έχουν ιδιαίτερο ενδιαφέρον στην πράξη, διότι τα γραφήματα που εμφανίζονται σε διάφορες εφαρμογές είναι τεράστια σε μέγεθος, οπότε γίνεται πολύ προκλητικό το πρόβλημα του αποδοτικού υπολογισμού μιας ακριβούς απάντησης. Ο μόνος τρόπος να επιτευχθεί αυτό είναι μέσω ειδικά σχεδιασμένων αλγορίθμων, που επεξεργάζονται με έξυπνο τρόπο την δομή του γραφήματος.

Σε αυτήν την διατριβή παρουσιάζουμε αποδοτικούς αλγορίθμους για ορισμένα προβλήματα συνεκτικότητας σε γραφήματα. Το πρώτο πρόβλημα που θα μας απα-

σχολήσει είναι ο υπολογισμός των k-συνεκτικών συνιστωσών. Οι k-συνεκτικές συνιστώσες ενός γραφήματος είναι τα μεγιστικά σύνολα κόμβων που παραμένουν συνδεδεμένοι μεταξύ τους ακόμη και αν αφαιρεθούν $k-1$ ακμές από το γράφημα (οπότε πρέπει να αφαιρέσουμε τουλάχιστον k ακμές για να καταστρέψουμε την συνεκτικότητα μεταξύ τους). Για την περίπτωση $k \leq 3$, υπάρχουν ήδη γνωστοί αλγόριθμοι γραμμικού χρόνου για αυτό το πρόβλημα. Δεδομένης της έρευνας που έχει γίνει πάνω στο ζήτημα της k-συνεκτικότητας, είναι εύλογο να υποθέσει κανείς ότι υπάρχουν αλγόριθμοι γραμμικού χρόνου και για οποιοδήποτε $k \geq 4$, αλλά αυτό δεν έχει αποδειχτεί ακόμη. Σε αυτήν την διατριβή, παρέχουμε τους πρώτους αλγορίθμους γραμμικού χρόνου για τις περιπτώσεις $k = 4$ και $k = 5$, ενισχύοντας έτσι την υπόθεση ότι αυτό το πρόβλημα έχει λύση γραμμικού χρόνου για οποιοδήποτε k. Η δυσκολία σε αυτό το πρόβλημα έγκειται στο ότι υπάρχουν πολλοί συνδυασμοί συνόλων ακμών που μπορούν να καταστρέψουν την συνεκτικότητα του γραφήματος με την αφαίρεσή τους. Τέτοια σύνολα κόμβων ονομάζονται *τομές*. Επομένως, η πρόκληση είναι να γίνει μία προσεκτική επιλογή τομών, ώστε να λάβουμε την διαμέριση στις k-συνεκτικές συνιστώσες. Προκειμένου να λύσουμε αυτό το πρόβλημα, βασιζόμαστε σε μία κατηγοριοποίηση των τομών μικρού μεγέθους (τριάδες ή τετράδες ακμών), με βάση την δομή που λαμβάνουμε μετά από μία καθοδική διερεύνηση του γραφήματος (DFS). Ειδικά για την περίπτωση $k = 5$, αυτό που μας καθοδηγεί στην επιλογή των τομών είναι μία θεωρητική ανάλυση της δομής τους, δηλαδή των τρόπων με τους οποίους μπορεί να εμπλέκονται μεταξύ τους, συμμεριζόμενες υποσύνολα ακμών. Πιστεύουμε ότι αυτή η ανάλυση μπορεί να έχει εφαρμογές και στην γενικότερη εκδοχή του προβλήματος, ή και σε άλλα σχετικά προβλήματα συνεκτικότητας.

Έπειτα, ασχολούμαστε με προβλήματα συνεκτικότητας σε *δυναμικά* γραφήματα. Ένα γράφημα ονομάζεται δυναμικό όταν μεταβάλλεται με την πάροδο του χρόνου, συνήθως με την μορφή εισαγωγών ή αφαιρέσεων κόμβων ή ακμών. Τέτοια γραφήματα εμφανίζονται με φυσικό τρόπο σε πολλές εφαρμογές. Η μεταβαλλόμενη δομή τους δυσχεραίνει σημαντικά την αποδοτικότητα στους υπολογισμούς σε σχέση με την περίπτωση όπου το γράφημα είναι στατικό. Μάλιστα, υπάρχουν πολλά παραδείγματα προβλημάτων όπου έχει αποδειχτεί (με βάση αναγωγές σε δημοφιλείς και αρκετά πιστευτές εικασίες) ότι γενικά δεν μπορούμε να κάνουμε κάτι καλύτερο από το να εφαρμόζουμε τον βέλτιστο στατικό αλγόριθμο μετά από κάθε αλλαγή στο γράφημα. Παρά ταύτα, υπάρχουν αρκετά προβλήματα στα οποία μπορούμε να

έχουμε αρκετά πιο αποδοτικές λύσεις. Η βασική ιδέα είναι ότι είτε κάθε μεμονωμένη αλλαγή (προσθήκη ή διαγραφή κόμβου ή ακμής) επιφέρει μια σχετικά μικρή μεταβολή στο γράφημα, ή ότι οι αλλαγές που μπορούν να επηρεάσουν ριζικά τις παραμέτρους που θέλουμε να υπολογίσουμε είναι σχετικά λίγες ή αντισταθμιστικά διαχειρίσιμες.

Μία βασική κατηγορία προβλημάτων σε δυναμικά γραφήματα προϋποθέτει ότι υπάρχει ένα όριο στο πλήθος των αλλαγών που μπορεί να έχει υποστεί το γράφημα σε κάθε δεδομένη χρονική στιγμή. Τέτοιες καταστάσεις εμφανίζονται π.χ. σε οδικά δίκτυα, όπου αναμένεται κάποιοι δρόμοι να πάψουν να λειτουργούν για ένα διάστημα (π.χ. λόγω έργων συντήρησης), αλλά έπειτα προβλέπεται να επανέλθει η λειτουργία τους. Σε αυτήν την διατριβή, ασχολούμαστε με το πρόβλημα του πώς μπορούμε να απαντούμε αποδοτικά σε ερωτήματα συνεκτικότητας όταν το γράφημα έχει υποστεί ένα περιορισμένο πλήθος απωλειών κόμβων. Πιο συγκεκριμένα, ο σκοπός είναι να κατασκευάσουμε μία δομή δεδομένων, η οποία να μπορεί να επεξεργαστεί αποδοτικά την πληροφορία ότι ένα (περιορισμένο σε πλήθος) σύνολο κόμβων έχει απενεργοποιηθεί, ώστε να μπορεί να απαντήσει αποδοτικά σε ερωτήματα της μορφής "δοθέντων δύο κόμβων x και y (που είναι ακόμη ενεργοί), παραμένουν οι x και y συνδεδεμένοι διά μέσου μονοπατιών που αποφεύγουν τους ανενεργούς κόμβους;". Σε αυτό το πρόβλημα, θέλουμε να βελτιστοποιήσουμε ταυτόχρονα (1) τον χρόνο κατασκευής της δομής δεδομένων, (2) τον χώρο που καταλαμβάνει, (3) τον χρόνο που χρειάζεται για να επεξεργαστεί την πληροφορία των απωλειών, και (4) τον χρόνο για να απαντήσει τα ερωτήματα συνεκτικότητας. Αυτό είναι ένα βασικό πρόβλημα συνεκτικότητας, και έχουν προταθεί διάφορες λύσεις στην βιβλιογραφία, αλλά καμία τους δεν είναι απολύτως βέλτιστη, διότι συνήθως η κάθε μια πλεονεκτεί από ορισμένες μονάχα απόψεις σε σχέση με τις άλλες. Η δομή δεδομένων που προτείνουμε εμείς, αν και δεν είναι βέλτιστη, είναι σίγουρα η πιο απλή και πρακτική λύση που έχει προταθεί μέχρι σήμερα, και η επίδοσή της είναι συγκρίσιμη με τις καλύτερες γνωστές λύσεις (και μάλιστα παρουσιάζει και μια θεωρητική βελτίωση από ορισμένες απόψεις).

Επιπλέον, ασχολούμαστε με το πρόβλημα του να απαντούμε αποδοτικά σε ερωτήματα συνεκτικότητας όταν το γράφημα έχει υποστεί απώλειες ακμών. Συγκεκριμένα, ασχολούμαστε με την περίπτωση όπου το γράφημα μπορεί να έχει χάσει μέχρι και τέσσερις ακμές. Πιο συγκεκριμένα, δείχνουμε ότι μπορούμε να κατασκευάσουμε σε γραμμικό χρόνο μία δομή δεδομένων που μπορεί να απαντάει σε

σταθερό χρόνο ερωτήματα της μορφής "δεδομένου ότι έχει χαθεί ένα σύνολο ακμών E' από το γράφημα, με $|E'| \leq 4$, υπάρχει μονοπάτι μεταξύ των κόμβων x και y που δεν χρησιμοποιεί ακμές από το E';". Αυτή η δομή μάς χρειάζεται στον αλγόριθμο που υπολογίζει τις 5-συνεκτικές συνιστώσες, όμως συνιστά και ένα αποτέλεσμα ανεξάρτητου ενδιαφέροντος, καθώς αποτελεί την πρώτη ουσιαστικά βέλτιστη λύση σε αυτό το πρόβλημα.

Τέλος, ασχολούμαστε με το πρόβλημα του υπολογισμού των μεγιστικών k-συνεκτικών υπογραφημάτων ενός γραφήματος. Αν και αυτό το πρόβλημα συγγενεύει στενά με εκείνο του υπολογισμού των k-συνεκτικών συνιστωσών, ο υπολογισμός των μεγιστικών k-συνεκτικών υπογραφημάτων παρουσιάζει μια ιδαίτερη δυσκολία, εφόσον δεν έχουν βρεθεί ακόμη εξ ίσου αποδοτικοί αλγόριθμοι για αυτό, παρά την εκτεταμένη έρευνα που έχει γίνει μέχρι σήμερα. Το σημαντικότερο αποτέλεσμά μας εδώ είναι ότι παρέχουμε ένα γενικό πλαίσιο για την αποδοτική διατήρηση των μεγιστικών k-συνεκτικών υπογραφημάτων καθώς το γράφημα υφίσταται εισαγωγές κόμβων ή ακμών, μέσω αναγωγής σε αλγορίθμους για την διατήρηση των k-συνεκτικών συνιστωσών. Η δομή που προτείνουμε έχει την μορφή ενός δέντρου που αναπαριστά την επαναλαμβανόμενη διαμέριση στις k-συνεκτικές συνιστώσες, μέχρι να φτάσουμε στα μεγιστικά k-συνεκτικά υπογραφήματα. Αυτή η ιδέα αποκαλύπτει και από μία άλλη άποψη την ιδιαίτερη δυσκολία που έχει το πρόβλημα των μεγιστικών k-συνεκτικών υπογραφημάτων, σε σχέση με αυτό των k-συνεκτικών συνιστωσών. Ως μια απτή εφαρμογή αυτής της ιδέας, παρέχουμε αποδοτικούς αλγορίθμους για την διατήρηση των μεγιστικών 3-συνεκτικών υπογραφημάτων, καθώς για την περίπτωση $k = 3$ μπορούμε να βασιστούμε σε γνωστούς αλγορίθμους και δομές δεδομένων για την διατήρηση των 3-συνεκτικών συνιστωσών. Τέλος, παρέχουμε αποδοτικές κατασκευές για αραιά υπογραφήματα που έχουν τα ίδια μεγιστικά k-συνεκτικά υπογραφήματα με το αρχικό γράφημα. Τέτοιες κατασκευές μπορούν να επιταχύνουν διάφορους υπολογισμούς που αφορούν τα μεγιστικά k-συνεκτικά υπογραφήματα.

Chapter 1

Introduction

1.1 Objective
1.2 Overview of our results
1.3 Organization

1.1 Objective

Our goal in this thesis is to provide efficient algorithms for some connectivity problems in undirected graphs, in the static, dynamic, and sensitivity setting. Specifically, we aim at algorithms that either (1) provide time and space bounds that are asymptotically optimal, or (2) improve the previous best known bounds, or (3) are relatively simple to describe, analyze and implement, while providing bounds that compare very well with the previous best. (Ideally, the "or" here should be inclusive.) In order to discuss the problems that we wanted to solve, we assume some familiarity with standard graph-theoretic terminology, that can be found e.g. in [20] or [52], and we also refer to Chapter 2.

First, we consider the problem of computing the k-edge-connected components. Here we focus on the case $k \in \{4, 5\}$, because for $k \leq 3$ it is already known how to compute the k-edge-connected components in linear time. In particular, we provide the first linear-time algorithms for computing the k-edge-connected components in the case where $k = 4$ or $k = 5$ (see Chapters 4 and 5, respectively).

Then we consider the problem of designing an oracle for efficiently answering connectivity queries in the presence of failures. The goal here is to preprocess a graph in order to build a data structure that can efficiently answer connectivity queries for pairs of vertices, after receiving information about a bounded number of failures (of vertices or edges).[1] Here we want to simultaneously optimize (1) the preprocessing time, (2) the space usage of the data structure, (3) the time to handle the information of failures (updates), and (4) the time to answer the queries. In Chapter 6, we provide an (essentially) optimal solution for the case where we want to be able to handle at most four edge-failures. In Chapter 7, we provide a solution that can handle an arbitrary (but fixed) number of vertex failures; our solution is very efficient, and it is arguably the simplest that has been proposed for this problem.

Finally, we consider the problem of maintaining the maximal k-edge-connected subgraphs of a graph after insertions of vertices or edges. We focus primarily on the case $k = 3$, because here we can rely on existing data structures in order to provide algorithms that are more efficient than re-computing the solution from scratch after every insertion.

1.2 Overview of our results

1.2.1 Computing the 4-edge-connected components

In Chapter 4 we provide a linear-time algorithm for computing the 4-edge-connected components of an undirected multigraph. This result is based on our paper with title "Computing the 4-Edge-Connected Components of a Graph in Linear Time", which was done in collaboration with Loukas Georgiadis and Giuseppe F. Italiano, and has been presented at the European Symposium on Algorithms (ESA), 2021. We note that in 2021 another group of researchers has independently reached the same result [50]. However, the algorithm that we provide here has a linear-time implementation in the pointer-machine model of computation [65], and thus it provides a theoretical improvement over [50] (which relies on the RAM model of computation to achieve linear time). Furthermore, in Section 4.5 we provide a very simple linear-time algorithm for determining whether a graph is 4-edge-connected.

[1] A nice way to think about this problem is in the context of "emergency planning" [60].

The general idea for computing the 4-edge-connected components can be described as follows. First, we use a construction described in [21], that reduces this computation to 3-edge-connected graphs. Then, the problem reduces to the computation of all 3-cuts of a 3-edge-connected graph. To perform this computation efficiently, we rely on a DFS-tree T of the graph, and we provide a typology of 3-cuts w.r.t. T. Specifically, we distinguish three types of 3-cuts, depending on the number of tree-edges of T that they contain (notice that a 3-cut must contain at least one tree-edge of T). Then, we can compute all three types of 3-cuts separately. The case of 3-cuts that contain exactly one tree-edge is the easiest one. The case of 3-cuts that contain exactly two tree-edges is the most demanding, and we further distinguish it into various subcases. Finally, the case of 3-cuts that consist of three tree-edges can be easily reduced to the previous two cases, as described in [50]. For the computation of 3-cuts, we rely on some DFS-based parameters that we introduce in Chapter 3. In particular, all the parameters that we use here can be computed with linear-time algorithms with a pointer-machine implementation.

1.2.2 Computing the 5-edge-connected components

In Chapter 5 we provide the first linear-time algorithm for computing the 5-edge-connected components of an undirected multigraph. There were probably good indications that this computation can be performed in linear time, but no such algorithm was actually known prior to this work. Thus, our results answers a theoretical question, and sheds light on the possibility that a solution may exist for general k. Furthermore, although the algorithm that we provide is quite extensive and broken up into several pieces, it can have an almost-linear time implementation with the use of elementary data structures.

This algorithm can be considered as a follow-up of previous work on computing the 4-edge-connected components in linear time. Specifically, we follow a DFS-based approach in order to compute a collection of 4-edge cuts, that is rich enough in properties for our purposes. However, in dealing with the computation of the 5-edge-connected components, we are faced with unique challenges that do not appear when dealing with lower connectivity. The problem is that the 4-edge cuts in 3-edge-connected graphs are entangled in various complicated ways, that make it difficult to organize them in a compact way. Here we provide a novel analysis of those cuts,

that reveals the existence of various interesting structures. These can be exploited so that we can disentangle and collect only those cuts that are essential in computing the 5-edge-connected components. This analysis may provide a clue for a general solution for the k-edge-connected components, or other related graph connectivity problems.

We note that the problem of computing the k-edge-connected components is related to the problem of computing a Gomory-Hu tree [39], which provides the edge-connectivity for all pairs of vertices.[2] A very recent line of impressive work [47, 3, 2, 4] has culminated in a randomized Monte Carlo algorithm for computing a Gomory-Hu tree of a weighted graph in $m^{1+o(1)}$ time, where m is the number of edges of the graph. Then, given a Gomory-Hu tree of the graph, we can derive the partition of the k-edge-connected components in linear time, for any fixed k. Thus, the k-edge-connected components can be computed with a randomized Monte Carlo algorithm in $m^{1+o(1)}$ time, for any k. However, the question still remains, whether we can compute the k-edge-connected components deterministically in linear time, for any $k > 5$.

The results of Chapter 5 are based on our paper with title "Computing the 5-Edge-Connected Components in Linear Time", which was presented at the ACM-SIAM Symposium on Discrete Algorithms (SODA), 2024.

1.2.3 Connectivity queries under four edge failures

In Chapter 6 we provide an (essentially) optimal solution for answering connectivity queries in the presence of at most four edge failures. Specifically, we provide the following result, that is summarized in Proposition 6.1. Let G be a connected graph with n vertices and m edges. Then, in linear time, we can construct a data structure with size $O(n)$, that we can use in order to answer connectivity queries in the presence of at most four edge failures in constant time. Specifically, given an edge-set E' with $|E'| \leq 4$, and two vertices x and y, we can determine whether x and y are connected in $G \setminus E'$ in $O(1)$ time.

We note that this result is a special instance of the general problem of designing an oracle that answers connectivity queries in the presence of edge-failures [60]. The currently best bounds for this problem are given by Duan and Pettie in [26], where they show how to construct an oracle of $O(m \log\log n)$ (or $O(m)$) size[3], so

[2] For more information, we refer to Section 5.1.2.

[3] The time-bounds for constructing the oracle are not specified.

that, given a set E' of at most d edges, one can answer connectivity queries in $G \setminus E'$ in $O(min\{\frac{\log\log n}{\log\log\log n}, \frac{\log d}{\log\log n}\})$ time, after a $O(d^2 \log\log n)$-time (or $O(d^2 \log^{\epsilon} n)$-time, for any $\epsilon > 0$) preprocessing. Thus, the oracle that we provide improves on the state of the art in the case where d is a fixed constant, upper bounded by 4. It is an interesting question whether we can achieve the same bounds for larger fixed d. We believe that this is probably the case, but it appears that this is a very challenging combinatorial problem.

We achieve this result by using a DFS-tree T of G, and by making a creative use of some of the DFS parameters that we introduce in Chapter 3, in order to reconstruct on a high-level the connected components of the graph upon removal of a set of (at most four) edges. More specifically, given a set E' of at most four edge-failures, we consider all the different topologies of the edges in E' w.r.t. T. Thus, we distinguish various cases, depending e.g. on the number of tree-edges that are contained in E', or their ancestry relation on T, and we show how to handle each particular case separately, in order to determine the connectivity relation of the connected components of $T \setminus E'$ in $G \setminus E'$.

The result of Chapter 6 was provided in our paper with title "Computing the 5-Edge-Connected Components in Linear Time", because we need it in order to provide our linear-time algorithm for computing the 5-edge-connected components. More specifically, we use Proposition 6.1 in order to provide an oracle for checking the *essentiality* of 4-cuts, in an on-line manner, in constant time per query. (See Proposition 5.4; we refer to Section 2.3 for the definition of the essential 4-cuts.) However, the data structure described in Proposition 6.1 is a result of independent interest.

1.2.4 Connectivity queries under vertex failures

In Chapter 7 we provide an oracle for efficiently answering connectivity queries in the presence of vertex failures. Specifically, the input to this problem are an undirected graph G with n vertices and m edges, and a fixed integer $d_\star$ ($d_\star \ll n$). Then, the goal is to construct a data structure $\mathcal{D}$ that can be used in order to answer efficiently connectivity queries in the presence of at most $d_\star$ vertex-failures. More precisely, given a set of vertices F, with $|F| \leq d_\star$, we must be able to efficiently derive an oracle from $\mathcal{D}$, which can efficiently answer queries of the form "are the vertices x and y connected in $G \setminus F$?". In this problem, we want to simultaneously optimize

the following parameters: (1) the construction time of $\mathcal{D}$ (preprocessing time), (2) the space usage of $\mathcal{D}$, (3) the time to derive the oracle from $\mathcal{D}$ given F (update time), and (4) the time to answer a connectivity query in $G \setminus F$.

We provide a deterministic data structure for this problem that has preprocessing time $O(d_\star m \log n)$, uses space $O(d_\star m \log n)$, and has $O(d^4 \log n)$ update time and $O(d)$ query time. Although this is not an optimal solution, the bounds that we provide compare very well with the previous best, and even improve them in some respects. (For more details, we refer to Section 7.1.2, and especially Tables 8.1 and 8.2.) We note that this is a problem with various parameters, and thus it is very difficult to optimize all of them simultaneously. In fact, there are at least three different known solutions for this problem in the literature, every one of which is better than the others in some respects. (For example, in the update time of [55], there is no dependency on n; however, this is superexponential in $d_\star$, whereas in the data structures of [26] and [49] the dependency on $d_\star$ is polynomial.) Perhaps the most important aspect of our own solution is the simplicity of our approach: this is arguably the simplest solution that has been proposed for this problem. Furthermore, it uses only standard textbook data structures. We believe that it is important to have a simple and practical solution for this very basic connectivity problem.

Our solution relies on a DFS tree T of G. Given a set of failed vertices F, the high-level idea is to reconstruct the connectivity relation of some of the connected components of $T \setminus F$ in $G \setminus F$, guided by the non-tree edges of the graph. (Notice that we cannot afford to reconstruct the connectivity relation for *all* the connected components of $T \setminus F$, because their number can be as high as $n-1$, even when $|F| = 1$.) The crucial observation is that there are only at most $|F|$ connected components of $T \setminus F$ that are ancestors of failed vertices (which we call *internal components*), and that these are enough in order to efficiently capture the connectivity between the remaining components as well (which we call *hanging subtrees*). We rely on 2D-range-emptiness data structures [19], in order to efficiently perform queries for the existence of non-tree edges that connect subtrees of T.

A very useful property of our data structure is that it can be efficiently adapted to changes in the sensitivity parameter $d_\star$. Thus, we can rely on the already computed data structure, in order to augment it so that it can handle more failures. To be specific, let $\mathcal{D}(d_\star)$ denote the data structure that handles up to $d_\star$ failures on G, and let $\mathcal{T}(d_\star)$ denote the time for initializing $\mathcal{D}(d_\star)$. Then, if we have computed $\mathcal{D}(d_\star)$, for

some $d_\star \geq 0$, we can derive $\mathcal{D}(d_\star + d')$, for any $d' \geq 0$, in time $O(\mathcal{T}(d'))$, by adding some extra items on $\mathcal{D}(d_\star)$. (Thus, we can avoid recomputing $\mathcal{D}(d_\star + d')$ from scratch, which would take time $\mathcal{T}(d_\star + d')$, for any $d' \geq 0$.) This also allows us to free some space, if later on we want to handle less failures, by simply discarding the extra items that we have computed. We note that it is very natural to ask whether a solution for a sensitivity problem satisfies such a property, and we call it *flexibility* of the data structure (w.r.t. the sensitivity parameter $d_\star$). As far as we know, we are the first to take notice of this aspect of the problem.

The result of Chapter 7 is based on our paper with title "Connectivity Queries under Vertex Failures: Not Optimal, but Practical", which was presented at the European Symposium on Algorithms (ESA), 2023.

1.2.5 On computing the maximal k-edge-connected subgraphs

In Chapter 8 we provide the following new results on maximal k-edge-connected subgraphs of undirected graphs.

1. A general framework for maintaining the maximal k-edge-connected subgraphs upon insertions of edges or vertices, by successively partitioning the graph into its k-edge-connected components. This defines a decomposition tree, which can be maintained by using algorithms for the incremental maintenance of the k-edge-connected components as black boxes at every level of the tree.

2. As a concrete application of this framework, we provide two algorithms for the incremental maintenance of the maximal 3-edge-connected subgraphs. These algorithms allow for vertex and edge insertions, interspersed with queries asking whether two vertices belong to the same maximal 3-edge-connected subgraph, and there is a trade-off between their time- and space-complexity. Specifically, the first algorithm has $O(m\alpha(m,n)+n^2\log^2 n)$ total running time and uses $O(n)$ space, where m is the number of edge insertions and queries, and n is the total number of vertices inserted starting from an empty graph. The second algorithm performs the same operations in faster $O(m\alpha(m,n) + n^2\alpha(n,n))$ time in total, using $O(n^2)$ space.

3. We provide efficient constructions of (almost) sparse spanning subgraphs that have the same maximal k-edge-connected subgraphs as the original graph. We

refer to such subgraphs as k-*certificates*. We use those certificates to speed up the computation of the maximal k-edge-connected subgraphs in the static and the fully-dynamic setting.

4. Finally, we give a simple reduction for computing the maximal k-edge-connected subgraphs to fully dynamic mincut. By using Thorup's fully dynamic mincut algorithm [66], we obtain a deterministic algorithm that computes the maximal k-edge-connected subgraphs in $O(m+k^{O(1)}n\sqrt{n}\log^{O(1)} n)$ time, for $k = \log^{O(1)} n$.

The results of Chapter 8 are a joint work with Loukas Georgiadis, Giuseppe F. Italiano, and Debasish Pattanayak.

1.3 Organization

In Chapter 2 we provide some basic definitions and notation that we will use throughout. In Chapter 3 we introduce some DFS-based concepts that we will use in Chapters 4, 5 and 6, and we provide efficient algorithms for their computation. In Chapters 4 and 5 we provide our results that concern the computation of the 4-edge and the 5-edge-connected components, respectively. In Chapter 6 we provide the oracle for answering connectivity queries in the presence of at most four edge-failures. In Chapter 7 we provide the oracle for answering connectivity queries in the presence of vertex failures. It is important to note that Chapter 7 is essentially self-contained; in particular, the "*low*" points that are used in Chapter 7 are different from those that are introduced in Chapter 3. Finally, in Chapter 8 we provide our results that concern the computation and the incremental maintenance of the maximal k-edge-connected subgraphs.

Chapter 2

Preliminaries

2.1 Basic graph terminology

2.2 Partitions and atoms

2.3 Edge-connectivity and k-edge-connected components

2.4 Maximal k-edge-connected subgraphs

2.5 Notation

2.1 Basic graph terminology

In this work, all graphs considered are undirected multigraphs (i.e., they may have parallel edges). We use standard graph-theoretic terminology, that can be found e.g. in [20] or [52]. Let $G = (V, E)$ be a graph. We let $V(G)$ and $E(G)$ denote the vertex-set and the edge-set of G, respectively. (That is, we have $V(G) = V$ and $E(G) = E$.) Since G is a multigraph, it may have multiple edges of the form (x, y), for two vertices x and y. Thus, in order to be precise, we should also include an index to specify the edge we are referring to. That is, every edge $e \in E$ should be written as (x, y, i), where x and y are the endpoints of e, and i is a unique identifier of $e \in E$. However, we keep our notation simple (i.e., we identify edges just with the tuple of their endpoints), and this will not affect our arguments.

For every two subsets X and Y of V, we let $E_G[X, Y]$ denote the set of the edges of G with one endpoint in X and the other endpoint in Y. We may skip the subscript "G" from this notation if the reference graph is clear from the context. Although it is

not necessary that X and Y are disjoint, wherever we use this notation in the sequel we have that X and Y are disjoint. We let $\partial_G(X)$ denote $E_G[X, V \setminus X]$. Again, we may skip the subscript "G" when no confusion arises. For a singleton $\{v\}$ that consists of a vertex v, we may simply write $\partial(v)$ instead of $\partial(\{v\})$. If X is a set of vertices of G, we let $G[X]$ denote the induced subgraph on X. This is the graph with vertex set X and edge set $\{(x, y) \in E \mid x \in X \text{ and } y \in X\}$. If C is a subset of edges of G, we use $G \setminus C$ to denote the graph $(V, E \setminus C)$. If C consists of a single edge e, we may use the simplified notation $G \setminus e := G \setminus \{e\}$.

A *path* P in G is an alternating sequence $x_1, e_1, \ldots, x_{k-1}, e_{k-1}, x_k$, with $k \geq 1$, of vertices and edges of G, starting with a vertex x_1 and ending with a vertex x_k, such that $e_i = (x_i, x_{i+1})$ for every $i \in \{1, \ldots, k-1\}$. In this case, we say that P is a path from x_1 to x_k in G, and these are called the beginning and the end, respectively, of P. Furthermore, we say that P passes from the vertices $x_1, \ldots, x_k$, and uses the edges $e_1, \ldots, e_{k-1}$. If P and Q are two paths such that the end of P coincides with the beginning of Q, then $P + Q$ denotes the path that is formed by the concatenation of P and Q (by discarding either the end of P or the beginning of Q, in order to keep a single copy of it as the concatenation point). Thus, if P is a path from x to y, and Q is a path from y to z, then $P + Q$ is a path from x to z. For every two vertices x and y of G, we say that x is connected with y if there is a path from x to y in G. Notice that this defines an equivalence relation on $V(G)$; its equivalence classes are called the *connected components* of G. In particular, if $V(G)$ is the only connected component of G, then G is called a *connected* graph. Otherwise, it is called disconnected.

Let G be a connected graph. A bipartition $\{X, V \setminus X\}$ of $V(G)$ is called a *cut* of G. The corresponding edge-set $C = E[X, V \setminus X]$ has the property that its removal from G increases the number of connected components at least by one. Since G is connected, it is not difficult to verify that C is uniquely determined by X (i.e., $\{X, V \setminus X\}$ is the only bipartition of $V(G)$ whose corresponding edge-set is C). Thus, we also call C a cut of G. We will be using the term "cut" to denote interchangeably a bipartition of G and the edge-set that is derived from it. It will be clear from the context whether the term cut refers to a bipartition or its corresponding edge-set. In particular, whenever we speak of the "edges" of a cut, we consider it as an edge-set. Notice that a set C of edges with the property that $G \setminus C$ is disconnected is not necessarily a cut of G. However, this is definitely the case when C is minimal w.r.t. this property (i.e., if no proper subset C' of C has the property that $G \setminus C'$ is disconnected). In this work, we

will deal exclusively with *edge-minimal* cuts. An edge-minimal cut C is a set of edges with the property that $G \setminus C$ is disconnected, but $G \setminus C'$ is connected for every proper subset C' of C. Notice that C has the property that $G \setminus C$ consists of two connected components X and $V \setminus X$. These are called the *sides* of C, and they have the property that $C = E[X, V \setminus X]$. From now on, the term "cut" will always mean "edge-minimal cut". A cut with k edges is called a k-cut. We let $\mathcal{C}_{kcuts}(G)$ denote the collection of all k-cuts of G. Since the reference graph will always be clear from the context, we will simply denote this as $\mathcal{C}_{kcuts}$. Following standard terminology, we refer to the 1-cuts as *bridges*.

Let G be a graph and let $\{V_1, \ldots, V_k\}$ be a partition of $V(G)$. By shrinking every one of $V_1, \ldots, V_k$ into a single node, and by ignoring the self-loops that may be formed, we get a new graph Q, which is called a *quotient graph* of G. Thus, the vertex-set of Q is given by $\{V_1, \ldots, V_k\}$. In order to describe the edge-set of Q precisely, we need the quotient map $q : V(G) \to V(Q)$, which maps every $x \in V(G)$ into the unique set from $\{V_1, \ldots, V_k\}$ that contains it. By slightly abusing notation, we extend the quotient map as follows. For every edge $e = (x, y) \in E(G)$, we let $q(e) = (q(x), q(y))$.[1] Then, the edge-set of Q is given by $\{q(e) \mid e = (x, y) \in E(G) \text{ and } q(x) \neq q(y)\}$. Thus, if there are two distinct edges $(x, y), (x', y') \in E(G)$ such that $q(x) \neq q(y)$, $q(x) = q(x')$ and $q(y) = q(y')$, then Q contains at least two parallel edges of the form $(q(x), q(y))$.

2.2 Partitions and atoms

Let P be a partition of a set V. Then we say that P *separates* two elements $x, y \in V$ if and only if x and y belong to different sets from P. A *refinement* P' of P is a partition with the property that every set in P' is a subset of a set in P. Equivalently, P' is a refinement of P if and only if every two elements separated by P are also separated by P'. The common refinement of two partitions P and Q is the unique partition R with the property that two elements are separated by R if and only if they are separated by either P or Q. Equivalently, R is given by the collection of the non-empty intersections of the form $X \cap Y$, where $X \in P$ and $Y \in Q$.

Let $\mathcal{P}$ be a collection of partitions of a set V. Then, the *atoms* of $\mathcal{P}$, denoted as

[1] More precisely, for every $e = (x, y, i) \in E(G)$, we let $q(e) = (q(x), q(y), i)$, where i is the unique identifier of e.

atoms$(\mathcal{P})$, is the common refinement of all partitions in $\mathcal{P}$. In other words, the partition *atoms*$(\mathcal{P})$ is defined by the property that two elements x and y of V are separated by *atoms*$(\mathcal{P})$ if and only if they are separated by a partition in $\mathcal{P}$.

We are particularly interested in collections of bipartitions. Two bipartitions $P = \{X, Y\}$ and $Q = \{X', Y'\}$ of V are called *parallel* if at least one of the intersections $X \cap X'$, $X \cap Y'$, $Y \cap X'$, $Y \cap Y'$ is empty. (Notice that, in general, at most one of those intersections may be empty, unless $P = Q$, in which case precisely two of those intersections are empty.) Otherwise, if none of the intersections $X \cap X'$, $X \cap Y'$, $Y \cap X'$, $Y \cap Y'$ is empty, then we say that P and Q *cross*. Notice that a collection of bipartitions of a set with n elements can have size $\Omega(2^n)$. However, the number of partitions in a collection of bipartitions that are pairwise parallel is bounded by $O(n)$ [24].

This terminology concerning partitions can be naturally applied to (collections of) cuts, since these are defined as bipartitions of the vertex set of a graph. Thus, we may speak of cuts that are parallel, or that cross. Also, if $\mathcal{C}$ is a collection of cuts, then the partition *atoms*$(\mathcal{C})$ is defined. (Regardless of whether $\mathcal{C}$ was considered as a collection of sets of edges, the expression *atoms*$(\mathcal{C})$ interprets $\mathcal{C}$ as a collection of bipartitions.) A collection of cuts that are pairwise parallel is called a parallel family of cuts. Thus, a parallel family of cuts of a graph G contains $O(|V(G)|)$ cuts.

2.3 Edge-connectivity and k-edge-connected components

It is natural to ask how "well connected" is a graph, and various concepts have been developed to capture this notion. Let G be a connected graph. We say that two vertices x and y of G are *k-edge-connected*, if we have to remove at least k edges from G in order to disconnect them. Equivalently, by Menger's theorem we have that x and y are k-edge-connected if and only if there are k edge-disjoint paths from x to y (see, e.g., [52]). The maximum k such that x and y are k-edge-connected is called the *edge-connectivity* of x and y, denoted as $\lambda(x, y)$.

It is not difficult to see that the relation of k-edge-connectivity defines an equivalence relation on $V(G)$; its equivalence classes are called the *k-edge-connected components* of G. Equivalently, a k-edge-connected component of G is a maximal set of vertices with the property that every pair of vertices x, y in it has $\lambda(x, y) \geq k$ (and so we have to remove at least k edges in order to disconnect them). Notice that, if two

vertices x and y are not k-edge-connected, then there is a k'-cut that separates them, for some $k' < k$. Thus, the collection of the k-edge-connected components is given by *atoms*$(\mathcal{C}_{\mathit{1cuts}} \cup \cdots \cup \mathcal{C}_{\mathit{(k-1)cuts}})$. If $V(G)$ is the unique k-edge-connected component of G, then G is called k-edge-connected. Equivalently, G is k-edge-connected if and only if we have to remove at least k edges in order to disconnect it.

In Chapter 5 we provide a novel analysis of the structure of 4-cuts in 3-edge-connected graphs, that is useful in order to compute the 5-edge-connected components in linear time. A key idea in our analysis is to consider the 4-cuts that separate at least one pair of vertices that are 4-edge-connected. We call those 4-cuts *essential*, because they strictly refine the collection of the 4-edge-connected components. On the other hand, there are 4-cuts that separate vertices that are only 3-edge-connected, but not 4-edge-connected. These 4-cuts can be discarded for the purpose of computing the atoms of $\mathcal{C}_{\mathit{3cuts}} \cup \mathcal{C}_{\mathit{4cuts}}$, because the separations that are induced by them are captured by the separations according to the $\mathcal{C}_{\mathit{3cuts}}$ (i.e., the 4-edge-connected components).

2.4 Maximal k-edge-connected subgraphs

Let $G = (V, E)$ be a connected undirected multigraph with m edges and n vertices, and let $S \subseteq V$ be a subset of vertices of G. We say that the induced subgraph $G[S]$ is a *maximal k-edge-connected subgraph* of G if (1) $G[S]$ is k-edge-connected and, (2) no proper superset of S has this property. Unlike 2-edge connectivity, for $k \geq 3$ the k-edge-connected components of G do not necessarily correspond to maximal k-edge-connected subgraphs. Indeed, for $k \geq 3$, two vertices in the subgraph induced by a k-edge-connected component may not be k-edge-connected in this subgraph, as some of the k edge-disjoint paths may go outside of the component; see Figure 2.1. Also, Figure 2.2 shows a graph where almost all its vertices are 3-edge-connected but has only trivial maximal 3-edge-connected subgraphs. Notice that, if S is a subset of V, then $\lambda_{G[S]}(x, y) \leq \lambda_G(x, y)$ for any pair of vertices $x, y \in S$, since every path in $G[S]$ is also a path in G. Thus, every maximal k-edge-connected subgraph lies within the subgraph induced by a k-edge-connected component.

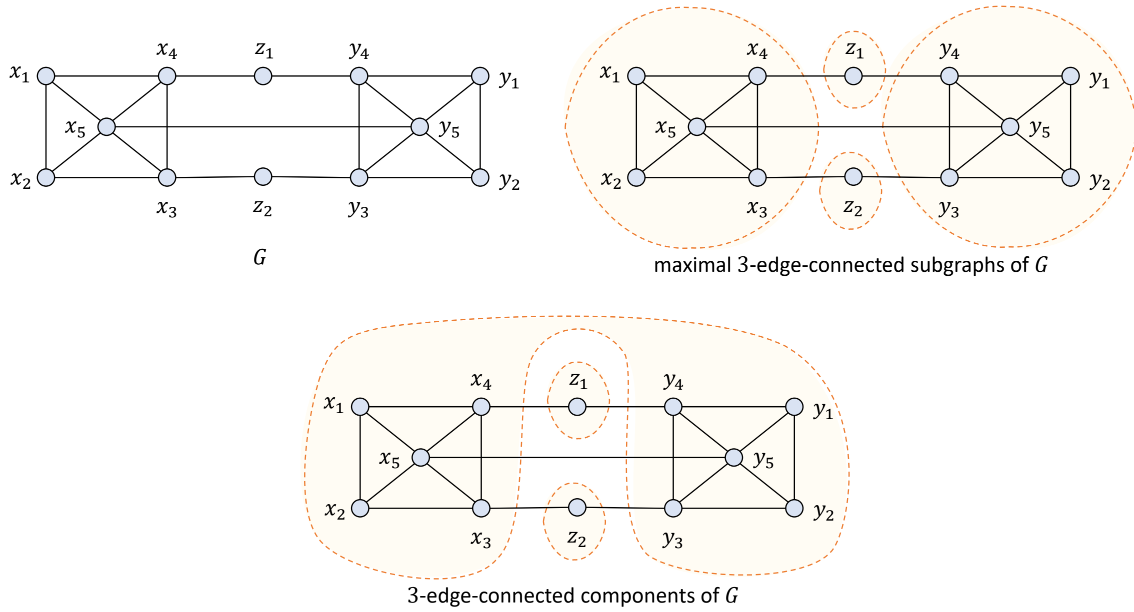


Figure 2.1: A 2-edge-connected graph G, its maximal 3-edge-connected subgraphs, and its 3-edge-connected components. Note that while any two vertices x_i and y_j are 3-edge-connected, they do not belong to the same maximal 3-edge-connected subgraph.

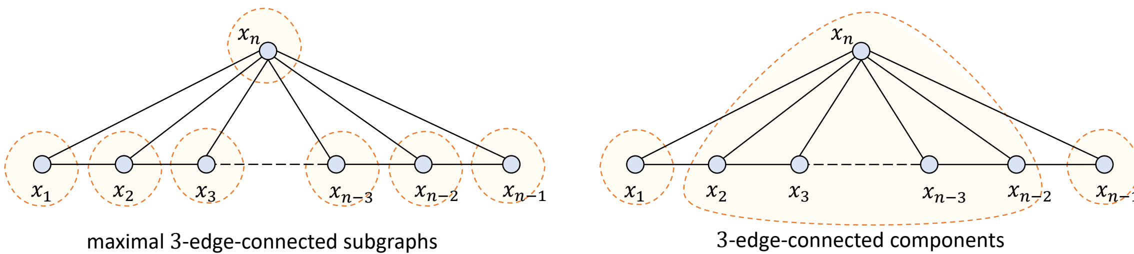


Figure 2.2: A 2-edge-connected graph G with only trivial maximal 3-edge-connected subgraphs, despite that almost all vertices are 3-edge-connected.

2.5 Notation

Here we introduce some notation that we will use throughout.

For any two positive integers i and k such that $i \in \{1, \ldots, k\}$, we define the "cyclic" addition and subtraction as:

$$i +_k 1 = \begin{cases} i+1 & \text{if } i < k \\ 1 & \text{if } i = k \end{cases}$$

$$i -_k 1 = \begin{cases} i-1 & \text{if } i > 1 \\ k & \text{if } i = 1 \end{cases}$$

For any two sets A and B such that $A \cap B = \emptyset$, we use the notation $A \sqcup B$ to denote the union $A \cup B$, while emphasizing the fact that $A \cap B = \emptyset$. This notation will be very convenient for our argumentation, because it packs more information in a single symbol. Also, whenever we use the expression $A \subset B$, we imply that A is a *proper* subset of B (and thus $A \neq B$). Otherwise, if $A = B$ is allowed, then we write $A \subseteq B$.

If L is a sorted list of elements and x is an element in L, we use $\mathit{next}_L(x)$ and $\mathit{prev}_L(x)$ to denote the successor and the predecessor, respectively, of x in L. We use $\perp$ to denote the end-of-list element. A *segment* of L is a sublist of consecutive elements. For convenience, sometimes we may view a list as a set (and write, e.g., $x \in L$).

If $f : X \to Y$ is a function and $\mathcal{C}$ is a collection of subsets of Y, we let $f^{-1}(\mathcal{C})$ denote the collection $\{f^{-1}(C) \mid C \in \mathcal{C}\}$ of subsets of X.

For every graph that we consider, we assume a total ordering of its edge set (e.g., lexicographic order). If $p = \{e, e'\}$ is a pair of edges of a graph and $e < e'$, then we let $\vec{p}$ denote the ordered pair of edges (e, e').

Chapter 3

Concepts defined on a DFS Tree

3.1 Basic definitions
3.2 Properties of the DFS parameters
3.3 Computing the *low*-edges
3.4 Computing the *high*-edges
3.5 Computing the leftmost and the rightmost edges
3.6 Computing the maximum points
3.7 Pointer-machine algorithms for some DFS parameters
3.8 Two lemmata concerning paths
3.9 An oracle for back-edge queries
3.10 Segments of vertices that have the same *high* point

Throughout this chapter we assume that G is a connected graph with n vertices and m edges, and r is a vertex of G. In Section 3.1 we present the parameters that are defined w.r.t. a DFS-tree of G, and we will use throughout in the rest of this work. In Section 3.2 we state and prove some simple properties that are satisfied by the DFS parameters. In Sections 3.3, 3.4, 3.5 and 3.6, we show how to compute the *low* edges, the *high* edges, the leftmost and the rightmost edges, and the M points, respectively. In Section 3.7 we provide alternative linear-time algorithms for the computation of some of the DFS parameters, that are implementable in the pointer machine model

and will be useful in Chapter 4. In Section 3.8 we prove two lemmata that concern the structure of paths w.r.t. a DFS-tree. In Section 3.9 we present an oracle for back-edge queries that we will use in our oracle for connectivity queries in the presence of at most four edge-failures in Section 6. Finally, we conclude with Section 3.10 that deals with the computation of the decreasingly ordered segments that consist of vertices that have the same *high* (or $high_2$) point, and are maximal w.r.t. the property that their elements are related as ancestor and descendant. These segments are involved in the computation of Type-$3\beta ii$ 4-cuts, in Section 5.8.2.

3.1 Basic definitions

Let T be a DFS-tree of G with start vertex r [63]. We identify the vertices of G with their order of visit by the DFS. (Thus, $r = 1$, and the last vertex visited by G is n.) For a vertex $v \neq r$ of G, we let $p(v)$ denote the parent of v on T. (Thus, v is a child of $p(v)$.) We let $T[v, u]$ denote the simple path from v to u on T, for any two vertices v and u of G. We use $T[v, u)$, $T(v, u]$ or $T(u, v)$, in order to denote the path $T[v, u]$ minus the vertex on the side of the parenthesis. If v lies on the tree-path $T[r, u]$, then we say that v is an ancestor of u (equivalently, u is a descendant of v). Notice that if v is an ancestor of u, then $v \leq u$. (The converse is not necessarily true.) If v is an ancestor of u such that $v \neq u$, then we say that v is a proper ancestor of u (equivalently, u is a proper descendant of v). We extend the ancestry relation to tree-edges. If $(u, p(u))$ and $(v, p(v))$ are two tree-edges, then we say that $(v, p(v))$ is an ancestor of $(u, p(u))$ (or equivalently, $(u, p(u))$ is a descendant of $(v, p(v))$) if and only if v is an ancestor of u. We let $T(v)$ denote the set of descendants of a vertex v. (Notice that this is a subtree of T.) The number of descendants of v is denoted as $ND(v)$ (i.e., $ND(v) = |T(v)|$). We note that $ND(v)$ can be computed easily during the DFS, because it satisfies the recursive formula $ND(v) = ND(c_1) + \cdots + ND(c_k) + 1$, where $c_1, \ldots, c_k$ are the children of v. We use the *ND* values in order to check the ancestry relation in constant time. Specifically, given two vertices u and v, we have that u is a descendant of v if and only if $v \leq u \leq v + ND(v) - 1$. Equivalently, we have $T(v) = \{v, v + 1, \ldots, v + ND(v) - 1\}$.

A DFS traversal imposes an organization of the edges of the graph that is very rich in properties. Specifically, every non-tree of G has its endpoints related as ancestor and descendant on T [63]. Thus, the non-tree edges of G are called "back-edges".

Whenever we let (x, y) denote a back-edge, we always assume that x is the higher endpoint of (x, y) (i.e., $x > y$). Thus, x is the endpoint of (x, y) that is a descendant of y. For a vertex $v \neq r$, we say that a back-edge (x, y) leaps over v if x is a descendant of v and y is a proper ancestor of v. We let $B(v)$ denote the set of the back-edges that leap over v. Recently, the sets of leaping back-edges were used in order to solve various graph-connectivity problems (see [38, 36]). The usefulness of those sets is intimated by the fact that if we delete the tree-edge $(v, p(v))$ from G, then the subtree $T(v)$ of T is connected with the rest of the graph through the back-edges in $B(v)$. Thus, e.g., we can test if an edge $(v, p(v))$ is a bridge by checking whether the set $B(v)$ is non-empty. In general, we can extract a lot of useful information from those sets, that can help us solve various connectivity problems. We note that we do not explicitly compute the sets of leaping back-edges, as their total size can be excessively large (i.e., it can be $\Omega(n^2)$ even in graphs with $O(n)$ number of edges). Instead, we compute some parameters that summarize the information that is contained in those sets (e.g., by considering the distribution of the endpoints of the back-edges that are contained in them). Here we will define some parameters that we will use throughout. Others that are more specialized, and probably of a more restricted scope, are developed and analyzed on the spot, in the sections that follow.

Let $v \neq r$ be a vertex. We let *bcount*(v) denote the number of back-edges that leap over v (i.e., *bcount*$(v) = |B(v)|$). We let *SumDesc*(v) denote the sum of the higher endpoints of the back-edges in $B(v)$, and we let *SumAnc*(v) denote the sum of the lower endpoints of the back-edges in $B(v)$. Similarly, we let *XorDesc*(v) denote the *XOR* of the higher endpoints of the back-edges in $B(v)$, and we let *XorAnc*(v) denote the *XOR* of the lower endpoints of the back-edges in $B(v)$. We introduce use values *XorDesc*(v) and *XorAnc*(v) because they help us retrieve back-edges from $B(v)$. Specifically, supposing that we know the *XOR* X of the higher endpoints of the back-edges in $B(v) \setminus \{e\}$, and the *XOR* Y of the lower endpoints of the back-edges in $B(v) \setminus \{e\}$, where e is a back-edge in $B(v)$, then we can retrieve the endpoints of e with the values $X \oplus$ *XorDesc*(v) and $Y \oplus$ *XorAnc*(v). The values *SumDesc* and *SumAnc* are used in order to draw inferences for the existence of back-edges. We note that these parameters satisfy a recursive formula that allows us to compute them in linear time in total, for all vertices. Specifically, let *In*(z) denote the set of the back-edges with lower endpoint z, for every vertex z. Also, let *Out*(z) denote the set of the back-edges with higher endpoint z, for every vertex z. Then, *bcount*$(v) =$ *bcount*$(c_1) + \cdots +$

$bcount(c_k) + |Out(v)| - |In(v)|$, where $c_1, \dots, c_k$ are the children of v. Similarly, we have $SumDesc(v) = SumDesc(c_1) + \dots + SumDesc(c_k) + SumDesc(Out(v)) - SumDesc(In(v))$, where we let $SumDesc(S)$ denote the sum of the higher endpoints of the back-edges in a set S of back-edges. The analogous relations hold for $SumAnc(v)$, $XorAnc(v)$ and $XorDesc(v)$. Thus, we can compute all these parameters with a bottom-up procedure (e.g., during the backtracking of the DFS), in total linear time, for all vertices $v \neq r$.

3.1.1 *low* and *high* edges

Now we consider parameters that are defined in relation to the lower endpoints of the back-edges in $B(v)$, for a vertex $v \neq r$. First, let $(v, z_1), \dots, (v, z_s)$ be the list of the back-edges with higher endpoint v, sorted in increasing order w.r.t. their lower endpoint. (Notice that these back-edges belong to $B(v)$.) Then we let $l_i(v)$ denote z_i, for every $i \in \{1, \dots, s\}$. If $i > s$, then we let $l_i(v) = v$. The vertex $l_1(v)$ is of particular importance, and we may denote it simply as $l(v)$. Thus, we can know e.g. if there is a back-edge that stems from v, by checking whether $l(v) < v$.

Now let $(x_1, y_1), \dots, (x_k, y_k)$ be the list of the back-edges in $B(v)$ sorted in increasing order w.r.t. their lower endpoint. We note that such a sorting may not be unique, but we suppose that we have fixed one. Then (w.r.t. this sorting) we call (x_i, y_i) the low_i-edge of v, for every $i \in \{1, \dots, k\}$. The lower endpoint of the low_i-edge of v is called the low_i point of v, and we denote it as $low_i(v)$ (i.e., we have $low_i(v) = y_i$). Notice that the definition of the low_i points of v is independent of the sorting of the back-edges in $B(v)$, provided only that this is in increasing order w.r.t. the lower endpoints. If we want to reference the low_i-edge of v with its endpoints, then we denote it as $(lowD_i(v), low_i(v))$. Of particular importance is the low_1 point of v, which we may simply denote as $low(v)$. The *low* points have been introduced several decades ago, in order to solve various graph problems with a DFS-based approach (see, e.g., [63]). In Section 3.3 we show how to compute the low_i-edges of all vertices, for every $i \in \{1, \dots, k\}$, where k is a fixed integer, in total linear time (see Proposition 3.2).

Now let v be a vertex, and let $c_1, \dots, c_t$ be the children of v sorted in increasing order w.r.t. their *low* point (breaking ties arbitrarily). In other words, we have $low(c_1) \leq \dots \leq low(c_t)$. Then we call c_i the *lowi* child of v. Once we have computed the *low* points of all vertices, we note that it is easy to construct the lists of the *low* children of all vertices in $O(n)$ time in total, using bucket-sort.

Now let $v \neq r$ be a vertex, and let $(x_1, y_1), \ldots, (x_k, y_k)$ be the list of the back-edges in $B(v)$ sorted in decreasing order w.r.t. their lower endpoint. Again, we note that such a sorting may not be unique, but we suppose that we have fixed one. Then (w.r.t. this sorting) we call (x_i, y_i) the $high_i$-edge of v, for every $i \in \{1, \ldots, k\}$. The lower endpoint of the $high_i$-edge of v is called the $high_i$ point of v, and we denote it as $high_i(v)$ (i.e., we have $high_i(v) = y_i$). Notice that the definition of the $high_i$ points of v is independent of the sorting of the back-edges in $B(v)$, provided only that this is in decreasing order w.r.t. the lower endpoints. If we want to reference the $high_i$-edge of v with its endpoints, then we denote it as $(highD_i(v), high_i(v))$. Of particular importance is the $high_1$ point of v, which we may simply denote as $high(v)$. Also, we denote the $high_1$-edge of v as $e_{high}(v)$. The *high* points have been introduced relatively recently (as a concept dual to the *low* points) in order to solve various problems of low connectivity with a DFS-based approach (see, e.g., [38, 36]). One of the reasons that the *high* points are useful is that they let us know whether there exists a back-edge that leaps over a vertex u, but not over a specific proper ancestor v of u. (Specifically, this is equivalent to $high(u) \geq v$.) Thus, if that is the case, then we know that if we remove both $(u, p(u))$ and $(v, p(v))$ from the graph, then u is connected with $p(u)$ through a path that uses $e_{high}(u)$. In Section 3.4 we show how to compute the $high_i$-edges of all vertices, for every $i \in \{1, \ldots, k\}$, where k is a fixed integer, in total linear time (see Proposition 3.3).

3.1.2 Maximum points, leftmost and rightmost edges

Now we consider concepts that are defined in relation to the higher endpoints of the leaping back-edges. Let $v \neq r$ be a vertex. We let $M(v)$ denote the maximum vertex that is an ancestor of the higher endpoints of the back-edges in $B(v)$. Equivalently, $M(v)$ is the nearest common ancestor of the higher endpoints of the back-edges in $B(v)$. (If $B(v) = \emptyset$, then we let $M(v) := \bot$.) Notice that $M(v)$ (if it exists) is a descendant of v. For every vertex x, we let $M^{-1}(x)$ denote the list of all vertices v with $M(v) = x$, sorted in decreasing order. Thus, we have that all vertices in $M^{-1}(x)$ have x as a common descendant, and therefore they are related as ancestor and descendant. For every vertex $v \in M^{-1}(x)$, we let $nextM(v)$ and $prevM(v)$ denote the successor and the predecessor, respectively, of v in $M^{-1}(x)$. Equivalently, we have that $nextM(v)$ (resp., $prevM(v)$) is the greatest proper ancestor (resp., the lowest proper

descendant) u of v such that $M(u) = M(v)$. We also let *lastM*(v) denote the lowest vertex in $M^{-1}(M(v))$.

We extend the concept of the M points in general sets of back-edges. Thus, if S is a set of back-edges, then we let $M(S)$ denote the nearest common ancestor of the higher endpoints of the back-edges in S. (Thus, we have $M(v) = M(B(v))$.) We introduce a notation for the M points of some special sets of back-edges. Let c be a descendant of a vertex $v \neq r$, and let S be the set of the back-edges that leap over v and stem from the subtree of c (i.e., $S = \{(x, y) \in B(v) \mid x \text{ is a descendant of } c\}$). Then we denote $M(S)$ as $M(v, c)$. Also, let $\widetilde{S}$ be the set of the back-edges in $B(v)$ that stem from proper descendants of $M(v)$ (i.e., $\widetilde{S} = \{(x, y) \in B(v) \mid x \text{ is a proper descendant of } M(v)\}$). Then we denote $M(\widetilde{S})$ as $\widetilde{M}(v)$. In Section 3.6 we deal with the computation of the M points. This relies on the computation of the leftmost and the rightmost points, which we define next.

Let $v \neq r$ be a vertex, and let $(x_1, y_1), \ldots, (x_k, y_k)$ be the list of the back-edges in $B(v)$ sorted in increasing order w.r.t. their higher endpoint. We note that such a sorting may not be unique, but we suppose that we have fixed one. Then we call (x_i, y_i) the i-th leftmost edge of v. We denote x_i as $L_i(v)$, and we call it the i-th leftmost point of v. Of particular importance is the first leftmost edge of v, which we denote as $e_L(v)$. On the other hand, let $(x_1, y_1), \ldots, (x_k, y_k)$ be the list of the back-edges in $B(v)$ sorted in decreasing order w.r.t. their higher endpoint. Again, such a sorting may not be unique, but we suppose that we have fixed one. Then we call (x_i, y_i) the i-th rightmost edge of v. We denote x_i as $R_i(v)$, and we call it the i-th rightmost point of v. Of particular importance is the first rightmost edge of v, which we denote as $e_R(v)$. (We note that the edges $e_L(v)$ and $e_R(v)$ were used in [50], with different notation.) In Section 3.5 we extend the concepts of the leftmost and the rightmost edges, and provide an efficient method to compute them (see Proposition 3.4).

3.2 Properties of the DFS parameters

Lemma 3.1. *Let u and v be two vertices $\neq r$ such that v is an ancestor of u and $M(v)$ is a descendant of u. Then $M(v)$ is a descendant of $M(u)$.*

Proof. Let (x, y) be a back-edge in $B(v)$. Then x is a descendant of $M(v)$, and therefore a descendant of u. Futhermore, y is a proper ancestor of v, and therefore a proper

ancestor of u. This shows that $(x, y) \in B(u)$, and thus x is a descendant of $M(u)$. Due to the generality of $(x, y) \in B(v)$, this implies that $M(v)$ is a descendant of $M(u)$. $\square$

Lemma 3.2. *Let u and v be two vertices $\neq r$ such that v is an ancestor of u and $M(v)$ is a descendant of $M(u)$. Then $B(v) \subseteq B(u)$.*

Proof. Let (x, y) be a back-edge in $B(v)$. Then x is a descendant of $M(v)$, and therefore a descendant of $M(u)$. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Due to the generality of $(x, y) \in B(v)$, we conclude that $B(v) \subseteq B(u)$. $\square$

Lemma 3.3. *Let u and v be two vertices $\neq r$ such that u is a descendant of v and $high(u) = high(v)$. Then $B(u) \subseteq B(v)$.*

Proof. Let (x, y) be a back-edge in $B(u)$. Then x is a descendant of u, and therefore a descendant of v. Furthermore, y is an ancestor of $high(u)$, and therefore an ancestor of $high(v)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u)$, we conclude that $B(u) \subseteq B(v)$. $\square$

Lemma 3.4. *Let v and v' be two vertices such that $M(v) = M(v')$. Then $low(v) = low(v')$.*

Proof. Since $M(v) = M(v')$, we have that v and v' are related as ancestor and descendant. Thus, we may assume w.l.o.g. that v' is an ancestor of v. Then, Lemma 3.2 implies that $B(v') \subseteq B(v)$. This implies that $low(v) \leq low(v')$. Now let (x, y) be a back-edge in $B(v)$ such that $y = low(v)$. Then x is a descendant of v, and therefore a descendant of v'. Furthermore, both y and v' have v as a common descendant, and therefore they are related as ancestor and descendant. Then, $y = low(v) \leq low(v') < v'$ implies that y is a proper ancestor of v'. This shows that $(x, y) \in B(v')$, and therefore $y \geq low(v')$. Thus, we conclude that $low(v) = low(v')$. $\square$

Lemma 3.5. *Let u and v be two vertices $\neq r$ such that v is an ancestor of u and $high(u) = high(v)$. Then $low(v) \leq low(u)$.*

Proof. By Lemma 3.3 we have that $B(u) \subseteq B(v)$, and thus we get $low(v) \leq low(u)$ as an immediate consequence. $\square$

Lemma 3.6. *Let u and v be two vertices $\neq r$ such that $M(u) = M(v)$, v is a proper ancestor of u, and $B(u) \neq B(v)$. Then $high(u)$ is a descendant of v.*

Proof. By Lemma 3.2 we have $B(v) \subseteq B(u)$. Since u is a common descendant of v and *high*(u), we have that v and *high*(u) are related as ancestor and descendant. Now let us suppose, for the sake of contradiction, that *high*(u) is not a descendant of v. This implies that *high*(u) is a proper ancestor of v. Now let (x, y) be a back-edge in $B(u)$. Then we have that x is a descendant of u, and therefore a descendant of v. Furthermore, y is an ancestor of *high*(u), and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(v)$. Thus, since $B(v) \subseteq B(u)$, we have that $B(u) = B(v)$, in contradiction to the assumption $B(u) \neq B(v)$. Thus, we conclude that *high*(u) is a descendant of v. □

Lemma 3.7. *Let u and v be two vertices $\neq r$. Then the following are equivalent:*

(1) $B(u) = B(v)$

(2) $M(u) = M(v)$ *and high*$(u) =$ *high*(v)

(3) $M(u) = M(v)$ *and bcount*$(u) =$ *bcount*(v)

Proof. (1) obviously implies (2) and (3). Conversely, we will show that either of (2) and (3) also implies (1). Let us first assume (2). Let (x, y) be a back-edge in $B(u)$. Then x is a descendant of $M(u)$, and therefore a descendant of $M(v)$. Furthermore, y is an ancestor of *high*(u), and therefore an ancestor of *high*(v), and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(v)$. Similarly, we can show the reverse inclusion, and therefore we have $B(u) = B(v)$.

Now let us assume (3). If $B(u) = \emptyset$, then $M(u) = \perp$, and therefore $M(v) = \perp$, and therefore $B(v) = \emptyset$. So let us assume that $B(u) \neq \emptyset$. Then $M(u)$ is defined, and therefore $M(u) = M(v)$ implies that $M(u)$ is a common descendant of u and v, and therefore u and v are related as ancestor and descendant. Thus, we may assume w.l.o.g. that u is a descendant of v. Then, since $M(u) = M(v)$, Lemma 3.2 implies that $B(v) \subseteq B(u)$. Therefore, *bcount*$(u) =$ *bcount*(v) implies that $B(u) = B(v)$. □

The following provides a criterion that characterizes 3-edge-connected graphs, and we will use it throughout without explicit mention.

Proposition 3.1. *A connected graph G is 3-edge-connected if and only if: for every vertex $v \neq r$ we have bcount$(v) > 1$, and for every two distinct vertices u and v such that $r \notin \{u, v\}$ we have $B(u) \neq B(v)$.*

Proof. ($\Rightarrow$) Let $v \neq r$ be a vertex, let $A = T(v)$ and let $B = T(r) \setminus T(v)$. Notice that A and B are connected subgraphs of $G \setminus \{(v, p(v))\}$. Then it is easy to see that $\{(v, p(v))\} \cup B(v)$ is a cut of G. Thus, since G is 3-edge-connected, we have $|B(v)| > 1$.

Now let us suppose, for the sake of contradiction, that there exist two distinct vertices u and v such that $r \notin \{u, v\}$ and $B(u) = B(v)$. We have already established that $B(u) \neq \emptyset$ and $B(v) \neq \emptyset$. Thus, $B(u) = B(v)$ implies that there is a back-edge $(x, y) \in B(u) \cap B(v)$. Then x is a common descendant of u and v, and therefore u and v are related as ancestor and descendant. Thus, we may assume w.l.o.g. that v is a proper ancestor of u. Now consider the parts $A = T(u)$, $B = T(v) \setminus T(u)$ and $C = T(r) \setminus T(v)$. Then A, B and C are non-empty subtrees of T. Since $B(u) = B(v)$, the following two implications are immediate: first, there is no back-edge from A to B, and second, there is no back-edge from B to C. Thus, B becomes disconnected from the rest of the graph in $G \setminus \{(u, p(u)), (v, p(v))\}$, in contradiction to the fact that G is 3-edge-connected.

($\Leftarrow$) Let us suppose, for the sake of contradiction, that G is not 3-edge-connected. Thus, there is either a 1-cut or a 2-cut of G. First, let us assume that there is a bridge e of G. Then e is a tree-edge, and so it has the form $(v, p(v))$ for a vertex $v \neq r$. Notice that $T(v)$ and $T(r) \setminus T(v)$ are two connected subgraphs of $G \setminus \{(v, p(v))\}$. Thus, these must be the connected components of $G \setminus \{e\}$. But this implies that there is no back-edge from A to B, in contradiction to $\mathit{bcount}(v) > 1$. This shows that G has no bridges.

Thus, there is at least one 2-cut C of G. Then, either C consists of one tree-edge and one back-edge, or it consists of two tree-edges. The first case can be excluded with an argument that is similar to the one that we used in order to show that G has no bridges. (In particular, here we rely again on $\mathit{bcount}(v) > 1$, for every vertex $v \neq r$.) So let us assume that C consists of two tree-edges $(u, p(u))$ and $(v, p(v))$, for some vertices u, v with $r \notin \{u, v\}$. Let us suppose, for the sake of contradiction, that u and v are not related as ancestor and descendant. Since $\mathit{bcount}(u) > 1$, there is a back-edge $(x, y) \in B(u)$. Then, since v is not related as ancestor and descendant with u, we have that the tree-paths $T[x, u]$ and $T[p(u), y]$ remain intact in $G \setminus \{(v, p(v))\}$. But this implies that u remains connected with $p(u)$ in $G \setminus \{(u, p(u)), (v, p(v))\}$ through the path $T[u, x], (x, y), T[y, p(u)]$, in contradiction to the fact that $\{(u, p(u)), (v, p(v))\}$ is a 2-cut of G. This shows that u and v are related as ancestor and descendant, and so we may assume w.l.o.g. that v is a proper ancestor of u. Then, notice that

$X = T(u)$, $Y = T(v) \setminus T(u)$ and $Z = T(r) \setminus T(v)$ are three connected subgraphs of $G \setminus \{(u, p(u)), (v, p(v))\}$. Thus, there cannot exist a back-edge from X to Y, because otherwise u remains connected with $p(u)$ in $G \setminus C$. Similarly, there cannot exist a back-edge from Y to Z, because otherwise v remains connected with $p(v)$ in $G \setminus C$. But then it is easy to see that $B(u) = B(v)$, a contradiction. This shows that G is 3-edge-connected. □

Lemma 3.8. *Let v and v' be two vertices with $M(v) = M(v')$ such that v' is a proper ancestor of v. If G is 3-edge-connected, then v' is an ancestor of high(v).*

Proof. We have that *high*(v) is a proper ancestor of v. Thus, since v' and *high*(v) have v as a common descendant, they are related as ancestor and descendant. Let us suppose, for the sake of contradiction, that v' is not an ancestor of *high*(v). Then, we have that v' is a proper descendant of *high*(v). Since $M(v) = M(v')$ and v' is a proper ancestor of v, by Lemma 3.2 we have that $B(v') \subseteq B(v)$. Now let (x, y) be a back-edge in $B(v)$. Then x is a descendant of v, and therefore a descendant of v'. Furthermore, y is an ancestor of *high*(v), and therefore a proper ancestor of v'. This shows that $(x, y) \in B(v')$. Due to the generality of $(x, y) \in B(v)$, this implies that $B(v) \subseteq B(v')$. But then, since $B(v') \subseteq B(v)$, we have that $B(v') = B(v)$, in contradiction to the fact that the graph is 3-edge-connected. Thus, we conclude that v' is an ancestor of *high*(v). □

Lemma 3.9. *Let $v \neq r$ be a vertex and let e be a back-edge in $B(v)$ such that $M(B(v) \setminus \{e\}) \neq M(v)$. Then, either $e = e_L(v)$ or $e = e_R(v)$.*

Proof. Let X be the set of the higher endpoints of the back-edges in $B(v)$. Then we have $M(v) = nca(X)$, $L_1(v) = min(X)$ and $R_1(v) = max(X)$. We claim that $M(v) = nca\{L_1(v), R_1(v)\}$. First, we obviously have that $M(v)$ is an ancestor of $nca\{L_1(v), R_1(v)\}$. Conversely, $nca\{L_1(v), R_1(v)\}$ is an ancestor of both $L_1(v)$ and $R_1(v)$. Since $L_1(v) \leq R_1(v)$, this implies that $nca\{L_1(v), R_1(v)\}$ is an ancestor of every vertex z such that $L_1(v) \leq z \leq R_1(v)$. Thus, since $L_1(v) = min(X)$ and $R_1(v) = max(X)$, we have that $nca\{L_1(v), R_1(v)\}$ is an ancestor of every vertex in X, and therefore $nca\{L_1(v), R_1(v)\}$ is an ancestor of $M(v)$. Thus, we have $M(v) = nca\{L_1(v), R_1(v)\}$.

Now let X' be the set of the higher endpoints of the back-edges in $B(v) \setminus \{e\}$. Then we have $X' \subseteq X$ and $M(B(v) \setminus \{e\}) = nca(X')$. Since $M(B(v) \setminus \{e\}) \neq M(v)$ and $M(v) = nca\{L_1(v), R_1(v)\}$, this implies that we cannot have both $L_1(v)$ and $R_1(v)$

in X'. Thus, either $L_1(v) \notin X'$ or $R_1(v) \notin X'$. Let us assume that $L_1(v) \notin X'$. This implies that e is the only back-edge in $B(v)$ whose higher endpoint is $L_1(v)$. Thus, by definition we have $e = e_L(v)$. Similarly, if we have $R_1(v) \notin X'$, then we can infer that $e = e_R(v)$. $\square$

The following two lemmata show that the leftmost and the rightmost points are useful for testing inclusion relations of the B sets.

Lemma 3.10. *Let u, v ($\neq r$) be two vertices such that u is a descendant of v. Then $B(v) \subseteq B(u)$ if and only if $L_1(v) \in T(u)$ and $R_1(v) \in T(u)$.*

Proof. ($\Rightarrow$) Let (x, y) and (x', y') be the leftmost and the rightmost back-edges of v, respectively. Then $B(v) \subseteq B(u)$ implies that $(x, y) \in B(u)$ and $(x', y') \in B(u)$. This implies that $x \in T(u)$ and $x' \in T(u)$. Thus we have $L_1(v) \in T(u)$ and $R_1(v) \in T(u)$ (by the definition of $L_1(v)$ and $R_1(v)$).
($\Leftarrow$) Let (x, y) be a back-edge in $B(v)$. Then (by the definition of $L_1(v)$ and $R_1(v)$) we have $L_1(v) \leq x \leq R_1(v)$. Now $L_1(v) \in T(u) \Rightarrow u \leq L_1(v)$, and so $L_1(v) \leq x$ implies that $u \leq x$. Similarly, $R_1(v) \in T(u) \Rightarrow R_1(v) < u + ND(u)$, and so $x \leq R_1(v)$ implies that $x < u + ND(u)$. Thus we have $x \in T(u)$. Furthermore, $(x, y) \in B(v) \Rightarrow y < v$, and since u is a descendant of v this implies that $y < u$. We conclude that $(x, y) \in B(u)$. $\square$

Lemma 3.11. *Let u, v ($\neq r$) be two vertices such that u is a descendant of v and $bcount(v) \geq 2$. Then $B(v) = B(u) \sqcup \{e\}$, for a back-edge e, if and only if $bcount(v) = bcount(u) + 1$ and either (1) $L_1(v) \notin T(u)$, $L_2(v) \in T(u)$ and $R_1(v) \in T(u)$, or (2) $R_1(v) \notin T(u)$, $R_2(v) \in T(u)$ and $L_1(v) \in T(u)$.*

Proof. ($\Rightarrow$) $bcount(v) = bcount(u) + 1$ is an obvious implication of $B(v) = B(u) \sqcup \{e\}$. Now, since there exists a back-edge $e \in B(v) \setminus B(u)$, by Lemma 3.10 we have that either $L_1(v) \notin T(u)$ or $R_1(v) \notin T(u)$. Suppose that $L_1(v) \notin T(u)$ (the proof for the other case is similar). Then the first leftmost back-edge of v is not in $B(u)$, and so e is the first leftmost back-edge of v. Every other back-edge in $B(v)$ is contained in $B(u)$, and therefore we have $L_2(v) \in T(u)$ and $R_1(v) \in T(u)$ (and these vertices are defined, since $bcount(v) \geq 2$).
($\Leftarrow$) Suppose that (1) is true (the proof for the other case is similar). Let (x, y) be a back-edge in $B(v)$. Then (by the definition of $L_1(v)$ and $R_1(v)$) we have that $L_1(v) \leq x \leq R_1(v)$. If $x \neq L_1(v)$, then $L_2(v) \leq x \leq R_1(v)$. Since $L_2(v) \in T(u)$ and

$R_1(v) \in T(u)$, this implies that $x \in T(u)$. Furthermore, $(x, y) \in B(v)$ implies that y is a proper ancestor of v, and therefore y is a proper ancestor of u. This shows that $(x, y) \in B(u)$. Thus we infer that every back-edge $(x, y) \in B(v)$ with $L_2(v) \leq x \leq R_1(v)$ is in $B(u)$ $(*)$. Now, since $bcount(v) = bcount(u) + 1$, we have that there exists at least one back-edge $e = (x, y) \in B(v) \setminus B(u)$, and so it has $x = L_1(v)$. Furthermore, this is the only back-edge with this property (otherwise, $bcount(v) = bcount(u) + 1$ is contradicted by $(*)$). Thus, by $(*)$ we have that $B(v) = B(u) \sqcup \{e\}$. □

Lemma 3.12. *Let $w \neq r$ be a vertex, and let u and v be two descendants of w such that $M(u) = M(w, x) \neq \bot$ and $M(v) = M(w, y) \neq \bot$, where x and y are descendants of different children of $M(w)$. Then u and v are not related as ancestor and descendant.*

Proof. Let c_1 be the child of $M(w)$ that is an ancestor of x, and let c_2 be the child of $M(w)$ that is an ancestor of y. By assumption we have that $c_1 \neq c_2$. Now let us suppose, for the sake of contradiction, that u is not a descendant of c_1. Since $M(u) = M(w, x)$, we have that $M(u)$ is a common descendant of u and x, and therefore u and x are related as ancestor and descendant. Since u is not a descendant of c_1, we cannot have that u is a descendant of x. Thus, u is a proper ancestor of x. Then, we have that x is a common descendant of u and c_1, and therefore u and c_1 are related as ancestor and descendant. Thus, we have that u is a proper ancestor of c_1. This implies that u is an ancestor of $M(w)$, and therefore an ancestor of c_2. Now, since $M(w, y)$ is defined, we have that there is a back-edge $(z, t) \in B(w)$ such that z is a descendant of y. Then, z is a descendant of c_2, and therefore a descendant of u. Furthermore, t is a proper ancestor of w, and therefore a proper ancestor of u. This shows that $(z, t) \in B(u)$. This implies that z is a descendant of $M(u)$. Since $M(u) = M(w, x) \neq \bot$, we have that there is a back-edge $(z', t') \in B(u)$ such that z' is a descendant of x. Since $(z, t) \in B(u)$ and $(z', t') \in B(u)$, we have that $M(u)$ is an ancestor of $nca\{z, z'\}$. But z is a descendant of c_1, whereas z' is a descendant of c_2. This implies that $nca\{z, z'\} = M(w)$, and therefore $M(u)$ is an ancestor of $M(w)$, contradicting the fact $M(u) = M(w, x)$ (which implies that $M(u)$ is a descendant of x, and therefore of c_1). Thus, we have that u is a descendant of c_1. Similarly, we can show that v is a descendant of c_2. Thus, since u and v are descendants of different children of $M(w)$, we have that they cannot be related as ancestor and descendant. □

Lemma 3.13. *Let $v \neq r$ be a vertex with $B(v) \neq \emptyset$ such that there is no back-edge of the form $(M(v), z)$ in $B(v)$. Then $low(c_2) < v$, where c_2 is the low2 child of $M(v)$.*

Proof. Let (x, y) be a back-edge in $B(v)$. Then we have that x is a descendant of $M(v)$. Since there is no back-edge of the form $(M(v), z)$ in $B(v)$, we have that $x \neq M(v)$. Thus, x is a proper descendant of $M(v)$. This shows that $M(v)$ has at least one child. Furthermore, it cannot be the case that $M(v)$ has only one child, because otherwise all the back-edges of $B(v)$ would stem from the subtree of this child, and so $M(v)$ would be a descendant of its child, which is absurd. Thus, $M(v)$ has at least two children, and so it makes sense to consider the *low2* child c_2 of $M(v)$.

Let us suppose, for the sake of contradiction, that *low*$(c_2) \geq v$. This implies that, of all the children of $M(v)$, only the *low1* child c_1 of $M(v)$ may have *low*$(c_1) < v$. Let (x, y) be a back-edge in $B(v)$. Then we have that x is a descendant of $M(v)$. Since there is no back-edge of the form $(M(v), z)$ in $B(v)$, we have that $x \neq M(v)$, and therefore x is a proper descendant of $M(v)$. Let c be the child of $M(v)$ that is an ancestor of x. Since $(x, y) \in B(v)$, we have that y is a proper ancestor of v, and therefore a proper ancestor of $M(v)$, and therefore a proper ancestor of c. This shows that $(x, y) \in B(c)$. Since y is a proper ancestor of v, we have $y < v$. Thus, since $(x, y) \in B(c)$, we have that *low*$(c) \leq y < v$. This implies that $c = c_1$. Due to the generality of $(x, y) \in B(v)$, this implies that $M(v)$ is a descendant of c_1, which is absurd. Thus, our supposition cannot be true, and therefore we have that *low*$(c_2) < v$. □

Lemma 3.14. *Let C be an edge-minimal cut of G. Let $(v_1, p(v_1)), \ldots, (v_k, p(v_k))$ be the list of the tree-edges in C. Let us assume w.l.o.g. that v_1 is the lowest among $v_1, \ldots, v_k$. Then v_1 is a common ancestor of $\{v_1, \ldots, v_k\}$. Furthermore, suppose that C contains a back-edge e. Then $e \in B(v_1) \cup \cdots \cup B(v_k)$.*

Proof. Let us suppose, for the sake of contradiction, that v_1 is not a common ancestor of $\{v_1, \ldots, v_k\}$. This means that at least one among $\{v_1, \ldots, v_k\}$ is not a descendant of v_1. Now let I be the collection of all indices in $\{1, \ldots, k\}$ such that v_i is a descendant of v_1, for every $i \in I$. Thus, we have $I \subset \{1, \ldots, k\}$. Now let $C_I = \{(v_i, p(v_i)) \mid i \in I\}$, and let C' be the subset of C that consists of all the back-edges in C. Then, since $I \subset \{1, \ldots, k\}$, we have $C_I \cup C' \subset C$. Thus, since C is an edge-minimal cut of G, we have that $G' = G \setminus (C_I \cup C')$ is connected. Thus, there is a path P from v_1 to $p(v_1)$ in G'. Then, by Lemma 3.19 we have that the first occurrence of an edge in P that leads outside of $T(v_1)$ is either $(v_1, p(v_1))$ or a back-edge that leaps over v_1. The first case is rejected, since $(v_1, p(v_1)) \in C_I$. Thus, the first occurrence of an edge in P that leads outside of $T(v_1)$ is a back-edge (x, y) that leaps over v_1. Now consider

the part P' of P from v_1 up to, and including, x. Then we have that P' avoids the tree-edges from C that have the form $(v, p(v))$ where v is a descendant of v_1 (since P has this property). Also, P' avoids all the back-edges from C (since P has this property). Furthermore, P' avoids the tree-edges of the form $(v, p(v))$ where v is not a descendant of v_1, because it is the initial part of P that lies entirely within $T(v_1)$. This shows that P' is a path in $G \setminus C$. Since v_1 is the minimum among $\{v_1, \dots, v_k\}$, we have that no vertex in $\{v_1, \dots, v_k\}$ is a proper ancestor of v_1. Thus, the tree-path $T[p(v_1), r]$ remains intact in $G \setminus C$. But then, $P' + (x, y) + T[y, p(v_1)]$ is a path from v_1 to $p(v_1)$ in $G \setminus C$, in contradiction to the fact that C is an edge-minimal cut of G. This shows that v_1 is a common ancestor of $\{v_1, \dots, v_k\}$.

Now let e be a back-edge in C. Let us suppose, for the sake of contradiction, that $e \notin B(v_1) \cup \dots \cup B(v_k)$. Let $e = (x, y)$. Then we have that none of $v_1, \dots, v_k$, can be on the tree-path $T[x, y]$. Thus, $T[x, y]$ remains intact in $G \setminus C$, and therefore the endpoints of e remain connected in $G \setminus C$, in contradiction to the fact that C is an edge-minimal cut of G. We conclude that $e \in B(v_1) \cup \dots \cup B(v_k)$. □

3.3 Computing the *low*-edges

Let $v \neq r$ be a vertex. The definition of the low_i-edges of v, for $i = 1, 2, \dots$, assumes any ordering of the back-edges in $B(v)$ that it is increasing w.r.t. the lower endpoints. For computational purposes (basically, for convenience in our arguments), we will fix such an ordering for sets of back-edges, which we call the *low* ordering. Let $(x_1, y_1), \dots, (x_t, y_t)$ be a list of back-edges. Then we say that this list is sorted in the *low* ordering if it is increasing w.r.t. the lower endpoints, and also satisfies $x_i \leq x_{i+1}$, for every $i \in \{1, \dots, t-1\}$ such that $y_i = y_{i+1}$.[1]

Now let $v \neq r$ be a vertex, and let $(x_1, y_1), \dots, (x_t, y_t)$ be the list of the back-edges in $B(v)$ sorted in the *low* ordering. Then we let (x_i, y_i) be the low_i-edge of v, for every $i \in \{1, \dots, t\}$. We assume that the *low* ordering is applied for every set of leaping back-edges, and the low_i-edges correspond to this ordering. Then we have the following.

[1]Here we have to be a little more precise. Since we consider multigraphs, we may have several back-edges of the form (x, y) in a set of back-edges. Then we assume a unique integer identifier that is assigned to every edge of the graph, and the ties here are broken according to those identifiers. However, for the sake of simplicity, we will not make explicit use of this information in what follows.

Lemma 3.15. *Let $v \neq r$ be a vertex, and let $(v, z_1), \ldots, (v, z_s)$ be the list of the back-edges with higher endpoint v, sorted in the low ordering. Let e be the low_k-edge of v, for some $k \geq 1$. Then, either $e \in \{(v, z_1), \ldots, (v, z_k)\}$, or there is a child c of v such that e is the $low_{k'}$-edge of c, for some $k' \leq k$.*

Proof. Let $(x_1, y_1), \ldots, (x_t, y_t)$ be the list of the back-edges in $B(v)$ sorted in the *low* ordering. Then we have $e = (x_k, y_k)$.

First, let us suppose, for the sake of contradiction, that $s > k$ and $e \in \{(v, z_{k+1}), \ldots, (v, z_s)\}$. Then, since the back-edges in $(v, z_1), \ldots, (v, z_s)$ are sorted in increasing order w.r.t. their lower endpoint, we have that, for every $i \in \{1, \ldots, s\}$, there is a $j \in \{i, \ldots, t\}$ such that $(v, z_i) = (x_j, y_j)$. Since $e \in \{(v, z_{k+1})), \ldots, (v, z_s)\}$, there is an $i \in \{k+1, \ldots, s\}$ such that $e = (v, z_i)$. But then we have $e = (x_j, y_j)$ for some $j \in \{k+1, \ldots, t\}$, contradicting the fact that $e = (x_k, y_k)$. This shows that either $e \in \{(v, z_1), \ldots, (v, z_k)\}$, or e does not belong to the set $\{(v, z_1), \ldots, (v, z_s)\}$ at all.

Now let us assume that $e \notin \{(v, z_1), \ldots, (v, z_s)\}$. Then, since $e = (x_k, y_k)$, we have $x_k \neq v$, and therefore x_k is a proper descendant of v. So let c be the child of v that is an ancestor of x_k. Then we have $e \in B(c)$. Let $(x'_1, y'_1), \ldots, (x'_{t'}, y'_{t'})$ be the list of the back-edges in $B(c)$ sorted in the *low* ordering. Then we have that the low_i-edge of c, for any $i \in \{1, \ldots, t'\}$, is (x'_i, y'_i). Now let us suppose, for the sake of contradiction, that $t' > k$ and $e \in \{(x'_{k+1}, y'_{k+1}), \ldots, (x'_{t'}, y'_{t'})\}$. So let i be the index in $\{k+1, \ldots, t'\}$ such that $e = (x'_i, y'_i)$. Since $e = (x_k, y_k) \in B(v)$ we have that y_k is a proper ancestor of v, and therefore $y_k < v$. Thus, since $y_k = y'_i$, we have $y'_i < v$. Since the back-edges in $(x'_1, y'_1), \ldots, (x'_{t'}, y'_{t'})$ are sorted in increasing order w.r.t. their lower endpoint, we have $y'_j \leq y'_i$, and therefore $y'_j < v$, for every $j \in \{1, \ldots, i\}$. Now let j be an index in $\{1, \ldots, i\}$. Then we have that x'_j is a descendant of c, and therefore a descendant of v. Since (x'_j, y'_j) is a back-edge, we have that x'_j is a descendant of y'_j. Thus, x'_j is a common descendant of v and y'_j, and therefore v and y'_j are related as ancestor and descendant. Since $j \in \{1, \ldots, i\}$, we have $y'_j < v$, and therefore y'_j is a proper ancestor of v. This shows that all back-edges in $(x'_1, y'_1), \ldots, (x'_i, y'_i)$ leap over v. Thus, since $(x'_1, y'_1), \ldots, (x'_i, y'_i)$ and $(x_1, y_1), \ldots, (x_t, y_t)$ are sequences of back-edges sorted in the *low* ordering and $\{(x'_1, y'_1), \ldots, (x'_i, y'_i)\} \subseteq \{(x_1, y_1), \ldots, (x_t, y_t)\}$, we have $(x'_j, y'_j) = (x_{j'}, y_{j'})$, where $j' \geq j$, for every $j \in \{1, \ldots, i\}$. But since $i \geq k+1$, this implies that $e = (x'_i, y'_i) = (x_{i'}, y_{i'})$ for some $i' \geq k+1$, contradicting the fact that $e = (x_k, y_k)$. This shows that $e \in \{(x'_1, y'_1), \ldots, (x'_k, y'_k)\}$. □

Lemma 3.15 provides enough information in order to find the low_i-edge of v, for any $i \geq 1$. Specifically, it is sufficient to search for it in the list of the first i back-edges of the form (v, z) with the lowest lower endpoint, plus the lists of the low_1-,…,low_i-edges of all the children of v. A procedure that computes the low_1-,…,low_k-edges of all vertices, for any fixed k, is shown in Algorithm 1. The idea is to process the vertices in a bottom-up fashion (e.g., in decreasing order w.r.t. the DFS numbering). Thus, for every vertex v that we process, we have computed the low_1-,…,low_k-edges of its children, and therefore we have to check among those, plus the k back-edges of the form (v, z) with the lowest lower endpoint, in order to get the low_1-,…,low_k-edges of v. For our purposes in this work, k will be a fixed constant (at most 4), and thus the dependency of the running time on k does not matter for us. However, we use balanced binary-search trees (BST) in order to get an algorithm with $O(m + nk \log k)$ time. Our result is summarized in Proposition 3.2.

Proposition 3.2. *Let k be any fixed integer. Algorithm 1 computes the low_i-edges of all vertices, for all $i \in \{1, \ldots, k\}$. Furthermore, it runs in $O(m + nk \log k)$ time.*

Proof. We will prove correctness inductively, by establishing that, whenever the **for** loop in Line 4 processes a vertex v, we have that the low_i-edges of the children of v have been correctly computed, for every $i \in \{1, \ldots, k\}$. Initially, this is trivially true (because we process the vertices in decreasing DFS order, and so the first vertex that we process is a leaf). So let us suppose that the **for** loop in Line 4 starts processing a vertex v for which we have computed the low_i-edges of its children, for every $i \in \{1, \ldots, k\}$. It is sufficient to show that, by the time the processing of v is done, we have correctly computed the low_i-edge of v, for every $i \in \{1, \ldots, k\}$.

Let $(v, z_1), \ldots, (v, z_s)$ be the list of the back-edges with higher endpoint v, sorted in the *low* ordering (and thus in increasing order w.r.t. their lower endpoint). Let also $c_1, \ldots, c_t$ be the children of v sorted in increasing order. In Line 6 we fill $BST[v]$ with the first k (non-null) back-edges from $\{(v, z_1), \ldots, (v, z_s)\}$. Notice that these are all back-edges that leap over v. Then, we may insert more back-edges into $BST[v]$ in Line 13 or in Line 19. In either case, the back-edge (x, y) that we insert into $BST[v]$ is a back-edge in $B(c)$, for a child c of v, that satisfies $y < v$. Since $(x, y) \in B(c)$, we have that x is a descendant of c, and therefore a descendant of v. Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Then, $y < v$ implies that y

Algorithm 1: Compute the low_i-edges of all vertices, for all $i \in \{1, \ldots, k\}$

```
1  compute the low ordering of the adjacency list of every vertex
2  initialize an empty balanced binary-search tree BST[v] that stores back-edges,
     for every vertex v
3  // BST[v] sorts the edges it stores w.r.t. the low ordering
4  for v ← n to v = 2 do
5      let L = (v, z_1), ..., (v, z_s) be the list of the back-edges with higher endpoint
         v, sorted in the low ordering
6      fill BST[v] with the first k (non-null) back-edges from L
7      let c_1, ..., c_t be the children of v sorted in increasing order
8      for c ← c_1 to c = c_t do
9          foreach i ∈ {1, ..., k} do
10             let (x, y) be the low_i-edge of c
11             if y ≥ v then continue
12             if BST[v] has less than k entries then
13                 insert (x, y) into BST[v]
14             end
15             else
16                 let (x', y') be the k-th entry of BST[v]
17                 if y < y' then
18                     delete the k-th entry of BST[v]
19                     insert (x, y) into BST[v]
20                 end
21             end
22         end
23     end
24     foreach i ∈ {1, ..., k} do
25         let the low_i-edge of v be the i-th entry in BST[v]
26     end
27 end
```

is a proper ancestor of v. Thus, we have that (x, y) is a back-edge in $B(v)$. This shows that, when we reach Line 24, we have that all elements of $BST[v]$ are back-edges that

leap over v. Furthermore, notice that, when a back-edge is deleted from *BST*$[v]$ in Line 18, this is because its lower endpoint is greater than that of the back-edge that is to be inserted. Thus, when we reach Line 24, we have that the back-edges in *BST*$[v]$ are those with the k-th lowest lower endpoints among the back-edges that we have met during the processing of v.

Let $(x_1, y_1), \dots, (x_N, y_N)$ be the list of the back-edges in $B(v)$ sorted in the *low* ordering. Let i be an index in $\{1, \dots, k\}$, and let (x, y) be the low_i-edge of v. Thus, we have $(x, y) = (x_i, y_i)$. According to Lemma 3.15, we have that (x, y) is either in $\{(v, z_1), \dots, (v, z_i)\}$, or it is the $low_{i'}$-edge of a child c of v, for an $i' \in \{1, \dots, i\}$. In the first case, we have that (x, y) was inserted into *BST*$[v]$ in Line 6. In the second case, due to the inductive hypothesis, we have that (x, y) was processed at some point during the **for** loop in Line 9 (during the processing of a child c of v). We will show that, when we reach Line 24, we have that (x, y) is contained in *BST*$[v]$.

First, let us suppose, for the sake of contradiction, that (x, y) was inserted into *BST*$[v]$ at some point, but later on it was deleted. Then the deletion took place in Line 18, due to the existence of a back-edge $(z, w) \in B(c)$ that has $w < y$, for a child c of v, and (x, y) was the k-th entry of *BST*$[v]$. But then, due to the sorting of *BST*$[v]$, this implies that there are k back-edges in $B(v)$ that precede (x, y) in the *low* ordering, contradicting the fact that (x, y) is the low_i-edge of v, for some $i \leq k$. Thus, it is impossible that (x, y) was inserted into *BST*$[v]$ at some point, but later on it was deleted. Now let us suppose, for the sake of contradiction, that (x, y) was processed at some point, but it was not inserted into *BST*$[v]$. This implies that (x, y) was met during the processing of a child c of v, Line 17 was reached, but the condition in this line was not satisfied. Thus, the k-th entry (x', y') of *BST*$[v]$ had $y' \leq y$. Since the back-edges in *BST*$[v]$ are sorted in the *low* ordering, we cannot have that (x', y') is a predecessor of (x, y) in this ordering, because otherwise we contradict the fact that (x, y) is the low_i-edge of v, where $i \leq k$. Thus, we have that either $y < y'$, or $y = y'$ and $x < x'$, (or $(x, y) = (x', y')$, but the precedence is given to (x, y)). Thus, since $y' \leq y$, we have $y = y'$, and therefore $x \leq x'$. Let c' be the child whose processing led to the insertion of (x', y') in *BST*$[v]$. Then, since we process the children of v in increasing order, we have that $c' \leq c$. Thus, since x is a descendant of c, and x' is a descendant of c', and $x \leq x'$, we have that $c' = c$. But then, since (x, y) precedes (x', y') in the *low* orderding, we have that (x, y) was met before (x', y') during the **for** loop in Line 9, due to the selection in Line 10, a contradiction. This shows that, when

we reach Line 24, we have that (x, y) is contained in *BST*$[v]$.

Now let k' be the maximum index in $\{1, \ldots, k\}$ such that the $low_{k'}$-edge of v is not null. (We note that, if $k' < k$, then $|B(v)| = k'$.) Then we have that, when we reach Line 24, *BST*$[v]$ is filled with k' back-edges from $B(v)$. Furthermore, we have established that the low_i-edge of v is included in *BST*$[v]$, for every $i \in \{1, \ldots, k'\}$. Thus, since the sorting of *BST*$[v]$ corresponds to the sorting of the list of back-edges in $B(v)$ that provides the *low*-edges, we have that the i-th entry of *BST*$[v]$ is the low_i-edge of v, for every $i \in \{1, \ldots, k'\}$.

This establishes the correctness of Algorithm 1. It is not difficult to argue about its complexity. First, the sorting of the adjacency lists in Line 1 can be performed in $O(m)$ time with bucket-sort (since the *low* ordering is basically a variant of lexicographic order). Now, whenever the **for** loop in Line 4 processes a vertex v, we have that *BST*$[v]$ is filled with at most k edges in Line 6. Then, the **for** loop in Line 9 processes at most $k \cdot \mathit{numChild}(v)$ back-edges in total, where $\mathit{numChild}(v)$ denotes the number of the children of v. Every one of those back-edges may be inserted into *BST*$[v]$ either in Line 13 or in Line 19. Furthermore, it may force a deletion from *BST*$[v]$ in Line 18. Thus, the total number of *BST* operations during the processing of v is $O(k(\mathit{numChild}(v)+1))$. Thus, the total number of *BST* operations during the course of Algorithm 1 is $O(kn)$. Since every one of those operations is performed on a balanced binary-search tree with no more than k entries, we have that it incurs cost $O(\log k)$. Thus, the total cost of *BST* operations during the course of Algorithm 1 is $O(nk \log k)$. We conclude that Algorithm 1 runs in $O(m + nk \log k)$ time. □

3.4 Computing the *high*-edges

Let $v \neq r$ be a vertex. The definition of the $high_i$-edges of v, for $i = 1, 2, \ldots$, assumes any ordering of the back-edges in $B(v)$ that it is decreasing w.r.t. the lower endpoints. For convenience in our arguments, we will fix such an ordering for sets of back-edges, which we call the *high* ordering. Let $(x_1, y_1), \ldots, (x_t, y_t)$ be a list of back-edges sorted in decreasing order w.r.t. the lower endpoints, while also satisfying $x_i \leq x_{i+1}$, for every $i \in \{1, \ldots, t-1\}$ such that $y_i = y_{i+1}$. Then we say that this list is sorted in the *high* ordering. If $(x_1, y_1), \ldots, (x_t, y_t)$ is the list of the back-edges in $B(v)$ sorted in the *high* ordering, then we let (x_i, y_i) be the $high_i$-edge of v, for every $i \in \{1, \ldots, t\}$. We

assume that the *high* ordering is applied for every set of leaping back-edges, and the $high_i$-edges correspond to this ordering.

Now let k be a fixed positive integer. In order to compute the $high_i$-edges of all vertices, for every $i \in \{1, \ldots, k\}$, the idea is to process all the back-edges according to the *high* ordering. For every vertex v, we maintain a variable $minIndex[v]$, that stores the minimum i such that the $high_i$-edge of v is not yet computed. (Initially, we set $minIndex[v] \leftarrow 1$.) Then, for every back-edge (x, y) that we process, we ascend the tree-path $T[x, y)$, and we have that the $high_i$-edge of v is (x, y), for every $v \in T[x, y)$ such that $minIndex[v] = i$. In order to implement this idea efficiently, whenever we ascend the path $T[x, y)$ during the processing of a back-edge (x, y), we have to avoid the vertices for which the $high_i$ edges are computed, for every $i \in \{1, \ldots, k\}$. (In other words, we have to avoid the vertices that have $minIndex = k + 1$.) We can achieve this with the use of a DSU data structure. Specifically, the DSU data structure maintains sets of vertices that have $minIndex = k + 1$, or they are singletons. The sets maintained by the data structure are subtrees of T. The operations supported by this data structure are $\texttt{find}(v)$ and $\texttt{unite}(u, v)$. The operation $\texttt{find}(v)$ returns the root of the subtree maintained by the DSU that contains v. The operation $\texttt{unite}(u, v)$ unites the subtree that contains u with the subtree that contains v into a larger subtree. Whenever $\texttt{unite}(u, v)$ is called, we have $v = p(u)$. We use those operations whenever we ascend the tree-path $T[x, y)$, during the processing of a back-edge (x, y). Thus, whenever we meet a vertex v which has $minIndex[v] = k + 1$, we first unite it with its parent $p(v)$, if $p(v)$ also has $minIndex[p(v)] = k + 1$, and then we move to $\texttt{find}(p(v))$. Thus, the next time that we meet v, we can jump immediately to the root of the subtree maintained by the DSU that contains $p(v)$. This idea for computing the $high_i$-edges of all vertices, for every $i \in \{1, \ldots, k\}$, is shown in Algorithm 2. The proof of correctness and the complexity of this algorithm is given in Proposition 3.3.

Proposition 3.3. *Let k be any fixed positive integer. Then Algorithm 2 computes the $high_i$-edges of all vertices $\neq r$, for every $i \in \{1, \ldots, k\}$, in $O(m + kn)$ time in total.*

Proof. We will prove correctness inductively, by establishing that whenever we reach the condition of the **while** loop in Line 12, we have:

(1) For every vertex z such that $minIndex[z] \leq k$, the $high_i$-edge of z has been correctly computed, for every $i < minIndex[z]$.

Algorithm 2: Compute the $high_i$-edges of all vertices, for all $i \in \{1, \ldots, k\}$

```
1  sort the adjacency list of every vertex in increasing order w.r.t. the higher
     endpoints of its edges
2  initialize an array minIndex, for every vertex v ≠ r
3  foreach vertex v ≠ r do
4      foreach i ∈ {1, ..., k} do
5          let the high_i-edge of v be ⊥
6      end
7      let minIndex[v] ← 1
8  end
9  for y ← n − 1 to y = 1 do
10     foreach back-edge (x, y) in the adjacency list of y do
11         let v ← x
12         while v ≠ y do
13             if minIndex[v] = k + 1 then
14                 while minIndex[p[v]] = k + 1 do
15                     unite(v, p(v))
16                     v ← find(v)
17                 end
18                 v ← p(v)
19             end
20             if v = y then break
21             let i ← minIndex[v]
22             let the high_i-edge of v be (x, y)
23             minIndex[v] ← i + 1
24             v ← p(v)
25         end
26     end
27 end
```

(2) For every vertex z such that *minIndex*$[z] \leq k$, no back-edge that has been processed so far by the **for** loop in Line 10 prior to the processing of (x, y) is the $high_i$-edge of z, for any $i \geq$ *minIndex*$[z]$.

(3) v lies on the tree-path $T[x, y]$.

(4) Every vertex v' that lies on the tree-path $T[x, y)$ and is a proper descendant of v such that (x, y) is the $high_i$-edge of v' for some $i \in \{1, \ldots, k\}$, has its $high_i$-edge correctly computed and *minIndex*$[v'] = i + 1$.

(5) For every vertex z such that *minIndex*$[z] > k$, we have *minIndex*$[z] = k+1$. In this case, the $high_i$-edge of z has been correctly computed, for every $i \in \{1, \ldots, k\}$. Furthermore, the set that is maintained by the DSU data structure and contains z is a subtree of T.

(6) Every set that is maintained by the DSU data structure and is not a singleton, has the property that all its vertices have *minIndex* $= k + 1$.

First, let us consider the first time that we reach the condition of the **while** loop in Line 12. Then, all vertices have *minIndex* 1. Thus, properties (1) and (5) are trivially satisfied. Furthermore, since the **for** loop in Line 10 has not processed any back-edge prior to (x, y), condition (2) is trivially satisfied. Also, since $v = x$ and x has no proper descendants on the tree-path $T[x, y)$, we have that (3) and (4) are trivially satisfied. Finally, we have not performed any DSU operations yet, and so every set maintained by the DSU data structure is a singleton. Thus, (6) is also satisfied. This establishes the base step of our induction.

Now suppose that we reach the condition of the **while** loop in Line 12 and our inductive hypothesis is true. First, suppose that $v = y$. This implies that either the computation will stop (in which case there is nothing to show), or the **for** loop in Line 10 will start processing a new back-edge (x', y') (we note that y' is not necessarily y, because the **for** loop in Line 9 may have changed the value of the variable "y"). Then, the invariants (1), (5) and (6) are obviously maintained. Property (4) implies that every vertex v' whose $high_i$-edge is (x, y), for some $i \in \{1, \ldots, k\}$, has its $high_i$-edge correctly computed and *minIndex*$[v'] = i+1$. This fact, in conjunction with (2), implies that (2) will also hold true when we reach for the first time the condition of the **while** loop in Line 12 during the processing of (x', y'). Finally, we will have (3) (since $v = x'$) and condition (4) will be trivially true (because x' has no proper descendants on the tree-path $T[x', y')$). Thus, the inductive hypothesis will still be true. So let us suppose that, when we reach the condition of the **while** loop in Line 12, our inductive hypothesis is true and $v \neq y$. Then we will enter the **while** loop in Line 12. Here we

distinguish two cases: either (i) $\mathit{minIndex}[v] \leq k$, or (ii) $\mathit{minIndex}[v] > k$.

Let us consider case (i) first. Then the condition in Line 13 is not satisfied, and therefore we go to Line 21, and we will set $i \leftarrow \mathit{minIndex}[v]$. By property (2) we have that no back-edge that has been processed so far by the **for** loop in Line 10 prior to the processing of (x, y) is the high_i-edge of v. By property (3) we have that (x, y) is a back-edge in $B(v)$. Thus, since the **for** loop in Line 9 processes the vertices "y" in decreasing order, and since the **for** loop in Line 10 processes the incoming back-edges to y in increasing order w.r.t. their higher endpoint (due to the sorting in Line 1), we have that the high_i-edge of v is (x, y). Thus, the assignment in Line 22 is correct. Then, in Line 23 we let $\mathit{minIndex}[v] \leftarrow i+1$. Let $v' = p(v)$. Then we reach the condition of the **while** loop in Line 12, and the "v" variable holds the value v'. The invariant (1) is maintained, because we have only increased $\mathit{minIndex}[v]$ by one, and we have correctly computed all high_j-edges of v, for $j < \mathit{minIndex}[v]$. Since v had $\mathit{minIndex} \leq k$, by condition (6) we had that v was in a singleton set of the DSU data structure. Thus, since we performed no DSU operations, (6) and (5) are maintainted. Since v was on the tree-path $T[x, y)$, we have that $p(v)$ is on the tree-path $T[x, y]$. Thus, (3) is also true. Property (2) is obviously maintained, and (4) is still true because we have correctly established that (x, y) is the high_i-edge of v. Thus, the inductive hypothesis still holds.

Now let us consider case (ii). By property (5) we have $\mathit{minIndex}[v] = k + 1$. Thus, the condition in Line 13 is satisfied. First, suppose that the condition of the **while** loop in Line 14 is not true. Then we simply perform $v \leftarrow p(v)$ in Line 18, and we go to Line 20. Then, properties (1), (2), (5) and (6) are obviously maintained. By property (3), v was on the tree-path $T[x, y)$ (since $v \neq y$). Thus, $p(v)$ is on the tree-path $T[x, y]$, and so (3) is also true. By property (5) we have that the high_i-edge of v was correctly computed, for every $i \in \{1, \ldots, k\}$. Thus, property (4) is also maintained. Thus, all the invariants are maintained, and so, when we reach Line 20, we can argue as previously, in order to show that when reach the condition of the **while** loop in Line 12 again, our inductive hypothesis will still be true.

So let us suppose that the condition of the **while** loop in Line 14 is true. Then we will unite the set that contains v with the set that contains its parent, with a call to `unite`$(v, p(v))$. By property (5) we have that both those sets are subtrees of T. Thus, joining two subtrees with a parent-edge maintaints this invariant. Notice that invariant (6) is maintained. Next, due to the convention we made in the main text

concerning the calls to `find`, we have that `find`(v) will return the root of the new subtree that is formed. Now we repeat the **while** loop in Line 14, until we reach a vertex v such that *minIndex*$[p(v)] \neq k+1$. Since this process does not change the values *minIndex* of the vertices, by property (5) we have *minIndex*$[p(v)] \leq k$. Then we go to Line 18. Notice that the invariants (1) and (2) are maintained, as well as (5) and (6). We claim that, when we reach Line 18, we have $p(v) \in T[x,y]$. To see this, first observe that the **while** loop in Line 14 was ascending the tree-path $T[x,y)$, starting from a vertex v_0 on $T[x,y)$ (due to property (3)). Then, all vertices from v_0 up to v have *minIndex* $k+1$ (due to (5), (6), and the fact that the condition of the **while** loop in Line 14 was repeatedly satisfied). Then, by (5) we have that the $high_i$-edges of v are correctly computed, for every $i \in \{1,\dots,k\}$. Since the **for** loop in Line 9 processes the vertices "y" in decreasing order, we have that no back-edge that was processed so far by the **for** loop in Line 10 has low enough lower endpoint to be a back-edge in $B(y)$. Thus, no $high_i$-edge of y is computed as yet, for any $i \in \{1,\dots,k\}$, and therefore *minIndex*$[y]=1$. Since all vertices on the tree-path from v_0 to v have *minIndex* $k+1$, we have that v is still a proper descendant of y, and therefore $p(v) \in T[x,y]$. Thus, invariant (3) is maintained. Finally, since all vertices on the tree-path from v_0 to v have all their $high_i$-edges computed, for all $i \in \{1,\dots,k\}$, we have that invariant (4) is maintained too. Thus, when we reach Line 20, we have that all the properties of the inductive hypothesis are satisfied. Then, from this point we can argue as previously, in order to show that when we reach the condition of the **while** loop in Line 12 again, our inductive hypothesis will still be true.

This shows the correctness of Algorithm 2. It remains to analyze its complexity. The sorting of the adjacency lists in Line 1 can be performed in $O(m)$ time with bucket-sort. Whenever we enter the **while** loop in Line 12, by the inductive hypothesis we have that, when we reach Line 20, we have that either $v=y$ or v is a vertex that has its $high_i$-edge yet to be computed, for some $i \in \{1,\dots,k\}$, but this will be computed correctly in Line 22. The case $v=y$ can be charged to the back-edge (x,y) that is currently processed by the **for** loop in Line 10. The other case can be charged to the $high_i$-edge of v. Thus, the **while** loop in Line 12 will be entered $O(m+kn)$ times in total. Whenever we enter the **while** loop in Line 12, the first run of the **while** loop in Line 14 can be charged to this entry. The remaining entries can be charged to the calls to `unite`, all of which are non-trivial (i.e., they join sets that were previously not united). This is because the call to `find`(v) in Line 16 returns the root of the subtree

maintained by the DSU data structure and contains v. Thus, after the assignment $v \leftarrow \texttt{find}(v)$ in Line 16, we have that v is not in the same set of the DSU data structure that contains $p(v)$. Therefore, in the next run of the **while** loop in Line 14, the operation $\texttt{unite}(v, p(v))$ will unite two disjoint sets. Thus, since the number of non-trivial calls to `unite` is $O(n)$, we have that the number of times that we enter the **while** loop in Line 14 is dominated by the number of times that we enter the **while** loop in Line 12. The final thing to do is to bound the cost of the DSU operations. Since the tree of the `unite` operations is known beforehand (i.e., it coincides with T), we can use the data structure of Gabow and Tarjan [33]. Thus, any sequence of m' `find` and `unite` operations can be performed in $O(m' + n)$ time in total. In our case, we have $m' = O(m + nk)$. We conclude that Algorithm 2 has an $O(m + kn)$-time implementation. □

3.5 Computing the leftmost and the rightmost edges

Let $v \neq r$ be a vertex, and let $(x_1, y_1), \ldots, (x_k, y_k)$ be the list of the back-edges in $B(v)$ sorted in lexicographic order. In other words, we have $x_1 \leq \cdots \leq x_k$, and if $x_i = x_{i+1}$ then $y_i \leq y_{i+1}$, for any $i \in \{1, \ldots, k-1\}$.[2] Now let c be a descendant of v. We are interested in computing some subsets of $B(v)$ that consist of back-edges whose higher endpoint is a descendant of c. Notice that, if i is the lowest index in $\{1, \ldots, k\}$ such that $x_i \in T(c)$, then, due to the lexicographic order, the set of the back-edges in $B(v)$ whose higher endpoint is a descendant of c is a segment of $B(v)$ starting from (x_i, y_i).

For every $t = 1, 2, \ldots$, we let $L(v, c, t)$ denote the set of the first t back-edges in $B(v)$ whose higher endpoint is a descendant of c. More precisely, $L(v, c, t)$ is defined as follows. Let i be the lowest index in $\{1, \ldots, k\}$ such that x_i is a descendant of c. If such an index does not exist, then we let $L(v, c, t) = \emptyset$. Otherwise, let j be the maximum index in $\{i, \ldots, i + t - 1\}$ such that x_j is a descendant of c. Then $L(v, c, t) = \{(x_i, y_i), \ldots, (x_j, y_j)\}$. Similarly, we let $R(v, c, t)$ denote the set of the last t back-edges in $B(v)$ whose higher endpoint is a descendant of c. More precisely, $R(v, c, t)$ is defined as follows. Let i' be the greatest index in $\{1, \ldots, k\}$ such that $x_{i'}$ is a descendant of c. If such an index does not exist, then we let $R(v, c, t) = \emptyset$. Otherwise,

[2]Notice that, since we consider multigraphs, we may have that an entry (x, y) appears several times; then we break ties according to the unique identifiers of the edges of the graph.

let j' be the minimum index in $\{i'-t+1,\dots,i'\}$ such that $x_{j'}$ is a descendant of c. Then $R(v,c,t)=\{(x_{j'},y_{j'}),\dots,(x_{i'},y_{i'})\}$.

The challenge in computing the sets $L(v,c,t)$ and $R(v,c,t)$ is that we do not have direct access to the list $B(v)$. Thus, the straightforward way to compute $L(v,c,t)$ is to start processing the vertices in $T(c)$ in increasing order, starting from c. For every vertex x that we process, we scan the adjacency list of x for back-edges of the form (x,y). If $y<v$, then we have $(x,y)\in B(v)$, and therefore we collect (x,y). We continue this process until we have collected t back-edges, or we have exhausted the search in the subtree of c. Similarly, in order to compute $R(v,c,t)$ we perform the same search, starting from the greatest descendant of c (i.e., $c+\mathit{ND}(c)-1$), and we process the vertices in decreasing order. Obviously, this method may take $O(m')$ time, where m' is the number of edges with one endpoint in $T(c)$. Notice that m' can be as large as $\Omega(m)$. Thus, this method is impractical if the number of L or R sets that we want to compute is $\Omega(n)$.

Thus, we will assume that the queries for the L and R sets are given in batches, to be performed in an off-line manner. We will focus on computing the L sets. The method and the arguments for the R sets are similar. Thus, let $L(v_1,c_1,t_1),\dots,L(v_N,c_N,t_N)$ be the L sets that we have to compute. (We assume that c_i is a descendant of v_i, for every $i\in\{1,\dots,N\}$, and $t_i\geq 1$.) The idea is to apply the straightforward algorithm that we described above, but we process the queries in an order that is convenient for us. Specifically, we processs the vertices v_i in a bottom-up fashion. By doing so, we can avoid vertices that previously were unable to provide a leaping back-edge. More precisely, if we meet a vertex x that is a descendant of c_i, during the processing of a query $L(v_i,c_i,t_i)$, then, if we have $l(x)\geq v_i$, we can be certain that there is no back-edge of the form (x,y) in any of $B(v_i),\dots,B(v_N)$. Thus, we mark x as "inactive", and we attach it to a segment of inactive vertices which we can bypass at once, because they cannot provide a leaping back-edge with low enough lower endpoint anymore. Initially, we assume that all vertices are active.

Thus, for every vertex x, we maintain a boolean attribute *is_active*(x), which is *true* if and only if x is active. We use a disjoint-set union data structure *DSU* on the set of inactive vertices, that maintains the partition of the maximal segments (w.r.t. the DFS numbering) of inactive vertices. Thus, if two vertices x and $x+1$ are inactive, then they belong to the same set maintained by the DSU data structure. *DSU* supports the operations `find`(x) and `unite`(x,y). We use `find`(x), on an inactive vertex x, in

order to return a representative of the segment of inactive vertices that contains x. With the operation `unite`(x, y) we unite the segment that contains x with the segment that contains y (we assume that $y = x + 1$). Also, for every vertex x, we maintain two pointers *left*(x) and *right*(x). These are only used when x is an inactive vertex, which is a representative of its segment S. In this case, *left*(x) points to the greatest vertex that is lower than x and not in S, and *right*(x) points to the lowest vertex that is greater than x and not in S. Then, by definition, we have that *left*(x) (resp., *right*(x)) is either an active vertex, or $\perp$. We maintain the invariant that, whenever a DSU operation changes the representative of a segment, it passes the *left* and the *right* pointer of the old representative to the new one (so that we can correctly retrieve the endpoints of the corresponding maximal segment).

Now we can describe in more detail the procedure for computing the sets $L(v_1, c_1, t_1), \dots, L(v_N, c_N, t_N)$. Recall that we process these queries in decreasing order w.r.t. the vertices v_i. We assume that the edges in the adjacency list of every vertex are sorted in increasing order w.r.t. their lower endpoint. Now, in order to answer a query $L(v, c, t)$, we begin the search from the lowest active vertex x that is a descendant of c. This is because all descendants of c that are lower than x are incapable of providing a back-edge that leaps over v. If c is active, then we have $x = c$. Otherwise, we have that c is inactive, and we use $z \leftarrow$ `find`(c) in order to get the representative z of the maximal segment of inactive vertices that contains c. Then we have $x =$ *right*(z). If $x = \perp$ or x is not a descendant of c, then we know that there are no back-edges with higher endpoint in $T(c)$ that leap over v (and so we have $L(v, c, t) = \emptyset$). Otherwise, we check whether $l(x) < v$. If that is the case, then we start traversing the adjacency list of x, in order to get back-edges of the form (x, y) that leap over v. We keep doing that until we have either collected t back-edges for $L(v, c, t)$, or we have reached a back-edge that does not leap over v (because its lower endpoint is not low enough), or we have reached the end of the adjacency list. If we have gathered less than t back-edges that leap over v, then we continue the search in the lowest active vertex that is greater than x and still a descendant of c. Otherwise, if we have $l(x) \geq v$, then we mark x as inactive, so as not to process it again. Then we have to properly update the segments. To do so, we have to check whether $x - 1$ or $x + 1$ is an inactive vertex. If none of those vertices is inactive, then there is nothing we have to do. So let us suppose that $x - 1$ is an inactive vertex. Then we expand the segment that contains $x - 1$ with a call `unite`$(x - 1, x)$, and we set the *right* pointer of the representative of

this segment to $x+1$. Then, if $x+1$ is inactive, we have to further expand the segment with a call `unite`$(x, x+1)$; otherwise, we are done. On the other hand, if only $x+1$ is inactive, then we expand the segment that contains it with a call `unite`$(x, x+1)$, and we set the *left* pointer of the representative of this segment to $x-1$. Now, after updating the segments, we proceed to the lowest active vertex that is greater than x and a descendant of c. Then we repeat the same process.

Algorithm 3: Compute the sets $L(v_1, c_1, t_1), \ldots, L(v_N, c_N, t_N)$, where c_i is a descendant of v_i, for every $i \in \{1, \ldots, N\}$

```
1  let Q be the list of all triples (v_1, c_1, t_1), ..., (v_N, c_N, t_N) sorted in decreasing order w.r.t. the
     first component
2  sort the adjacency list of every vertex in increasing order w.r.t. the lower endpoints
3  initialize a boolean array is_active with n entries
4  foreach vertex v do set is_active[v] ← true
5  foreach vertex v do initialize two pointers v.left ← v − 1 and v.right ← v + 1
6  foreach (v, c, t) ∈ Q do
7      let L(v, c, t) ← ∅
8      set counter ← 0 // counter for the number of back-edges we have collected
9      let x ← c
10     if is_active(x) = false then  x ← find(x).right
11     while x ≠ ⊥ and x is a descendant of c do
12         if l(x) < v then
13             let e = (x, y) be the first back-edge in the adjacency list of x
14             while y < v and counter < t do
15                 insert (x, y) into L(v, c, t)
16                 counter ← counter + 1
17                 let e = (x, y) be the next back-edge in the adjacency list of x
18                 if e = ⊥ then break
19             end
20             if counter = t then break
21         end
22         else
23             is_active(x) ← false
24             if is_active(x − 1) = false and is_active(x + 1) = true then
25                 unite(x − 1, x), find(x).right ← x + 1
26             end
27             if is_active(x − 1) = true and is_active(x + 1) = false then
28                 unite(x, x + 1), find(x).left ← x − 1
29             end
30             if is_active(x − 1) = false and is_active(x + 1) = false then
31                 unite(x − 1, x), unite(x, x + 1)
32             end
33         end
34         x ← x + 1
35         if x = ⊥ then break
36         if is_active(x) = false then  x ← find(x).right
37     end
38 end
```

This procedure for computing the sets $L(v_1, c_1, t_1), \ldots, L(v_N, c_N, t_N)$ is shown in Algorithm 3. The proof of correctness and linear complexity is given in Proposition 3.4. After that, we describe the minor changes that we have to make to Algorithm 3 in order to get an algorithm that computes sets of the form $R(v_1, c_1, t_1), \ldots, R(v_N, c_N, t_N)$, with the same time-bound guarantees.

Proposition 3.4. *Let $L(v_1, c_1, t_1), \ldots, L(v_N, c_N, t_N)$ be a collection of queries, where c_i is a descendant of v_i, for every $i \in \{1, \ldots, N\}$, and $t_i \geq 1$. Then Algorithm 3 correctly computes the sets $L(v_1, c_1, t_1), \ldots, L(v_N, c_N, t_N)$. Furthermore, it runs in $O(t_1 + \cdots + t_N + m)$ time.*

Proof. We will prove correctness inductively, by establishing the following: Whenever the **for** loop in Line 6 processes a triple $(v, c, t) \in Q$, we have that (1) every vertex x with $l(x) < v$ is active, and every inactive vertex x has $l(x) \geq v$, and (2) for every inactive vertex x, $\texttt{find}(x)$.*left* is the greatest active vertex x' such that $x' < x$, and $\texttt{find}(x)$.*right* is the lowest active vertex x' such that $x' > x$.

Initially, all vertices are active. Thus, the inductive hypothesis is trivially true before entering the **for** loop in Line 6. Now suppose that the inductive hypothesis holds by the time the **for** loop in Line 6 processes a triple $(v, c, t) \in Q$. Then we will show that $L(v, c, t)$ will be correctly computed, and the inductive hypothesis will still hold for the next triple (v', c', t') that will be processed by the **for** loop in Line 6.

Now, given the query $L(v, c, t)$, the first thing to do it to find the lowest descendant x of c that can provide a back-edge of the form $(x, y) \in B(v)$. Notice that x satisfies the property $l(x) \leq y < v$, and therefore it is active, according to (1). Thus, we first check whether c is active, in Line 10. If c is not active, then by (1) it has $l(c) \geq v$, and therefore it is proper to set $x \leftarrow \texttt{find}(x)$.*right* in Line 10. According to (2), now x is the lowest active vertex that is greater than c. Otherwise, if c is active, then we have $x = c$, due to the assignment in Line 9. Now, if $x = \perp$ or x is great enough to not be a descendant of c, then we will not enter the **while** loop in Line 11, and the computation of $L(v, c, t)$ is over (i.e., we have $L(v, c, t) = \emptyset$). This is correct, because there is no descendant x of c that can provide a back-edge of the form $(x, y) \in B(v)$. Notice that the inductive hypothesis will still hold for the next query, because we have made no changes in the underlying data structures.

Otherwise, if we enter the **while** loop in Line 11, then we have that x is the lowest active descendant of c, and therefore in Line 12 we check whether it can provide a back-edge of the form $(x, y) \in B(v)$. Notice that this is equivalent to $l(x) < v$. The

necessity was already shown. To prove sufficiency, we note that $l(x) < v$ implies that there is a back-edge of the form (x, y) such that $y < v$. We have that x is a descendant of c, and therefore a descendant of v (due to the assumption concerning the queries). Then, since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Thus, $y < v$ implies that y is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Now, if the condition in Line 12 is satisfied, then we have to gather as many back-edges of the form $(x, y) \in B(v)$ as x can provide, provided that we do not exceed t. This is precisely the purpose of Lines 13 to 20. Correctness follows from the fact that the adjacency list of x is sorted in increasing order w.r.t. the lower endpoints. The variable *counter* counts precisely the number of back-edges (x', y) that we have gathered, where x' is a descendant of c and $(x', y) \in B(v)$. (*counter* has been initialized to 0 in Line 8.) Thus, if the condition in Line 20 is satisfied, then it is proper to stop the computation of $L(v, c, t)$, because it has been correctly computed. Notice that the inductive hypothesis holds true, because we have made no changes in the underlying data structure. Otherwise, if *counter* $< t$, then we have to proceed to the lowest active descendant of c that is greater than x. This is precisely the purpose of Lines 34 to 36. Thus, first we move to $x + 1$. Now, if $x \neq \perp$ and x is active, then we go to the condition of the **while** loop in Line 11. Otherwise, if x is inactive, then it is correct to set $x \leftarrow$ `find`(x).*right* in Line 36, because now x has moved to its lowest active successor, due to (2) of the inductive hypothesis.

Now suppose that the condition in Line 12 is false. Then we have $l(x) \geq v$, and therefore we set the mode of x to inactive, in Line 23. Notice that point of (1) of the inductive hypothesis is maintained, because we process the triples (v', c', t') in decreasing order w.r.t. their first component. Thus, the next such triple has $v' \leq v$, and therefore we have $l(x) \geq v \geq v'$. Now, since x is made inactive, we have to maintain invariant (2). This is the purpose of Lines 24 to 32. The idea in those lines is the following. First, if both $x - 1$ and $x + 1$ are active, then there is nothing to do, because `find`$(x) = x$, and the pointers x.*left* and x.*right* have been initialized to $x - 1$ and $x + 1$, respectively, in Line 5. Otherwise, if $x - 1$ is inactive and $x + 1$ is active, then we join x to the segment of inactive vertices that precede it with a call `unite`$(x - 1, x)$ in Line 25. Then, we have to set `find`(x).*right* $\leftarrow x + 1$, which is done in Line 25. We work similarly, if $x - 1$ is active and $x + 1$ is inactive. Finally, if both $x - 1$ and $x + 1$ are inactive, then it is sufficient to join x to the segments of inactive vertices that

precede it and succeed it, which is done in Lines 31 and 31. We only have to make sure that the *left* pointer of the new representative is the same as the *left* pointer of what was previously the representative of the segment that contained $x-1$, and the *right* pointer of the new representative is the same as the *right* pointer of what was previously the representative of the segment that contained $x+1$. Thus, the point (2) of the inductive hypothesis is maintained.

This establishes the correctness of Algorithm 3 (taking also into account our presentation of the general idea in the main text). It remains to argue about the complexity of Algorithm 3. First, the sorting of the triples in Line 1 can be performed in $O(n+N)$ time in total with bucket-sort. Also, the sorting of the adjacency lists in Line 2 can be performed in $O(m)$ time in total with bucket-sort. Whenever the **for** loop in Line 6 processes a triple (v,c,t), we have that the **while** loop in Line 11 will only process those x that either have $l(x)<v$, and so they will provide at least one back-edge for $L(v,c,t)$, or they have $l(x)\geq v$, in which case they will be made inactive, and will not be accessed again in this **while** loop for any further triple. Thus, the total number of runs of the **while** loop in Line 11 is $O(n+t_1+\cdots+t_N)$. Every x that we encounter that has $l(x)<v$, will initiate the **while** loop in Line 14. But the number of runs of this **while** loop will not exceed t (for (v,c,t)). Thus, the total number of runs of the **while** loop in Line 14 is $O(t_1+\cdots+t_N)$. Finally, the `unite` operations that we perform with the DSU data structure have the form `unite`$(x,x+1)$. Since the tree-structure of the calls to `unite` is predetermined (it is essentially a path), we can use the data structure of Gabow and Tarjan [33] that performs a sequence of m' DSU operations in $O(m'+n)$ time, when applied on a set of n elements. Notice that the number of calls to the DSU data structure is $O(n+t_1+\cdots+t_N)$. Thus, the most expensive time expressions that we have gathered are $O(m)$, $O(n+N)$ and $O(n+t_1+\cdots+t_N)$. Since $t_i\geq 1$, for every $i\in\{1,\dots,N\}$, we have $N\leq t_1+\cdots+t_N$. We conclude that Algorithm 3 runs in $O(m+t_1+\cdots+t_N)$ time. □

With only minor changes to Algorithm 3 we can also answer queries of the form $R(v_1,c_1,t_1),\dots,R(v_N,c_N,t_N)$, where c_i is a descendant of v_i, for every $i\in\{1,\dots,N\}$, and $t_i\geq 1$, in $O(t_1+\cdots+t_N+m)$ time in total. These changes are as follows. First, in order to compute a query of the form $R(v,c,t)$, we start the search from the greatest active descendant of c. Thus, we replace Line 9 with "$x\leftarrow c+ND(c)-1$", and Line 10 with "$x\leftarrow$ `find`(x).*left*". Then, since we want to process the active descendants

of c in decreasing order, we replace Line 34 with "$x \leftarrow x - 1$", and Line 36 with $x \leftarrow$ `find`(x).*left*". Now the proof of correctness is similar as in Proposition 3.4. In particular, we can argue using the same inductive hypothesis.

3.6 Computing the maximum points

Given a vertex $v \neq r$, we will need an efficient method to compute the values $M(v)$ and $\widetilde{M}(v)$, as well as values of the form $M(v,c)$ and $M(B(v) \setminus S)$, where c is a descendant of v, and S is a subset of $B(v)$. In Section 3.7, we show how to compute the values $M(v)$ and $\widetilde{M}(v)$, for all vertices $v \neq r$, in linear time in total. Alternatively, we have $M(v) = M(v,v)$. Also, let c_1 and c_2 be the *low1* and the *low2* child of $M(v)$, respectively. Then it is easy to see that $\widetilde{M}(v) = M(v)$ if ($c_2 \neq \bot$ and) $low(c_2) < v$, and $\widetilde{M}(v) = M(v, c_1)$ otherwise. Thus, the computation of both $M(v)$ and $\widetilde{M}(v)$ can be reduced to the computation of values of the form $M(v,c)$, where c is a descendant of v.

Now let $v \neq r$ be a vertex, and let c be a descendant of v. Let $(x_1, y_1), \dots, (x_k, y_k)$ be the list of the back-edges in $B(v) \cap B(c)$, sorted in lexicographic order (and thus in increasing order w.r.t. their higher endpoint). Then, following the notation in Section 3.5, we let $L(v,c,t)$ denote the set $\{(x_1,y_1), \dots, (x_t,y_t)\}$. Similarly, we let $R(v,c,t)$ denote the set $\{(x_k,y_k), \dots, (x_{k-t+1}, y_{k-t+1})\}$. Then we have the following.

Proposition 3.5. *Let $(v_1,c_1), \dots, (v_N,c_N)$ be a sequence of pairs of vertices such that $v_i \neq r$ and c_i is a descendant of v_i, for every $i \in \{1,\dots,N\}$. Then the values $M(v_1,c_1),\dots,M(v_N,c_N)$ can be computed in $O(m+N)$ time in total.*

Proof. First, we compute the sets $L(v_1,c_1,1),\dots,L(v_N,c_N,1)$ and $R(v_1,c_1,1),\dots,R(v_N,c_N,1)$. According to Proposition 3.4 (and the comments after Algorithm 3), this takes $O(m+N)$ time in total. Then, for every $i \in \{1,\dots,N\}$, we gather the higher endpoint $L(v_i,c_i)$ of the back-edge in $L(v_i,c_i,1)$, and the higher endpoint $R(v_i,c_i)$ of the back-edge in $R(v_i,c_i,1)$. We claim that $M(v_i,c_i) = nca\{L(v_i,c_i), R(v_i,c_i)\}$. To see this, let $(x_1,y_1),\dots,(x_k,y_k)$ be the list of the back-edges in $B(v_i) \cap B(c_i)$, sorted in lexicographic order, so that $L(v_i,c_i) = x_1$ and $R(v_i,c_i) = x_k$. By definition, we have $M(v_i,c_i) = nca\{x_1,\dots,x_k\}$. Thus, $z = nca\{x_1,x_k\}$ is a descendant of $M(v_i,c_i)$. Since $z = nca\{x_1,x_k\}$, we have that z is an ancestor of both x_1 and x_k. This implies that $z \leq x_1 \leq x_k \leq z + ND(z) - 1$. Thus, for every

$j \in \{1, \ldots, k\}$ we have $z \leq x_j \leq z + ND(z) - 1$. This implies that z is an ancestor of all vertices in $\{x_1, \ldots, x_k\}$, and thus z is an ancestor of $M(v_i, c_i)$. This shows that $z = M(v_i, c_i)$.

Thus, we can compute the values $M(v_1, c_1), \ldots, M(v_N, c_N)$, by answering the *nca* queries *nca*$\{L(v_1, c_1), R(v_1, c_1)\}, \ldots$ *nca*$\{L(v_N, c_N), R(v_N, c_N)\}$. By [40] or [12], we know that there is a linear-time preprocessing of T, so that we can answer a collection of N *nca* queries in $O(N)$ time in total. We conclude that the values $M(v_1, c_1), \ldots, M(v_N, c_N)$ can be computed in $O(m + N)$ time in total. □

Similarly, the computation of the values of the form $M(B(v) \setminus S)$ utilizes the leftmost and the rightmost points, as shown by the following.

Lemma 3.16. *Let $v \neq r$ be a vertex, let S be a subset of $B(v)$ with $|S| = k$, and let D be the multiset of the higher endpoints of the back-edges in S. Then $M(B(v) \setminus S) =$ nca$(\{L_1(v), \ldots, L_{k+1}(v), R_1(v), \ldots, R_{k+1}(v)\} \setminus D)$.*

Proof. Let x and y be the the minimum and the maximum, respectively, among the higher endpoints of the back-edges in $B(v) \setminus S$. We claim that *nca*$\{x, y\} = M(B(v) \setminus S)$. First, it is clear that $M(B(v) \setminus S)$ is an ancestor of *nca*$\{x, y\}$. Conversely, let z be the higher endpoint of an edge in $B(v) \setminus S$. Then we have $x \leq z \leq y$. Thus, since *nca*$\{x, y\}$ is an ancestor of both x and y, we have that *nca*$\{x, y\}$ is an ancestor of z. Due to the generality of z, this implies that *nca*$\{x, y\}$ is an ancestor of $M(B(v) \setminus S)$. This shows that $M(B(v) \setminus S) =$ *nca*$\{x, y\}$.

Now let D be the *multiset* of the higher endpoints of the back-edges in S. (That is, if there are t distinct back-edges of the form $(x, y_1), \ldots, (x, y_t)$ in S, for some $t \geq 1$, then D contains at least t multiple entries for x.) We also consider $\{L_1(v), \ldots, L_{k+1}(v), R_1(v), \ldots, R_{k+1}(v)\}$ as a multiset.

It is clear that $M(B(v) \setminus S)$ is an ancestor of *nca*$(\{L_1(v), \ldots, L_{k+1}(v), R_1(v), \ldots, R_{k+1}(v)\} \setminus D)$. To see the converse, notice that $x \in \{L_1(v), \ldots, L_{k+1}(v)\} \setminus D$ and $y \in \{R_1(v), \ldots, R_{k+1}(v)\} \setminus D$, since $|D| = k$. Thus, we have that *nca*$(\{L_1(v), \ldots, L_{k+1}(v), R_1(v), \ldots, R_{k+1}(v)\} \setminus D)$ is an ancestor of *nca*$\{x, y\}$, and therefore an ancestor of $M(B(v) \setminus S)$. This shows that $M(B(v) \setminus S) =$ *nca*$(\{L_1(v), \ldots, L_{k+1}(v), R_1(v), \ldots, R_{k+1}(v)\} \setminus D)$. □

Proposition 3.6. *Let $(v_1, S_1), \ldots, (v_N, S_N)$ be a collection of pairs of vertices and sets of back-edges, such that $v_i \neq r$ and $\emptyset \neq S_i \subseteq B(v_i)$ for every $i \in \{1, \ldots, N\}$. Then the values*

$M(B(v_1) \setminus S_1), \dots, M(B(v_N) \setminus S_N)$ can be computed in $O(m + |S_1| + \cdots + |S_N|)$ time in total.

Proof. For every $i \in \{1, \dots, N\}$, let $k_i = |S_i|$. Then we compute the sets $L(v_i, v_i, k_{i+1})$ and $R(v_i, v_i, k_{i+1})$, for every $i \in \{1, \dots, N\}$. According to Proposition 3.4 (and the comments after Algorithm 3), this takes $O(m + |S_1| + \cdots + |S_N|)$ time in total. Then, for every $i \in \{1, \dots, N\}$, we can compute the $L_1(v), \dots, L_{k+1}(v)$ and the $R_1(v), \dots, R_{k+1}(v)$ values, by gathering the higher endpoints of the back-edges in $L(v_i, v_i, k_{i+1}) \cup R(v_i, v_i, k_{i+1})$. Let D_i be the set of the higher endpoints of the back-edges in S_i. Then, by Lemma 3.16 we have $M(B(v_i) \setminus S_i) = nca(\{L_1(v_i), \dots, L_{k+1}(v_i), R_1(v_i), \dots, R_{k+1}(v_i)\} \setminus D_i)$. (We note that the sets $\{L_1(v_i), \dots, L_{k+1}(v_i), R_1(v_i), \dots, R_{k+1}(v_i)\} \setminus D_i$ can be computed in $O(n + |S_1| + \cdots + |S_N|)$ time in total with bucket-sort.) Notice that $nca(\{L_1(v_i), \dots, L_{k+1}(v_i), R_1(v_i), \dots, R_{k+1}(v_i)\} \setminus D_i)$ can be broken up into $O(k_i)$ *nca* queries. Thus, we can compute all values $M(B(v_i) \setminus S_i)$ with the use of $O(k_1 + \cdots + k_N) = O(|S_1| + \cdots + |S_N|)$ *nca* queries. By [40] or [12], we know that there is a linear-time preprocessing of T, so that we can answer a collection of N' *nca* queries in $O(N')$ time in total. We conclude that the values $M(B(v_1) \setminus S_1), \dots, M(B(v_N) \setminus S_N)$ can be computed in $O(m + |S_1| + \cdots + |S_N|)$ time in total. □

3.7 Pointer-machine algorithms for some DFS parameters

In this section we provide alternative linear-time algorithms for computing some parameters such as $M(v)$, $L_1(v)$, $R_1(v)$, and a few others that we introduce here. The characteristic of those alternative algorithms is that they are implementable in the pointer-machine model of computation (as can be easily discerned from the pseudocode), and so they will be useful in establishing the result of Chapter 4, for computing the 4-edge-connected components in linear time in the pointer-machine model. By contrast, the algorithms that we provided in the previous sections utilize the data structure of Gabow and Tarjan [33], which is implemented in the RAM model.

First, we introduce a few more parameters. For every vertex v, we assume that the list of the children of v is sorted in increasing order w.r.t. the *low* point (breaking ties arbitrarily). We call the k-th child of v in this ordering the *lowk* child of v, denoted as $c_k(v)$. (If v has less than k children, then we let $c_k(v) := \bot$.) Thus, we

have $low(c_1(v)) \leq low(c_2(v)) \leq \dots$. For every vertex $v \neq r$, and every $k \geq 1$, we let $M_{lowk}(v)$ denote the nearest common ancestor of the higher endpoints of the back-edges that leap over v and start from a descendant of the $lowk$ child of v. (In other words, $M_{lowk}(v) = M(v, c_k(v))$.) Now let $v \neq r$ be a vertex such that *nextM*$(v) \neq \bot$. Then, if $B($*nextM*$(v)) \neq B(v)$, we have that $B($*nextM*$(v)) \subset B(v)$, and so there is at least one back-edge in $B(v) \setminus B($*nextM*$(v))$. We let $low_M(v)$ denote the lowest lower endpoint of the back-edges in $B(v) \setminus B($*nextM*$(v))$. Let $(x, low_M(v))$ be any back-edge in $B(v) \setminus B($*nextM*$(v))$. Then we let $low_M D(v) = x$ (an arbitrary higher endpoint suffices for our purposes).

In the following, we provide linear-time algorithms for computing the M, $\widetilde{M}$, M_{low1}, M_{low2}, L_1, L_2, R_1, R_2 points of all vertices $v \neq r$, as well as the $(low_M D(v), low_M(v))$-edges of all vertices $v \neq r$ with *nextM*$(v) \neq \bot$.

3.7.1 Computing all $M(v)$

We can compute all $M(v)$ by using the leftmost and the rightmost points $L_1(v)$ and $R_1(v)$. That is, $M(v)$ is the nearest common ancestor of $L_1(v)$ and $R_1(v)$. Thus, if we have all $L_1(v)$ and $R_1(v)$ computed (see Section 3.7.4), then we can use an offline linear-time algorithm for answering *nca* queries in linear time [12].

However, here we will provide a different algorithm for computing all $M(v)$, which is much easier to implement. We refer to Algorithm 4. This is a recursive algorithm that processes the vertices in a bottom-up fashion. For every vertex v that we process, we first determine whether $M(v) = \bot$, which is equivalent to $low(v) \geq v$ (see Line 3). So let us assume that $low(v) < v$. Now, first we check whether $l(v) < v$. If that is the case, then there is a back-edge of the form $(v, l(v))$, and so this is in $B(v)$. Since the higher endpoint of this back-edge is obviously an ancestor of the higher endpoints of all back-edges that leap over v, in this case we set $M(v) \leftarrow v$. Otherwise, we check whether $c_2(v)$ exists, and whether $low(c_2(v)) < v$. If that is the case, then there is a back-edge (x, y) such that x is a descendant of $c_2(v)$ and y is a proper ancestor of v. Furthermore, since $low(c_1(v))$ must also be lower than v in this case, there is a back-edge (x', y') such that x' is a descendant of $c_1(v)$ and y' is a proper ancestor of v. Thus, since both (x, y) and (x', y') are in $B(v)$ and $nca(x, x') = v$, we set again $M(v) \leftarrow v$. Otherwise, if $c_2(v)$ does not exist or it has $low(c_2(v)) \geq v$, then $M(v)$ must be a descendant of $c_1(v)$. More precisely, by Lemma 3.1 we have that $M(v)$

must be a descendant of $M(c_1(v))$. Thus, we follow the same procedure for $M(c_1(v))$. Eventually we will reach $M(v)$. It should be clear that this establishes the correctness of Algorithm 4. Proposition 3.7 shows that it runs in linear time.

Algorithm 4: Compute all $M(v)$, for all vertices $v \neq r$

```
1  for v = n to v = 2 do
2  |   let x ← v
3  |   if low(x) ≥ v then continue
4  |   while M(v) = ⊥ do
5  |   |   if l(x) < v then M(v) ← x, break
6  |   |   if c2(x) ≠ ⊥ and low(c2(x)) < v then
7  |   |   |   M(v) ← x
8  |   |   |   break
9  |   |   end
10 |   |   x ← M(c1(x))
11 |   end
12 end
```

Proposition 3.7. *Algorithm 4 runs in linear time.*

Proof. The only reason that Algorithm 4 may fail to run in linear time is that, when we process a vertex $v \neq r$ in order to compute $M(v)$, we may have to descend to the M value of the *low1* child of v in Line 10. Since this may be repeated $\Omega(n)$ times in order to compute $M(v)$, we may have to perform $\Omega(n^2)$ steps overall during the course of the algorithm. We will show that this is impossible to occur. To do this, we define $S(v)$, for a vertex $v \neq r$, to be the set of all x that we had to descend to in Line 10, during the computation of $M(v)$. Our goal is to bound $\sum_{v \neq r} |S(v)|$. Notice that all vertices in $S(v)$ (if there are any) are descendants of $c_1(v)$, and the greatest vertex in $S(v)$ is $M(v)$. Furthermore, notice that if x is a vertex in $S(v)$, then every vertex in $S(v)$ which is greater than x is a proper descendant of x.

Now let v and v' be two distinct vertices $\neq r$ such that $S(v) \cap S(v') \neq \emptyset$. Since $S(v)$ consists of descendants of v and $S(v')$ consists of descendants of v', we have that v and v' have a common descendant, and therefore they are related as ancestor and descendant. Thus we may assume w.l.o.g. that v' is a proper ancestor of v. Let

us suppose, for the sake of contradiction, that $M(v')$ is not related as ancestor and descendant with v. Let x be a vertex in $S(v) \cap S(v')$. Then x is an ancestor of both $M(v)$ and $M(v')$. Since v is an ancestor of $M(v)$ and $M(v')$ is not related as ancestor and descendant with v, we have that the nearest common ancestor of $M(v)$ and $M(v')$ is a proper ancestor of v. Thus, we have that x is a proper ancestor of v. But this is impossible, since all vertices in $S(v)$ are proper descendants of v. Thus we have that $M(v')$ and v are related as ancestor and descendant. Let us suppose, for the sake of contradiction, that $M(v')$ is a proper ancestor of v. Then all vertices in $S(v')$ are proper ancestors of v (since all of them are ancestors of $M(v')$), and so none if them is in $S(v)$, a contradiction. Thus we have shown that $M(v')$ is a descendant of v, and therefore Lemma 3.1 implies that $M(v')$ is a descendant of $M(v)$.

Now let $x_1, \dots, x_k$ be the vertices in $S(v')$ sorted in increasing order, and let $x_0 = v'$. Since x_0 is a proper ancestor of v, we may consider the greatest index $i \in \{0, \dots, k\}$ such that x_i is a proper ancestor of v. Since $M(v') = x_k$ is a descendant of $M(v)$, and therefore a descendant of v, we have that $i \neq k$, and so x_{i+1} is well-defined. Notice that none of $x_0, \dots, x_i$ is in $S(v)$. Since x_i is a proper ancestor of v, we have that v is a descendant of a child of x_i. Since none of $x_0, \dots, x_i$ is in $S(v)$ and $S(v) \cap S(v') \neq \emptyset$, we have that a $x_j \in \{x_{i+1}, \dots, x_k\}$ is in $S(v)$. Then x_j is a descendant of the *low1* child of x_i, and also a descendant of v. Thus, v cannot be a descendant of any child of x_i other than $c_1(x_i)$.

Let x be a vertex in $S(v) \cap S(v')$. Then x is a descendant of x_{i+1} and an ancestor of $M(v)$. Thus, x_{i+1} is an ancestor of $M(v)$. Since v is also an ancestor of $M(v)$, we have that x_{i+1} is related as ancestor and descendant with v. Since i is the largest index in $\{0, \dots, k\}$ such that x_i is a proper ancestor of v, we have that x_{i+1} is a descendant of v. Since $x_{i+1} = M(c_1(x_i))$ and $c_1(x_i)$ is an ancestor of v and x_{i+1} is a descendant of v, by Lemma 3.1 we have that x_{i+1} is a descendant of $M(v)$. Since x_{i+1} is an ancestor of $M(v)$, we infer that $x_{i+1} = M(v)$. Then, all vertices in $\{x_{i+2}, \dots, x_k\}$ are proper descendants of $M(v)$, and therefore none of them is in $S(v)$. Since also none of $\{x_0, \dots, x_i\}$ is in $S(v)$, we thus have that $S(v) \cap S(v') = \{M(v)\}$.

Thus we have shown that, for any two vertices v and v' with $v > v' \neq r$, if $S(v) \cap S(v')$ is not empty then $S(v) \cap S(v') = \{M(v)\}$. Now we define, for a vertex $v \neq r$, the set $S'(v) = S(v) \setminus \{M(v)\}$. Then we have that $S'(v) \cap S'(v) = \emptyset$ for any two distinct vertices v and v' ($\neq r$), because $S(v) \cap S(v')$ is either $\{M(v)\}$, or $\{M(v')\}$, or $\emptyset$. Thus we have $\sum_{v \neq r} |S(v)| \leq \sum_{v \neq r}(|S'(v)| + 1) < \sum_{v \neq r} |S'(v)| + n < 2n$, and so

Algorithm 4 runs in linear time. □

3.7.2 Computing all $\widetilde{M}(v)$, $M_{low1}(v)$ and $M_{low2}(v)$

The idea for computing all $\widetilde{M}(v)$, $M_{low1}(v)$ and $M_{low2}(v)$, for $v \neq r$, is similar as that for computing all $M(v)$. First, let us discuss the computation of $\widetilde{M}(v)$, which is shown in Algorithm 5. Here, the first thing we do is to check whether the *low1* child c_1 of $M(v)$ exists and has $low(c_1) < v$. If this is not true, then we have $\widetilde{M}(v) = \bot$ (see Line 6). Otherwise, we check whether the *low2* child c_2 of $M(v)$ exists and has $low(c_2) < v$. If this is the case, then we have $\widetilde{M}(v) = M(v)$ (see Line 8). Otherwise, we have to search for $\widetilde{M}(v)$ in $T(c_1)$. Now we apply repeatedly the following procedure, until we reach $\widetilde{M}(v)$. Let $x \neq M(v)$ be a vertex for which we have established that $\widetilde{M}(v)$ is a descendant of x. Then, the first thing we do is to check whether $l(x) < v$. If this is the case, then we have $\widetilde{M}(v) = x$. Otherwise, we check whether the *low2* child c_2' of x exists and has $low(c_2') < v$. If this is the case, then we have $\widetilde{M}(v) = x$. Otherwise, we deduce that $\widetilde{M}(v)$ is a descendant of the *low1* child of x. This idea is captured in Lines 9 to 17. In order to achieve linear time, we have to avoid visiting vertices whose l point and the *low* point of their *low2* child are not low enough to be proper ancestors of the current vertex v that we process. Thus, we use a variable *currentM*$[x]$ to bypass vertices that are provably unable to be the $\widetilde{M}$ point of vertices anymore (since we process the vertices in a bottom-up fashion). Thus, if we have established that $\widetilde{M}(v)$ is a descendant of a vertex x, we can go immediately to *currentM*$[x]$ to continue the search for $\widetilde{M}(v)$. (Initially, we set *currentM*$[x] \leftarrow x$, and we update the *currentM* pointers appropriately.)

We use a similar idea to compute all $M_{low1}(v)$ and $M_{low2}(v)$, and the corresponding procedure is shown in Algorithm 6. The proof of correctness of both Algorithm 5 and Algorithm 6 is given in Proposition 3.8.

Lemma 3.17. *Let v and v' be two vertices such that v' is an ancestor of v with $M(v') = M(v)$. Then, $\widetilde{M}(v')$ (resp. $M_{low1}(v')$, $M_{low2}(v')$), if it is defined, is a descendant of $\widetilde{M}(v)$ (resp. $M_{low1}(v)$, $M_{low2}(v)$).*

Proof. Let v' be an ancestor of v such that $M(v') = M(v)$.

Assume, first, that $\widetilde{M}(v')$ is defined. Then, there exists a back-edge $(x, y) \in B(v')$ such that x is a proper descendant of $M(v')$. Since $M(v') = M(v)$, x is a proper descendant of $M(v)$. Furthermore, since y is a proper ancestor of v', it is also a

Algorithm 5: Compute all $\widetilde{M}(v)$, for all vertices $v \neq r$

```
1  initialize an array currentM with n entries
2  foreach vertex v do currentM[v] ← v
3  for v = n to v = 2 do
4      x ← M(v)
5      c ← low1 child of m
6      if low(c) ≥ v then M̃(v) ← ⊥, continue
7      c′ ← low2 child of m
8      if low(c′) < v then M̃(v) ← x, continue
9      x ← currentM[c]
10     while M̃(v) = ⊥ do
11         if l(x) < v then M̃(v) ← x, break
12         c_1 ← low1 child of x
13         c_2 ← low2 child of x
14         if low(c_2) < v then M̃(v) ← x, break
15         x ← currentM[c_1]
16     end
17     currentM[c] ← x
18 end
```

proper ancestor of v. This shows that $(x, y) \in B(v)$, and $\widetilde{M}(v)$ is an ancestor of x. Due to the generality of (x, y), we conclude that $\widetilde{M}(v)$ is an ancestor of $\widetilde{M}(v')$.

Now assume that $M_{low1}(v')$ is defined. Then, there exists a back-edge $(x, y) \in B(v')$ such that x is a descendant of the *low1* child of $M(v')$. Since $M(v') = M(v)$, x is a descendant of the *low1* child of $M(v)$. Furthermore, since y is a proper ancestor of v', it is also a proper ancestor of v. This shows that $(x, y) \in B(v)$, and $M_{low1}(v)$ is an ancestor of x. Due to the generality of (x, y), we conclude that $M_{low1}(v)$ is an ancestor of $M_{low1}(v')$.

Similarly, we can see that, if $M_{low2}(v')$ is defined, then $M_{low2}(v)$ is an ancestor of $M_{low2}(v')$. □

Proposition 3.8. *Algorithms 5 and 6 compute all* $\widetilde{M}(v)$*,* $M_{low1}(v)$ *and* $M_{low2}(v)$*, for all vertices* $v \neq r$*, in total linear time.*

Proof. We will show that Algorithm 6 correctly computes all $M_{low1}(v)$, for all $v \neq r$,

Algorithm 6: Compute all $M_{low1}(v)$ and $M_{low2}(v)$, for all vertices $v \neq r$

1. initialize an array *currentM* with n entries
2. // Compute all $M_{low1}(v)$
3. **foreach** *vertex* v **do** *currentM*$[v] \leftarrow v$
4. **for** $v = n$ *to* $v = 2$ **do**
5. $\quad x \leftarrow M(v)$
6. $\quad c \leftarrow$ *low1* child of x
7. $\quad$ **if** $low(c) \geq v$ **then** $M_{low1}(v) \leftarrow \perp$, **continue**
8. $\quad x \leftarrow$ *currentM*$[c]$
9. $\quad$ **while** $M_{low1}(v) = \perp$ **do**
10. $\quad\quad$ **if** $l(x) < v$ **then** $M_{low1}(v) \leftarrow x$, **break**
11. $\quad\quad c_1 \leftarrow$ *low1* child of x
12. $\quad\quad c_2 \leftarrow$ *low2* child of x
13. $\quad\quad$ **if** $low(c_2) < v$ **then** $M_{low1}(v) \leftarrow x$, **break**
14. $\quad\quad x \leftarrow$ *currentM*$[c_1]$
15. $\quad$ **end**
16. $\quad$ *currentM*$[c] \leftarrow x$
17. **end**
18. // Compute all $M_{low2}(v)$
19. **foreach** *vertex* v **do** *currentM*$[v] \leftarrow v$
20. **for** $v = n$ *to* $v = 2$ **do**
21. $\quad x \leftarrow M(v)$
22. $\quad c \leftarrow$ *low2* child of x
23. $\quad$ **if** $low(c) \geq v$ **then** $M_{low2}(v) \leftarrow \perp$, **continue**
24. $\quad x \leftarrow$ *currentM*$[c]$
25. $\quad$ **while** $M_{low2}(v) = \perp$ **do**
26. $\quad\quad$ **if** $l(x) < v$ **then** $M_{low2}(v) \leftarrow x$, **break**
27. $\quad\quad c_1 \leftarrow$ *low1* child of x
28. $\quad\quad c_2 \leftarrow$ *low2* child of x
29. $\quad\quad$ **if** $low(c_2) < v$ **then** $M_{low2}(v) \leftarrow x$, **break**
30. $\quad\quad x \leftarrow$ *currentM*$[c_1]$
31. $\quad$ **end**
32. $\quad$ *currentM*$[c] \leftarrow x$
33. **end**

in total linear time. The proofs for the other cases are similar. So let v be a vertex $\neq r$. Since we are interested in the back-edges $(z,w) \in B(v)$ where z is a descendant of the *low1* child c of $M(v)$, we first have to check whether *low*$(c) < v$. If *low*$(c) \geq v$, then there is no such back-edge, and therefore we set $M_{low1}(v) \leftarrow \perp$ (in Line 7). If *low*$(c) < v$, then $M_{low1}(v)$ is defined, and in Line 8 we assign x the value *currentM*$[c]$. We claim that, at that moment, *currentM*$[c]$ is an ancestor of $M_{low1}(v)$, and every *currentM*$[c_1]$ that we will access in the **while** loop in Line 14 is also an ancestor of $M_{low1}(v)$; furthermore, when we reach Line 16, *currentM*$[c]$ is assigned $M_{low1}(v)$. We will see this inductively. Suppose, then, that this was the case for every vertex $v' > v$, and let us see what happens when we process v. Let c be the *low1* child of $M(v)$. Initially, *currentM*$[c]$ was set to be c. Now, if *currentM*$[c]$ is still c, then $M_{low1}(v)$ is a descendant of c (by definition). Otherwise, due to the inductive hypothesis, *currentM*$[c]$ had been assigned $M_{low1}(v')$ during the processing of a vertex $v' > v$ with $M(v') = M(v)$. This implies that v' is a descendant of v, and therefore by Lemma 3.17 we have that $M_{low1}(v')$ is an ancestor of $M_{low1}(v)$. In any case, then, we have that $x =$ *currentM*$[c]$ is an ancestor of $M_{low1}(v)$.

Now we enter the **while** loop in Line 9. If either $l(x) < v$ or *low*$(c_2) < v$, where c_2 is the *low2* child of x, we have that $M_{low1}(v)$ is an ancestor of x. Since x is also an ancestor of $M_{low1}(v)$, we correctly set $M_{low1}(v) \leftarrow x$ (in Lines 10 or 13). Otherwise, we have that $M_{low1}(v)$ is a descendant of the *low1* child c_1 of x. Now, due to the inductive hypothesis, *currentM*$[c_1]$ is either c_1 or $M_{low1}(v')$ for a vertex $v' > v$ with $M(v') = x$. In the first case we obviously have that *currentM*$[c_1]$ is an ancestor of $M_{low1}(v)$. Now assume that the second case is true, and let (z,w) be a back-edge with z a descendant of c_1 and w a proper ancestor of v. Then, since $v' > v$ and v, v' have z as a common descendant, we have that v is ancestor of v', and therefore w is a proper ancestor of v'. This shows that z is a descendant of $M_{low1}(v')$. Thus, due to the generality of (z,w), we have that $M_{low1}(v)$ is a descendant of $M_{low1}(v')$. In any case, then, we have that *currentM*$[c_1]$ is an ancestor of $M_{low1}(v)$. Thus we set $x \leftarrow$ *currentM*$[c_1]$ and we continue the **while** loop, until we have that $x = M_{low1}(v)$, in which case we will set *currentM*$[c] \leftarrow x$ in Line 16. Thus we have proved that Algorithm 6 correctly computes $M_{low1}(v)$, for every vertex $v \neq r$, and that, during the processing of a vertex v, every *currentM*$[c]$ that we access is an ancestor of $M_{low1}(v)$ (until, in Line 16, we assign *currentM*$[c]$ to $M_{low1}(v)$).

Now, to prove the linear time-complexity, let $S(v) = \{m_1, \ldots, m_k\}$, ordered in-

creasingly, denote the (possibly empty) set of all vertices that we had to descend to before leaving the **while** loop in Lines 9-15. (Thus, if $k \geq 1$, then $m_k = M_{low1}(v)$.) In other words, $S(v)$ contains all vertices that were assigned to the variable x in Line 14. We will show that Algorithm 6 runs in linear time, by showing that, for every two vertices v and v', $v \neq v'$ implies that $S(v) \cap S(v') \subseteq \{M_{low1}(v)\}$, where we have $S(v) \cap S(v') = \{M_{low1}(v)\}$ only if $M_{low1}(v) = M_{low1}(v')$. Of course, it is definitely the case that $S(v) \cap S(v') = \emptyset$ if v and v' are not related as ancestor and descendant, since the **while** loop descends to descendants of the vertex under processing. So let v' be a proper ancestor of v. If $M_{low1}(v')$ is not a descendant of the *low1* child c of $M(v)$, then we obviously have $S(v) \cap S(v') = \emptyset$ (since $S(v)$ consists of descendants of c, but the **while** loop during the computation of $M_{low1}(v')$ will not descend to the subtree of c). Thus we may assume that $M_{low1}(v')$ is a descendant of c. Now, let $S(v') = \{m_1, \dots, m_k\}$ and let $m_0 =$ *currentM*$[c']$ (during the processing of v'), where c' is the *low1* child of $M(v')$. We will show that every m_i, for $i \in \{1, \dots, k\}$, is either an ancestor of $M(v)$ or a descendant of $M_{low1}(v)$. (This obviously implies that $S(v') \cap S(v) \subseteq \{M_{low1}(v)\}$.)

First, observe that $M(v')$ is either an ancestor of $M(v)$ or a descendant of $M_{low1}(v)$. To see this, suppose that $M(v')$ is not an ancestor of $M(v)$. Since $M_{low1}(v')$ is a descendant of c, there is at least one back-edge (x, y) in $B(v')$ with x a descendant of c. Then, since y is a proper ancestor of v' and v' is a proper ancestor of v, we have that (x, y) is in $B(v)$, and therefore x is a descendant of $M_{low1}(v)$. Now let (x', y') be a back-edge in $B(v')$. If x' is a descendant of a vertex in $T(c, v']$, but not a descendant of c, then the nearest common ancestor of x and x' is in $T[p(c), v'] = T[M(v), v']$, and therefore $M(v')$ is an ancestor of $M(v)$, contradicting our supposition. Thus, x' is a descendant of c. Furthermore, y' is a proper ancestor of v, and therefore $(x', y') \in B(v)$. Thus, x' is a descendant of $M_{low1}(v)$. Due to the generality of $(x', y') \in B(v')$, we conclude that $M(v')$ is a descendant of $M_{low1}(v)$. Thus we have shown that $M(v')$ is either an ancestor of $M(v)$ or a descendant of $M_{low1}(v)$.

Now, if $M(v')$ is a descendant of $M_{low1}(v)$, then we obviously have $S(v) \cap S(v') = \emptyset$ (because the vertices in $S(v')$ are proper descendants of $M(v')$, but the vertices in $S(v)$ are ancestors of $M_{low1}(v)$). Let us assume, then, that $M(v')$ is an ancestor of $M(v)$. If $M(v')$ coincides with $M(v)$, then $c' = c$, and so m_0 coincides with *currentM*$[c]$, which is a descendant of $M_{low1}(v)$ (since $M_{low1}(v)$ has already been calculated), and therefore every m_i, for every $i \in \{1, \dots, k\}$, is a proper descendant of $M_{low1}(v)$ (since m_1, if it

exists, is a proper descendant of m_0), and so we have $S(v')\cap S(v)=\emptyset$. So let us assume that $M(v')$ is a proper ancestor of $M(v)$. Then, c' is an ancestor of $M(v)$. Suppose that m_0 is not an ancestor of $M(v)$. This means that *currentM*$[c'] \neq c'$, and therefore there is a vertex $\widetilde{v} > v'$ with $M(\widetilde{v}) = M(v')$ and $M_{low1}(\widetilde{v}) =$ *currentM*$[c']$. Furthermore, since m_0 is not an ancestor of $M(v)$, it must be a descendant of c. Now, since v' is an ancestor of v and $M(v')$ is a proper ancestor of $M(v)$, Lemma 3.1 implies that $M(v')$ is a proper ancestor of v. Since $M(v') = M(\widetilde{v})$, this implies that $M(\widetilde{v})$ is a proper ancestor of v, and therefore $\widetilde{v}$ is a proper ancestor of v. Now let (x, y) be a back-edge in $B(\widetilde{v})$ such that x is a descendant of $M_{low1}(\widetilde{v}) =$ *currentM*$[c'] = m_0$. Then, since m_0 is a descendant of c, x is also descendant of c. Furthermore, since $\widetilde{v}$ is an ancestor of v, y is a proper ancestor of v. This shows that x is a descendant of $M_{low1}(v)$. Due to the generality of (x, y), we conclude that $M_{low1}(\widetilde{v})$ is a descendant of $M_{low1}(v)$. Thus we have shown that m_0 is either an ancestor of $M(v)$ or a descendant of $M_{low1}(v)$.

Now let us assume that m_i is either an ancestor of $M(v)$ or a descendant of $M_{low1}(v)$, for some $i \in \{0, \dots, k-1\}$. We will prove that the same is true for m_{i+1}. If m_i is a descendant of $M_{low1}(v)$, then the same is true for m_{i+1}. Let us assume, then, that m_i is an ancestor of $M(v)$. Now we have that $m_{i+1} =$ *currentM*$[c_1]$, where c_1 is the *low1* child of m_i. If $m_i = M(v)$, then we have $c_1 = c$, and therefore *currentM*$[c_1] =$ *currentM*$[c]$ is a descendant of $M_{low1}(v)$ (since $M_{low1}(v)$ has already been computed). Suppose, then, that m_i is a proper ancestor of $M(v)$. Then, c_1 is an ancestor of $M(v)$. If *currentM*$[c_1] = c_1$, we obviously have that *currentM*$[c_1]$ is an ancestor of $M(v)$. Otherwise, if *currentM*$[c_1] \neq c_1$, there is a vertex $\widetilde{v}$ such that $M(\widetilde{v}) = m_i$ and *currentM*$[c_1] = M_{low1}(\widetilde{v})$. Assume, first, that $\widetilde{v}$ is an ancestor of v. Suppose that $M_{low1}(\widetilde{v})$ is not an ancestor of $M(v)$. Then it must be a descendant of c (because all m_i, for $i \in \{0, \dots, k-1\}$ are ancestors of $M_{low1}(v')$, and we have assumed that $M_{low1}(v')$ is a descendant of c). Let (x, y) be a back-edge in $B(\widetilde{v})$ with x a descendant of $M_{low1}(\widetilde{v})$. Then x is a descendant of c. Furthermore, y is a proper ancestor of $\widetilde{v}$, and therefore a proper ancestor of v. This shows that x is a descendant of $M_{low1}(v)$. Due to the generality of (x, y), we conclude that $M_{low1}(\widetilde{v})$ is a descendant of $M_{low1}(v)$. Thus, if $\widetilde{v}$ is an ancestor of v, $M_{low1}(\widetilde{v})$ is either an ancestor of $M(v)$ or a descendant of $M_{low1}(v)$. Suppose, now, that $\widetilde{v}$ is a descendant of v. Let (x, y) be a back-edge in $B(v)$. Then, x is a descendant of $M(v)$, and therefore a descendant of c_1. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of $\widetilde{v}$. This shows that x is a descendant of $M_{low1}(\widetilde{v})$. Due to the generality of (x, y), we conclude that $M(v)$ is a descendant

of $M_{low1}(\widetilde{v})$. In any case, then, m_{i+1} is either an ancestor of $M(v)$ or a descendant of $M_{low1}(v)$. Thus, $S(v) \cap S(v') \subseteq \{M_{low1}(v)\}$ is established. $\square$

3.7.3 Computing all $low_M(v)$ and $low_M D(v)$

In order to compute all $low_M(v)$ and $low_M D(v)$ efficiently, we process the vertices in a bottom-up fashion. For every vertex v that we process, we check whether $u = prevM(v) \neq \perp$. If that is the case, then $low_M(u)$ is defined and it lies on the simple tree-path $T(u, v]$. Thus we descend the path $T(u, v]$, starting from v, following the *low1* children of the vertices on the path; for every vertex y that we encounter we check whether there exists a back-edge (x, y) with $x \in T(M(v))$. The first y with this property is $low_M(u)$, and we set $(low_M D(u), low_M(u)) \leftarrow (x, y)$.

To achieve linear running time, we let $In[y]$ denote the list of all vertices x for which there exists a back-edge (x, y). Furthermore, we have the elements of $In[y]$ sorted in increasing order (this can be done easily in linear time with bucket sort). When we process a vertex y as we descend $T(u, v]$, during the processing of v, we traverse $In[y]$ starting from the element we accessed the last time we traversed $In[y]$ (or, if this is the first time we traverse $In[y]$, from the first element of $In[y]$). Thus, we need a variable $currentBackEdge[y]$ to store the element of $In[y]$ we accessed the last time we traversed $In[y]$. Now, for every $x \in In[y]$ that we meet, we check whether $x \in T(M(v))$. If that is the case, then we set $(low_M D(u), low_M(u)) \leftarrow (x, y)$; otherwise, we move to the next element of $In[y]$. If we reach the end of $In[y]$, then we descend the path $T(u, v]$ by moving to the *low1* child of y. In fact, if $prevM(c_1(y)) \neq \perp$, then we may descend immediately to $low_M(prevM(c_1(y)))$. This ensures that $In[y]$ will not be accessed again. Algorithm 7 shows how to compute all pairs $(low_M D(v), low_M(v))$, for all vertices v with $nextM(v) \neq \perp$, in total linear time. Proposition 3.9 establishes the correctness and the linearity of this algorithm.

Proposition 3.9. *Algorithm 7 correctly computes all pairs $(low_M D(v), low_M(v))$, for all vertices v with $nextM(v) \neq \perp$, in total linear time.*

Proof. Let v be a vertex with $prevM(v) = u \neq \perp$. We will prove inductively that $(low_M D(u), low_M(u))$ will be computed correctly, and that $low_M D(u)$ will be the lowest vertex which is a descendant of $M(u)$ such that $(low_M D(u), low_M(u))$ is a back-edge. So let us assume that we have run Algorithm 7 and we have correctly computed all pairs $(low_M D(u'), low_M(u'))$, for all vertices $v' > v$ with $prevM(v') = u' \neq \perp$, and $low_M D(u')$ is

Algorithm 7: Compute $(low_M D(v), low_M(v))$, for every vertex $v \neq r$ with $nextM(v) \neq \bot$

```
1  calculate In[y], for every vertex y, and have its elements sorted in increasing
     order
2  foreach vertex y do currentBackEdge[y] ← first element of In[y]
3  foreach vertex v do low_M(v) ← ⊥
4  for v ← n to v ← 2 do
5  |   if prevM(v) = ⊥ then continue
6  |   u ← prevM(v)
7  |   y ← v
8  |   while low_M(u) = ⊥ do
9  |   |   while currentBackEdge[y] ≠ ⊥ do
10 |   |   |   x ← currentBackEdge[y]
11 |   |   |   if x < M(u) then
12 |   |   |   |   currentBackEdge[y] ← next element of In[y]
13 |   |   |   end
14 |   |   |   else
15 |   |   |   |   if x < M(u) + ND(M(u)) then
16 |   |   |   |   |   (low_M D(u), low_M(u)) ← (x, y)
17 |   |   |   |   end
18 |   |   |   |   break
19 |   |   |   end
20 |   |   end
21 |   |   if low_M(u) = ⊥ then
22 |   |   |   if prevM(c_1(y)) = ⊥ then y ← c_1(y)
23 |   |   |   else y ← low_M(prevM(c_1(y)))
24 |   |   end
25 |   end
26 end
```

the lowest vertex in $T(M(u'))$ such that $(low_M D(u'), low_M(u'))$ is a back-edge. Suppose also that we have currently descended the path $T(u, v]$, we have reached y, and $low_M(v) \geq y$.

Let us assume, first, that $low_M(v) = y$, and let (x, y) be the back-edge such that $x \in T(M(v))$ and x is minimal with this property. The **while** loop in line 9 will search the list of incoming back-edges to y, starting from *currentBackEdge*$[y]$. If *currentBackEdge*$[y]$ is the first element of *In*$[y]$, then is it certainly true that x will be found. Otherwise, let $x' =$ *currentBackEdge*$[y]$. Due to the inductive hypothesis, we have that $(x', y) = (low_M D(u'), low_M(u'))$, for a vertex u' with *nextM*$(u') = v' > v$. Then, y is in $T(u', v']$, but also in $T(u, v]$, and thus it is a common descendant of v and v'. This means that v and v' are related as ancestor and descendant. In particular, since $v' > v$, we have that v is an ancestor of v'. Furthermore, since y is an ancestor of u, it is also an ancestor of $M(u) = M(v)$; therefore, since v' is an ancestor of y, it is also an ancestor of $M(v)$. Since v is an ancestor of v', this implies that $M(v')$ is an ancestor of $M(v)$. Since $M(v') = M(u')$ and $M(v) = M(u)$, we thus have that $M(u')$ is an ancestor of $M(u)$, and therefore $M(u') \leq M(u)$. Thus, since x' is the lowest descendant of $M(u')$ such that (x', y) is a back-edge, and x is the lowest descendant of $M(u)$ such that (x, y) is a back-edge, we have $x' \leq x$. This shows that x will be accessed during the **while** loop in line 9.

Now let us assume that $low_M(v) \neq y$. This means that $low_M(u)$ is greater than y, and we have to descend the path $T(u, y)$ in order to find it. First, let c be the child of y in the direction of u. Then we have $low(c) < v$ (since $M(v) = M(u)$ is a descendant of u, and therefore a descendant of c, and we have $low(M(v)) < v$). If there was another child c' of y with $low(c') < v$, this would imply that $M(v) = y$, which is absurd, since y is a proper ancestor of u, and therefore a proper ancestor of $M(u) = M(v)$. This means that c is the *low1* child of v, and thus we may descend to $c_1(y) = y'$. Now we have $low_M(u) \geq y'$. If *prevM*$(y') = \bot$, then we simply traverse the list of incoming back-edges to y', in line 9, and repeat the same process. Otherwise, let $u' =$ *prevM*(y'). Due to the inductive hypothesis, we know that $low_M(u')$ has been computed correctly. Since y' is an ancestor of u, it is also an ancestor of $M(u) = M(v)$. Furthermore, y' is a descendant of v. Thus, $M(y')$ is an ancestor of $M(v)$, and therefore $M(u')$ is an ancestor of $M(u)$ (since $M(y') = M(u')$ and $M(v) = M(u)$). This means that u' is an ancestor of $M(u)$. Now we see that $low_M(u)$ lies on $T(u, low_M(u')]$. (For otherwise, $(low_M D(u), low_M(u))$ would be a back-edge in $B(u')$ with $low_M(u) \geq y' =$ *nextM*(u') and $low_M(u) < low_M(u')$, contradicting the minimality of $low_M(u')$). Thus we may descend immediately to $low_M(u')$. Then we traverse the list of incoming back-edges to $low_M(u')$, in line 9, and repeat the same process. Eventually we will reach $low_M(u)$ and have it

computed correctly. It should be clear that no vertex on the path $T(low_M(u), v)$ will be traversed again, and this ensures the linear complexity of Algorithm 7. $\square$

3.7.4 Computing all $L_1(v)$, $L_2(v)$, $R_1(v)$ and $R_2(v)$

Here we will show how to compute the first and the second leftmost and rightmost points of every vertex $v \neq r$ in linear time. Let $(x_1, y_1), \ldots, (x_k, y_k)$ be the back-edges in $B(v)$ sorted in increasing order w.r.t. their higher endpoint. Then $L_1(v) = x_1$ and $L_2(v) = x_2$. The idea to compute x_1 and x_2 is pretty straightforward: we visit all vertices in $T(v)$ in increasing order, and for every $x \in T(v)$ that we visit we check whether there exists a y in the adjacency list of x such that (x, y) is a back-edge leaping over v. The first such x that we find is $L_1(v)$; the second one is $L_2(v)$, and then we stop the search (for v).

Now, to derive a linear-time algorithm from this idea, we only have to make sure that, during the search for $L_1(v)$ and $L_2(v)$, we visit only the vertices $x \in T(v)$ that may provide a leaping back-edge over v, and we ignore all other vertices in $T(v)$ that provably cannot. Thus, we put all vertices, in increasing order, in a list L (signifying that they are available for search), and we process all vertices $v \neq r$ in a bottom-up fashion. For every vertex v that we process, we visit all $x \in T(v)$, in increasing order, that are still available in L, and for every x that we visit we check whether we can find a leaping back-edge over v in the adjacency list of x. If that is the case, then we set $L_1(v) \leftarrow x$ or $L_2(v) \leftarrow x$ (depending on whether $L_1(v)$ has been already computed). Otherwise, if x provided no leaping back-edge, then it is deleted from L. In the meantime, we also remove all vertices y in the adjacency list of x that do not form a leaping back-edge (x, y) over v. Algorithm 8 is an implementation of this idea. *Out*$[x]$ is a doubly linked list containing all y in the adjacency list of x such that (x, y) is a back-edge.

Algorithm 8: Compute $L_1(v), L_2(v)$, for all vertices $v \neq r$

```
1  join all vertices in a doubly linked list L, in increasing order
2  for v ← n to 2 do
3      x ← v
4      t ← 1
5      while x is a descendant of v and t ≠ 3 do
6          isEmpty ← true
7          foreach y ∈ Out[x] do
8              if y < v then
9                  isEmpty ← false
10                 L_t(v) ← x
11                 t ← t + 1
12             end
13             else
14                 delete y from Out[x]
15             end
16             if t = 3 then break
17         end
18         next ← next of x in L
19         if isEmpty then delete x from L
20         x ← next
21     end
22 end
```

Theorem 3.1. *Algorithm 8 correctly computes $L_1(v)$ and $L_2(v)$, for every vertex $v \neq r$. Its time-complexity is linear to the size of the graph.*

Proof. Correctness follows easily if we observe that whenever we process a vertex v, all back-edges that leap over v are still available for search. (This is because we process the vertices in a bottom-up fashion; thus, whenever we remove a back-edge (x, y), during the processing of a vertex u, this is because (x, y) does not leap over u, and so it does not leap over v either, for any vertex $v < u$.) In particular, the leftmost back-edges of v are still accessible from the lists L and *Out* (and so are the leftmost points). The linear complexity is obvious, because whenever a back-edge is accessed it is either deleted from the list in which it is contained, or it corresponds to an external

point. Furthermore, whenever a vertex $x \in L$ is accessed it is either deleted from L or it corresponds to an external point. □

For computing $R_1(v)$ and $R_2(v)$ for all vertices $v \neq r$, we use an algorithm similar to Algorithm 8. The only difference is that, for a vertex $v \neq r$, we begin the search for $R_1(v)$ and $R_2(v)$ from the greatest vertex in $T(v)$ that still exists in L, and we traverse L in the reverse direction. Thus, in Line 3 we set "$x \leftarrow R_1(c)$", where c is the greatest child of x (or $x \leftarrow v$, if x is childless), and we replace Line 18 with "*next* $\leftarrow$ previous of x in L". The time complexity does not change. The proof of correctness is essentially contained in that of Theorem 3.1.

3.8 Two lemmata concerning paths

Lemma 3.18. *Let u and v be two vertices, and let P be a path in G from u to v. Then, P passes from an ancestor of nca$\{u,v\}$.*

Proof. Let w be the lowest vertex that is used by P. Let us suppose, for the sake of contradiction, that w is not an ancestor of *nca*$\{u,v\}$. Then, either w is not an ancestor of u, or w is not an ancestor of v. Let us assume w.l.o.g. that w is not an ancestor of u. Since u is not a descendant of w, we may consider the first predecessor z of w in P that is not a descendant of w. Let z' be the successor of z in P. Then, we have that P uses the edge (z,z'), and z' is a descendant of w. Let us suppose, first, that (z,z') is a tree-edge. Then, since z' is a descendant of w, but z is not, we have that z cannot be a child of z'. Thus, z is the parent of z'. But since z' is a descendant of w and its parent is not, we have that $z' = w$, and therefore z is the parent of w. But this contradicts the minimality of w. Thus, we have that (z,z') is a back-edge. Then, since z' is a descendant of w, but z is not, we have that z cannot be a descendant of z', and therefore it is an ancestor of z'. Then, z' is a common descendant of w and z, and therefore w and z are related as ancestor and descendant. But since z is not a descendant of w, it must be a proper ancestor of w, and therefore $z < w$. This again contradicts the minimality of w. Thus, our initial supposition cannot be true, and therefore w is an ancestor of *nca*$\{u,v\}$. □

Lemma 3.19. *Let u be a vertex and let v be a proper ancestor of u. Let P be a path that starts from a descendant of u and ends in v. Then, the first occurrence of an edge that is*

used by P and leads outside of the subtree of u is either a back-edge that leaps over u or the tree-edge $(u, p(u))$.

Proof. Let (x, y) be the first occurrence of an edge that is used by P and leads outside of the subtree of u. We may assume w.l.o.g. that x is a descendant of u and y is not a descendant of u. Then, we have that y is not a descendant of x, because otherwise it would be a descendant of u. Suppose first that (x, y) is a tree-edge. Then, since y is not a descendant of x, it must be the parent of x. Thus, since x is a descendant of u but its parent is not, we have that $x = u$ and therefore $y = p(u)$. Now let us suppose that (x, y) is a back-edge. Then, since y is not a descendant of x, it must be a proper ancestor of x. Thus, x is a common descendant of u and y, and therefore u and y are related as ancestor and descendant. Then, since y is not a descendant of u, we have that y is a proper ancestor of u. This shows that (x, y) is a back-edge that leaps over u. □

3.9 An oracle for back-edge queries

Our goal in this section is to prove the following.

Lemma 3.20. *Let T be a DFS-tree of a connected graph G. We can construct in linear time a data structure of size $O(n)$ that we can use in order to answer in constant time queries of the form: given three vertices u, v, w, such that u is a proper descendant of v, and v is a proper descendant of w, is there a back-edge $(x, y) \in B(v) \setminus B(u)$ such that $y \leq w$?*

Proof. First we compute the *low* points of all vertices. This takes linear time. Then we compute, for every vertex v that is not a leaf, a child $c(v)$ of v that has the lowest *low* point among all the children of v (breaking ties arbitrarily). We call this the *low* child of v. Then, starting from any vertex v that is either r or a vertex that is not the *low* child of its parent, we consider the path that starts from v and ends in a leaf by following the *low* children. In other words, this is the path $v, c(v), c(c(v)), \dots$. Notice that every vertex v of G belongs to precisely one such path. We call this the *low* path that contains v, and we maintain a pointer from v to the *low* path that contains it. Furthermore, we consider those paths indexed, starting from their lowest vertex. In other words, if $v, c(v), c(c(v)), \dots$ is a *low* path, then v has index 1, $c(v)$ has index 2, and so on, on this path. We also maintain, for every vertex v, a pointer to the index

of v on the *low* path that contains it.

Now, for every vertex v, we compute a value $m(v)$ that is defined as follows. If v has less than two children, then $m(v) := l(v)$. Otherwise, let $c_1, \ldots, c_k$ be the list of the children of v, excluding $c(v)$. Then, $m(v) := \mathit{min}\{l(v), \mathit{low}(c_1), \ldots, \mathit{low}(c_k)\}$. Notice that the m values of all vertices can be easily computed in total linear time. Now, for every *low* path, we initialize a data structure for answering range-minimum queries w.r.t. the m values. (We consider a *low* path as an array, corresponding to the indexes of its vertices.) More precisely, for every *low* path P we initialize a range-minimum query data structure RMQ_P. We can use RMQ_P in order to answer queries of the form: given two vertices u and v on P with indices i and j, respectively, such that $i \leq j$, return the minimum of $m(u_1), \ldots, m(u_{j-i+1})$, where $u_1, \ldots, u_{j-i+1}$ is the set of the vertices on P with indices $i, i+1, \ldots, j$. Using the RMQ data structure described, e.g., in [9], the initialization of all those data structures takes $O(n)$ time in total (because the total size of the *low* paths is $O(n)$), and every range-minimum query on every such data structure can be answered in $O(1)$ worst-case time. This completes the description of the data structure and its construction.

Now let u, v, w be three vertices such that u is a proper descendant of v, and v is a proper descendant of w. We will show how we can determine in constant time whether there is a back-edge $(x, y) \in B(v) \setminus B(u)$ such that $y \leq w$. First, suppose that $\mathit{low}(v) > w$. Then we know that there is no back-edge $(x, y) \in B(v)$ such that $y \leq w$, and we are done. So let us assume that $\mathit{low}(v) \leq w$. If $l(v) \leq w$, then $(v, l(v))$ is a back-edge in $B(v) \setminus B(u)$ such that $l(v) \leq w$, and we are done. So let us assume that $l(v) > w$.

First, suppose that u does not belong to the same *low* path as v. Then we claim that there is a proper descendant c' of v on the *low* path P that contains v such that $l(c') \leq w$. To see this, let us assume the contrary. Now, since $l(v) > w$ and $\mathit{low}(v) \leq w$, we have that $\mathit{low}(c(v)) \leq w$ (because $c(v)$ has the lowest *low* point among all the children of v). Then, by assumption, we have $l(c(v)) > w$. Thus, since $\mathit{low}(c(v)) \leq w$, we have that $c(c(v))$ exists, and $\mathit{low}(c(c(v))) \leq w$. Then, again by assumption, we have $l(c(c(v))) > w$. Therefore, since $\mathit{low}(c(c(v))) \leq w$, we have that $c(c(c(v)))$ exists, and $\mathit{low}(c(c(c(v)))) \leq w$. We can see that this process must continue endlessly, in contradiction to the fact that the graph contains a finite number of vertices. Thus, there is indeed a proper descendant c' of v on P such that $l(c') \leq w$. Then we can see that $(c', l(c'))$ is a back-edge in $B(v) \setminus B(u)$ such that $l(c') \leq w$, and we are done.

So let us assume that u and v belong to the same *low* path P. Let i be the index of v on P, and let j be the index of $p(u)$ on P. (We note that $i \leq j$.) Let m' be the answer to the range-minimum query on RMQ_P on the range $[i, j]$. Then we claim that there is a back-edge $(x, y) \in B(v) \setminus B(u)$ such that $y \leq w$ if and only if $m' \leq w$. So let us suppose, first, that there is a back-edge $(x, y) \in B(v) \setminus B(u)$ such that $y \leq w$. Then x is a descendant of v, but not a descendant of u. Now let z be the maximum vertex on $T[v, p(u)]$ (i.e, the one closest to $p(u)$) such that x is a descendant of z. Then, we have that either $x = z$, or x is a descendant of a child c' of z such that $c' \neq c(z)$ (since all the vertices on $T[v, p(u)]$ are part of the *low* path that contains u and v). If $x = z$, then we have that $l(z) \leq y \leq w$, and therefore $m(z) \leq w$. Therefore, since $m' \leq m(z)$, we have $m' \leq w$, as desired. Otherwise, suppose that x is a descendant of a child c' of z such that $c' \neq c(z)$. Then, we have that $(x, y) \in B(c')$, and therefore $low(c') \leq y \leq w$. Then, since $m(z) \leq low(c')$, we have that $m(z) \leq w$. And since $m' \leq m(z)$, this shows that $m' \leq w$.

Conversely, let us suppose that $m' \leq w$. There is a vertex z on the tree-path $T[v, p(u)]$ such that $m' = m(z)$. Then, we have that either $m' = l(z)$, or there is a child c' of z, with $c' \neq c(z)$, such that $low(c') = m'$. If $m' = l(z)$, then, since $m' \leq w$ and w is a proper ancestor of v, we have that there is a back-edge $(z, l(z))$. Then we have that $(z, l(z)) \in B(v)$, but $(z, l(z)) \notin B(u)$ (since z is a proper ancestor of u). Thus, $(z, l(z))$ is the desired back-edge. Otherwise, suppose that there is a child c' of z, with $c' \neq c(z)$, such that $low(c') = m'$. Then, there is a back-edge $(x, y) \in B(c')$ such that $y = low(c')$. Since $low(c') = m' \leq w$, we can see that $(x, y) \in B(v)$. But since c' and $c(z)$ are two different children of z, we have that x is cannot be a descendant of u (because u is a descendant of $c(z)$ and x is a descendant of c'), and therefore we have $(x, y) \notin B(u)$. Thus, (x, y) is a back-edge in $B(v) \setminus B(u)$ such that $y \leq w$.

This concludes the method by which we can determine in constant time whether there exists a back-edge $(x, y) \in B(v) \setminus B(u)$ such that $y \leq w$. This process is shown in Algorithm 9. $\square$

Algorithm 9: Determine whether there is a back-edge $(x, y) \in B(v) \setminus B(u)$ such that $y \leq w$, where u is a proper descendant of v, and v is a proper descendant of w

1. **if** $low(v) > w$ **then** **return false**
2. **if** $l(v) \leq w$ **then** **return true**
3. **if** *u and v do not belong to the same low path* **then** **return true**
4. let P be the *low* path that contains u and v
5. let i be the index of v on P, and let j be the index of $p(u)$ on P
6. let m' be the answer to the range-minimum query on RMQ_R on the range $[i, j]$
7. **if** $m' \leq w$ **then** **return true**
8. **return false**

3.10 Segments of vertices that have the same *high* point

Throughout this section, we assume that G is a 3-edge-connected graph. According to Proposition 3.1, this implies that $|B(v)| > 2$ for every vertex $v \neq r$, and therefore the $high_1$ and $high_2$ points of v are defined.

Let x be a vertex of G, and let $H(x)$ be the list of all vertices $v \neq r$ such that $high(v) = x$, sorted in decreasing order. For a vertex $v \in H(x)$, we let $S(v)$ denote the segment of $H(x)$ that contains v and is maximal w.r.t. the property that its elements are related as ancestor and descendant. The collection of those segments constitutes a partition of $H(x)$, as shown in the following.

Lemma 3.21. *Let x be a vertex. Then, the collection $\mathcal{S}$ of all segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant is a partition of $H(x)$.*

Proof. Every vertex $v \in H(x)$ is contained in $S(v) \in \mathcal{S}$, and therefore the collection of all segments in $\mathcal{S}$ covers $H(x)$. Now let S and S' be two distinct segments in $\mathcal{S}$. Let us suppose, for the sake of contradiction, that $S \cap S' \neq \emptyset$. Then there is a vertex $z \in S \cap S'$.

We will show that all vertices in $S \cup S'$ are related as ancestor and descendant. So let u and u' be two vertices in $S \cup S'$. If both u and u' are either in S or in S', then we have that u and u' are related as ancestor and descendant. So let assume w.l.o.g. that $u \in S$ and $u' \in S'$. Since $u, z \in S$, we have that u and z are related as ancestor

and descendant. Also, since $u', z \in S'$, we have that u' and z are related as ancestor and descendant. Thus, we have the following cases to consider: either (1) both u and u' are ancestors of z, or (2) one of u and u' is an ancestor of z, and the other is a descendant of z, or (3) both u and u' are descendants of z.

In case (1), we have that z is a common descendant of u and u', and therefore u and u' are related as ancestor and descendant. In case (2), we may assume w.l.o.g. that u is an ancestor of z, and u' is a descendant of z. Then, we obviously have that u is an ancestor of u' (due to the transitivity of the ancestry relation). So let us consider case (3). This implies that $u \geq z$ and $u' \geq z$. If $u \geq u'$, then we have $u \geq u' \geq z$. Then, since $H(x)$ is sorted in decreasing order and S is a segment of $H(x)$ and $u, z \in S$, we have $u' \in S$. Then, $u \geq u'$ implies that u' is an ancestor of u. Otherwise, supppose that $u < u'$. Then we have $u' > u \geq z$. Then, since $H(x)$ is sorted in decreasing order and S' is a segment of $H(x)$ and $u', z \in S'$, we have $u \in S'$. Then, $u < u'$ implies that u is a proper ancestor of u'. Thus, in any case we have shown that u and u' are related as ancestor and descendant.

Now we will show that $S \cup S'$ is a segment of $H(x)$. So let us suppose, for the sake of contradiction, that $S \cup S'$ is not a segment of $H(x)$. Since $H(x)$ is sorted in decreasing order, this means that there are two vertices u and u' in $S \cup S'$, with $u > u'$, and a vertex $w \in H(x)$, such that $u > w > u'$ and $w \notin S \cup S'$. Notice that we cannot have that both u and u' are either in S or in S', because S and S' are segments of $H(x)$, and therefore $u > w > u'$ implies that $w \in S$ or $w \in S'$, respectively. Thus, we may assume w.l.o.g. that $u \in S$ and $u' \in S'$. Since u and z are in S, we have that u and z are related as ancestor and descendant. First, let us suppose that z is a descendant of u. This implies that $z \geq u$. Thus, we have $z \geq u > w > u'$. Since S' is a segment of $H(x)$ that contains both z and u', this implies that $w \in S'$, a contradiction. Thus, we have that z is a proper ancestor of u, and therefore $u > z$. Since u' and z are in S', we have that u' and z are related as ancestor and descendant. Let us suppose that u' is a descendant of z. This implies that $u' \geq z$. Then, we have $u > w > u' \geq z$. Since S is a segment of $H(x)$ that contains both u and z, this implies that $w \in S$, a contradiction. Thus, we have that u' is a proper ancestor of z, and therefore $z > u'$. Thus, since $u > w > u'$ and $u > z > u'$, we have that either $u > w \geq z$ or $z \geq w > u'$. Any of those cases implies that either $w \in S$ or $w \in S'$, since S and S' are segments of $H(x)$. A contradiction. This shows that $S \cup S'$ is a segment of $H(x)$.

Thus, we have shown that $S \cup S'$ is a segment of $H(x)$ with the property that its

elements are related as ancestor and descendant. But since $S \neq S'$, this contradicts the maximality of both S and S' with this property. We conclude that the segments in $\mathcal{S}$ partition $H(x)$. $\square$

For every vertex x, we will need to compute the collection of the segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant. This can be done with a straightforward method that is shown in Algorithm 10. The idea is to traverse the list $H(x)$, and greedily collect all consecutive vertices that are related as ancestor and descendant in order to get a segment. The proof of correctness in given in Lemma 3.22.

Algorithm 10: Compute the collection $\mathcal{S}$ of the segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant

```
1  let 𝒮 ← ∅
2  let z be the first element of H(x)
3  while z ≠ ⊥ do
4      let S ← {z}
5      let z' ← next_H(x)(z)
6      while z' ≠ ⊥ and z' is an ancestor of z do
7          insert z' into S
8          z' ← next_H(x)(z')
9      end
10     insert S into 𝒮
11     z ← z'
12 end
```

Lemma 3.22. *Let x be a vertex. Then, Algorithm 10 correctly computes the collection $\mathcal{S}$ of the segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant. The running time of Algorithm 10 is $O(|H(x)|)$.*

Proof. The **while** loop in Line 3 begins the processing of $H(x)$ from its first vertex z. Then, the **while** loop in Line 6 collects all the consecutive successors z' of z that are ancestors of z, and stops until it reaches a vertex z' that is not an ancestor of z. Let S be the resulting set (in Line 10). Then, we have that all vertices in S have z as a common descendant, and therefore all of them are related as ancestor and descendant. Furthermore, by construction, S is a segment of $H(x)$. Then, notice that

S is a maximal segment of $H(x)$ with the property that its elements are related as ancestor and descendant, because the successor of the lowest element in S is not an ancestor of z, and therefore it is not related as ancestor and descendant with z (because, if it was, it would be an ancestor of z, due to the ordering of $H(x)$). By Lemma 3.21, we have that no other segment of $\mathcal{S}$ intersects with S. Thus, it is proper to move on to the processing of z', and always move forward in processing the vertices of $H(x)$. Thus, we can see that the **while** loop in Line 3 correctly computes the segment $S(z)$, for every z that it processes, and every vertex in $H(x)$ will be inserted in such a segment eventually (either at the beginning of the **while** loop of Line 3 in Line 4, or by the **while** loop of Line 6 in Line 7). It is easy to see that the number of steps performed by Algorithm 10 is $O(|H(x)|)$. □

For every vertex x, we also define the list $\widetilde{H}(x)$ that consists of all vertices $v \neq r$ such that either $high_1(v) = x$ or $high_2(v) = x$, sorted in decreasing order. Notice that, for every vertex $v \neq r$, there are at most two distinct x and x' such that $v \in \widetilde{H}(x)$ and $v \in \widetilde{H}(x')$. More precisely, if a vertex $v \neq r$ satisfies that $high_1(v) \neq high_2(v)$, then v belongs to both $\widetilde{H}(high_1(v))$ and $\widetilde{H}(high_2(v))$. But there is no other set of the form $\widetilde{H}(x)$ that contains v. Thus, the collection $\{\widetilde{H}(x) \mid x \text{ is a vertex}\}$ has total size $O(n)$. For every vertex $v \neq r$, we let $\widetilde{S}_1(v)$ denote the segment of $\widetilde{H}(high_1(v))$ that contains v and is maximal w.r.t. the property that its elements are related as ancestor and descendant. Similarly, we let $\widetilde{S}_2(v)$ denote the segment of $\widetilde{H}(high_2(v))$ that contains v and is maximal w.r.t. the property that its elements are related as ancestor and descendant. We can see that the collection of the segments of $\widetilde{H}(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant constitutes a partition of $\widetilde{H}(x)$. The proof of this property is precisely the same as in Lemma 3.21, because it only relies on the fact that $\widetilde{H}(x)$ is sorted in decreasing order. Furthermore, we can apply a procedure as that shown in Algorithm 10, in order to compute the collection of all those maximal segments in $O(|\widetilde{H}(x)|)$ time. We state this result in the following lemma, which has the same proof as Lemma 3.22.

Lemma 3.23. *Let x be a vertex. Then, in $O(|\widetilde{H}(x)|)$ time, we can compute the collection of the segments of $\widetilde{H}(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant.*

Proof. We can use Algorithm 10, where we have replaced every occurrence of "$H(x)$" with "$\widetilde{H}(x)$". The proof of correctness is the same as in Lemma 3.22. □

Since every vertex $v \neq r$ belongs to at most two sets of the form $\widetilde{H}(x)$, for a vertex x, the collection $\bigcup\{\mathcal{S}(x) \mid x \text{ is a vertex }\}$, where $\mathcal{S}(x)$ is the collection of the segments of $\widetilde{H}(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant, has total size $O(n)$. That is, $\sum_x \sum_{S \in \mathcal{S}(x)} |S| = O(n)$.

We conclude this section with the following lemma, which shows that the vertices in every segment from $\mathcal{S}(x)$ are sorted in decreasing order w.r.t. their *low* point.

Lemma 3.24. *Let x be a vertex, and let u and v be two vertices in $\widetilde{H}(x)$ such that u is a descendant of v. Then $low(u) \geq low(v)$.*

Proof. Since $u \in \widetilde{H}(x)$, we have that either $high_1(u) = x$ or $high_2(u) = x$. Since $v \in \widetilde{H}(x)$, we have that either $high_1(v) = x$ or $high_2(v) = x$. In either case then, we have that x is a proper ancestor of v.

Let us suppose first that $high_1(u) = x$. Let (z, w) be a back-edge in $B(u)$. Then z is a descendant of u, and therefore a descendant of v. Furthermore, we have that w is an ancestor of $high_1(u)$, and therefore an ancestor of x, and therefore a proper ancestor of v. This shows that $(z, w) \in B(v)$. Due to the generality of $(z, w) \in B(u)$, this implies that $B(u) \subseteq B(v)$. From this we infer that $low(u) \geq low(v)$.

Now let us suppose that $high_1(u) \neq x$. This implies that $high_2(u) = x$. Let (z, w) be a back-edge in $B(u) \setminus \{e_{high}(u)\}$ (such a back-edge exists, because the graph is 3-edge-connected, and therefore $|B(u)| > 1$). Then, we have that z is a descendant of u, and therefore a descendant of v. Furthermore, w is an ancestor of $high_2(u)$, and therefore an ancestor of x, and therefore a proper ancestor of v. This shows that $(z, w) \in B(v)$. Due to the generality of $(z, w) \in B(u) \setminus \{e_{high}(u)\}$, this implies that $B(u) \setminus \{e_{high}(u)\} \subseteq B(v)$. Since $high_1(u) \neq x$ and $high_2(u) = x$, we have $high_1(u) \neq high_2(u)$. This implies that $high_1(u) > high_2(u)$, and therefore $low(u) < high_1(u)$. Thus, the *low* point of u is given by the lowest lower endpoint of all back-edges in $B(u) \setminus \{e_{high}(u)\}$. Since this set is a subset of $B(v)$, we conclude that $low(v) \leq low(u)$. □

Chapter 4

Computing the 4-Edge-Connected Components in Linear Time

4.1 Introduction

4.2 3-cuts on a DFS tree

4.3 Computing all 3-cuts of a 3-edge-connected graph

4.4 Computing the 4-edge-connected components

4.5 Testing 4-edge connectivity

4.1 Introduction

In this chapter, we present a linear-time algorithm for computing the 4-edge-connected components of an undirected multigraph. This algorithm uses elementary data structures, and it is implementable in the pointer machine model of computation [65]. We also provide a very simple algorithm for testing the 4-edge-connectivity of a graph in linear time.

The general idea for computing the 4-edge-connected components can be described as follows. First, we use a construction described in [21], that reduces this computation to 3-edge-connected graphs. This construction can be completed in linear time, using any algorithm for computing the 3-edge-connected components of an undirected multigraph (e.g., [67]). Now, given a 3-edge-connected graph, we compute

the collection of all 3-cuts of the graph, and then the atoms induced by this collection. Since the collection of all 3-cuts of a 3-edge-connected graph forms a parallel family of 3-cuts (see e.g. [24]), we can compute those atoms in linear time using Algorithm 17. Thus, the whole problem reduces to the computation of all 3-cuts of a 3-edge-connected graph. To perform this computation efficiently, we rely on a DFS-tree T of the graph, and we provide a typology of 3-cuts w.r.t. T. Specifically, we distinguish three types of 3-cuts, depending on the number of tree-edges of T that they contain (notice that a 3-cut must contain at least one tree-edge of T). Then, we can compute all three types of 3-cuts separately. The case of 3-cuts that contain exactly one tree-edge is the easiest one. The case of 3-cuts that contain exactly two tree-edges is the most demanding, and we further distinguish it into various sub-cases. Finally, the case of 3-cuts that consist of three tree-edges can be reduced to the previous two cases, as shown in [50]. For the computation of 3-cuts, we rely on some of the DFS-based parameters that were introduced in Chapter 3. In particular, all those parameters that we use here can be computed with linear-time algorithms that have a pointer-machine implementation.

4.2 3-cuts on a DFS tree

Here we provide a typology of the 3-cuts of a connected graph, according to their topology on a DFS tree. This will be useful in order to show how to compute them in Section 4.3, using an appropriate algorithm for each type. Furthermore, it will be useful in order to compute the 4-edge-connected components in Section 4.4. We refer to Figure 4.1 for the typology of 3-cuts that we provide in this section. Our goal here is precisely to show that this figure exhausts all possibilities for the 3-cuts on a DFS tree. We assume throughout that we work on a connected graph G, with an r-rooted DFS tree T.

Let C be a 3-cut of G. Then observe that C must contain at least one tree-edge (otherwise its removal from G does not disconnect it). Thus we initially distinguish three types of 3-cuts, Type-1, Type-2, and Type-3, depending on whether they contain one, two, or three tree-edges, respectively.

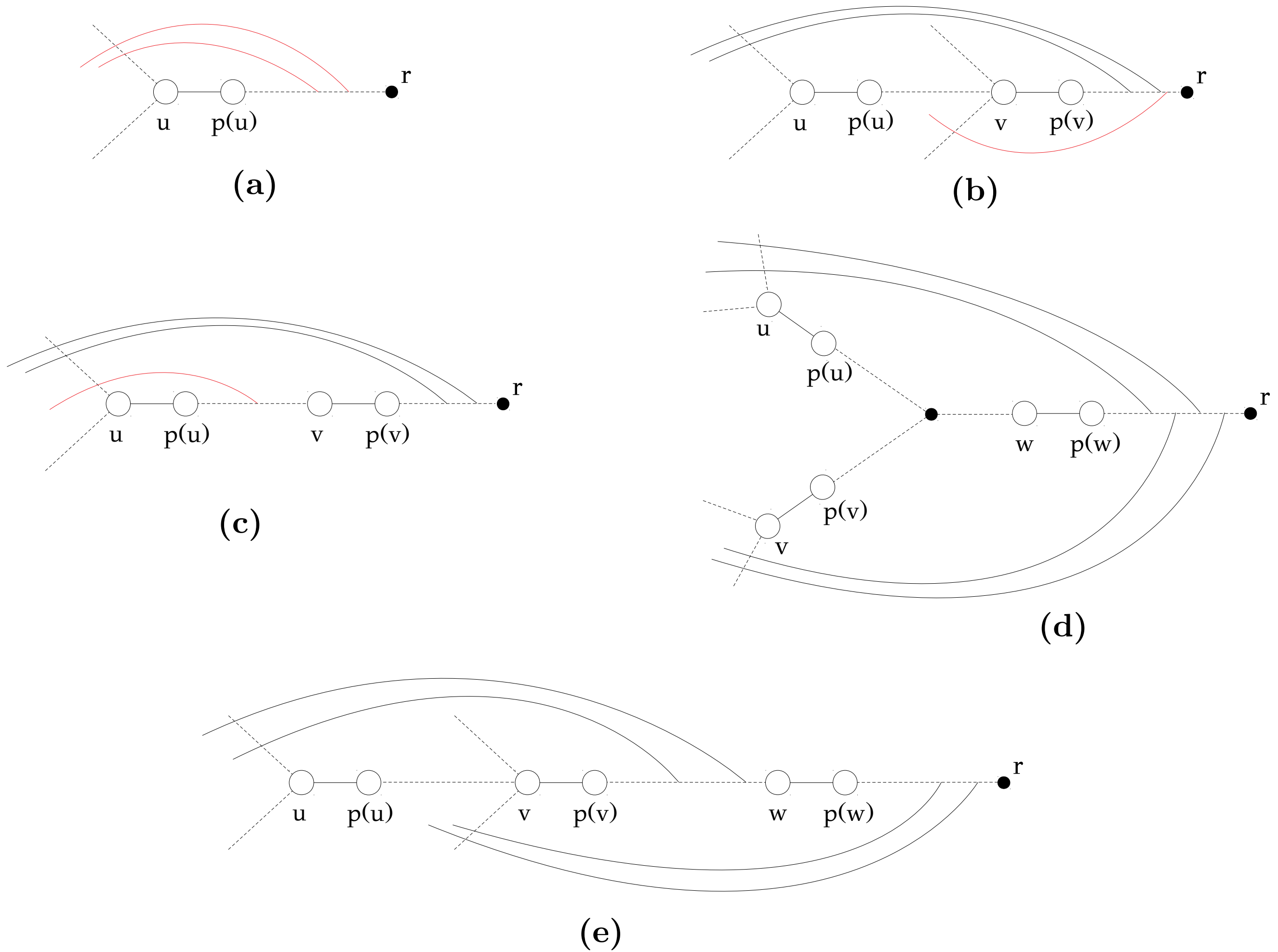


Figure 4.1: The types of 3-cuts with respect to a DFS tree. **(a)** (Type-1) One tree-edge $(u, p(u))$ and two back-edges. **(b)** (Type-2, "upper case") Two tree-edges $(u, p(u))$ and $(v, p(v))$, where u is a descendant of v, and one back-edge in $B(v) \setminus B(u)$. **(c)** (Type-2, "lower case") Two tree-edges $(u, p(u))$ and $(v, p(v))$, where u is a descendant of v, and one back-edge in $B(u) \setminus B(v)$. **(d)** (Type-3α) Three tree-edges $(u, p(u))$, $(v, p(v))$ and $(w, p(w))$, where w is an ancestor of u and v, but u and v are not related as ancestor and descendant. **(d)** (Type-3β) Three tree-edges $(u, p(u))$, $(v, p(v))$ and $(w, p(w))$, where u is a descendant of v, and v is a descendant of w.

4.2.1 Type-1 3-cuts

For Type-1 3-cuts we have the following.

Lemma 4.1. *Let $u \neq r$ be a vertex. Then there exist two distinct back-edges e_1 and e_2 such that $C = \{(u, p(u)), e_1, e_2\}$ is a 3-cut if and only if $B(u) = \{e_1, e_2\}$.*

Proof. ($\Rightarrow$) First we will show that both e_1 and e_2 must be in $B(u)$. Let us suppose, for the sake of contradiction, that this is not true. We may assume w.l.o.g. that e_1 is not in $B(u)$. Let $e_1 = (x, y)$. Since C is a 3-cut of G, we have that $G' = G \setminus \{(u, p(u)), e_2\}$ is connected. In particular, since $e_1 \notin B(u)$, we have that x is connected with y in G' through the path $T[x, y]$ (which remains intact in G'). But then x remains connected with y in $G' \setminus e_1$ through this path, in contradiction to the fact that C is a 3-cut of G. Thus we have that e_1 must leap over u. Similarly, we have that e_2 must leap over u. Thus, $\{e_1, e_2\} \subseteq B(u)$. Now let us suppose, for the sake of contradiction, that $B(u) \neq \{e_1, e_2\}$. This implies that there is a back-edge $(x, y) \in B(u) \setminus \{e_1, e_2\}$. But then u remains connected with $p(u)$ in $G \setminus C$ through the path $T[u, x], (x, y), T[y, p(u)]$, in contradiction to the fact that C is a 3-cut of G. Thus we have that $B(u) = \{e_1, e_2\}$.

($\Leftarrow$) Consider the parts $A = T(u)$ and $B = T(r) \setminus T(u)$. Then $B(u) = \{e_1, e_2\}$ implies that the only back-edges from A to B are e_1 and e_2. Thus, A and B become disconnected in $G \setminus C$. Furthermore, observe that no proper subset of C can disconnect A and B upon removal. □

Lemma 4.2. *Let $C = \{(u, p(u)), e_1, e_2\}$ be a Type-1 3-cut. Then the connected components of $G \setminus C$ are $T(u)$ and $T(r) \setminus T(u)$.*

Proof. It is easy to see that $T(u)$ and $T(r) \setminus T(u)$ become separated in $G \setminus C$, and that each of them is connected in $G \setminus C$. □

4.2.2 Type-2 3-cuts

Lemma 4.3. *Let u and v be two vertices $\neq r$, and let e be a back-edge such that $C = \{(u, p(u)), (v, p(v)), e\}$ is a 3-cut. Then u and v are related as ancestor and descendant.*

Proof. Let us suppose, for the sake of contradiction, that u and v are not related as ancestor and descendant. Then we have that e cannot leap over both u and v (for otherwise u and v would be common ancestors of the higher endpoint of e). So we may assume w.l.o.g. that e does not leap over u. Since C is a 3-cut, we have that

$G \setminus \{(u,p(u))\}$ is connected, and so $B(u)$ is non-empty. Thus, there is a back-edge $(x,y) \in B(u)$. Then we have $e \neq (x,y)$. Furthermore, since v is not related as ancestor and descendant with u, we have that $v \notin T[x,u]$ and $v \notin T[p(u),y]$. But this implies that u remains connected with $p(u)$ in $G \setminus C$ through the path $T[u,x],(x,y),T[y,p(u)]$, in contradiction to the fact that C is a 3-cut of G. We conclude that u and v are related as ancestor and descendant. □

Lemma 4.4. *Let $C' = \{(u,p(u)),(v,p(v)),e\}$ be a Type-2 3-cut, where v is an ancestor of u. Then the connected components of $G \setminus C'$ are $T(v) \setminus T(u)$ and $T(u) \cup (T(r) \setminus T(v))$.*

Proof. Consider the parts $A = T(u)$, $B = T(v) \setminus T(u)$, and $C = T(r) \setminus T(v)$. Notice that every one of those parts is connected in $G \setminus C'$. Then, since $G \setminus C'$ consists of two connected components, we have that these components are either (i) $A \cup B$ and C, or (ii) $A \cup C$ and B, or (iii) $B \cup C$ and A. Case (i) is rejected, because u here remains connected with $p(u)$ in $G \setminus C'$. Case (iii) is rejected, because v here remains connected with $p(v)$ in $G \setminus C'$. We conclude that the connected components of $G \setminus C'$ are given by (ii). □

Lemma 4.5. *Let u and v be two vertices $\neq r$. Then there is a back-edge e such that $C' = \{(u,p(u)),(v,p(v)),e\}$ is a 3-cut if and only if neither of $B(u)$ and $B(v)$ is empty, and either (1) $B(v) = B(u) \sqcup \{e\}$ or (2) $B(u) = B(v) \sqcup \{e\}$.*

Proof. ($\Rightarrow$) By Lemma 4.3 we have that u and v are related as ancestor and descendant. We may assume w.l.o.g. that v is a proper ancestor of u. Consider the parts $A = T(u)$, $B = T(v) \setminus T(u)$, and $C = T(r) \setminus T(v)$. By Lemma 4.4 we have that the connected components of $G \setminus C'$ are B and $A \cup C$. Thus, in $G \setminus C'$ there is no back-edge from A to B, and no back-edge from B to C $(*)$. However, since $G \setminus \{(u,p(u)),(v,p(v))\}$ is connected, there must exist either a back-edge from A to B, or a back-edge from B to C.

First, let us suppose that there is a back-edge from A to B in G. Then $(*)$ implies that this back-edge is unique, it coincides with e, and there is no back-edge from B to C. Thus, e is in $B(u) \setminus B(v)$, and there are no back-edges in $B(v) \setminus B(u)$. Thus, $B(v) \sqcup \{e\} \subseteq B(u)$. Conversely, a back-edge in $B(u) \setminus \{e\}$ must be a back-edge from A to C, and therefore a back-edge in $B(v)$. This shows that $B(v) \sqcup \{e\} = B(u)$.

Now let us suppose that there is a back-edge from B to C in G. Then $(*)$ implies that this back-edge is unique, it coincides with e, and there is no back-edge from A

to B. Thus, e is in $B(v) \setminus B(u)$, and there are no back-edges in $B(u) \setminus B(v)$. Thus, $B(u) \sqcup \{e\} \subseteq B(v)$. Conversely, a back-edge in $B(v) \setminus \{e\}$ must be a back-edge from A to C, and therefore a back-edge in $B(u)$. This shows that $B(u) \sqcup \{e\} = B(v)$.

Notice that $B(u) \neq \emptyset$ (resp., $B(v) \neq \emptyset$), because C' is a 3-cut of G, and therefore $(u, p(u))$ (resp., $(v, p(v))$) is not a bridge.

($\Leftarrow$) Since neither of $B(u)$ and $B(v)$ is empty, we have that either of (1) and (2) implies that $B(u) \cap B(v) \neq \emptyset$. Thus, u and v are related as ancestor and descendant (because they are ancestors of the higher endpoint of any of the back-edges that leaps over both of them). We will consider only case (1), in both possible ancestry relations between u and v, because case (2) is symmetric to (1).

First, let us suppose that v is an ancestor of u. We have that v is a proper ancestor of u, because $B(v) = B(u) \sqcup \{e\}$ implies that $B(v) \neq B(u)$. We consider the parts $A = T(u)$, $B = T(v) \setminus T(u)$, and $C = T(r) \setminus T(v)$. Then $B(v) = B(u) \sqcup \{e\}$ implies that e is the unique back-edge from B to C, and there is no back-edge from A to B. Furthermore, since $B(u) \neq \emptyset$, we have that there is at least one back-edge from A to C. Thus, it is easy to see that $G \setminus C'$ is not connected, but $G \setminus C''$ is connected for any proper subset C'' of C'.

Now let us suppose that u is an ancestor of v. We have that u is a proper ancestor of v, because $B(v) = B(u) \sqcup \{e\}$ implies that $B(v) \neq B(u)$. We consider the parts $A = T(v)$, $B = T(u) \setminus T(v)$, and $C = T(r) \setminus T(u)$. Then $B(v) = B(u) \sqcup \{e\}$ implies that e is the unique back-edge from A to B, and there is no back-edge from B to C. Furthermore, since $B(v) \neq \emptyset$, we have that there is at least one back-edge from A to C. Thus, it is easy to see that $G \setminus C'$ is not connected, but $G \setminus C''$ is connected for any proper subset C'' of C'. □

According to Lemma 4.5, we distinguish two types of Type-2 3-cuts, depending on whether (1) or (2) is satisfied. Following the terminology of [50], we call them the "upper case" and the "lower case", respectively. We refer to Figure 4.1 for an illustration.

4.2.3 Type-3 3-cuts

Lemma 4.6. *Let u, v, w be three vertices $\neq r$ such that $C = \{(u, p(u)), (v, p(v)), (w, p(w))\}$ is a 3-cut. Then one of u, v, w is a common ancestor of the other two.*

Proof. Let us suppose, for the sake of contradiction, that none of u, v, w is a common ancestor of the other two. This is equivalent to saying that at least one of u, v, w is not related as ancestor and descendant with the other two. So we may assume w.l.o.g. that w is not related as ancestor and descendant with u and v. Since C is a 3-cut, we have that $(w, p(w))$ is not a cut-edge, and so $B(w)$ is not empty. Let (x, y) be a back-edge in $B(w)$. Then x is descendant of w and y is a proper ancestor of w. Since w is not related as ancestor and descendant with u and v, we have that neither of u and v is in $T[x, w]$ or $T[p(w), y]$. But this implies that w remains connected with $p(w)$ in $G \setminus C$ through the path $T[w, x], (x, y), T[y, p(w)]$, in contradiction to the fact that C is a 3-cut of G. Thus, we have that one of u, v, w is a common ancestor of the other two. □

Let u, v, w be three vertices $\neq r$ such that $C' = \{(u, p(u)), (v, p(v)), (w, p(w))\}$ is a 3-cut. Then, according to Lemma 4.6 we have that one of u, v, w is a common ancestor of the other two. Let us assume w.l.o.g. that w is a common ancestor of u and v. Then, if u and v are not related as ancestor and descendant, we call C a Type-3α 3-cut. Otherwise, we call C a Type-3β 3-cut. We refer to Figure 4.1 for an illustration.

Lemma 4.7. *Let u, v, w be three vertices $\neq r$ such that $C' = \{(u, p(u)), (v, p(v)), (w, p(w))\}$ is a Type-3α 3-cut, where w is a common ancestor of u and v. Then the connected components of $G \setminus C'$ are given by $T(u) \cup T(v) \cup (T(r) \setminus T(w))$ and $T(w) \setminus (T(u) \cup T(v))$.*

Proof. Consider the parts $A = T(u)$, $B = T(v)$, $C = T(w) \setminus (T(u) \cup T(v))$ and $D = T(r) \setminus T(w)$. Then every one of A, B, C and D remains connected in $G \setminus C'$. Notice that there is no back-edge from A to C, because otherwise we would have that u remains connected with $p(u)$ in $G \setminus C'$. Similarly, there is no back-edge from B to C, because otherwise we would have that v remains connected with $p(v)$ in $G \setminus C'$. Finally, there is no back-edge from C to D, because otherwise we would have that w remains connected with $p(w)$ in $G \setminus C'$. This shows that C becomes disconnected from $A \cup B \cup D$ in $G \setminus C'$. Since C' is a 3-cut of G, we have that $G \setminus C'$ consists of two connected components. Since C remains connected in $G \setminus C'$, but it is disconnected from $A \cup B \cup D$ in $G \setminus C'$, we have that the connected components of $G \setminus C'$ are C and $A \cup B \cup D$. □

Lemma 4.8. *Let u, v, w be three vertices $\neq r$ such that $C' = \{(u, p(u)), (v, p(v)), (w, p(w))\}$ is a Type-3β 3-cut, where w is an ancestor of v, and v is an ancestor of u. Then the connected components of $G \setminus C'$ are given by $T(u) \cup (T(w) \setminus T(v))$ and $(T(v) \setminus T(u)) \cup (T(r) \setminus T(w))$.*

Proof. Consider the parts $A = T(u)$, $B = T(v) \setminus T(u)$, $C = T(w) \setminus T(v)$ and $D = T(r) \setminus T(w)$. Then every one of A, B, C and D remains connected in $G \setminus C'$. Notice that A and B cannot be in the same connected component of $G \setminus C'$, since $u \in A$ and $p(u) \in B$ (1). Similarly, B and C cannot be in the same connected component of $G \setminus C'$, since $v \in B$ and $p(v) \in C$ (2). Finally, C and D cannot be in the same connected component of $G \setminus C'$, since $w \in C$ and $p(w) \in D$ (3).

Let us suppose, for the sake of contradiction, that A and D are in the same connected component X of $G \setminus C'$. By (1) we have that B cannot be in X. By (3) we have that C cannot be in X. Since C' is a 3-cut of G, we have that $G \setminus C'$ consists of two connected components. Thus, one of them is $A \cup D$, and the other must be $B \cup C$. But this contradicts (2). Thus we have that A and D do not belong to the same connected component of $G \setminus C'$ (4).

Now let us suppose, for the sake of contradiction, that A and C do not belong to the same connected component of $G \setminus C'$. Then (1) and (4) imply that A is a connected component of $G \setminus C'$. But also (2) and (3) imply that C is a connected component of $G \setminus C'$. Thus, $G \setminus C'$ consists of at least three connected components, a contradiction. Thus we have that A and C belong to the same connected component X of $G \setminus C'$. By (1) we have that B does not belong to X. And by (3) we have that D does not belong to X. Thus, the connected components of $G \setminus C'$ are $A \cup C$ and $B \cup D$. ☐

4.3 Computing all 3-cuts of a 3-edge-connected graph

In the following we assume that G is a 3-edge-connected graph with n vertices and m edges, and we have fixed a DFS tree T of G with root r. Thus, for every vertex $v \neq r$ of G, we have that $B(v)$ contains at least two back-edges.

4.3.1 Computing Type-1 3-cuts

According to Lemma 4.1, $\{(u, p(u)), e_1, e_2\}$ is a 3-cut of G, where $u \neq r$ and e_1, e_2 are back-edges, if and only if $B(u) = \{e_1, e_2\}$. Thus, we can determine easily whether a tree-edge $(u, p(u))$ forms a 3-cut with two back-edges, by checking whether $\mathit{bcount}(u) = 2$. In this case, we need to know the two back-edges that are in $B(u)$. For this purpose, we may use $e_1 = (\mathit{lowD}_1(u), \mathit{low}_1(u))$ and $e_2 = (\mathit{lowD}_2(u), \mathit{low}_2(u))$, since these are two distinct back-edges that leap over u.

4.3.2 Computing Type-2 3-cuts

Let u and v be two vertices $\neq r$. According to Lemma 4.3, if $\{(u,p(u)),(v,p(v)),e\}$ is a 3-cut, where e is a back-edge, then u and v are related as ancestor and descendant. So we may assume w.l.o.g. that v is an ancestor of u. Then, by Lemma 4.5, we have that either (1) $B(v)=B(u)\sqcup\{e\}$ or (2) $B(u)=B(v)\sqcup\{e\}$. We treat cases (1) and (2) with a different algorithm, in Sections 4.3.2.1 and 4.3.2.2, respectively. We note that cases (1) and (2) correspond to the "upper case" and the "lower case", respectively. (For an illustration, we refer to Figure 4.1.)

4.3.2.1 The upper case

Let $v\neq r$ be a vertex, and suppose that there is a proper descendant u of v and a back-edge e such that $B(v)=B(u)\sqcup\{e\}$. Then, by Lemma 4.9 below we have that u is unique. Now, if such a u exists, by Lemma 4.10 we have that one of the following holds (see also Figure 4.2): either (1) $M(u)=\widetilde{M}(v)$, or (2) $M(u)=M_{low1}(v)$, or (3) $M(u)=M_{low2}(v)$. Furthermore, Lemma 4.10 shows how to find the back-edge e in every case. Thus, there are three different cases to consider in order to find u (and e). In every case, we rely on Lemma 4.11, which says that u is the lowest proper descendant of v in $M^{-1}(M(u))$. Thus, we consider the three different cases $x=\widetilde{M}(v)$, $x=M_{low1}(v)$ and $x=M_{low2}(v)$, and in each case we traverse the list $M^{-1}(x)$ in order to find the lowest proper descendant u of v (if it exists). Then we use Lemma 4.12, 4.13 or 4.14, respectively, in order to verify that u has indeed the desired property.

This is basically the idea in order to find all 3-cuts of the form $\{(u,p(u)),(v,p(v)),e\}$, where v is an ancestor of u and e is a back-edge such that $B(v)=B(u)\sqcup\{e\}$. However, this is not a linear-time procedure, since we may have to traverse the lists $M^{-1}(x)$ an excessive number of times. In order to resolve this, we process the vertices v in a bottom-up fashion, and we keep in a variable *currentVertex*$[x]$, for every vertex x, the lowest element in $M^{-1}(x)$ that is a proper descendant of v (if it exists), where v is the current vertex under process. To update this variable appropriately during the processing of v, we just scan the list $M^{-1}(x)$, each time starting from *currentVertex*$[x]$, until we reach the lowest vertex in $M^{-1}(x)$ that is greater than v. The procedure for computing all Type-2 3-cuts in the upper case is shown in Algorithm 11.

Algorithm 11: Find all 3-cuts $\{(u,p(u)),(v,p(v)),e\}$, where u is a descendant of v and $B(v) = B(u) \sqcup \{e\}$, for a back-edge e.

```
1  // x = M̃(v)
2  for v ← n to v = 1 do
3      x ← M̃(v)
4      if x = ⊥ then continue
5      let u be the lowest vertex in M^{-1}(x) which is greater than v
6      // check the condition in Lemma 4.12
7      if bcount(v) = bcount(u) + 1 and l_2(M(v)) ≥ v and (c_2(M(v)) = ⊥ or
        low(c_2(M(v))) ≥ v) then
8          mark {(u,p(u)),(v,p(v)),(M(v),l(M(v)))} as a 3-cut
9      end
10 end
11 // x = M_low1(v)
12 for v ← n to v = 1 do
13     x ← M_low1(v)
14     if x = ⊥ then continue
15     let u be the lowest vertex in M^{-1}(x) which is greater than v
16     // check the condition in Lemma 4.13
17     if bcount(v) = bcount(u) + 1 and low_2(M_low2(v)) ≥ v and (c_3(M(v)) = ⊥ or
        low(c_3(M(v))) ≥ v) then
18         mark {(u,p(u)),(v,p(v)),(M_low2(v),l(M_low2(v)))} as a 3-cut
19     end
20 end
21 // x = M_low2(v)
22 for v ← n to v = 1 do
23     x ← M_low2(v)
24     if x = ⊥ then continue
25     let u be the lowest vertex in M^{-1}(x) which is greater than v
26     // check the condition in Lemma 4.14
27     if bcount(v) = bcount(u) + 1 and low_2(M_low1(v)) ≥ v and (c_3(M(v)) = ⊥ or
        low(c_3(M(v))) ≥ v) then
28         mark {(u,p(u)),(v,p(v)),(M_low1(v),l(M_low1(v)))} as a 3-cut
29     end
30 end
```

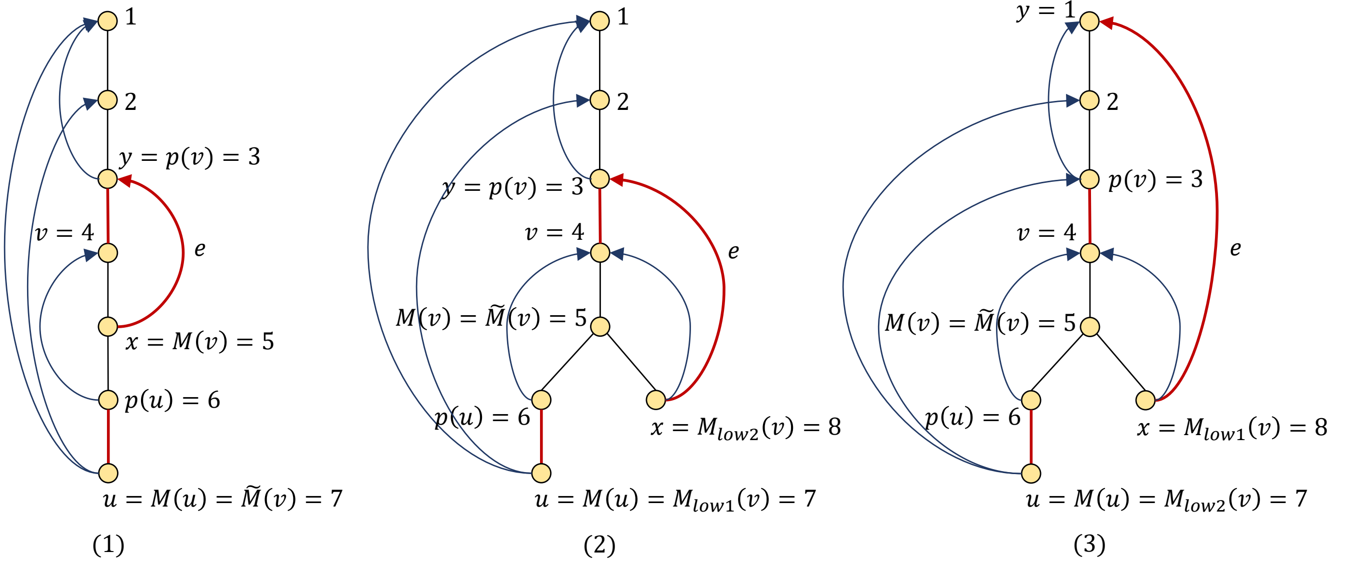


Figure 4.2: An illustration of the three cases for type-2 cuts $\{(u,p(u)),(v,p(v)),e\}$ (shown with red edges) where v is an ancestor of u, e is a back-edge, and $B(v) = B(u) \sqcup \{e\}$. Vertices are numbered in DFS order and back-edges are shown directed from descendant to ancestor in the DFS tree.

Lemma 4.9. *Let $v \neq r$ be a vertex, and suppose that there is a proper descendant u of v and a back-edge e such that $B(v) = B(u) \sqcup \{e\}$. Then u is unique with this property.*

Proof. Let us suppose, for the sake of contradiction, that there are two distinct proper descendants u and u' of v, and two back-edges e and e' (not necessarily distinct), such that $B(v) = B(u) \sqcup \{e\}$ and $B(v) = B(u') \sqcup \{e'\}$. Then we have $B(u) \sqcup \{e\} = B(u') \sqcup \{e'\}$. Since $bcount(u) > 1$ and $bcount(u') > 1$ (since the graph is 3-edge-connected), we infer that $B(u) \cap B(u') \neq \emptyset$, and thus u and u' are related as ancestor and descendant. Thus we can assume, without loss of generality, that u' is an ancestor of u. Now let $(x,y) \in B(u)$. Then x is a descendant of u, and therefore a descendant of u'. Furthermore, since $B(v) = B(u) \sqcup \{e\}$, we have $(x,y) \in B(v)$, and so y is a proper ancestor of v, and therefore a proper ancestor of u'. This shows that $(x,y) \in B(u')$, and thus we have $B(u) \subseteq B(u')$. In conjunction with $B(u) \sqcup \{e\} = B(u') \sqcup \{e'\}$ (which implies that $|B(u)| = |B(u')|$), we infer that $B(u) = B(u')$ (and $e = e'$). This contradicts the fact that the graph is 3-edge-connected (see Proposition 3.1). □

Lemma 4.10. *Let u and v be two vertices $\neq r$ such that v is a proper ancestor of u with $B(v) = B(u) \sqcup \{e\}$, for a back-edge e. Then, we have that either (1) $M(u) = \widetilde{M}(v)$, or (2) $M(u) = M_{low1}(v)$, or (3) $M(u) = M_{low2}(v)$. According to whether (1), or (2), or (3), is true, we have $e = (M(v), l_1(M(v)))$, or $e = (M_{low2}(v), l_1(M_{low2}(v)))$, or $e = (M_{low1}(v), l_1(M_{low1}(v)))$, respectively.*

Proof. First let us suppose, for the sake of contradiction, that $M(v)$ is a descendant of u. Let (x, y) be a back-edge in $B(v)$. Then x is a descendant of $M(v)$, and therefore a descendant of u. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Due to the generality of $(x, y) \in B(v)$, this implies that $B(v) \subseteq B(u)$, in contradiction to $B(v) = B(u) \sqcup \{e\}$. Thus we have that $M(v)$ is not a descendant of u.

Now suppose that $e = (M(v), z)$, for a vertex z. Let (x, y) be a back-edge in $B(u)$. Then we have $x \neq M(v)$, and $B(v) = B(u) \sqcup \{e\}$ implies that $(x, y) \in B(v)$. This implies that x is a descendant of $\widetilde{M}(v)$. Due the generality of $(x, y) \in B(u)$, this shows that $M(u)$ is a descendant of $\widetilde{M}(v)$. Conversely, let (x, y) be a back-edge in $B(v)$ such that $x \neq M(v)$. Then $(x, y) \neq e$, and so $B(v) = B(u) \sqcup \{e\}$ implies that $(x, y) \in B(u)$, which further implies that x is a descendant of $M(u)$. Due to the generality of $(x, y) \in B(v)$ with the property that $x \neq M(v)$, we infer that $\widetilde{M}(v)$ is a descendant of $M(u)$. Thus we have that $M(u) = \widetilde{M}(v)$. Notice that there is an $i \in \{1, 2, \dots\}$ such that $z = l_i(M(v))$. Let us suppose, for the sake of contradiction, that $i > 1$. Since $(M(v), z) \in B(v)$, we have that $z < v$. This implies that $l_j(M(v)) < v$, for every $j \in \{1, \dots, i\}$. Since $M(v)$ is not a descendant of u, there are no back-edges of the form $(M(v), z')$ in $B(u)$. In particular, none of the back-edges $(M(v), l_j(M(v)))$, for $j \in \{1, \dots, i\}$, are in $B(u)$. But all of them are in $B(v)$. Since $i > 1$, this implies that there are more that one back-edges in $B(v) \setminus B(u)$, in contradiction to $B(v) = B(u) \sqcup \{e\}$. Thus we have that $e = (M(v), l_1(M(v)))$.

Now suppose that there are vertices w and z, with $w \neq M(v)$, such that $e = (w, z)$. We can show that w is not a descendant of u, by using the same argument that we used in order to show that $M(v)$ is not a descendant of u. Furthermore, we can use the same argument that we used before in order to show that $z = l_1(w)$. Now, since $e \in B(v)$ and $w \neq M(v)$, we have that w is a descendant of a child of $M(v)$. Let us suppose, for the sake of contradiction, that w is a descendant of the *lowi* child of $M(v)$, for some $i > 2$. Then we have *low*$(c_1(M(v))) < v$ and *low*$(c_2(M(v))) < v$. This implies that there are back-edges $e_1 = (x_1, y_1)$ and $e_2 = (x_2, y_2)$, such that x_1 is a descendant

of the *low1* child of $M(v)$, x_2 is a descendant of the *low2* child of $M(v)$, and both y_1 and y_2 are proper ancestors of v. Thus, we have that $e_1, e_2 \in B(v)$. But since none of e_1 and e_2 can be e, by $B(v) = B(u) \sqcup \{e\}$ we have that $e_1, e_2 \in B(u)$. This implies that $M(u)$ is an ancestor of both x_1 and x_2, and therefore it is an ancestor of $M(v)$. But this contradicts the fact that $M(v)$ is not a descendant of u. Thus we have that w is a descendant of either $c_1(M(v))$ or $c_2(M(v))$.

Let us suppose that w is a descendant of $c_1(M(v))$ (the other case is treated similarly). Notice that there cannot be any back-edge of the form $(M(v), z')$ in $B(v)$, for this would imply the existence of at least two back-edges in $B(v) \setminus B(u)$. Thus, we must have $low(c_2(M(v))) < v$ (for otherwise we would have that all back-edges that leap over v have their higher endpoint in $T(c_1(M(v)))$, which would imply that $M(v)$ is a descendant of $c_1(M(v))$, which is absurd). This means that there must exist at least one back-edge (x, y) such that x is a descendant of $c_2(M(v))$ and y is a proper ancestor of v. Then $(x, y) \in B(v)$ and $(x, y) \neq e$, and so $B(v) = B(u) \sqcup \{e\}$ implies that $(x, y) \in B(u)$. This implies that e must be the unique back-edge in $B(v)$ whose higher endpoint is a descendant of $c_1(M(v))$, for otherwise we would have that $M(u)$ is an ancestor of both $c_1(M(v))$ and $c_2(M(v))$, which would imply that $M(u)$ is an ancestor of $M(v)$, which would imply that $M(v)$ is a descendant of u. Thus, we have that $w = M_{low1}(v)$.

Now let (x, y) be a back-edge in $B(u)$. Then $B(v) = B(u) \sqcup \{e\}$ implies that $(x, y) \in B(v) \setminus \{e\}$, and so x cannot be a descendant of $c_1(M(v))$. Thus x is a descendant of $c_i(M(v))$, for some $i > 1$. But we cannot have $i \neq 2$, because this would imply that $M(u)$ is an ancestor of $M(v)$. Thus, $i = 2$. Thus we have that x is a descendant of $M_{low2}(v)$. Due to the generality of $(x, y) \in B(u)$, this implies that $M(u)$ is a descendant of $M_{low2}(v)$. Conversely, let (x, y) be a back-edge in $B(v)$ such that x is a descendant of $c_2(M(v))$. Then $(x, y) \neq e$, and therefore $B(v) = B(u) \sqcup \{e\}$ implies that $(x, y) \in B(u)$. This implies that x is a descendant of $M(u)$. Due to the generality of $(x, y) \in B(v)$ with the property that x is a descendant of $c_2(M(v))$, this shows that $M_{low2}(v)$ is a descendant of $M(u)$. Thus we have $M(u) = M_{low2}(v)$. □

Lemma 4.11. *Let u and v be two vertices $\neq r$ such that v is a proper ancestor of u. If there is a back-edge e such that $B(v) = B(u) \sqcup \{e\}$, then u is the lowest element in $M^{-1}(M(u))$ which is greater than v.*

Proof. Let us suppose, for the sake of contradiction, that there is a vertex u' with

$M(u') = M(u)$, which is greater than v and lower than u. Then $M(u') = M(u)$ implies that $B(u') \subseteq B(u)$. Furthermore, since the graph is 3-edge-connected, we have $B(u') \subset B(u)$. This means that there is a back-edge $(x, y) \in B(u) \setminus B(u')$. Then x is a descendant of $M(u)$, and therefore a descendant of u'. Thus, y cannot be a proper ancestor of u', for this would imply that $(x, y) \in B(u')$. But $B(v) = B(u) \sqcup \{e\}$ implies that $(x, y) \in B(v)$, and so y must be a proper ancestor of v, and therefore a proper ancestor of u', a contradiction. We conclude that u is the lowest element in $M^{-1}(M(u))$ which is greater than v. □

First, let us consider the case that u is the lowest proper descendant of v with $M(u) = \widetilde{M}(v)$. In this case, we have the following.

Lemma 4.12 (Case (1)). *Let u be the lowest vertex in $M^{-1}(\widetilde{M}(v))$ that is greater than v. Then there exists a back-edge e such that $B(v) = B(u) \sqcup \{e\}$ if and only if: bcount(v) = bcount$(u) + 1$, $l_2(M(v)) \geq v$, and either $M(v)$ has no low2 child, or low$(c_2(M(v))) \geq v$.*

Proof. $(\Rightarrow)$ *bcount*(v) = *bcount*$(u)+1$ is an immediate consequence of $B(v) = B(u) \sqcup \{e\}$. Furthermore, $B(v) = B(u) \sqcup \{e\}$ implies that $M(v)$ is an ancestor of $M(u)$. But it cannot be the case that $M(v) = M(u)$ (for otherwise, v being an ancestor of u would imply that $B(v)$ is a subset of $B(u)$); thus, $M(v)$ is a proper ancestor of $M(u)$. Since $M(u) = \widetilde{M}(v)$, this implies that there is no back-edge (x, y) with x a descendant of a child c of $M(v)$, with $c \neq c_1(M(v))$, and y a proper ancestor of v (otherwise, we would have $\widetilde{M}(v) = M(v)$). This means that *low*$(c) \geq v$, for every child c of $M(v)$ with $c \neq c_1(M(v))$. In other words, either $M(v)$ has no *low2* child, or *low*$(c_2(M(v))) \geq v$. This also means that there exists a back-edge $\widetilde{e} = (M(v), l(M(v)))$. Obviously, $\widetilde{e} = e$, since $\widetilde{e} \in B(v) \setminus B(u)$. Now, if we had $l_2(M(v)) < v$, then there would exist a back-edge $e' = (M(v), l_2(M(v))) \neq e$. But then we would have $e' \in B(v) \setminus B(u)$, contradicting $B(v) = B(u) \sqcup \{e\}$.

$(\Leftarrow)$ Let $(x, y) \in B(v)$. Since $M(v)$ either has no *low2* child, or *low*$(c_2(M(v))) \geq v$, we have that x is either $M(v)$ or a descendant of $\widetilde{M}(v)$. Let $\widetilde{B}(v) = \{(x, y) \in B(v) \mid x \text{ is a descendant of } \widetilde{M}(v)\}$. Then we have $B(v) = \widetilde{B}(v) \sqcup \{(M(v), z) \mid (M(v), z) \text{ is a back-edge with } z < v\}$. Now, if $(x, y) \in \widetilde{B}(v)$, then x is a descendant of $\widetilde{M}(v)$, and therefore a descendant of $M(u)$. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of u. This means that $\widetilde{B}(v) \subseteq B(u)$. Now, since $l_2(M(v)) \geq v$, there is at most one back-edge $e = (M(v), z)$ with $z < v$ (which thus satisfies $z = l(M(v))$). But such a back-edge must necessarily exist, for otherwise we

would have $B(v) = \widetilde{B}(v) \subseteq B(u)$, contradicting $bcount(v) = bcount(u) + 1$. Now, since $B(v) = \widetilde{B}(v) \sqcup \{(M(v), z) \mid (M(v), z) \text{ is a back-edge with } z < v\}$ and $|\{(M(v), z) \mid (M(v), z) \text{ is a back-edge with } z < v\}| = 1$, we have $B(v) = \widetilde{B}(v) \sqcup \{(M(v), l(M(v)))\}$. And since $\widetilde{B}(v) \subseteq B(u)$ and $bcount(v) = bcount(u) + 1$, we conclude that $B(v) = B(u) \sqcup \{(M(v), l(M(v)))\}$. □

Now let us consider the case where u is the lowest proper descendant of v with $M(u) = M_{low1}(v)$. If $l(M(v)) < v$, then we have that $e = (M(v), l(M(v)))$, and $M(u) = \widetilde{M}(v)$. Thus, this possibility is included in the previous case. So we may assume that $l(M(v)) \geq v$. In this case, we have the following.

Lemma 4.13 (Case (2)). *Let $l(M(v)) \geq v$, and let u be the lowest vertex in $M^{-1}(M_{low1}(v))$ that is greater than v. Then there exists a back-edge e such that $B(v) = B(u) \sqcup \{e\}$ if and only if: $bcount(v) = bcount(u) + 1$, $low_2(M_{low2}(v)) \geq v$, and either $M(v)$ has no low3 child, or $low(c_3(M(v))) \geq v$.*

Proof. (⇒) $bcount(v) = bcount(u)+1$ is an immediate consequence of $B(v) = B(u) \sqcup \{e\}$. Now, $l(M(v)) \geq v$ implies that every back-edge $(x, y) \in B(v)$ has $x \in T(c)$, where c is a child of $M(v)$. Let (x, y) be a back-edge in $B(v)$ with $x \notin T(c_1(M(v)))$. Then $(x, y) \notin B(u)$, and therefore $B(v) = B(u) \sqcup \{e\}$ implies that $(x, y) = e$. This means that $x \in T(c_2(M(v)))$, $low_2(M_{low2}(v)) \geq v$, and either $M(v)$ has no *low3* child, or $low(c_3(M(v))) \geq v$.
(⇐) $l(M(v)) \geq v$ implies that every back-edge $(x,y) \in B(v)$ has $x \in T(c)$, where c is a child of $M(v)$. Furthermore, it implies that there exists at least one back-edge $(x, y) \in B(v)$ with $x \in T(c_2(M(v)))$. Now let $B_{low1}(v) = \{(x, y) \in B(v) \mid x \text{ is a descendant of } c_1(M(v))\}$. Then, since $low_2(M_{low2}(v)) \geq v$, and either $M(v)$ has no *low3* child, or $low(c_3(M(v))) \geq v$, we have that $B(v) = B_{low1}(v) \sqcup \{(M_{low2}(v), l(M_{low2}(v)))\}$. Since $M_{low1}(v) = M(u)$ and $v < u$, we have that $B_{low1}(v) \subseteq B(u)$. Now, from $B(v) = B_{low1}(v) \sqcup \{(M_{low2}(v), l(M_{low2}(v)))\}$, $B_{low1}(v) \subseteq B(u)$, and $bcount(v) = bcount(u)+1$, we conclude that $B(v) = B(u) \sqcup \{(M_{low2}(v), l(M_{low2}(v)))\}$. □

Finally, let us assume that we are in the case where u is the lowest proper descendant of v with $M(u) = M_{low2}(v)$. If $l(M(v)) < v$, then we have that $e = (M(v), l(M(v)))$. Thus, since $M(u) = M_{low2}(v)$ and $B(v) = B(u) \sqcup \{e\}$, we have that all back-edges in $B(v) \setminus \{e\}$ stem from $T(c_2(M(v)))$, which is impossible (because this implies that $low(c_1(M(v))) < v$, and thus there is at least one back-edge in $B(v)$ that stems from

$T(c_1(M(v)))$. Thus, we may assume that $l(M(v)) \geq v$. In this case, we have the following.

Lemma 4.14 (Case (3)). *Let $l(M(v)) \geq v$, and let u be the lowest vertex in $M^{-1}(M_{low2}(v))$ which is strictly greater than v. Then there exists a back-edge e such that $B(v) = B(u) \sqcup \{e\}$ if and only if: bcount$(v) =$ bcount$(u) + 1$, low$_2(M_{low1}(v)) \geq v$, and either $M(v)$ has no low3 child, or low$(c_3(M(v))) \geq v$.*

Proof. ($\Rightarrow$) *bcount*$(v) =$ *bcount*$(u)+1$ is an immediate consequence of $B(v) = B(u) \sqcup \{e\}$. Now, $l(M(v)) \geq v$ implies that every back-edge $(x, y) \in B(v)$ has $x \in T(c)$, where c is a child of $M(v)$. Let (x, y) be a back-edge in $B(v)$ with $x \notin T(c_2(M(v)))$. Then $(x, y) \notin B(u)$, and therefore $B(v) = B(u) \sqcup \{e\}$ implies that $(x, y) = e$. This means that $x \in T(c_1(M(v)))$, *low*$_2(M_{low1}(v)) \geq v$, and either $M(v)$ has no *low3* child, or *low*$(c_3(M(v))) \geq v$.

($\Leftarrow$) $l(M(v)) \geq v$ implies that every back-edge $(x, y) \in B(v)$ has $x \in T(c)$, where c is a child of $M(v)$. Furthermore, it implies that there exists at least one back-edge $(x, y) \in B(v)$ with $x \in T(c_1(M(v)))$. Now let $B_{low2}(v) = \{(x, y) \in B(v) \mid x \text{ is a descendant of } c_2(M(v))\}$. Then, since *low*$_2(M_{low1}(v)) \geq v$, and either $M(v)$ has no *low3* child, or *low*$(c_3(M(v))) \geq v$, we have that $B(v) = B_{low2}(v) \sqcup \{(M_{low1}(v), l(M_{low1}(v)))\}$. Since $M_{low2}(v) = M(u)$ and $v < u$, we have that $B_{low2}(v) \subseteq B(u)$. Now, from $B(v) = B_{low2}(v) \sqcup \{(M_{low1}(v), l(M_{low1}(v)))\}$, $B_{low2}(v) \subseteq B(u)$, and *bcount*$(v) =$ *bcount*$(u)+1$, we conclude that $B(v) = B(u) \sqcup \{(M_{low1}(v), l(M_{low1}(v)))\}$. □

4.3.2.2 The lower case

Let $u \neq r$ be a vertex, and suppose that there is a proper ancestor v of u with $v \neq r$ and a back-edge e such that $B(u) = B(v) \sqcup \{e\}$. Then, by Lemma 4.15 we have that v is unique. Now, if such a v exists, by Lemma 4.16 we have that one of the following holds (see also Figure 4.3): either (1) $M(v) = M(u)$, or (2) $M(v) = \widetilde{M}(u)$, or (3) $M(v) = M_{low1}(u)$. Furthermore, in either case we have $e = (highD(u), high(u))$. Thus, there are three different cases to consider in order to find v (and e). In every case, we rely on Lemma 4.17, which says that v is the greatest proper ancestor of u in $M^{-1}(M(v))$. Thus, we consider the three different cases $x = M(u)$, $x = \widetilde{M}(u)$ and $x = M_{low1}(u)$, and in each case we traverse the list $M^{-1}(x)$ in order to find the greatest proper ancestor v of u (if it exists). Then we use Lemma 4.18 in order to verify that u has indeed the desired property (it is sufficient to check only *bcount*$(u) =$ *bcount*$(v)+1$,

since x is obviously a descendant of $M(u)$). Furthermore, in each case we provide alternative and more easily computable characterizations of e. Notice that for the case $x = M(u)$ it is sufficient to check only $v = \mathit{nextM}(u)$ (and so we do not have to traverse the list $M^{-1}(x)$), because only $v = \mathit{nextM}(u)$ may have $\mathit{bcount}(v) = \mathit{bcount}(u) - 1$, as demanded by the condition in Lemma 4.18.

This is basically the idea in order to find all 3-cuts of the form $\{(u, p(u)), (v, p(v)), e\}$, where v is an ancestor of u, and e is a back-edge such that $B(u) = B(v) \sqcup \{e\}$. However, this is not a linear-time procedure, since we may have to traverse the lists $M^{-1}(x)$ an excessive number of times. In order to resolve this, we can use a similar idea as in the "upper case": that is, we process the vertices u in a bottom-up fashion, and we introduce a variable $\mathit{currentVertex}[x]$, for every vertex x, which points to the greatest element in $M^{-1}(x)$ that is a proper ancestor of u (if it exists), where u is the current vertex under process. The implementation of this idea is shown in Algorithm 12.

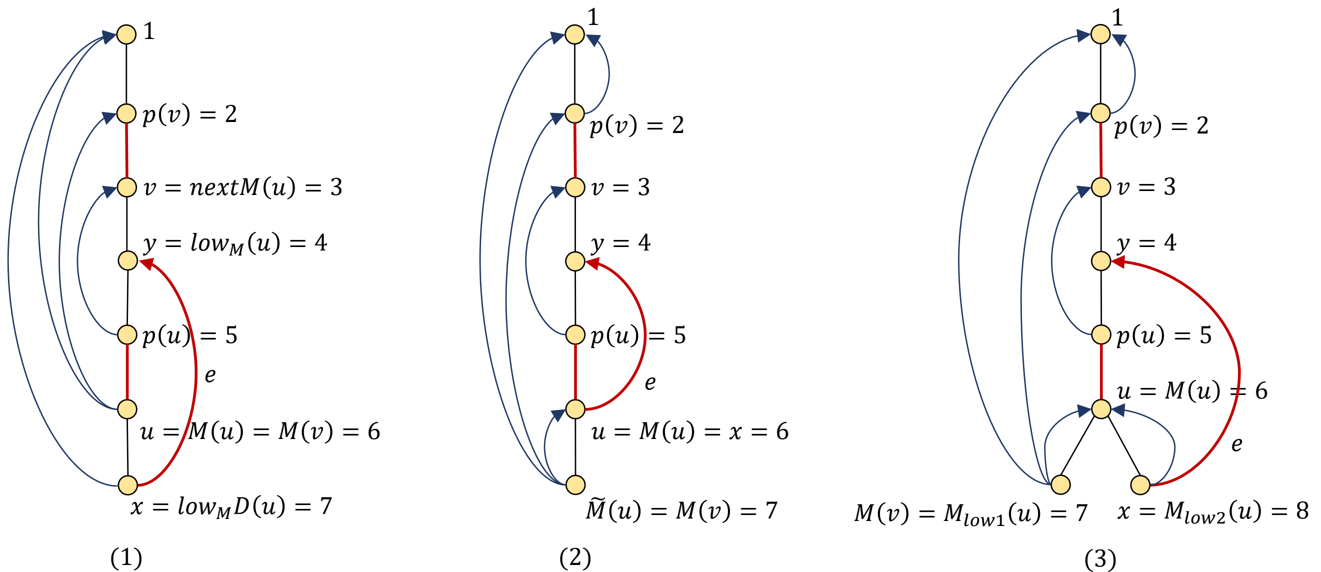


Figure 4.3: An illustration of the three cases for type-2 cuts $\{(u, p(u)), (v, p(v)), e\}$ (shown with red edges) where v is an ancestor of u, e is a back-edge, and $B(u) = B(v) \sqcup \{e\}$. Vertices are numbered in DFS order and back-edges are shown directed from descendant to ancestor in the DFS tree.

Algorithm 12: Find all 3-cuts $\{(u,p(u)),(v,p(v)),e\}$, where u is a descendant of v and $B(u)=B(v)\sqcup\{e\}$, for a back-edge e.

```
1  // x = M(v)
2  // just check whether the condition in Lemma 4.18 is satisfied for
      nextM(u)
3  if bcount(u) = bcount(nextM(u)) + 1 then
4  |  mark {(u,p(u)), (nextM(u), p(nextM(u))), (low_M D(u), low_M(u))} as a 3-cut
5  end
6  // x = M~(u)
7  for u ← n to u = 1 do
8  |  x ← M~(u)
9  |  if x = ⊥ then continue
10 |  let v be the greatest vertex in M^{-1}(x) which is lower than u
11 |  // check the condition in Lemma 4.18
12 |  if bcount(u) = bcount(v) + 1 then
13 |  |  mark {(u,p(u)), (v,p(v)), (M(u), l(M(u)))} as a 3-cut
14 |  end
15 end
16 // x = M_low1(u)
17 for u ← n to u = 1 do
18 |  x ← M_low1(u)
19 |  if x = ⊥ then continue
20 |  let v be the greatest vertex in M^{-1}(x) which is lower than u
21 |  // check the condition in Lemma 4.18
22 |  if bcount(u) = bcount(v) + 1 then
23 |  |  mark {(u,p(u)), (v,p(v)), (M_low2(u), l(M_low2(u)))} as a 3-cut
24 |  end
25 end
```

Lemma 4.15. *Let $u \neq r$ be a vertex, and suppose that there is a proper ancestor v of u with $v \neq r$ and a back-edge e such that $B(u) = B(v) \sqcup \{e\}$. Then v is unique with this property.*

Proof. Let us suppose, for the sake of contradiction, that there are two distinct proper ancestors v and v' of u, with $v, v' \neq r$, and two back-edges e and e' (not necessarily distinct), such that $B(u) = B(v) \sqcup \{e\}$ and $B(u) = B(v') \sqcup \{e'\}$. Then v and v' are related as ancestor and descendant, since they have a common descendant. Thus we may assume, without loss of generality, that v' is an ancestor of v. Let (x, y) be a back-edge in $B(v')$. Then, y is a proper ancestor of v', and therefore a proper ancestor of v. Furthermore, $B(u) = B(v') \sqcup \{e'\}$ implies that $B(v') \subseteq B(u)$, and therefore $(x, y) \in B(u)$. Thus, x is a descendant of u, and therefore a descendant of v. This shows that $(x, y) \in B(v)$, and thus we have $B(v') \subseteq B(v)$. Now, since $B(u) = B(v) \sqcup \{e\}$ and $B(u) = B(v') \sqcup \{e'\}$, we have $B(v) \sqcup \{e\} = B(v') \sqcup \{e'\}$. Therefore, $|B(v)| = |B(v')|$. In conjunction with $B(v') \subseteq B(v)$, this implies that $B(v) = B(v')$ (and $e = e'$), contradicting the fact that the graph is 3-edge-connected (see Proposition 3.1). □

Lemma 4.16. *Let u and v be two vertices $\neq r$ such that v is a proper ancestor of u with $B(u) = B(v) \sqcup \{e\}$, for a back-edge e. Then, we have that either (1) $M(v) = M(u)$, or (2) $M(v) = \widetilde{M}(u)$, or (3) $M(v) = M_{low1}(u)$. In either case, $e = (highD(u), high(u))$.*

Proof. Let $(x_1, y_1), \dots, (x_k, y_k)$ be the back-edges in $B(u)$ sorted in decreasing order w.r.t. their lower endpoint. Suppose that (x_i, y_i) is in $B(v)$ for some $i \in \{1, \dots, k\}$. Then y_i is a proper ancestor of v. This implies that y_j is a proper ancestor of v, for every $j \in \{i, \dots, k\}$. Thus, all the back-edges $(x_i, y_i), \dots, (x_k, y_k)$ are in $B(v)$, since all of $x_1, \dots, x_k$ are descendants of v. Now, $B(u) = B(v) \sqcup \{e\}$ implies that only one back-edge in $B(u)$ is not in $B(v)$. Thus, this must be (x_1, y_1). This shows that $y_1 = high(u)$. Since $(x_1, y_1) \in B(u) \setminus B(v)$ and x_1 is a descendant of v, we have that $high(u)$ is not a proper ancestor of v. Thus, (x_1, y_1) is the only back-edge of the form $(x, high(u))$ in $B(u)$, for otherwise we would have that there are at least two back-edges in $B(u) \setminus B(v)$, in contradiction to $B(u) = B(v) \sqcup \{e\}$. This shows that $(x_1, y_1) = (highD(u), high(u))$.

Let us suppose first that x_1 is a descendant of $M(v)$. Let (x, y) be a back-edge in $B(v)$. Then $B(u) = B(v) \sqcup \{e\}$ implies that $(x, y) \in B(u)$, and therefore x is a descendant of $M(u)$. Due to the generality of $(x, y) \in B(v)$, this shows that $M(v)$ is a descendant of $M(u)$. Conversely, let (x, y) be a back-edge in $B(u)$. Then $B(u) = B(v) \sqcup \{e\}$ implies that either $(x, y) = e$, or $(x, y) \in B(v)$. Thus, by our supposition

we have that x is a descendant of $M(v)$. Due to the generality of $(x,y) \in B(u)$, this implies that $M(u)$ is a descendant of $M(v)$, and so we have that $M(v) = M(u)$.

Now let us suppose that x_1 is not a descendant of $M(v)$. Let (x,y) be a back-edge in $B(v)$. Then $B(u) = B(v) \sqcup \{e\}$ implies that $(x,y) \in B(u)$. Thus, x is a descendant of $M(u)$, and this shows that $M(v)$ is a descendant of $M(u)$. Since x_1 is a descendant of $M(u)$, but not a descendant of $M(v)$, we thus have that $M(v)$ is a proper descendant of $M(u)$. Let us suppose that $x_1 = M(u)$. Then (x_1, y_1) is the unique back-edge of the form $(M(u), z)$ in $B(u)$, for otherwise we would have that $B(u) \setminus B(v)$ contains at least two back-edges. Now let (x,y) be a back-edge in $B(v)$. Then $B(u) = B(v) \sqcup \{e\}$ implies that $(x,y) \in B(u)$, and so x is a descendant of $M(u)$. But x cannot coincide with $M(u)$, for otherwise we would have that $M(v)$ is an ancestor of $M(u)$. This shows that x is a descendant of $\widetilde{M}(u)$, and so we have that $M(v)$ is a descendant of $\widetilde{M}(u)$. Conversely, let (x,y) be a back-edge in $B(u)$ such that $x \neq M(u)$. Then we have that $(x,y) \neq e$, and so $B(u) = B(v) \sqcup \{e\}$ implies that $(x,y) \in B(v)$. This implies that x is a descendant of $M(v)$. Due to the generality of $(x,y) \in B(u)$ with $x \neq M(u)$, we thus have that $\widetilde{M}(u)$ is a descendant of $M(v)$. This shows that $M(v) = \widetilde{M}(u)$.

Finally, let us suppose that x_1 is not a descendant of $M(v)$ and $x_1 \neq M(u)$. Let us suppose, for the sake of contradiction, that there is a back-edge of the form $(M(u), z)$ in $B(u)$. This implies that $(M(u), z) \neq e$, and so $B(u) = B(v) \sqcup \{e\}$ implies that $(M(u), z) \in B(v)$. But this implies that $M(u)$ is a descendant of $M(v)$, contradicting the fact that $M(v)$ is a proper descendant of $M(u)$. This shows that there are no back-edges of the form $(M(u), z)$ in $B(u)$. Thus we have that $low(c_2(M(u))) < u$ (for otherwise we would have that all the back-edges that leap over u stem from $T(c_1(M(u)))$, which would imply that $M(u)$ is a descendant of $c_1(M(u))$, which is absurd). This means that there is an $i > 1$ and a back-edge (x,y) such that x is a descendant of $c_i(M(u))$ and y is a proper ancestor of u. Then we have that $(x,y) \in B(u)$. Let us suppose, for the sake of contradiction, that $(x,y) \neq e$. Then $B(u) = B(v) \sqcup \{e\}$ implies that $(x,y) \in B(v)$, and so x is a descendant of $M(v)$. Furthermore, y is a proper ancestor of v, and thus $low(c_i(M(u))) < v$. Since $low(c_2(M(u))) \leq low(c_i(M(u)))$, we have that $low(c_1(M(u))) < v$. Thus, there is a back-edge (x', y') such that x' is a descendant of $c_1(M(u))$ and $y' < v$. But this implies that $(x', y') \in B(v)$, and so x' is a descendant of $M(v)$. Now we have that $M(v)$ is an ancestor of a descendant of $c_1(M(u))$ (i.e., x), and an ancestor of a descendant of $c_i(M(u))$ (i.e., x'), where $i \neq 1$. This implies that $M(v)$ is an ancestor of $M(u)$, a contradiction. Thus we have

shown that the only back-edge in $B(u)$ whose higher endpoint is not a descendant of $c_1(M(u))$ is e. Since such a back-edge must necessarily exist (since there are no back-edges of the form $(M(u), z)$ in $B(u)$), we have that e stems from $T(c_2(M(u)))$.

Now let (x, y) be a back-edge in $B(v)$. Then $B(u) = B(v) \sqcup \{e\}$ implies that $(x, y) \in B(u) \setminus \{e\}$. This implies that x is a descendant of $c_1(M(u))$. Thus, since $(x, y) \in B(u)$, we have that x is a descendant of $M_{low1}(u)$. Due to the generality of $(x, y) \in B(v)$, this shows that $M(v)$ is a descendant of $M_{low1}(u)$. Conversely, let (x, y) be a back-edge in $B(u)$ such that x is a descendant of $c_1(M(u))$. Then we have that $(x, y) \neq e$, and so $B(u) = B(v) \sqcup \{e\}$ implies that $(x, y) \in B(v)$. This implies that x is a descendant of $M(v)$. Due to the generality of $(x, y) \in B(u)$ with x a descendant of $c_1(M(u))$, this shows that $M_{low1}(u)$ is a descendant of $M(v)$. We conclude that $M(v) = M_{low1}(u)$.

□

Lemma 4.17. *Let u and v be two vertices $\neq r$ such that v is a proper ancestor of u. If there is a back-edge e such that $B(u) = B(v) \sqcup \{e\}$, then v is the greatest element in $M^{-1}(M(v))$ which is lower than u.*

Proof. Notice that $B(u) = B(v) \sqcup \{e\}$ implies that $M(u)$ is an ancestor of $M(v)$. Now let us suppose, for the sake of contradiction, that there is a vertex v' with $M(v') = M(v)$ which is greater than v and lower than u. Then $M(v') = M(v)$ implies that $B(v) \subseteq B(v')$. And since the graph is 3-edge-connected, we have $B(v) \subset B(v')$. This implies that there is a back-edge $(x, y) \in B(v') \setminus B(v)$. Then we have that x is a descendant of $M(v') = M(v)$, and therefore a descendant of $M(u)$. Furthermore, y is a proper ancestor of v' and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Then, $B(u) = B(v) \sqcup \{e\}$ and $(x, y) \notin B(v)$ implies that $(x, y) = e$. Notice that the previous argument also shows that $B(v') \subseteq B(u)$. And since the graph is 3-edge-connected, we have $B(v') \subset B(u)$. Thus, there is a back-edge $e' \in B(u) \setminus B(v')$. Since $e \in B(u) \cap B(v')$, we have that $e' \neq e$. Thus, $B(u) = B(v) \sqcup \{e\}$ implies that $e' \in B(v)$. But then $B(v) \subset B(v')$ implies that $e \in B(v')$, a contradiction. Thus we have that v is the greatest element in $M^{-1}(M(v))$ which is lower than u. □

Lemma 4.18. *Let u and v be two vertices $\neq r$ such that v is a proper ancestor of u. Then there is a back-edge e such that $B(u) = B(v) \sqcup \{e\}$ if and only if $M(v)$ is a descendant of $M(u)$ and $bcount(u) = bcount(v) + 1$.*

Proof. (⇒) *bcount*$(u) = $ *bcount*$(v)+1$ is an immediate consequence of $B(u) = B(v) \sqcup \{e\}$. Now let (x, y) be a back-edge in $B(v)$. Then $B(u) = B(v) \sqcup \{e\}$ implies that $(x, y) \in B(u)$, and so x is a descendant of $M(u)$. Due to the generality of $(x, y) \in B(v)$, this shows that $M(v)$ is a descendant of $M(u)$.

(⇐) Let (x, y) be a back-edge in $B(v)$. Then x is a descendant of $M(v)$, and therefore a descendant of $M(u)$. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$, and so we have $B(v) \subseteq B(u)$. Now *bcount*$(u) = $ *bcount*$(v) + 1$ implies that there is a back-edge e such that $B(u) = B(v) \sqcup \{e\}$. □

Now we provide different formulations for the back-edge e that satisfies $B(u) = B(v) \sqcup \{e\}$, which we can use instead of $(\mathit{highD}(u), \mathit{high}(u))$. According to Lemma 4.16, there are three cases to consider: either $M(v) = M(u)$, or $M(v) = \widetilde{M}(u)$, or $M(v) = M_{low1}(u)$. In the case where $M(v) = M(u)$, since $B(u) = B(v) \sqcup \{e\}$ we have $v = \mathit{nextM}(u)$. Thus, since e is the unique back-edge in $B(u) \setminus B(v)$, we have that e must coincide with $(\mathit{low}_M D(u), \mathit{low}_M(u))$. This explains Line 4 of Algorithm 12. Now let us consider the case where $M(v) = \widetilde{M}(u)$. Let us assume that $\widetilde{M}(u) \neq M(u)$, for otherwise we are back in the previous case. Then $\widetilde{M}(u) \neq M(u)$ implies that there is a back-edge of the form $(M(u), z)$, with $z < u$. Then, since $B(u) = B(v) \sqcup \{e\}$, this is the only back-edge of this form, and it coincides with e. Thus, $e = (M(u), l(M(u)))$. This explains Line 13 of Algorithm 12. Finally, let us consider the case where $M(v) = M_{low1}(u)$. Let us assume that $M_{low1}(u) \neq \widetilde{M}(u)$, for otherwise we are in the previous case. Then, since only one back-edge in $B(u)$ is not in $B(v)$, this must stem from the subtree of the *low2* child of $M(u)$, and thus it coincides with $(M_{low2}(u), l(M_{low2}(u)))$. This explains Line 23 of Algorithm 12.

4.3.3 Computing Type-3 3-cuts

In order to compute the Type-3 3-cuts, we use the idea described in [50], for reducing this case to the previous two types of 3-cuts. This idea works as follows. First, we remove all the tree-edges from the graph, and we compute the connected components. Then we contract every one of those components into a single node, and we simply recurse the computation of 3-cuts to the quotient graph. The proof of correctness and overall linear complexity follows from Lemma 5.26.

4.4 Computing the 4-edge-connected components

Now we consider how to compute the 4-edge-connected components of an undirected graph G in linear time. First, we reduce this problem to the computation of the 4-edge-connected components of a collection of auxiliary 3-edge-connected graphs.

4.4.1 Reducing the computation to 3-edge-connected graphs

Given an undirected graph G, we execute the following steps:

1. Compute the connected components of G.
2. For each connected component, compute its 2-edge-connected components (which are also 2-edge-connected components of G).
3. For each 2-edge-connected component, compute its 3-edge-connected components $C_1, \ldots, C_\ell$.
4. For each 3-edge-connected component C_i, compute a 3-edge-connected auxiliary graph H_i, such that for any two vertices x and y of G, we have that x and y are k-edge-connected in G, for $k \geq 3$, if and only if there is an $i' \in \{1, \ldots, t\}$ such that x and y are both vertices of $H_{i'}$ and they are k-edge-connected in $H_{i'}$.
5. Finally, compute the 4-edge-connected components of each H_i.

Steps 1–3 take overall linear time [63, 67]. We describe step 5 in the next section, so it remains to give the details of step 4. In order to perform this step efficiently, we rely on a construction described by Dinitz [21]. Let H be a 2-edge-connected component (subgraph) of G. We can construct a compact representation of the 2-cuts of H, which allows us to compute its 3-edge-connected components $C_1, \ldots, C_\ell$ in linear time [67]. Now, since the collection $\{C_1, \ldots, C_\ell\}$ constitutes a partition of the vertex set of H, we can form the quotient graph Q of H by shrinking each C_i into a single node. Graph Q has the structure of a tree of cycles [21]; in other words, Q is connected and every edge of Q belongs to a unique cycle. Let (C_i, C_j) and (C_i, C_k) be two edges of Q which belong to the same cycle. Then (C_i, C_j) and (C_i, C_k) correspond to two edges (x, y) and (x', y') of G, with $x, x' \in C_i$, $y \in C_j$ and $y' \in C_k$. If $x \neq x'$, then we add a virtual edge (x, x') to $G[C_i]$. (The idea is to attach (x, x') to $G[C_i]$ as a substitute for the cycle of Q that contains (C_i, C_j) and (C_i, C_k).) Now let $\overline{C_i}$ be the graph $G[C_i]$ plus all those

virtual edges. Then $\overline{C_i}$ is 3-edge-connected and its k-edge-connected components, for $k \geq 3$, are precisely those of G that are contained in C_i [21]. Thus we can compute the 4-edge-connected components of G by computing the 4-edge-connected components of the graphs $\overline{C_1}, \ldots, \overline{C_\ell}$ (which can easily be constructed in total linear time). Since every $\overline{C_i}$ is 3-edge-connected, we can apply Algorithm 13 of the following section in order to compute its 4-edge-connected components in linear time.

4.4.2 Splitting a 3-edge-connected graph according to its 3-cuts

Now we describe how to compute the 4-edge-connected components of a 3-edge-connected graph G in linear time. Let r be any fixed vertex of G, and let C be a minimum cut of G. By removing C from G, G becomes disconnected into two connected components. We let V_C denote the connected component of $G \setminus C$ that does not contain r, and we refer to the number of vertices of V_C as the *r-size* of the cut C. (Notice that these notions are relative to r.)

Let $G = (V, E)$ be a 3-edge-connected graph, and let $\mathcal{C}$ be the collection of the 3-cuts of G. If the collection $\mathcal{C}$ is empty, then G is 4-edge-connected, and V is the only 4-edge-connected component of G. Otherwise, let $C \in \mathcal{C}$ be a 3-cut of G. By removing C from G, G is separated into two connected components, and every 4-edge-connected component of G lies entirely within a connected component of $G \setminus C$. This observation suggests a recursive algorithm for computing the 4-edge-connected components of G, by successively splitting G into smaller graphs according to its 3-cuts. Thus, we start with a 3-cut C of G, and we perform the splitting operation shown in Figure 4.4. Then we take another 3-cut C' of G and we perform the same splitting operation on the part which contains (the corresponding 3-cut of) C'. We repeat this procedure until we have considered every 3-cut of G. When no more splits are possible, the connected components of the final split graph correspond (by ignoring the newly introduced vertices) to the 4-edge-connected components of G.

To implement this procedure in linear time, we must take care of two things. First, whenever we consider a 3-cut C of G, we have to be able to know which ends of the edges of C belong to the same connected component of $G \setminus C$. And second, since an edge e of a 3-cut of the original graph may correspond to two virtual edges of the split graph, we have to be able to know which is the virtual edge that corresponds to e. We tackle both these problems by locating the 3-cuts of G on a DFS tree T of G

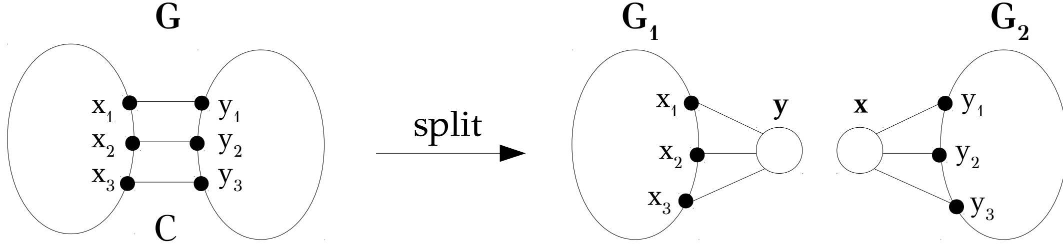


Figure 4.4: $C = \{(x_1,y_1),(x_2,y_2),(x_3,y_3)\}$ is a 3-cut of G, with $\{x_1,x_2,x_3\}$ and $\{y_1,y_2,y_3\}$ lying in different connected components of $G \setminus C$. The split operation of G at C consists of the removal of the edges of C from G, and the introduction of two new nodes x, y, and six virtual edges $(x_1,y),(x_2,y),(x_3,y),(x,y_1),(x,y_2),(x,y_3)$. Now the split graph is made of two connected components, G_1 and G_2. Every 3-cut $C' \neq C$ of G (or more precisely: a 3-cut that corresponds to C') lies entirely within G_1 or G_2. Conversely, every 3-cut of either G_1 or G_2 corresponds to a 3-cut of G. Thus, every 4-edge-connected component of G lies entirely within G_1 or G_2.

rooted at r, and by processing them in increasing order with respect to their r-size. By locating a 3-cut $C \in \mathcal{C}$ on T we can answer in $O(1)$ time which ends of the edges of C belong to the same connected component of $G \setminus C$. And then, by processing the 3-cuts of G in increasing order with respect to their size, we ensure that (the 3-cut that corresponds to) a 3-cut $C \in \mathcal{C}$ that we process lies in the split part of G that contains r.

Now, due to the analysis in the preceding sections, we can distinguish the following types of 3-cuts on a DFS tree T (see also Figure 4.1):

- (I) $\{(v,p(v)),(x_1,y_1),(x_2,y_2)\}$, where (x_1,y_1) and (x_2,y_2) are back-edges.
- (IIa) $\{(u,p(u)),(v,p(v)),(x,y)\}$, where u is a descendant of v and $(x,y) \in B(v)$.
- (IIb) $\{(u,p(u)),(v,p(v)),(x,y)\}$, where u is a descendant of v and $(x,y) \in B(u)$.
- (III) $\{(u,p(u)),(v,p(v)),(w,p(w))\}$, where w is an ancestor of both u and v, but u, v are not related as ancestor and descendant.

- (IV) $\{(u,p(u)),(v,p(v)),(w,p(w))\}$, where u is a descendant of v and v is a descendant of w.

Let r be the root of T. Then, for every 3-cut $C \in \mathcal{C}$, V_C is either $T(v)$, or $T(v) \setminus T(u)$, or $T(w) \setminus (T(u) \cup T(v))$, or $T(u) \cup (T(w) \setminus T(v))$, depending on whether C is of type (I), (II), (III), or (IV), respectively. Thus we can immediately calculate the size of C and the endpoints of its edges that lie in V_C. In particular, the size of C is either $ND(v)$, or $ND(v) - ND(u)$, or $ND(w) - ND(u) - ND(v)$, or $ND(u) + ND(w) - ND(v)$, depending on whether it is of type (I), (II), (III), or (IV), respectively; V_C contains either $\{v, x_1, x_2\}$, or $\{p(u), v, x\}$, or $\{p(u), v, y\}$, or $\{p(u), p(v), w\}$, or $\{u, p(v), w\}$, depending on whether C is of type (I), (IIa), (IIb), (III), or (IV), respectively.

Algorithm 13 shows how we can compute the 4-edge-connected components of G in linear time, by repeatedly splitting G into smaller graphs according to its 3-cuts. When we process a 3-cut C of G, we have to find the edges of the split graph that correspond to those of C, in order to delete them and replace them with (new) virtual edges. That is why we use the symbol v', for a vertex $v \in V$, to denote a vertex that corresponds to v in the split graph. (Initially, we set $v' \leftarrow v$.) Now, if (x,y) is an edge of C with $x \in V_C$, the edge of the split graph corresponding to (x,y) is (x',y'). Then we add two new vertices v_C and $\widetilde{v_C}$ to G, and the virtual edges $(x', \widetilde{v_C})$ and (v_C, y'). Finally, we let x correspond to v_C, and so we set $x' \leftarrow v_C$. This is sufficient, since we process the 3-cuts of G in increasing order with respect to their r-size, and so the next time we meet the edge (x,y) in a 3-cut, we can be certain that it corresponds to (v_C, y'). The correctness of this procedure is established with the same argument as Proposition 5.5, and it relies on the fact that the collection of 3-cuts of a 3-edge-connected graph is a parallel family of 3-cuts [24].

4.5 Testing 4-edge connectivity

In order to check whether a graph is 4-edge-connected, we first apply any of the known linear-time algorithms for testing 3-edge-connectivity (e.g., [63, 67]). Thus, let us assume that the graph is 3-edge-connected. Then we only have to check whether the graph has a 3-cut. To do this, we use a DFS-tree as in the previous sections. The cases of Type-1 and Type-3 3-cuts are the easiest ones: the computation of Type-1 3-cuts is in-itself simple, and that of the Type-3 3-cuts can be easily reduced to

Algorithm 13: Compute the 4-edge-connected components of a 3-edge-connected graph $G = (V, E)$

1. Find the collection $\mathcal{C}$ of the 3-cuts of G
2. Locate and classify the 3-cuts of G on a DFS tree of G rooted at r
3. For every $C \in \mathcal{C}$, calculate $size(C)$ (relative to r)
4. Sort $\mathcal{C}$ in increasing order w.r.t. the *size* of its elements
5. **foreach** $v \in V$ **do** Set $v' \leftarrow v$
6. **foreach** $C = \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\} \in \mathcal{C}$ **do**
7. Find the ends of the edges of C that lie in V_C // *Let those ends be* x_1,x_2 *and* x_3
8. Remove the edges (x'_1, y'_1),(x'_2, y'_2),(x'_3, y'_3) from G
9. Introduce two new vertices v_C and $\widetilde{v_C}$ to G
10. Add the edges $(x'_1, \widetilde{v_C})$,$(x'_2, \widetilde{v_C})$,$(x'_3, \widetilde{v_C})$,(v_C, y'_1),(v_C, y'_2),(v_C, y'_3) to G
11. Set $x'_1 \leftarrow v_C$, $x'_2 \leftarrow v_C$, $x'_3 \leftarrow v_C$
12. **end**
13. Output the connected components of G, ignoring the newly introduced vertices

the previous two cases. Thus, we only have to provide an algorithm that checks the existence of a Type-2 3-cut. Here, again, we distinguish between the upper and the lower case, and we discuss how to handle each in Sections 4.5.1 and 4.5.2, respectively.

4.5.1 The upper case

Here we provide a method to determine whether there exist vertices u, v and a back-edge e, such that u is a descendant of v and $B(v) = B(u) \sqcup \{e\}$. We have the following:

Lemma 4.19. *Let u, v be two vertices ($\neq r$) such that u is a descendant of v with $B(v) = B(u) \sqcup \{e\}$, for a back-edge e. Then the nearest ancestor w of u with $bcount(w) = bcount(u) + 1$ satisfies $B(w) = B(u) \sqcup \{e'\}$, for a back-edge e'.*

Proof. Let w be the nearest ancestor of u with $bcount(w) = bcount(u) + 1$. ($w$ exists since v is an ancestor of u with $bcount(v) = bcount(u) + 1$; furthermore, we have that w is a descendant of v.) Let (x, y) be a back-edge in $B(u)$. Then $B(v) = B(u) \sqcup \{e\}$ implies that $(x, y) \in B(v)$. Now, $(x, y) \in B(u)$ implies that x is a descendant of u,

and therefore it is a descendant of w. Furthermore, $(x, y) \in B(v)$ implies that y is a proper ancestor of v, and therefore it is a proper ancestor of w. This means that $(x, y) \in B(w)$, and so we have $B(u) \subseteq B(w)$. Now $\mathit{bcount}(w) = \mathit{bcount}(u) + 1$ implies that there exists a back-edge e' such that $B(w) = B(u) \sqcup \{e'\}$. □

Thus we may proceed as follows. According to Lemma 4.19, we only need to find, for every vertex $u \neq r$, the nearest ancestor v of u that satisfies $\mathit{bcount}(v) = \mathit{bcount}(u) + 1$ (if it exists). Then we may use Lemma 3.11 in order to check whether $B(v) = B(u) \sqcup \{e\}$, for a back-edge e. Now, to perform this search efficiently, for every vertex $v \neq r$ we find all descendants u of v for which v is the nearest ancestor with $\mathit{bcount}(v) = \mathit{bcount}(u) + 1$. Thus we need to have fast access to the vertices that have a specific number of leaping back-edges, and so we keep in a stack $\mathit{stack}[k]$ all vertices u we have encountered that have $\mathit{bcount}(u) = k$. (We need as many such stacks as there are edges in the graph.)

Now we process all vertices in a bottom-up fashion. For every vertex $v \neq r$ that we process, we push v in $\mathit{stack}[\mathit{bcount}(v)]$. (This means that the vertices in every stack $\mathit{stack}[k]$ are popped out in increasing order.) Then we pop out every element u of $\mathit{stack}[\mathit{bcount}(v) - 1]$, as long as it is a descendant of v, which is the case if and only if $u < v + \mathit{ND}(v)$. (If we meet a u that is not a descendant of v, then this means that $u \geq v + \mathit{ND}(v)$, and so no subsequent element in $\mathit{stack}[\mathit{bcount}(v) - 1]$ is a descendant of v.) This ensures that v is the greatest ancestor of u that has $\mathit{bcount}(v) = \mathit{bcount}(u) + 1$. (And conversely: for every vertex u, this process ensures that we will eventually find the greatest ancestor v of u that has $\mathit{bcount}(v) = \mathit{bcount}(u) + 1$, if it exists.) Then we check whether $B(v) = B(u) \sqcup \{e\}$, for a back-edge e, using the leftmost and rightmost points of v, according to Lemma 3.11. The implementation of this idea is shown in Algorithm 14.

4.5.2 The lower case

Here we provide a method to determine whether there exist vertices u, v and a back-edge e, such that u is a descendant of v and $B(u) = B(v) \sqcup \{e\}$. We have the following.

Lemma 4.20. *Let u, v be two vertices ($\neq r$) such that u is a descendant of v and $B(u) = B(v) \sqcup \{e\}$, for a back-edge e. Then v is the nearest ancestor of u with $\mathit{bcount}(v) = \mathit{bcount}(u) - 1$.*

Algorithm 14: Check whether there exists a 3-cut of the form $\{(u, p(u)), (v, p(v)), e\}$, for a back-edge e, where u is a descendant of v and $B(v) = B(u) \sqcup \{e\}$

```
1  initialize an array stack of empty stacks, of size m
2  for v ← n to 2 do
3      stack[bcount(v)].push(v)
4      while stack[bcount(v) − 1].top() is a descendant of v do
5          u ← stack[bcount(v) − 1].pop()
6          if L1(v) ∉ T(u) and L2(v) ∈ T(u) and R1(v) ∈ T(u) then
7              return true // the graph is not 4-edge-connected
8          end
9          if R1(v) ∉ T(u) and R2(v) ∈ T(u) and L1(v) ∈ T(u) then
10             return true // the graph is not 4-edge-connected
11         end
12     end
13 end
14 return false
```

Proof. $bcount(v) = bcount(u) - 1$ is an immediate consequence of $B(u) = B(v) \sqcup \{e\}$. Now suppose, for the sake of contradiction, that there exists an ancestor w of u such that $bcount(w) = bcount(u) - 1$ and w is a proper descendant of v. Let (x, y) be a back-edge in $B(v)$. Then we have $(x, y) \in B(u)$. This implies that x is a descendant of u, and therefore a descendant of w. Furthermore, since $y < v$ and $v < w$, we have that y is a proper ancestor of w. This shows that $(x, y) \in B(w)$. Thus, all the back-edges from $B(u) \setminus \{e\}$ are in $B(w)$. Since $bcount(w) = bcount(u) - 1$, this means that $B(w) = B(u) \setminus \{e\}$. Thus we have $B(w) = B(v)$, contradicting the fact that the graph is 3-edge-connected. □

Lemma 4.21. *Let u, v be two vertices ($\neq r$) such that v is an ancestor of u with $bcount(v) = bcount(u) - 1$. Then $B(u) = B(v) \sqcup \{e\}$, for a back-edge e, if and only if $B(v) \subseteq B(u)$.*

Proof. The equivalence is obvious. □

Thus we may proceed as follows. According to Lemma 4.20, it is sufficient to find, for every $u \neq r$, the nearest ancestor v of u that has $bcount(v) = bcount(u) - 1$ (if it

exists). Then, according to Lemma 4.21, we only have to check whether $B(v) \subseteq B(u)$. Now, to perform this search efficiently, for every vertex $v \neq r$ we find all descendants u of v for which v is the nearest ancestor with $bcount(u) = bcount(v) + 1$. Thus we need to have fast access to the vertices that have a specific number of leaping back-edges, and so we keep in a stack $stack[k]$ all vertices u we have encountered that have $bcount(u) = k$. (We need as many such stacks as there are edges in the graph.)

Now the idea is the same as in Algorithm 14 that we used for the upper case. Thus, we process all vertices in a bottom-up fashion. For every vertex $v \neq r$ that we process, we push v in $stack[bcount(v)]$. (This means that the vertices in every stack $stack[k]$ are popped out in increasing order.) Then we pop out every element u of $stack[bcount(v) + 1]$, as long as it is a descendant of v, which is the case if and only if $u < v + ND(v)$. (If we meet a u that is not a descendant of v, then this means that $u \geq v + ND(v)$, and so no subsequent element in $stack[bcount(v) + 1]$ is a descendant of v.) Then we simply check whether $B(v) \subseteq B(u)$ by using the leftmost and rightmost points of v, according to Lemma 3.10. The implementation of this idea is shown in Algorithm 15.

Algorithm 15: Check whether there exists a 3-cut of the form $\{(u, p(u)), (v, p(v)), e\}$, for a back-edge e, where u is a descendant of v and $B(u) = B(v) \sqcup \{e\}$

```
1  initialize an array stack of empty stacks, of size m
2  for v ← n to 2 do
3      stack[bcount(v)].push(v)
4      while stack[bcount(v) + 1].top() is a descendant of v do
5          u ← stack[bcount(v) + 1].pop()
6          if L1(v) ∈ T(u) and R1(v) ∈ T(u) then
7              return true // the graph is not 4-edge-connected
8          end
9      end
10 end
11 return false
```

Chapter 5

Computing the 5-Edge-Connected Components

5.1 Introduction

5.2 Properties of 4-cuts in 3-edge-connected graphs

5.3 Using a DFS-tree for some problems concerning 4-cuts

5.4 Computing the 5-edge-connected components

5.5 Computing a complete collection of 4-cuts

5.6 Computing Type-2 4-cuts

5.7 Computing Type-3α 4-cuts

5.8 Computing Type-3β 4-cuts

5.1 Introduction

5.1.1 Problem definition

Let $G = (V, E)$ be an undirected multigraph. We say that two vertices x and y of G are k-edge-connected if we have to remove at least k edges from G in order to destroy all paths from x to y. In general, a set of k edges with the property that its removal from G disconnects at least one pair of vertices, is called a k-edge cut of G. Equivalently, by Menger's theorem we have that x and y are k-edge-connected if

there are at least k edge-disjoint paths from x to y (see, e.g., [52]). We denote this condition at $x \equiv_k y$. It is easy to see that $\equiv_k$ is an equivalence relation on V. The equivalence classes of $\equiv_k$ are called the k-edge-connected components of G.

Determining the k-edge-connectivity relation is a fundamental graph connectivity problem. The case $k = 1$ coincides with the computation of the connected components, and can be solved easily with a standard graph traversal (like BFS or DFS). For $k = 2$, Tarjan [63] provided a linear-time algorithm, that essentially finds all the bridges of the graph. The case $k = 3$ was initially solved in linear time through the reduction of Galil and Italiano [34] from the triconnectivity algorithm of Hopcroft and Tarjan [45]. Afterwards, more linear-time algorithms were developed for $k = 3$, that did not rely on this reduction and were much simpler (see e.g., [52, 67]). Relatively recently, linear-time algorithms for the case $k = 4$ were presented [36, 50]. Although we are not aware of any specific application of the case $k = 5$ (or beyond), it is not known if we can compute the k-edge-connected components in linear time for $k \geq 5$ (not even with randomized algorithms), and this seems to be an intriguing problem. Thus, considering the case $k = 5$ seems to be the natural next step in order to determine whether this computation is possible in linear time for general fixed k.

5.1.2 Related work

The best time bounds that we have for computing the 5-edge-connected components are almost linear, and they are derived from solutions of more general versions of the problem that we consider. Specifically, Dinitz and Nossenson [23] have provided an algorithm for maintaining the relation of 5-edge-connectivity in incremental graphs. More precisely, starting from an empty graph, they show how to process a sequence of n insertions of vertices and m insertions of edges in $O(m + n \log^2 n)$ time in total, so that, at any point in this sequence, we can answer 5-edge-connectivity queries for pairs of vertices in constant time. Since the relation of 5-edge-connectivity with this algorithm is essentially maintained with the use of a disjoint-set union data structure (DSU), we can use this incremental algorithm in order to report the 5-edge-connected components of a graph G, after we have started from the empty graph and we have inserted from the beginning all the edges of G. Thus, we have an $O(m + n \log^2 n)$-time algorithm for computing the 5-edge-connected components. Since this comes from an incremental algorithm, it is reasonable to expect that this computation can

be performed even faster on a static graph. It seems difficult to achieve this from the work of Dinitz and Nossenson for the following reasons. First, this comes from an extended abstract, but we were not able to find the journal version that would contain the full details. And second, this algorithm relies on the 2-level cactus of the ($minimum + 1$)-cuts [24], which is quite involved, and we do not know how to construct it in linear time. Instead, we start anew the analysis of the structure of 4-cuts in 3-edge-connected graphs, that enables us to compute enough of them in linear time, so that we can derive the 5-edge-connected components.

We note that the problem of maintaining the k-edge-connectivity relation in dynamic graphs is a problem that has received a lot of attention. First, for the case $k \in \{2, 3\}$ there are optimal and almost optimal solutions for incremental graphs, that process a sequence of n vertex insertions and m edge insertions and queries in $O(n + m\alpha(m, n))$ time, where α is an inverse of Ackermann's function (see [69, 35, 59, 58]). For $k = 4$, Dinitz and Westbrook [25] presented an algorithm that processes a sequence of n vertex insertions and m edge insertions in $O(m + n \log n)$ time, so that, in the meantime, we can answer 4-edge-connectivity queries in constant time. Very recently, Jin and Sun [46] presented a deterministic algorithm for answering k-edge-connectivity queries in a fully dynamic graph in $n^{o(1)}$ worst case update and query time for any positive integer $k = (\log n)^{o(1)}$ for a graph with n vertices. This is a very remarkable result, but it is highly complicated, and it does not seem to provide an algorithm for computing the k-edge-connected components in, say, $O(n \cdot n^{o(1)})$ time, because it only computes the answer to the queries in response to them, without maintaining explicitly the k-edge-connected components (as do the algorithms e.g. in [69, 35, 59, 25]). However, this result, since it applies to fully dynamic graphs, makes it seem reasonable that the k-edge-connected components can be computed in (almost) linear time, for general fixed k.

Another route for computing the k-edge-connected components is given by Gomory-Hu trees [39]. A Gomory-Hu tree of a graph G is a weighted tree on the same vertex set as G, with the property that (1) the minimum weight of an edge on the tree-path that connects any two vertices x and y coincides with the edge-connectivity of x and y in G, and (2) by taking the connected components of the tree after removing such an edge we get a minimum cut of G that separates x and y. Thus, given a Gomory-Hu tree, we can easily compute the k-edge-connected components in linear time, for any fixed k, by simply removing all edges with weight less than k from the

tree, and then gathering the connected components. However, the computation of the Gomory-Hu tree itself is very demanding. The original algorithm of Gomory and Hu can take as much as $\Omega(mn)$ time for a graph with m edges and n vertices. In a recent breakthrough, Abboud et al. [2] provided a randomized Monte Carlo construction of Gomory-Hu trees that takes $\widetilde{O}(n^2)$[1] time in general weighted graphs with n vertices. Furthermore, using the recent $m^{1+o(1)}$-time max-flow algorithm of Chen et al. [17], Abboud et al. [2] provide a randomized Monte Carlo algorithm that runs in $m^{1+o(1)}$ time in unweighted graphs with m edges. More recently, Abboud et al. [4] extended this result to weighted graphs, and provided an $m^{1+o(1)}$-time construction of a Gomory-Hu tree. Thus, we can compute the k-edge-connected components of a graph, for any fixed k, with a randomized Monte Carlo algorithm in $m^{1+o(1)}$ time. For our purposes, it seems more fitting to use a *partial* Gomory-Hu tree, introduced by Hariharan et al. [41]. This has the same properties as a general Gomory-Hu tree (i.e., (1) and (2)), but it captures the k-edge-connectivity relation only up to a bounded k. Hariharan et al. [41] showed how to compute a partial Gomory-Hu tree, for edge-connectivity up to a fixed k, in expected $O(m + kn \log n)$ time. Thus, we get an algorithm of expected $O(m + kn \log n)$ time for computing the k-edge-connected components, for any fixed k.

From this general overview of the history of this subject (which omits several other related advances, such as determining the vertex-connectivity relation [54], or computing the k-edge-connected components in directed graphs [37]), we can see that determining various notions of edge-connectivity is an area of active interest. However, the precise computation of the k-edge-connectivity relation in linear time, for general fixed k, is still an elusive open problem, that demands a deeper understanding of the structure of cuts in undirected graphs.

5.1.3 Our contribution

Here we present a deterministic linear-time algorithm for computing the 5-edge-connected components of an undirected multigraph. This result relies on a novel analysis of the structure of 4-cuts in 3-edge-connected graphs. This analysis is crucial in order to guide us to a selection of enough 4-cuts that can provide the partition of the 5-edge-connected components. The second half of this work is devoted to

[1]The $\widetilde{O}$ notation hides polylogarithmic factors.

the development of a linear-time algorithm that computes a compact representation of all 4-cuts of a 3-edge-connected graph. (The precise meaning of this term will be given in the Technical Overview, in the following section.) The state of the art in deterministically computing even a single 4-cut is the algorithm of Gabow [32] that runs in $O(m + n \log n)$ time in a graph with n vertices and m edges. Thus, we present the first deterministic algorithm that computes a 4-cut of a graph, and tests the 5-edge-connectivity in linear time.

In addition to computing the 5-edge-connected components, we also provide a linear-time construction of an oracle that can answer in constant time queries of the form "given two vertices x and y, report a 4-cut that separates x and y, or determine that no such 4-cut exists" (see Corollary 5.11). In essence, we provide a data structure that retains the full functionality of a partial Gomory-Hu tree for 5-edge-connectivity.

An indispensable tool in our analysis is the concept of the *essential* 4-cuts. These are the 4-cuts that separate at least one pair of vertices that are 4-edge-connected. We do not know if this concept (or its generalization) has been used before in the literature, but it is reasonable to care about the essential 4-cuts when we want to compute the relation of 5-edge-connectivity. In fact, by retaining only the essential 4-cuts in the end, we have some convenient properties that enable us to derive efficiently the partition of the 5-edge-connected components. We show how to process a graph in linear time, so that we can check the essentiality of any given 4-cut in constant time. This relies on an oracle for answering connectivity queries in the presence of at most four edge-failures (see Proposition 6.1).

Finally, we note that our algorithm for computing the 5-edge-connected components, although it is quite extensive and broken up into several pieces, has an almost linear-time implementation with the use of elementary data structures. Specifically, the only sophisticated data structures that we use in order to achieve linear time are the DSU data structure of Gabow and Tarjan [33], and any linear-time algorithm for answering off-line NCA queries (e.g., [40] or [12]). We note that, in particular, the DSU data structure of Gabow and Tarjan utilizes the power of the RAM model of computation. Thus, it is still an open question whether the computation of the 5-edge-connected components can be performed in linear time without using the power of the RAM model. For practical purposes, however, there are implementations for those data structures that run in almost linear time, with an overhead of only an inverse of Ackermann's function [64]. Thus, in practice, one could use those implementations

for our algorithm, in order to achieve almost-linear time.

5.1.4 Technical overview

First, let us recall some definitions from Chapter 2. In this chapter, all graphs considered are undirected multigraphs. It is convenient to consider only *edge-minimal* cuts. Thus, whenever we consider a k-edge cut C of a connected graph G, we assume that C is minimal w.r.t. the property that $G \setminus C$ is disconnected. For simplicity, we call C a *k-cut* of G. If two vertices x and y belong to different connected components of $G \setminus C$, then we say that C separates x and y. Notice that two vertices of G are 5-edge-connected if and only if there is no k-cut that separates them, for any $k \leq 4$. A k-cut that separates at least one pair of k-edge-connected vertices is called an *essential* k-cut.

There is a duality between cuts of G and bipartitions of $V(G)$. ($V(G)$ denotes the vertex set of G.) Specifically, if C is a cut of G and X, Y are the connected components of $G \setminus C$, then we have $E_G[X, Y] = C$ (where $E_G[X, Y] = \{(x, y) \in E(G) \mid x \in X \text{ and } y \in Y\}$). Thus, we can view C either as a set of edges, or as the bipartition $\{X, Y\}$ of $V(G)$. X and Y are also called the *sides* of C. If X is a subset of $V(G)$, then we denote $E_G[X, V(G) \setminus X]$ as $\partial(X)$. If r is a vertex of G, and X is the side of a cut C of G that does not contain r, then we call $|X|$ the *r-size* of C.

Let C and C' be two cuts of G, with sides X, Y and X', Y', respectively. If at least one of the intersections $X \cap X'$, $X \cap Y'$, $Y \cap X'$ or $Y \cap Y'$ is empty, then we say that C and C' are *parallel*. A collection $\mathcal{C}$ of cuts of G that are pairwise parallel is called a *parallel family* of cuts of G. It is a known fact that a parallel family of cuts of a graph with n vertices contains at most $O(n)$ cuts (see, e.g., [24]).

If $\mathcal{P}$ is a collection of partitions of a set V, then we let *atoms*($\mathcal{P}$) denote the partition of V that is given by the mutual refinement of all partitions in $\mathcal{P}$. In other words, *atoms*($\mathcal{P}$) is defined by the property that two elements x and y of V belong to two different sets in *atoms*($\mathcal{P}$) if and only if there is a partition $P \in \mathcal{P}$ such that x and y belong to different sets in P. This terminology is convenient for the following reason. Let $\mathcal{C}_{kcuts}$ denote the collection of all k-cuts of a connected graph G, for every $k \geq 1$. Then the partition of the 5-edge-connected components of G is given by *atoms*($\mathcal{C}_{1cuts} \cup \mathcal{C}_{2cuts} \cup \mathcal{C}_{3cuts} \cup \mathcal{C}_{4cuts}$).

In Section 5.3 we provide the following results. First, we show that, given a graph

G and a vertex r of G, there is a linear-time preprocessing of G such that we can report the r-size of any 4-cut of G in $O(1)$ time (see Lemma 5.22). Second, there is a linear-time preprocessing of G such that, given a 4-cut C of G, we can check if C is an essential 4-cut in $O(1)$ time (see Proposition 5.4). And third, given a parallel family $\mathcal{C}$ of 4-cuts of G, we can compute the atoms of $\mathcal{C}$ in linear time (see Proposition 5.5). In order to establish Proposition 5.4, we utilize the oracle that we develop in Chapter 6, for answering connectivity queries in the presence of at most four edge-failures.

5.1.4.1 Reduction to 3-edge-connected graphs

We rely on a construction that was described by Dinitz [21], that enables us to reduce the computation of the 5-edge-connected components to 3-edge-connected graphs. (We note that this was also used by [36] and [50] in order to compute the 4-edge-connected components.) Specifically, [21] provided the following result. Let G be a graph, and let $S_1, \dots, S_t$ be the 3-edge-connected components of G. Then, we can augment the graphs $G[S_1], \dots, G[S_t]$ with the addition of $O(|V(G)|)$ artificial edges, so that the resulting graphs $G'[S_1], \dots, G'[S_t]$ have the property that (1) $G'[S_i]$ is 3-edge-connected for every $i \in \{1, \dots, t\}$, (2) for every k-edge-connected component S of G, for $k \geq 3$, there is an $i \in \{1, \dots, t\}$ such that $G'[S_i]$ contains S as a k-edge-connected component, and (3) for every $i \in \{1, \dots, t\}$, and every $k \geq 3$, a k-edge-connected component of $G'[S_i]$ is also a k-edge-connected component of G. We note that the auxiliary graphs $G'[S_1], \dots, G'[S_t]$ can be constructed easily in linear time in total, after computing the 3-edge-connected components of G (using, e.g., the algorithm from [67]). Thus, in order to compute the 5-edge-connected components of G in linear time, properties (1), (2) and (3) imply that it is enough to know how to compute in linear time the 5-edge-connected components of a 3-edge-connected graph.

We note that we do not know how to produce auxiliary 4-edge-connected graphs, with properties like (1), (2) and (3), that can provide the 5-edge-connected components. However, even if we knew how to do that, we would still be faced with the problem of computing enough 4-cuts in order to derive the 5-edge-connected components. Although the work of Gabow [32] shows that we can compute the k-edge-connected components of a $(k-1)$-edge-connected graph in $O(m+k^2 n \log(n/k))$ time, there are no indications that computing the 5-edge-connected components of a 4-edge-connected graph in *linear time* is a much easier problem than working directly on 3-edge-connected graphs.

5.1.4.2 Computing enough 4-cuts of a 3-edge-connected graph

Let G be a 3-edge-connected graph with n vertices and m edges. In order to compute the 5-edge-connected components of G, we have to solve simultaneously the following two problems. First, we have to compute a collection $\mathcal{C}$ of 4-cuts that are enough in order to provide the 5-edge-connected components. At the same time, we must be able to efficiently compute the atoms of $\mathcal{C}$ (in order to derive the 5-edge-connected components). The straightforward way to compute these atoms is to compute the bipartition of the connected components after the removal of every 4-cut in $\mathcal{C}$, and then return the mutual refinement of all those bipartitions. However, if the number of 4-cuts in $\mathcal{C}$ is $\Omega(n)$, then this procedure will take $\Omega(nm)$ time in total, which is very far from our linear-time goal. Nevertheless, if $\mathcal{C}$ is a parallel family of 4-cuts, then the computation of the atoms of $\mathcal{C}$ can be performed in $O(m+n)$ time (see Proposition 5.5). Furthermore, in this case we can construct in linear time an oracle of $O(n)$ size that can report in constant time a 4-cut that separates any given pair of vertices, or determine that no such 4-cut exists (see Corollary 5.10). Thus, our goal is precisely to compute a parallel family of 4-cuts that can provide the 5-edge-connected components.

It turns out that this is a highly non-trivial task. First of all, even computing a single 4-cut takes $O(m + n\log n)$ time with the state-of-the-art method (which is Gabow's mincut algorithm [32]). On the other hand, it would be impractical to compute *all* 4-cuts of the graph, no matter the algorithm used, since the number of all 4-cuts in a 3-edge-connected graph can be as high as $\Omega(n^2)$ even in graphs with $O(n)$ edges. Our approach, instead, is to compute a compact collection of all 4-cuts that has size $O(n)$. When we say a "compact collection", we mean that there is a procedure, through which, from this collection of 4-cuts, we can essentially derive all 4-cuts. At this point, it is necessary to precisely define our concepts. First, we have the following property of 4-cuts in 3-edge-connected graphs.

Lemma 5.1 (Implied 4-cut). *Let $\{e_1, e_2, e_3, e_4\}$ and $\{e_3, e_4, e_5, e_6\}$ be two distinct 4-cuts of a 3-edge-connected graph G. Then $\{e_1, e_2, e_5, e_6\}$ is also a 4-cut of G.*

Proof. See Lemma 5.5. □

Then, Lemma 5.1 motivates the following.

Definition 5.1 (Implicating sequences of 4-cuts). Let $\mathcal{C}$ be a collection of 4-cuts of

a 3-edge-connected graph G. Let $p_1, \ldots, p_{k+1}$ be a sequence of pairs of edges, and let $C_1, \ldots, C_k$ be a sequence of 4-cuts from $\mathcal{C}$, such that $C_i = p_i \cup p_{i+1}$ for every $i \in \{1, \ldots, k\}$, and $C = p_1 \cup p_{k+1}$ is a 4-cut of G. Then we say that C is implied from $\mathcal{C}$ through the pair of edges p_1 (or equivalently: through the pair of edges p_{k+1}). In this case, we say that $C_1, \ldots, C_k$ is an implicating sequence of $\mathcal{C}$. If $\mathcal{C}$ implies every 4-cut of G, then we say that $\mathcal{C}$ is a *complete collection* of 4-cuts of G.

One of our main results is the following.

Theorem 5.1. *Let G be a 3-edge-connected graph with m edges and n vertices. Then, in $O(m+n)$ time, we can compute a complete collection $\mathcal{C}$ of 4-cuts of G with $|\mathcal{C}| = O(n)$.*

Proof. See Theorem 5.3. □

It is not at all obvious why a complete collection of 4-cuts with size $O(n)$ should exist. Furthermore, computing such a collection in linear time seems to be a very difficult problem, considering that it is not even known how to compute a single 4-cut in linear time. In particular, with Theorem 5.1 we improve on the state of the art in computing a mincut of bounded cardinality as follows.

Corollary 5.1. *Let G be any graph. Then, in linear time, we can compute a k-cut of G, with $k \leq 4$, or determine that G is 5-edge-connected.*

Proof. From previous work [63, 67, 36, 50], we know that, in linear time, we can compute a k-cut of G, with $k \leq 3$, or determine that G is 4-edge-connected. So let us assume that G is 4-edge-connected. Then, Theorem 5.1 implies that, in linear time, we can compute a 4-cut of G, or determine that G is 5-edge-connected. □

More than half of this chapter is devoted to establishing Theorem 5.1. The high-level idea is to identify the 4-cuts on a DFS-tree of the graph. We can distinguish various types of 4-cuts on a DFS-tree, and there is enough structure that enables us to compute a specific selection of them, that implies all 4-cuts of the graph. For a detailed elaboration on this idea we refer to Section 5.5. We note that the bulk of this work would be significantly reduced if we had a simpler algorithm for computing a complete collection of 4-cuts with at most linear size. At the moment, we do not know any alternative method to do this. However, even computing a near-linear sized complete collection of 4-cuts (in near-linear time), would still provide a near-linear time algorithm for computing the 5-edge-connected components, by following the

same analysis. Thus, there is room for simplifying the computation a lot, by relaxing the strictness of the linear complexity. In any case, given a complete collection of 4-cuts, we are faced with the problem of how to use this package of information in order to derive the 5-edge-connected components. This is what we discuss next.

5.1.4.3 Unpacking the implicating sequences of a complete collection of 4-cuts

Given a complete collection $\mathcal{C}$ of 4-cuts of G, the challenge is to unpack as many 4-cuts as are needed, in order to derive the 5-edge-connected components. The first thing we do is to implicitly expand all implicating sequences of $\mathcal{C}$, and organize them in collections of pairs of edges that generate, in total, all the 4-cuts of the graph. The concept of *generating* 4-cuts is made precise in the following.

Definition 5.2. Let $\mathcal{F} = \{p_1, \dots, p_k\}$ be a collection of pairs of edges of G, with $k \geq 2$, such that $p_i \cup p_j$ is a 4-cut of G for every $i, j \in \{1, \dots, k\}$ with $i \neq j$. Then we say that $\mathcal{F}$ generates the collection of 4-cuts $\{p_i \cup p_j \mid i, j \in \{1, \dots, k\}, i \neq j\}$.

An important intermediate result that we use throughout is the following.

Proposition 5.1. *Let $\mathcal{C}$ be a collection of 4-cuts of G. Then, in $O(n + |\mathcal{C}|)$ time, we can construct a set $\{\mathcal{F}_1, \dots, \mathcal{F}_k\}$ of collections of pairs of edges, with $|\mathcal{F}_i| \geq 2$ for every $i \in \{1, \dots, k\}$, such that $\mathcal{F}_i$ generates a collection of 4-cuts implied by $\mathcal{C}$, and every 4-cut implied by $\mathcal{C}$ is generated by $\mathcal{F}_i$, for some $i \in \{1, \dots, k\}$. The total size of $\{\mathcal{F}_1, \dots, \mathcal{F}_k\}$ is $O(|\mathcal{C}|)$ (i.e., $|\mathcal{F}_1| + \dots + |\mathcal{F}_k| = O(|\mathcal{C}|)$).*

Proof. See Proposition 5.3. □

Thus, given a complete collection $\mathcal{C}$ of 4-cuts, the collections of pairs of edges that we get in Proposition 5.1 constitute an alternative compact representation of the collection of all 4-cuts of the graph. In order to establish Proposition 5.1, we use an algorithm that breaks up every 4-cut from $\mathcal{C}$ into its three different partitions into pairs of edges, and then greedily reassembles all implicating sequences of $\mathcal{C}$, by constructing maximal collections of pairs of edges that participate in an implicating sequence. Specifically, for every bipartition $\{p, q\}$ of a 4-cut $C \in \mathcal{C}$ into pairs of edges, we generate two elements (C, p) and (C, q). Then, we consider the elements (C, p) and (C, q) as connected, by introducing an artificial edge that joins them. Notice that $\mathcal{F} = \{p, q\}$ is a collection of pairs of edges that generates a 4-cut implied by $\mathcal{C}$. Then, we try to expand $\mathcal{F}$ as much as possible, into a collection of pairs of edges that

generates 4-cuts implied by $\mathcal{C}$, by tracing the implicating sequences of $\mathcal{C}$ that use p or q. Thus, if e.g. another 4-cut $C' \in \mathcal{C}$ contains the pair of edges p, and is partitioned as $C' = p \cup q'$, then we also consider the element (C', p) connected with (C', q'), and the element (C', p) connected with (C, p), so that $\mathcal{F}' = \{p, q, q'\}$ is a collection of pairs of edges that generates 4-cuts implied by $\mathcal{C}$. The precise method by which we create these collections of pairs of edges is shown in Algorithm 16. The output of Algorithm 16 has some nice properties that we analyze in Sections 5.2.4, 5.2.5 and 5.2.6.

Let $\mathcal{F}$ be a collection of pairs of edges that is returned by Algorithm 16. Then we distinguish three different cases for $\mathcal{F}$: either $|\mathcal{F}| > 3$, or $|\mathcal{F}| = 3$, or $|\mathcal{F}| = 2$. The collections of pairs of edges that have size more than 3 generate collections of 4-cuts that have a very convenient structure for computational purposes. These are discussed next.

5.1.4.4 Cyclic families of 4-cuts, and minimal 4-cuts

Notice that if we have a collection of k pairs of edges that generates a collection $\mathcal{C}$ of 4-cuts, then $|\mathcal{C}| = k(k-1)/2$. Now, the reason that the number of 4-cuts in 3-edge-connected graphs with n vertices can be as high as $\Omega(n^2)$ is essentially the existence of some families of 4-cuts that are captured in the following.[2]

Definition 5.3 (Cyclic family of 4-cuts). Let $\{p_1, \dots, p_k\}$, with $k \geq 3$, be a collection of pairs of edges of G that generates a collection $\mathcal{C}$ of 4-cuts of G. Suppose that there is a partition $\{X_1, \dots, X_k\}$ of $V(G)$ with the property that (1) $G[X_i]$ is connected for every $i \in \{1, \dots, k\}$, (2) $E[X_i, X_{i+1}] = p_i$ for every $i \in \{1, \dots, k-1\}$, and (3) $E[X_k, X_1] = p_k$. Then $\mathcal{C}$ is called a cyclic family of 4-cuts. (See Figure 5.5.)

Now, our claim above is supported by Proposition 5.2, Proposition 5.3, and Theorem 5.3. Specifically, Proposition 5.2 basically states that a collection of pairs of

[2]We note that Definition 5.3 is similar to the concept of a *circular partition* (given e.g. in [52] or [28]), that is used in the construction of the cactus representation of the minimum cuts of a graph. The difference is that here the 4-cuts are not necessarily mincuts. Thus, some convenient properties like Lemma 5.1 in [52] or Lemma 2.5 in [28] fail to hold. However, organizing the 4-cuts in cyclic families is still very useful for computational purposes. In particular, the cyclic families of 4-cuts that are produced by the output of Algorithm 16 on a complete collection of 4-cuts have some very convenient properties that we explore in Section 5.2 (most importantly, see Lemma 5.20).

edges with more than 3 pairs of edges generates a cyclic family of 4-cuts. Theorem 5.3 implies that there is a complete collection $\mathcal{C}$ of 4-cuts with size $O(n)$, and then Proposition 5.3 (applied on $\mathcal{C}$) implies that there is a set $\{\mathcal{F}_1, \dots, \mathcal{F}_k\}$ of collections of pairs of edges, with $|\mathcal{F}_1| + \dots + |\mathcal{F}_k| = O(|\mathcal{C}|) = O(n)$, that generate in total all 4-cuts of G. Thus, we have the following combinatorial result, which is also derived from [43].[3]

Corollary 5.2. *The number of* 4*-cuts in a* 3*-edge-connected graph with* n *vertices is* $O(n^2)$*.*

Now, given a cyclic family of 4-cuts $\mathcal{C}$ as in Definition 5.3, by Lemma 5.8 we have $\partial(X_i) = p_i \cup p_{i-1}$ for every $i \in \{2, \dots, k\}$, and $\partial(X_1) = p_1 \cup p_k$, and therefore we have $\{\partial(X_1), \dots, \partial(X_k)\} \subseteq \mathcal{C}$. The collection of 4-cuts $\mathcal{M} := \{\partial(X_1), \dots, \partial(X_k)\}$ is of particular importance, and we call it the collection of the $\mathcal{C}$*-minimal* 4-cuts. These 4-cuts are $\mathcal{C}$-"minimal" in the sense that one of their sides (i.e., X_i), is a subset of one of the sides of every 4-cut in $\mathcal{C}$. (Lemma 5.8 describes the structure of the sides of the 4-cuts in a cyclic family of 4-cuts; this can also be inferred from Figure 5.5.) The main reasons that $\mathcal{M}$ is important are the following. First, $\mathcal{M}$ is a parallel family of 4-cuts. Second, the atoms of $\mathcal{M}$ coincide with the atoms of $\mathcal{C}$. And third, given the collection $\mathcal{F}$ of pairs of edges that generates $\mathcal{C}$, we can compute $\mathcal{M}$ in $O(n+|\mathcal{F}|)$ time.

The first two points are almost immediate from the definition of minimal 4-cuts (see also Figure 5.5). On the other hand, the computation of the minimal 4-cuts is not entirely trivial. The problem is that, given the collection $\mathcal{F}$ of pairs of edges that generates a cyclic family of 4-cuts $\mathcal{C}$, it is not necessary that the pairs of edges in $\mathcal{F}$ are given in the order that is needed in order to form the $\mathcal{C}$-minimal 4-cuts. Thus, we have to determine the sequence of the pairs of edges in $\mathcal{F}$ that provides the $\mathcal{C}$-minimal 4-cuts. One way to achieve this can be roughly described as follows. First, we take any vertex $r \in V(G)$, and let us assume w.l.o.g. that $r \in X_1$. Then we pick any pair of edges p_i from $\mathcal{F}$. Then, notice that, among all 4-cuts of the form $p_i \cup p_j$, for $j \in \{1, \dots, k\} \setminus \{i\}$, we have that either $p_i \cup p_1$ or $p_i \cup p_k$ has the maximum r-size. Thus, we can determine one of the two pairs of edges that are incident to X_1 (i.e., either p_1 or p_k), by taking the maximum r-size of all 4-cuts of the form $p_i \cup p_j$, for $j \in \{1, \dots, k\} \setminus \{i\}$. So let suppose that we have determined

[3] It is possible that Corollary 5.2 can also be derived from the 2-level cactus representation of the (*minimum* + 1)-cuts of a graph [24]. However, here perhaps it is clearer why the number of 4-cuts in 3-edge-connected graphs is bounded by $O(n^2)$, and what are the responsible structures that this number can be as high as $\Omega(n^2)$.

that p_1 is one of the two pairs of edges from $\mathcal{F}$ that is incident to X_1. Then, notice that the 4-cuts $p_1 \cup p_2, \dots, p_1 \cup p_k$ are sorted in increasing order w.r.t. their r-size. Thus, it is sufficient to form all 4-cuts of the form $\{p_1 \cup p_i \mid i \in \{2, \dots, k\}\}$, and then sort them in increasing order w.r.t. their r-size. Then we can extract the sequence of pairs of edges $p_1, \dots, p_k$, which is what we need in order to find the $\mathcal{C}$-minimal 4-cuts (by taking the union of every two consecutive pairs of edges in this sequence, plus $p_1 \cup p_k$). This method demands $O(n)$ time in order to perform the sorting of the 4-cuts of the form $\{p_1 \cup p_i \mid i \in \{2, \dots, k\}\}$ (with bucket-sort). Thus, this method in itself is impractical for our purposes, because we may have to compute the minimal 4-cuts for $\Omega(n)$ collections of pairs of edges. However, given all the collections of pairs of edges beforehand, we can use this method to compute the minimal 4-cuts of the cyclic families that are generated by those collections with only one bucket-sort (that sorts all the 4-cuts that we will form and we need to have sorted, in increasing order w.r.t. their r-size). Thus, Algorithm 18 shows how we can compute all $\mathcal{C}_1$-,...,$\mathcal{C}_t$-minimal 4-cuts, where $\mathcal{C}_1, \dots, \mathcal{C}_t$ are cyclic families of 4-cuts that are generated by the collections of pairs of edges $\mathcal{F}_1, \dots, \mathcal{F}_t$, respectively. The running time of this algorithm is $O(n + |\mathcal{F}_1| + \dots + |\mathcal{F}_t|)$, as shown in Proposition 5.6.

Now let $\mathcal{C}$ be a complete collection of 4-cuts, and let $\mathcal{F}_1, \dots, \mathcal{F}_t$ be the collections of pairs of edges that are returned by Algorithm 16 on input $\mathcal{C}$ and have the property that $|\mathcal{F}_i| > 3$, for every $i \in \{1, \dots, t\}$. Then, Proposition 5.3 implies that $\mathcal{F}_i$ generates a collection $\mathcal{C}_i$ of 4-cuts implied by $\mathcal{C}$, for every $i \in \{1, \dots, t\}$, and by Proposition 5.2 we have that $\mathcal{C}_i$ is a cyclic family of 4-cuts. Thus, we can apply Algorithm 18 in order to derive the collection $\mathcal{M}_i$ of the $\mathcal{C}_i$-miminal 4-cuts, for every $i \in \{1, \dots, t\}$, in $O(n + |\mathcal{F}_1| + \dots + |\mathcal{F}_t|)$ time in total. As noted above, we have that $atoms(\mathcal{M}_i) = atoms(\mathcal{C}_i)$, and $\mathcal{M}_i$ is a parallel family of 4-cuts, for every $i \in \{1, \dots, t\}$. Thus, we have $atoms(\mathcal{C}_1 \cup \dots \cup \mathcal{C}_t) = atoms(\mathcal{M}_1 \cup \dots \cup \mathcal{M}_t)$. We note that this formula is not very useful for computing $atoms(\mathcal{C}_1 \cup \dots \cup \mathcal{C}_t)$, because there is no guarantee that $\mathcal{M}_1 \cup \dots \cup \mathcal{M}_t$ is a parallel family of 4-cuts. (In fact, Figure 5.12 provides a counterexample.) However, if we keep the subcollection $\mathcal{M}'$ of the essential 4-cuts in $\mathcal{M}_1 \cup \dots \cup \mathcal{M}_t$, then we have that $\mathcal{M}'$ is a parallel family of 4-cuts, as a consequence of Lemma 5.20. Thus, it is sufficient to keep only $\mathcal{M}'$ and compute $atoms(\mathcal{M}')$.

Now let $\mathcal{F}$ be a collection of pairs of edges that is returned by Algorithm 16 on input $\mathcal{C}$ and has $|\mathcal{F}| = 3$. By Proposition 5.3 we have that $\mathcal{F}$ generates a collection $\mathcal{C}'$ of 4-cuts implied by $\mathcal{C}$. If $\mathcal{C}'$ is not a cyclic family of 4-cuts, then we say that

$\mathcal{C}'$ is a *degenerate family* of 4-cuts, and Lemma 5.9 describes the structure of such a family (i.e., this is given by Figure 5.3(a), if $\mathcal{F} = \{\{e_1, e_2\}, \{e_3, e_4\}, \{e_5, e_6\}\}$). Then, by Corollary 5.4 we have that $\mathcal{C}'$, if it is not a cyclic family of 4-cuts, it has the property that all its 4-cuts are non-essential. Thus, if $\mathcal{C}'$ consists of three non-essential 4-cuts, then we can discard them. Otherwise, we have that $\mathcal{C}'$ is a cyclic family of 4-cuts. Since $|\mathcal{F}| = 3$, we have that all 4-cuts in $\mathcal{C}'$ are $\mathcal{C}'$-minimal. Thus, it is sufficient to keep only the essential 4-cuts from $\mathcal{C}'$. Then, these are all parallel among themselves, and also parallel with the 4-cuts in $\mathcal{M}'$ (due to Lemma 5.20).

Thus, we have shown how to extract enough 4-cuts from the collections of pairs of edges that are returned by Algorithm 16 and have size at least three. Now it remains to consider the collections of pairs of edges that have size 2.

5.1.4.5 Isolated and quasi-isolated 4-cuts

Let $\mathcal{C}$ be a complete collection of 4-cuts, and let $\mathcal{F}$ be a collection of pairs of edges that is returned by Algorithm 16 on input $\mathcal{C}$ and has $|\mathcal{F}| = 2$. They, by Proposition 5.3 we have that $\mathcal{F}$ generates a 4-cut C implied by $\mathcal{C}$. More precisely, by Lemma 5.15 we have $C \in \mathcal{C}$. Now, if there is another collection of pairs of edges $\mathcal{F}'$ that is returned by Algorithm 16 on input $\mathcal{C}$ that also generates C and has $|\mathcal{F}'| > 2$, then we have collected enough 4-cuts in order to capture the separation of $V(G)$ induced by C. Otherwise, we have that all collections of pairs of edges that are returned by Algorithm 16 on input $\mathcal{C}$ and generate C have size 2. Then, since $C \in \mathcal{C}$, these collections are the three different partitions of C into pairs of edges. We note that we can determine in linear time what are the 4-cuts from $\mathcal{C}$ with the property that all three partitions of them into pairs of edges are returned by Algorithm 16 on input $\mathcal{C}$. Now, for every such 4-cut C, we distinguish two different cases: either (1) there is no collection $\mathcal{F}$ of pairs of edges with $|\mathcal{F}| > 2$ that generates a collection of 4-cuts that includes C, or (2) the contrary of (1) is true. In case (1), we say that C is an *isolated* 4-cut. In case (2), we say that C is a *quasi-isolated* 4-cut. (Notice that the concept of quasi-isolated 4-cuts is relative to a collection $\mathcal{C}$ of 4-cuts that is given as input to Algorithm 16.)

The distinction between isolated and quasi-isolated 4-cuts is important, because by Corollary 5.6 we have that an essential isolated 4-cut is parallel with every essential 4-cut. On the other hand, there are examples where two essential quasi-isolated 4-cuts may cross (see Figure 5.15). However, there are two nice things that are very helpful here. First, the quasi-isolated 4-cuts are basically not needed for our purposes. More

precisely, by Lemma 5.21 we have that every pair of vertices that are separated by an essential quasi-isolated 4-cut, are also separated by a 4-cut that is generated by a collection of pairs of edges with size more than 2 that is returned by Algorithm 16. And second, we have enough information in order to identify the quasi-isolated 4-cuts, so that we can discard them. Specifically, by Corollary 5.9 we have that every essential quasi-isolated 4-cut shares a pair of edges with an essential $\mathcal{C}'$-minimal 4-cut, where $\mathcal{C}'$ is a cyclic family of 4-cuts that is generated by a collection of pairs of edges (with size at least 3) that is returned by Algorithm 16 on input $\mathcal{C}$. This is a property that distinguishes the quasi-isolated from the isolated 4-cuts, and we can use it in order to identify all the essential isolated 4-cuts, as shown in Proposition 5.7.

5.1.4.6 The full algorithm

In summary, these are the steps that we follow in order to compute the 5-edge-connected components of a 3-edge-connected graph G.

1. Compute the partition $\mathcal{P}_4$ of the 4-edge-connected components of G.
2. Compute a complete collection $\mathcal{C}$ of 4-cuts of G with size $O(n)$.
3. Compute the collections of pairs of edges $\mathcal{F}_1, \dots, \mathcal{F}_k$ that are returned by Algorithm 16 on input $\mathcal{C}$.
4. Let $I \subseteq \{1, \dots, k\}$ be the collection of indices such that, for every $i \in I$, either $|\mathcal{F}_i| > 3$, or $|\mathcal{F}_i| = 3$ and $\mathcal{F}_i$ generates at least one essential 4-cut. Let $\mathcal{C}_i$ be the cyclic family of 4-cuts generated by $\mathcal{F}_i$, for every $i \in I$.
5. Compute the collection $\mathcal{M}_i$ of the $\mathcal{C}_i$-minimal 4-cuts, for every $i \in I$.
6. Compute the subcollection $\mathcal{M}'$ of the essential 4-cuts in $\bigcup_{i \in I} \mathcal{M}_i$.
7. Compute the collection $\mathcal{ISO}$ of the essential isolated 4-cuts of G.
8. Let $\mathcal{P}_5$ be the refinement of *atoms*($\mathcal{M}'$) with *atoms*($\mathcal{ISO}$).
9. Return $\mathcal{P}_5$ refined by $\mathcal{P}_4$.

Step 1 can be performed in linear time from previous results (see [36] or [50]). By Theorem 5.3, Step 2 can be performed in linear time. Step 3 takes $O(n)$ time, according to Proposition 5.3. By Proposition 5.2, we know that every $\mathcal{F}_i$ with $|\mathcal{F}_i| > 3$ generates

a cyclic family of 4-cuts. By Corollary 5.5 we have that every $\mathcal{F}_i$ with $|\mathcal{F}_i| = 3$ that generates at least one essential 4-cut generates a cyclic family of 4-cuts. Thus, if we let I be the collection of indices in Step 4, then we have that $\mathcal{F}_i$ generates a cyclic family of 4-cuts for every $i \in I$. Then, it makes sense to perform Step 5, and this can be completed in $O(n)$ time, according to Proposition 5.6. Then, we can extract the subcollection $\mathcal{M}'$ of the essential 4-cuts in $\bigcup_{i\in I} \mathcal{M}_i$ in $O(n)$ time, after we have performed the preprocessing described in Proposition 5.4. Thus, Step 6 takes linear time. The computation in Step 7 also takes linear time, according to Proposition 5.7. Since we have that $\mathcal{M}'$ and $\mathcal{ISO}$ are parallel families of 4-cuts, we can compute *atoms*$(\mathcal{M}')$ and *atoms*$(\mathcal{ISO})$ in linear time, according to Proposition 5.5. Then, the mutual refinement of those partitions in Step 8 takes $O(n)$ time, by using bucket-sort. Finally, the refinement in Step 9 also takes $O(n)$ time with bucket-sort.

Now we will demonstrate the correctness of this procedure. First, notice that the partition of the 5-edge-connected components is a refinement of the partition returned in Step 9 (because the latter is a refinement of the partition of the 4-edge-connected components with the atoms of a specific collection of 4-cuts). Conversely, let x and y be two vertices of G that are not 5-edge-connected. Then, there is either a 3-cut or a 4-cut that separates x and y. If there is a 3-cut that separates x and y, then x and y belong to different 4-edge-connected components, and therefore they belong to different sets in the partition returned in Step 9. Otherwise, if x and y are 4-edge-connected, then there is an essential 4-cut C that separates them. Since $\mathcal{C}$ is a complete collection of 4-cuts, Proposition 5.3 implies that there is an $i \in \{1, \dots, k\}$ such that C is generated by $\mathcal{F}_i$. If $|\mathcal{F}_i| \geq 3$, then, since C is an essential 4-cut, Proposition 5.2 and Corollary 5.5 imply that $\mathcal{F}_i$ generates a cyclic family $\mathcal{C}_i$ of 4-cuts. Therefore, by Lemma 5.12 we have that x and y are separated by an essential $\mathcal{C}_i$-minimal 4-cut. Thus, x and y belong to different sets in *atoms*$(\mathcal{M}')$. Otherwise, C is either an isolated or a quasi-isolated 4-cut. If C is isolated, then it belongs to $\mathcal{ISO}$, and therefore x and y belong to different sets in *atoms*$(\mathcal{ISO})$. Otherwise, by Lemma 5.21 we have that x and y are separated by an essential $\mathcal{C}_i$-minimal 4-cut, for some $i \in I$. Thus, x and y belong to different sets in *atoms*$(\mathcal{M}')$. In either case, then, we have that x and y belong to different sets in the partition returned by Step 9. We conclude that this partition coincides with that of the 5-edge-connected components.

5.1.5 Organization of this chapter

In Section 5.2 we study the structure of 4-cuts in 3-edge-connected graphs. In Section 5.3 we present some applications of identifying 4-cuts on a DFS-tree, that we will need in order to establish our main result. In Section 5.4 we present the algorithm for computing the 5-edge-connected components of a 3-edge-connected graph. Section 5.5 gives an overview of the algorithm for computing a complete collection of 4-cuts of a 3-edge-connected graph. There, we provide a DFS-based classification of all 4-cuts of a 3-edge-connected graph, and briefly discuss the methods that we employ in order to compute the 4-cuts of each class. In Sections 5.6, 5.7 and 5.8 we provide the full details for computing the most demanding classes of 4-cuts.

5.2 Properties of 4-cuts in 3-edge-connected graphs

Throughout this section we assume that G is a 3-edge-connected graph. We also assume a total ordering of the edges of G (e.g., lexicographic order w.r.t. their endpoints). This is needed for Algorithm 16, and for analyzing its output. We let V denote $V(G)$. All graph-related elements, such as vertices, edges, cuts, etc., refer to G.

Lemma 5.2. *Let C_1 and C_2 be two distinct 4-cuts of G. Then $|C_1 \cap C_2| \neq 3$.*

Proof. Let us suppose, for the sake of contradiction, that $|C_1 \cap C_2| = 3$. Since C_1 and C_2 are 4-cuts, we have that $G' = G \setminus (C_1 \cap C_2)$ is connected. Let $e_1 = C_1 \setminus (C_1 \cap C_2)$ and $e_2 = C_2 \setminus (C_1 \cap C_2)$. Since C_1 and C_2 are distinct, we have that $e_1 \neq e_2$. And since both C_1 and C_2 are 4-cuts, we have that both $G' \setminus e_1$ and $G' \setminus e_2$ are disconnected. Thus, e_1 and e_2 are two distinct bridges of G', and so $V(G')$ can be partitioned into three sets X, Y, and Z, such that $E_{G'}[X, Z] = \{e_1\}$, $E_{G'}[Y, Z] = \{e_2\}$, and $E_{G'}[X, Y] = \emptyset$. Since G is 3-edge-connected, we have that $|\partial(X)| \geq 3$, $|\partial(Y)| \geq 3$ and $|\partial(Z)| \geq 3$. This can only be true if either (1) two of the edges from $C_1 \cap C_2$ connect X and Y, and the other edge connects either X and Z, or Y and Z, or (2) one edge from $C_1 \cap C_2$ connects X and Y, one edge from $C_1 \cap C_2$ connects X and Z, and one edge from $C_1 \cap C_2$ connects Y and Z. Let us consider case (1) first, and let us assume w.l.o.g. that an edge from $C_1 \cap C_2$ connects X and Z. But now we have that the two edges from $C_1 \cap C_2$ that connect X and Y, plus e_2, constitute a 3-cut of G (with sides Y and $V \setminus Y$). This contradicts the fact that C_2 is a 4-cut of G. Thus, only case (2) can be

true. But then we have that $\partial(X)$ consists of e_1 and two edges from $C_1 \cap C_2$. Therefore $\partial(X)$ is a proper subset of C_1 that disconnects G upon removal, contradicting the fact that C_1 is a 4-cut of G. We conclude that it is impossible to have $|C_1 \cap C_2| = 3$. $\square$

Lemma 5.3. *Let X be a subset of $V(G)$ such that $|\partial(X)| = 4$. Then $G[X]$ is connected, and it has at most one bridge. If $G[X]$ has a bridge e, then $G[X] \setminus e$ consists of two connected components Y_1 and Y_2 such that $|E[Y_1, V \setminus X]| = 2$ and $|E[Y_2, V \setminus X]| = 2$.*

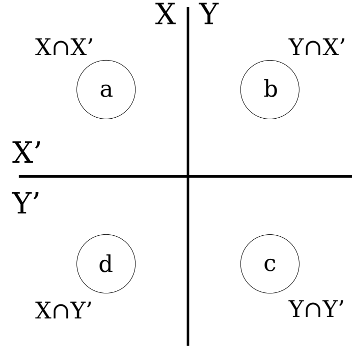
Proof. First, let us suppose, for the sake of contradiction, that $G[X]$ is not connected. Then, let S and S' be two distinct connected components of $G[X]$. Then we have that $\partial(S) \subseteq \partial(X)$, $\partial(S') \subseteq \partial(X)$, and $\partial(S) \cap \partial(S') = \emptyset$. This implies that $|\partial(S)| + |\partial(S')| = |\partial(S) \cup \partial(S')| \leq |\partial(X)| = 4$. Thus, at least one of $|\partial(S)|$ and $|\partial(S')|$ must be lower than 3, in contradiction to the fact that G is 3-edge-connected. This shows that $G[X]$ is connected.

Now, let us suppose, for the sake of contradiction, that $G[X]$ has at least two bridges e_1 and e_2. Then there is a partition $\{Z_1, Z_2, Z_3\}$ of X, such that all of $G[Z_1]$, $G[Z_2]$ and $G[Z_3]$, are connected, and such that $E[Z_1, Z_2] = \{e_1\}$, $E[Z_2, Z_3] = \{e_2\}$ and $E[Z_1, Z_3] = \emptyset$. Let $E_i = E[Z_i, V \setminus X]$, for $i \in \{1, 2, 3\}$. Then we have $E_1 \sqcup E_2 \sqcup E_3 = \partial(X)$, $\partial(Z_1) = E_1 \sqcup \{e_1\}$, $\partial(Z_3) = E_3 \sqcup \{e_2\}$, and $\partial(Z_2) = E_2 \sqcup \{e_1, e_2\}$. Since $|E_1| + |E_2| + |E_3| = 4$, $|\partial(Z_1)| = |E_1| + 1$, $|\partial(Z_3)| = |E_3| + 1$, and $|\partial(Z_2)| = |E_2| + 2$, we infer that at least one of $|\partial(Z_i)|$, for $i \in \{1, 2, 3\}$, is at most 2. This contradicts the fact that G is 3-edge-connected. Thus, $G[X]$ can have at most one bridge.

Now let e be a bridge of $G[X]$. Then $G[X] \setminus e$ consists of two connected components Y_1 and Y_2, and we have $E[Y_1, Y_2] = \{e\}$. Since G is 3-edge-connected, we have that both $|\partial(Y_1)|$ and $|\partial(Y_2)|$ must be at least 3. We have that $\partial(Y_i) = E[Y_i, V \setminus X] \cup \{e\}$, for $i \in \{1, 2\}$. Thus, both $E[Y_1, V \setminus X]$ and $E[Y_2, V \setminus X]$ must contain at least 2 edges. Since $E[Y_1, V \setminus X] \sqcup E[Y_2, V \setminus X] = \partial(X)$, this implies that $|E[Y_1, V \setminus X]| = 2$ and $|E[Y_2, V \setminus X]| = 2$ (due to $|\partial(X)| = 4$). $\square$

5.2.1 The structure of crossing 4-cuts of a 3-edge-connected graph

The following lemma is one of the cornerstones of our work. It motivates several concepts that we develop in order to analyze the structure of 4-cuts in 3-edge-connected graphs.



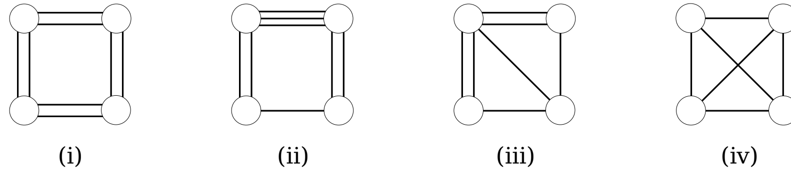


Figure 5.1: All possible crossings of two 4-cuts $C = \{X, Y\}$ and $C' = \{X', Y'\}$.

Lemma 5.4 (The Structure of Crossing 4-cuts). *Let C and C' be two 4-cuts of a 3-edge-connected graph. Then, cases **(i)** to **(iv)** in Figure 5.1 show all the different ways in which C and C' may cross.*

Proof. Let X and Y be the sides of C, and let X' and Y' be the sides of C'. We let $a = X \cap X'$, $b = Y \cap X'$, $c = Y \cap Y'$, and $d = X \cap Y'$ (see Figure 5.1). We will analyze all the different (non-isomorphic) ways in which a, b, c and d may be connected with edges. Thus, we have to determine all the different combinations of values for $|E[a, b]|$, $|E[a, c]|$, $|E[a, d]|$, $|E[b, c]|$, $|E[b, d]|$ and $|E[c, d]|$. There are two properties that guide us. First, the graph that is formed by a, b, c and d does not have any 1-cuts or 2-cuts, since the original graph is 3-edge-connected. In particular, we have $|\partial(a)| \geq 3$, $|\partial(b)| \geq 3$, $|\partial(c)| \geq 3$ and $|\partial(d)| \geq 3$. And second, we have $C = E[a, b] \cup E[a, c] \cup E[d, b] \cup E[d, c]$, $C' = E[a, c] \cup E[a, d] \cup E[b, c] \cup E[b, d]$, and $|C| = |C'| = 4$.

Since $|C| = 4$, there are at most four edges in $E[a, b]$. Thus, we will consider all possible values for $|E[a, b]|$. We will start by showing that $|E[a, b]| \neq 4$ and $|E[a, b]| \neq 0$.

First, suppose that $E[a, b]$ consists of four edges. Then, since $|C| = 4$, there are no edges in $E[a, c]$, $E[d, b]$ or $E[d, c]$. Thus, since $|\partial(d)| \geq 3$, there are at least three edges in $E[d, a]$. Then, since $|\partial(c)| \geq 3$, and since $E[a, c] = E[d, c] = \emptyset$, there must be at least three edges in $E[c, b]$. But then, since $E[d, a] \cup E[c, b] \subseteq C'$, we have $|E[d, a]| + |E[c, b]| \leq$

$|C'| = 4$, which is impossible, because $|E[d, a]| + |E[c, b]| \geq 6$. This shows that there cannot be four edges in $E[a, b]$.

Now suppose that $E[a, b]$ is empty. Then, since $|\partial(a)| \geq 3$, we have $|E[a, c]| + |E[a, d]| \geq 3$. Since $C' = E[a, c] \cup E[a, d] \cup E[b, c] \cup E[b, d]$ and $|C'| = 4$, this implies that $|E[b, c]| + |E[b, d]| \leq 1$. But then, since $|E[a, b]| = 0$, we have a contradiction to the fact $|\partial(b)| \geq 3$. This shows that $E[a, b]$ must contain at least one edge.

Now suppose that $E[a, b]$ consists of three edges. Then $E[a, c]$ contains at most one edge. Suppose that $E[a, c]$ contains one edge. Then, since $|C| = 4$, we have $E[d, b] = E[d, c] = \emptyset$. Then, since $|\partial(d)| \geq 3$, we have at least three edges in $E[d, a]$. Furthermore, since $|\partial(c)| \geq 3$ and $|E[a, c]| = 1$ and $E[d, c] = \emptyset$, we have at least two edges in $E[c, b]$. But then, since $E[d, a] \cup E[c, b] \subseteq C'$, we have $|E[d, a]| + |E[c, b]| \leq |C'| = 4$, which is impossible, because $|E[d, a]| + |E[c, b]| \geq 5$. This shows that $E[a, c] = \emptyset$. Then, since $|C| = 4$, we have that one of $E[d, b]$ and $E[d, c]$ consists of one edge, and the other is empty. Suppose that $E[d, b]$ contains one edge (and therefore $E[d, c] = \emptyset$). Then, since $|\partial(c)| \geq 3$ and $E[a, c] = E[d, c] = \emptyset$, we have that $E[c, b]$ contains at least three edges. Then, since $E[c, b] \cup E[d, b] \cup E[d, a] \subseteq C'$ and $|C'| = 4$, we have that $E[d, a]$ must be empty. But then we have $|\partial(d)| = |E[d, b]| = 1$, contradicting the fact that $|\partial(d)| \geq 3$. This shows that $E[d, b]$ is empty, and $E[d, c]$ consists of one edge. Then, since $|\partial(d)| \geq 3$, we have that $E[a, d]$ contains at least two edges. Similarly, since $|\partial(c)| \geq 3$, and $E[a, c] = \emptyset$ and $|E[c, d]| = 1$, we have that $E[b, c]$ contains at least two edges. Then, since $|C'| = 4$ and $E[a, d] \cup E[b, c] \subseteq C'$, we have $|E[a, d]| = 2$ and $|E[b, c]| = 2$. Thus, we are in case **(ii)**.

Now suppose that $E[a, b]$ consists of two edges. Then, since $|C| = 4$ and $E[a, b] \cup E[a, c] \subseteq C$, we have that $E[a, c]$ contains at most two edges. Let us suppose, for the sake of contradiction, that $|E[a, c]| = 2$. Then, since $|C| = 4$, we have $C = E[a, b] \cup E[a, c]$. Since $E[d, b] \cup E[d, c] \subseteq C$, this implies that $E[d, b] = E[d, c] = \emptyset$. Then, since $|\partial(d)| \geq 3$, we have that $E[d, a]$ contains at least three edges. But then, since $E[a, c] \cup E[a, d] \subseteq C'$, we have $|E[a, c]| + |E[a, d]| \leq |C'| = 4$, which contradicts the fact that $|E[a, c]| + |E[a, d]| \geq 5$. This shows that $E[a, c]$ contains less than two edges.

Let us assume first that $E[a, c] = 1$. Then, since $|C| = 4$ and $C = E[a, b] \cup E[a, c] \cup E[d, b] \cup E[d, c]$, we have that one of $E[d, b]$ and $E[d, c]$ consists of one edge, and the other is empty. Let us suppose, for the sake of contradiction, that $E[d, b]$ contains one edge and $E[d, c]$ is empty. Then, since $|C'| = 4$ and $C' = E[a, c] \cup E[a, d] \cup E[b, c] \cup E[b, d]$, we have $|E[a, d]| + |E[b, c]| = 2$. This implies that it cannot be that both $E[a, d]$ and

$E[b,c]$ contain at least two edges. Now, if $E[a,d]$ contains less than two edges, then $|E[d,b]| = 1$ and $E[d,c] = \emptyset$ imply that $|\partial(d)| < 3$, which is impossible. And if $E[b,c]$ contains less than two edges, then $|E[a,c]| = 1$ and $E[d,c] = \emptyset$ imply that $|\partial(c)| < 3$, which is also impossible. Thus, we have that $E[d,b] = \emptyset$ and $E[d,c]$ consists of one edge. Then, since $|C'| = 4$ and $C' = E[a,c] \cup E[a,d] \cup E[b,c] \cup E[b,d]$, we have $|E[a,d]| + |E[b,c]| = 3$. And then, since $|\partial(d)| \geq 3$ and $|\partial(c)| \geq 3$, we have $|E[a,d]| = 2$ and $|E[b,c]| = 1$. Thus, we are in case **(iii)**.

Now let us assume that $E[a,c] = \emptyset$. Then, since $|C| = 4$ and $C = E[a,b] \cup E[a,c] \cup E[d,b] \cup E[d,c]$, we have $|E[d,b]| + |E[d,c]| = 2$. This implies that either $|E[d,b]| = 0$ and $|E[d,c]| = 2$, or $|E[d,b]| = 1$ and $|E[d,c]| = 1$, or $|E[d,b]| = 2$ and $|E[d,c]| = 0$. Let us suppose first that $|E[d,b]| = 0$ and $|E[d,c]| = 2$. Then, since $|C'| = 4$ and $C' = E[a,c] \cup E[a,d] \cup E[b,c] \cup E[b,d]$, we have $|E[a,d]| + |E[b,c]| = 4$ (because $E[a,c] = E[b,d] = \emptyset$). Then, since $|\partial(d)| \geq 3$ and $|\partial(c)| \geq 3$, we have that either $|E[a,d]| = 3$ and $|E[b,c]| = 1$, or $|E[a,d]| = |E[b,c]| = 2$, or $|E[a,d]| = 1$ and $|E[b,c]| = 3$. The first and the third case correspond to **(ii)** (by permuting the labels a, b, c and d). The second case is precisely **(i)**. Now let us suppose that $|E[d,b]| = 1$ and $|E[d,c]| = 1$. Then, since $|C'| = 4$ and $C' = E[a,c] \cup E[a,d] \cup E[b,c] \cup E[b,d]$, we have $|E[a,d]| + |E[b,c]| = 3$. And then, since $|\partial(d)| \geq 3$ and $|\partial(c)| \geq 3$, we have $|E[a,d]| = 1$ and $|E[b,c]| = 2$. Thus, we are in case **(iii)** (by permuting the labels a, b, c and d). Finally, let us suppose that $|E[d,b]| = 2$ and $|E[d,c]| = 0$. Then, since $E[a,c] = E[d,c] = \emptyset$ and $|\partial(c)| \geq 3$, we have that $E[b,c]$ contains at least three edges. But then, since $E[b,c] \cup E[d,b] \subseteq C'$, we have $|E[b,c]| + |E[d,b]| \leq |C'| = 4$, which contradicts the fact that $|E[b,c]| + |E[d,b]| \geq 5$. Thus, this case is impossible.

Finally, suppose that $E[a,b]$ consists of one edge. Since we have considered all other cases for $|E[a,b]|$, it is sufficient to assume, due to the symmetry of our situation, that $|E[b,c]| = 1$, $|E[c,d]| = 1$ and $|E[d,a]| = 1$. (Because, if at least one of those values is different than 1, then we can properly relabel the corners of the square, and revert to one of the previous cases.) It remains to determine the values $|E[a,c]|$ and $|E[d,b]|$. Since $|C| = 4$ and $|E[a,b]| = |E[d,c]| = 1$, we have that either $|E[a,c]| = 2$ and $|E[d,b]| = 0$, or $|E[a,c]| = 1$ and $|E[d,b]| = 1$, or $|E[a,c]| = 0$ and $|E[d,b]| = 2$. The first and the last case are rejected, because they imply that $|\partial(d)| = 2$ and $|\partial(a)| = 2$, respectively. Thus, $|E[a,c]| = 1$ and $|E[d,b]| = 1$ are the only viable options, and thus we are in case **(iv)**. $\square$

The following is an obvious corollary of Lemma 5.4.

Corollary 5.3. *Let $C_1 = \{e_1, e_2, e_3, e_4\}$ and $C_2 = \{f_1, f_2, f_3, f_4\}$ be two essential 4-cuts that cross. Then, C_1 and C_2 cross as in Figure 5.2 (up to permuting the labels of the edges of each cut).*

Proof. Lemma 5.4 implies that the possible crossings of C_1 and C_2 are given by cases **(i)** to **(iv)** of Figure 5.1. Notice that only in case **(i)** we have that both of the 4-cuts that cross are essential. To see this, observe that in either of cases **(ii)** to **(iv)** we have that the lower corners of the square have degree 3. Therefore, no vertex that is contained in those corners can be 4-edge-connected with a vertex that is not contained in them. But the union of the lower corners is precisely a side of one of the 4-cuts C_1 and C_2. Thus, it cannot be that both C_1 and C_2 are essential 4-cuts in those cases. Therefore, we may assume w.l.o.g. (i.e., by possibly permuting the labels of the edges of each cut) that C_1 and C_2 cross as in Figure 5.2. □

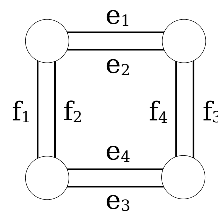


Figure 5.2: The crossing square of two essential 4-cuts $\{e_1, e_2, e_3, e_4\}$ and $\{f_1, f_2, f_3, f_4\}$.

5.2.2 Implied 4-cuts, and cyclic families of 4-cuts

The following is one of the implications of Lemma 5.4.

Lemma 5.5 (Implied 4-cut). *Let $C_1 = \{e_1, e_2, e_3, e_4\}$ and $C_2 = \{e_3, e_4, e_5, e_6\}$ be two distinct 4-cuts of a graph G. Then $C_3 = \{e_1, e_2, e_5, e_6\}$ is also a 4-cut of G.*

Proof. First, let us assume that C_1 and C_2 cross. By Lemma 5.4, we have that Figure 5.1 shows all the different ways in which C_1 and C_2 may cross. Then, since C_1 and C_2 share a pair of edges, notice that only case **(iv)** applies here, because this is the only case in which the crossing 4-cuts share two edges. Thus, C_1 and C_2 cross as in (a) of Figure 5.3. Then it is easy to see that Lemma 5.3 implies that the four corners of this

square are connected subgraphs of G. Then we observe that $\{e_1, e_2, e_5, e_6\}$ is indeed a 4-cut of G, because its deletion splits the graph into two connected components, but no proper subset of it has this property.

Now suppose that C_1 and C_2 are parallel. Let X be one of the two sides of C_1 and let X' be one of the two sides of C_2. Then, since C_1 and C_2 are parallel and distinct, we may assume w.l.o.g. that $X' \subset X$. Since C_1 and C_2 are distinct, we have $|C_1 \cap C_2| \neq 4$. By Lemma 5.2 we have $|C_1 \cap C_2| \neq 3$. Thus, $\{e_3, e_4\} \subseteq C_1 \cap C_2$ implies that $\{e_3, e_4\} = C_1 \cap C_2$. This implies that $\{e_1, e_2\} \cap \{e_5, e_6\} = \emptyset$. Since C_2 is a 4-cut, it is a 4-element set, and therefore $\{e_3, e_4\} \cap \{e_5, e_6\} = \emptyset$. Thus, we have $C_1 \cap \{e_5, e_6\} = \emptyset$. Since C_1 is a 4-cut of G, we have that the edges in C_1 are the only edges of G that join the sides of C_1. Thus, we have that either of e_5 and e_6 lies entirely within either $G[X]$ or $G[V \setminus X]$. Since $X' \subset X$ and both e_5 and e_6 have one endpoint in X', we infer that both e_5 and e_6 lie in $G[X]$.

Let $G' = G \setminus \{e_5, e_6\}$. Since $G[X]$ is connected, we have that $G'[X]$ consists of at most three connected components. Since X is one of the sides of C_1, Lemma 5.3 implies that $G[X]$ contains at most one bridge. Thus, it cannot be the case that $G'[X]$ consists of three connected components (because otherwise e_5 and e_6 would be two distinct bridges of $G[X]$). Let us suppose, for the sake of contradiction, that $G'[X]$ is connected. Let $G'' = G' \setminus \{e_3, e_4\}$. Then, since neither of e_3 and e_4 lies within $G[X]$, we have that $G''[X]$ is connected. Furthermore, since neither of $\{e_3, e_4, e_5, e_6\}$ lies within $G[V \setminus X]$, we have that $G''[V \setminus X]$ is connected. Then, notice that we have $E_{G''}[X, V \setminus X] = \{e_1, e_2\}$. But this implies that $G'' = G' \setminus \{e_3, e_4\} = G \setminus C_2$ is connected, in contradiction to the fact that C_2 is a 4-cut of G. This shows that $G'[X]$ is disconnected. Therefore, we have that $G'[X]$ consists of two connected components X_1 and X_2 such that $E[X_1, X_2] = \{e_5, e_6\}$. Notice that none of the subgraphs $G[X_1]$, $G[X_2]$ and $G[V \setminus X]$ contains edges from $\{e_1, e_2, e_3, e_4, e_5, e_6\}$.

Let $Y = V \setminus X$. Now we will determine the edge-sets $E[X_1, Y]$ and $E[X_2, Y]$. Since X is one of the sides of C_1, we have $C_1 = E[X, Y]$. Thus, since $E[X, Y] = E[X_1, Y] \sqcup E[X_2, Y]$, we have $E[X_1, Y] \sqcup E[X_2, Y] = \{e_1, e_2, e_3, e_4\}$. Notice that $\partial(X_1) = \{e_5, e_6\} \sqcup E[X_1, Y]$ and $\partial(X_2) = \{e_5, e_6\} \sqcup E[X_2, Y]$. Since G is 3-edge-connected, we have $|\partial(X_1)| \geq 3$ and $|\partial(X_2)| \geq 3$. Thus, we have that both $E[X_1, Y]$ and $E[X_2, Y]$ contain at least one edge from C_1.

Let us suppose, for the sake of contradiction, that one of $E[X_1, Y]$ and $E[X_2, Y]$ contains precisely one edge from C_1. Then we may assume w.l.o.g. that $E[X_1, Y]$

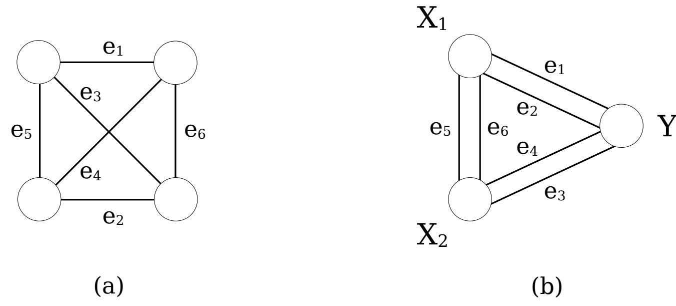


Figure 5.3: The possible arrangements of two distinct 4-cuts of the form $C_1 = \{e_1, e_2, e_3, e_4\}$ and $C_2 = \{e_3, e_4, e_5, e_6\}$ (up to swapping the labels e_3 and e_4). In (a), C_1 and C_2 cross. In (b), C_1 and C_2 are parallel. In either case, we have that $\{e_1, e_2, e_5, e_6\}$ is also a 4-cut.

contains precisely one edge from C_1. First, let us assume that $E[X_1, Y] = \{e_1\}$. Then we have $E[X_2, Y] = \{e_2, e_3, e_4\}$. But then we have that $G \setminus \{e_3, e_4, e_5, e_6\}$ is connected, because, in $G \setminus C_2$, X_1 is connected with $V \setminus X$ through e_1, and X_2 is connected with $V \setminus X$ through e_2. This contradicts the fact that C_2 is a 4-cut of G. Similarly, if we assume that $E[X_1, Y] = \{e_2\}$, then with the same reasoning we get a contradiction to the fact that C_2 is a 4-cut of G. Now let us assume that $E[X_1, Y] = \{e_3\}$. Then we have that $\partial(X_1) = \{e_3, e_5, e_6\}$. This implies that $\{e_3, e_5, e_6\}$ is a 3-cut of G, in contradiction to the fact that C_2 is a 4-cut of G. Similarly, if we assume that $E[X_1, Y] = \{e_4\}$, then with the same reasoning we get a contradiction to the fact that C_2 is a 4-cut of G. This shows that both $E[X_1, Y]$ and $E[X_2, Y]$ contain at least two edges from C_1. Since $E[X_1, Y] \sqcup E[X_2, Y] = C_1$ and $|C_1| = 4$, this implies that $|E[X_1, Y]| = |E[X_2, Y]| = 2$.

Let us suppose, for the sake of contradiction, that one of $E[X_1, Y]$ and $E[X_2, Y]$ is $\{e_1, e_3\}$. Then we may assume w.l.o.g. that $E[X_1, Y] = \{e_1, e_3\}$. Then it is easy to see that $\partial(X_1) = \{e_5, e_6, e_1, e_3\}$ is a 4-cut of G. But then we have $|\partial(X_1) \cap C_2| = 3$, contradicting Lemma 5.2. This shows that none of $E[X_1, Y]$ and $E[X_2, Y]$ is $\{e_1, e_3\}$. Similarly, we can show that none of $E[X_1, Y]$ and $E[X_2, Y]$ is $\{e_1, e_4\}$. Thus, we have that one of $E[X_1, Y]$ and $E[X_2, Y]$ is $\{e_1, e_2\}$, and the other is $\{e_3, e_4\}$. Let us assume w.l.o.g. that $E[X_1, Y] = \{e_1, e_2\}$ and $E[X_2, Y] = \{e_3, e_4\}$. Then we have a situation as that depicted in (b) of Figure 5.3. Then we observe that $\{e_1, e_2, e_5, e_6\}$ is indeed a 4-cut of G, since its deletion splits the graph into two connected components, but no proper subset of it has this property. □

Lemma 5.5 motivates the following definition.

Definition 5.4 (Implicating Sequence). Let $\mathcal{C}$ be a collection of 4-cuts of G, let $p_1, \ldots, p_{k+1}$ be a sequence of pairs of edges, and let $C_1, \ldots, C_k$ be a sequence of 4-cuts from $\mathcal{C}$, such that $C_i = p_i \cup p_{i+1}$ for every $i \in \{1, \ldots, k\}$. Then $C_1, \ldots, C_k$ is called an implicating sequence of $\mathcal{C}$. Furthermore, if $C = p_1 \cup p_{k+1}$ is a 4-cut of G, then we say that $\mathcal{C}$ implies C through the pair of edges p_1 (or equivalently: through the pair of edges p_{k+1}).

Note 5.1. We note the following two facts, which are immediate consequences of Definition 5.4 that we will use throughout. First, every 4-cut $C \in \mathcal{C}$ is implied by $\mathcal{C}$, through any pair of edges that is contained in C. And second, if $C_1, \ldots, C_k$ is an implicating sequence of $\mathcal{C}$, then, for every $i \in \{1, \ldots, k-1\}$, either $C_i = C_{i+1}$ or $C_i \cap C_{i+1} = p_{i+1}$ (as a consequence of $C_i = p_i \cup p_{i+1}$, $C_{i+1} = p_{i+1} \cup p_{i+2}$, and Lemma 5.2).

Lemma 5.6. *Let $\mathcal{C}$ be a collection of 4-cuts of G, and let $C_1, \ldots, C_k$ be an implicating sequence of $\mathcal{C}$ with $k > 1$. Let $p = C_1 \setminus C_2$, let $q = C_k \setminus C_{k-1}$, and suppose that $\emptyset \neq p \neq q \neq \emptyset$. Then $p \cup q$ is a 4-cut implied by $\mathcal{C}$ through p.*

Proof. This follows inductively by a repeated application of Lemma 5.5. First we consider the case $k = 2$. Thus, let C_1, C_2 be an implicating sequence of $\mathcal{C}$ such that $C_1 \setminus C_2 \neq C_2 \setminus C_1$. Then we have that $C_1 \cap C_2$ is a pair of edges. Thus, Lemma 5.5 implies that $C' = (C_1 \setminus C_2) \cup (C_2 \setminus C_1)$ is a 4-cut. By definition, we have that C' is implied by $\mathcal{C}$ through the pair of edges $C_1 \setminus C_2$.

Now let us suppose that the conclusion of the lemma holds for a $k \geq 2$. We will show that it also holds for $k + 1$. So let $C_1, \ldots, C_{k+1}$ be an implicating sequence of $\mathcal{C}$ such that $\emptyset \neq C_1 \setminus C_2 \neq C_{k+1} \setminus C_k \neq \emptyset$. Then there is a sequence $p_1, \ldots, p_{k+2}$ of pairs of edges such that $C_i = p_i \cup p_{i+1}$, for every $i \in \{1, \ldots, k+1\}$. If $C_k = C_{k-1}$, then $C_1, \ldots, C_{k-1}, C_{k+1}$ is an implicating sequence of $\mathcal{C}$ with $\emptyset \neq C_1 \setminus C_2 \neq C_{k+1} \setminus C_{k-1} \neq \emptyset$, and the conclusion holds due to the inductive hypothesis. So let us assume that $C_k \neq C_{k-1}$. Then we have $p_1 = C_1 \setminus C_2$, $p_{k+1} = C_k \setminus C_{k-1}$, $p_{k+2} = C_{k+1} \setminus C_k$, and $p_1 \neq p_{k+2}$.

Our goal is to show that $p_1 \cup p_{k+2}$ is a 4-cut of G (then, by definition, $C_1, \ldots, C_{k+1}$ is an implicating sequence of $\mathcal{C}$ that demonstrates that $p_1 \cup p_{k+2}$ is implied by $\mathcal{C}$ through p_1). Due to the inductive hypothesis, we have that either $p_1 = p_{k+1}$, or $p_1 \cup p_{k+1}$ is a 4-cut of G. If $p_1 = p_{k+1}$, then, since $C_{k+1} = p_{k+1} \cup p_{k+2}$, we have that $p_1 \cup p_{k+2}$ is a 4-cut of G. So let us assume that $p_1 \cup p_{k+1}$ is a 4-cut of G. Then, since $C_{k+1} = p_{k+1} \cup p_{k+2}$

and $p_1 \neq p_{k+2}$, we have that $p_1 \cup p_{k+1}$ and $p_{k+1} \cup p_{k+2}$ are two distinct 4-cuts. Thus, Lemma 5.5 implies that $p_1 \cup p_{k+2}$ is a 4-cut of G. This concludes the proof. □

We extend the terminology of Definition 5.4 as follows. Let $\mathcal{C}$ and $\mathcal{C}'$ be two collections of 4-cuts such that every 4-cut in $\mathcal{C}'$ is implied by $\mathcal{C}$. Then we say that $\mathcal{C}$ implies the collection of 4-cuts $\mathcal{C}'$.

Note 5.2. Despite what its name suggests, we note that the relation of implication between collections of 4-cuts is not transitive. In other words, if a collection of 4-cuts $\mathcal{C}$ implies a collection of 4-cuts $\mathcal{C}'$, and $\mathcal{C}'$ implies a collection of 4-cuts $\mathcal{C}''$, then it is not necessarily true that $\mathcal{C}$ implies $\mathcal{C}''$. An example for that is given in Figure 5.4. The next lemma provides a sufficient condition under which a kind of transitivity holds.

Lemma 5.7. *Let $\mathcal{C}$ be a collection of 4-cuts, and let C and C' be two 4-cuts such that $C \in \mathcal{C}$ and $\mathcal{C}$ implies C' through a pair of edges p. Suppose that $\{C, C'\}$ implies a 4-cut C'' through p. Then $\mathcal{C}$ implies C'' through p.*

Proof. Let us assume that $C'' \notin \{C, C'\}$, because otherwise the lemma follows trivially. Since C' is implied by $\mathcal{C}$ through the pair of edges p, we have that $p \subset C'$. Let $q = C' \setminus p$. Then we have $C' = p \cup q$. Since C'' is implied by C and C' through p, and since $p \subset C'$ and $C'' \neq C'$, we have that $q \subset C$ and $p \subset C''$. Furthermore, let $q' = C \setminus q$. Then, $C = q \cup q'$ and $C'' = p \cup q'$. Now, if $C' \in \mathcal{C}$, then we obviously have that C'' is implied by $\mathcal{C}$ through the pair of edges p. Otherwise, since C' is implied by $\mathcal{C}$ through p, we have that there is a sequence of 4-cuts $C_1, \dots, C_k$ in $\mathcal{C}$, and a sequence $p_1, \dots, p_{k+1}$ of pairs of edges, such that $p_1 = p$, $p_{k+1} = q$, and $C_i = p_i \cup p_{i+1}$, for every $i \in \{1, \dots, k\}$. Thus, the existence of the sequence of 4-cuts $C_1, \dots, C_k, C$ in $\mathcal{C}$, demonstrates that C'' is implied by $\mathcal{C}$ through the pair of edges p. □

A collection of 4-cuts that implies all 4-cuts of G, is called a *complete* collection of 4-cuts of G. One of our main contributions in this chapter is the following:

Theorem 5.2. *There is a linear-time algorithm that, given a 3-edge-connected graph G, computes a complete collection of 4-cuts of G, that has size $O(n)$.*

We prove Theorem 5.2 in Section 5.5. We note that it seems non-trivial to even establish the existence of a complete collection of 4-cuts of G that has size $O(n)$. This fact is implied through the analysis of the algorithm that computes $\mathcal{C}$.

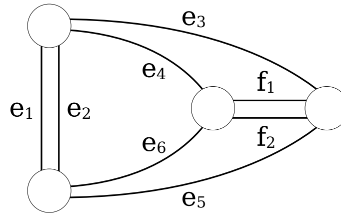


Figure 5.4: Let $\mathcal{C}$ be the collection of 4-cuts $\{\{e_1, e_2, e_3, e_4\}, \{e_3, e_5, f_1, f_2\}, \{e_4, e_6, f_1, f_2\}\}$. Then the collection of all 4-cuts implied by $\mathcal{C}$ is given by $\mathcal{C}' = \mathcal{C} \cup \{\{e_3, e_4, e_5, e_5\}\}$. Now notice that $\mathcal{C}'$ implies the 4-cut $\{e_1, e_2, e_5, e_6\}$. However, this 4-cut is not implied by $\mathcal{C}$.

Definition 5.5. Let $\mathcal{F} = \{p_1, \dots, p_k\}$, with $k \geq 2$, be a collection of pairs of edges with the property that $p_i \cup p_j$ is a 4-cut of G, for every $1 \leq i < j \leq k$, and let $\mathcal{C}$ be the collection of all such 4-cuts. Then we say that $\mathcal{F}$ generates $\mathcal{C}$.

The following concept is motivated by the structure of crossing 4-cuts that appears in **(i)** of Figure 5.1.

Definition 5.6. (Cyclic families of 4-cuts.) Let $\{p_1, \dots, p_k\}$, with $k \geq 3$, be a collection of pairs of edges of G that generates a collection $\mathcal{C}$ of 4-cuts of G. Suppose that there is a partition $\{X_1, \dots, X_k\}$ of $V(G)$ with the property that $G[X_i]$ is connected for every $i \in \{1, \dots, k\}$, and $E[X_i, X_{i+_k 1}] = p_i$ for every $i \in \{1, \dots, k\}$. Then $\mathcal{C}$ is called a cyclic family of 4-cuts. (See Figure 5.5.)

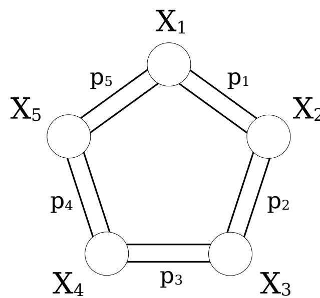


Figure 5.5: A cyclic family of 4-cuts generated by the collection of pairs of edges $\{p_1, p_2, p_3, p_4, p_5\}$.

In turns out that a cyclic family of 4-cuts has a lot of properties that are helpful in

order to derive efficiently the 5-edge-connected components from a complete collection of 4-cuts. The name "cyclic" refers to the structure of the graph that is formed by shrinking every X_i, $i \in \{1, \dots, k\}$, into a single vertex. This has the structure of a cycle (if we ignore edge multiplicities), as proved in the following lemma. Throughout this work, we may use Lemma 5.8 without explicit mention.

Lemma 5.8. *Let $\mathcal{F} = \{p_1, \dots, p_k\}$, with $k \geq 3$, be a collection of pairs of edges that generates a collection of 4-cuts of G such that there is a partition $\{X_1, \dots, X_k\}$ of $V(G)$ with the property that $G[X_i]$ is connected for every $i \in \{1, \dots, k\}$, and $E[X_i, X_{i+_k 1}] = p_i$ for every $i \in \{1, \dots, k\}$. Then $\partial(X_i) = p_i \cup p_{i-_k 1}$, for every $i \in \{1, \dots, k\}$. Furthermore, for every $i, j \in \{1, \dots, k\}$ with $i \neq j$, the connected components of $G \setminus (p_i \cup p_j)$ are given by $X_{i+_k 1} \cup X_{i+_k 2} \cup \dots \cup X_j$ and $X_{j+_k 1} \cup X_{j+_k 2} \cup \dots \cup X_i$. (See Figure 5.5.)*

Proof. Let $i \in \{1, \dots, k\}$. Since $\{X_1, \dots, X_k\}$ is a partition of V, we have $\partial(X_i) = (E[X_i, X_1] \cup \dots \cup E[X_i, X_k]) \setminus E[X_i, X_i]$. Since $E[X_i, X_{i+_k 1}] = p_i$ and $E[X_i, X_{i-_k 1}] = p_{i-_k 1}$, this implies that $p_i \cup p_{i-_k 1} \subseteq \partial(X_i)$. If we assume that $E[X_i, X_j] \neq \emptyset$ for some $j \in \{1, \dots, k\} \setminus \{i -_k 1, i, i +_k 1\}$, then it is easy to see that $G \setminus (p_i \cup p_{i-_k 1})$ remains connected, in contradiction to the fact that $\mathcal{F}$ generates a collection of 4-cuts of G. This shows that $\partial(X_i) = p_i \cup p_{i-_k 1}$.

Now let i and j be two distinct indices in $\{1, \dots, k\}$. If $|i - j| = 1$ or $\{i, j\} = \{1, k\}$, then we may assume w.l.o.g. that $j = i -_k 1$. Then we have shown that $p_i \cup p_j = \partial(X_i)$, and therefore the connected components of G are given by X_i and $(X_1 \cup \dots \cup X_k) \setminus X_i$. So let us assume that $|i - j| > 1$ and $\{i, j\} \neq \{1, k\}$. Then we have that none of p_i and p_j intersects with any of $E[X_{i+_k 1}, X_{i+_k 2}], \dots, E[X_{j-_k 1}, X_j]$ or $E[X_{j+_k 1}, X_{j+_k 2}], \dots, E[X_{i-_k 1}, X_i]$. Furthermore, we have that $E[X_i, X_{i+_k 1}] \setminus (p_i \cup p_j) = \emptyset$ and $E[X_j, X_{j+_k 1}] \setminus (p_i \cup p_j) = \emptyset$. Finally, we have that all graphs $G[X_1] \setminus (p_i \cup p_j), \dots, G[X_k] \setminus (p_i \cup p_j)$ remain connected. Thus, we can see that the connected components of $G \setminus (p_i \cup p_j)$ are given by $X_{i+_k 1} \cup X_{i+_k 2} \cup \dots \cup X_j$ and $X_{j+_k 1} \cup X_{j+_k 2} \cup \dots \cup X_i$. □

In Proposition 5.2, we show that if a collection of pairs of edges $\mathcal{F}$, with $|\mathcal{F}| > 3$, generates a collection of 4-cuts $\mathcal{C}$, then $\mathcal{C}$ is a cyclic family of 4-cuts. The next lemma analyzes the case $|\mathcal{F}| = 3$, which provides the base step in order to prove inductively Proposition 5.2.

Lemma 5.9. *Let $\mathcal{F} = \{p_1, p_2, p_3\}$ be a collection of pairs of edges that generates a collection of 4-cuts of G. Then $G' = G \setminus (p_1 \cup p_2 \cup p_3)$ consists of either three or four connected*

components. If G' consists of three connected components X_1, X_2, X_3, then (by possibly permuting the indices) we have $E[X_1, X_2] = p_1$, $E[X_2, X_3] = p_2$, and $E[X_3, X_1] = p_3$ (i.e., $\mathcal{F}$ generates a cyclic family of 4-cuts of G). If G' consists of four connected components, then $\mathcal{F}$ is maximal w.r.t. the property of generating a collection of 4-cuts of G. Furthermore, if $p_1 = \{e_1, e_2\}$, $p_2 = \{e_3, e_4\}$ and $p_3 = \{e_5, e_6\}$, then the quotient graph of G that is formed by shrinking the connected components of G' into single vertices is shown in (a) of Figure 5.3 (after possibly swapping the labels e_3 and e_4).

Proof. Let $p_1 = \{e_1, e_2\}$, $p_2 = \{e_3, e_4\}$ and $p_3 = \{e_5, e_6\}$. By assumption we have that $p_1 \cup p_2$ is a 4-cut of G, and so let $X, V \setminus X$ be the two connected components of $G \setminus (p_1 \cup p_2)$. Then, either (1) p_3 is contained entirely within $G[X]$ or $G[V \setminus X]$, or (2) both $G[X]$ and $G[V \setminus X]$ contain an edge from p_3. We will show that in case (1) G' consists of three connected components, in case (2) G' consists of four connected components, and in either case the claims of lemma hold true.

Let us consider case (1) first. Then we may assume, w.l.o.g., that p_3 lies entirely within $G[X]$. Since $G[X]$ is connected, we have that $G[X] \setminus p_3$ is split into at most three connected components. Now, if $G[X] \setminus p_3$ is connected, then $G \setminus (p_2 \cup p_3)$ is also connected (since $E_{G \setminus (p_2 \cup p_3)}[X, V \setminus X] = p_1$, and both $(G \setminus (p_2 \cup p_3))[X]$ and $(G \setminus (p_2 \cup p_3))[V \setminus X]$ remain connected), contradicting the fact that $p_2 \cup p_3$ is a 4-cut of G. Thus, $G[X] \setminus p_3$ is split into either two or three connected components. Let us assume, for the sake of contradiction, that $G[X] \setminus p_3$ is split into three connected components. This implies that both edges of p_3 are bridges of $G[X]$. But since $|\partial(X)| = |p_1 \cup p_2| = 4$, Lemma 5.3 implies that $G[X]$ contains at most one bridge, a contradiction. Thus, we have that $G[X] \setminus p_3$ consists of two connected components, C_1 and C_2, and $E[C_1, C_2] = p_3$.

Now we claim that either of $E[C_1, V \setminus X]$ and $E[C_2, V \setminus X]$ contains exactly two edges from $p_1 \cup p_2$. To see this, first notice that $\partial(C_1) = E[C_1, C_2] \sqcup E[C_1, V \setminus X]$, $\partial(C_2) = E[C_1, C_2] \sqcup E[C_2, V \setminus X]$, and $E[C_1, V \setminus X] \sqcup E[C_2, V \setminus X] = p_1 \cup p_2$. Since $E[C_1, C_2] = p_3$ and G is 3-edge-connected, this implies that either of $E[C_1, V \setminus X]$ and $E[C_2, V \setminus X]$ contains at least one edge from $p_1 \cup p_2$. Let us suppose, for the sake of contradiction, that one of $E[C_1, V \setminus X]$ and $E[C_2, V \setminus X]$ contains exactly one edge from $p_1 \cup p_2$. Then, w.l.o.g., we may assume that $E[C_1, V \setminus X] = \{e_1\}$ (the other cases are treated similarly). Then we have $\partial(C_1) = \{e_1, e_5, e_6\}$, and therefore $G \setminus \{e_1, e_5, e_6\}$ is not connected. But this contradicts the fact that $p_1 \cup p_3$ is a 4-cut of G. This shows that either of $E[C_1, V \setminus X]$ and $E[C_2, V \setminus X]$ contains at least two edges from $p_1 \cup p_2$. Then, since $E[C_1, V \setminus X] \sqcup E[C_2, V \setminus X] = p_1 \cup p_2$ and $|p_1 \cup p_2| = 4$, we infer that either

of $E[C_1, V \setminus X]$ and $E[C_2, V \setminus X]$ contains exactly two edges from $p_1 \cup p_2$.

Now, if $E[C_1, V \setminus X] = p_1$ (and $E[C_2, V \setminus X] = p_2$), or $E[C_1, V \setminus X] = p_2$ (and $E[C_2, V \setminus X] = p_1$), then we basically have the lemma for the case that G' consists of three connected components. But if we assume the contrary, then, w.l.o.g., let $E[C_1, V \setminus X] = \{e_1, e_3\}$ (and $E[C_2, V \setminus X] = \{e_2, e_4\}$). But this means that $p_3 \cup p_1$ is not a 4-cut of G (because C_1 remains connected with $V \setminus X$ in $G \setminus (p_3 \cup p_1)$ through e_3, and C_2 remains connected with $V \setminus X$ in $G \setminus (p_3 \cup p_1)$ through e_4), a contradiction.

Now let us consider case (2). Then we may assume, w.l.o.g., that e_5 is contained in $G[X]$ and e_6 is contained in $G[V \setminus X]$. Then $G[X] \setminus e_5$ is split into at most two connected components. Let us suppose, for the sake of contradiction, that $G[X] \setminus e_5$ is connected. Then $G[X] \setminus (p_3 \cup p_1)$ is also connected (because the only edge from $p_3 \cup p_1$ that lies in $G[X]$ is e_5), and therefore $p_3 \cup p_1$ is not a 4-cut of G (because the endpoints of e_5 remain connected in $G \setminus (p_3 \cup p_1)$), a contradiction. Thus, we have that e_5 is a bridge of $G[X]$. Similarly, e_6 is a bridge of $G[V \setminus X]$. So let C_1, C_2 be the connected components of $G[X] \setminus e_5$, and let C_3, C_4 be the connected components of $G[V \setminus X] \setminus e_6$. Then we have $E[C_1, C_2] = e_5$ and $E[C_3, C_4] = e_6$. This shows that G' consists of four connected components, C_1, C_2, C_3, C_4.

Now we have to consider how the vertex sets C_1, C_2, C_3, C_4 are interconnected using the edges from $p_1 \cup p_2$. First, it is not difficult to see that every one of C_1, C_2, C_3, C_4 must have exactly two edges from $p_1 \cup p_2$ as boundary edges, because otherwise we violate the fact that G is 3-edge-connected. Then, we can see that no one of C_1, C_2, C_3, C_4 can have either both edges from p_1 or both edges from p_2 as boundary edges, because otherwise we violate the fact that $p_1 \cup p_3$ and $p_2 \cup p_3$ are 4-cuts of G (and therefore no proper subsets of them can destroy the connectivity of G upon removal). Finally, we can see that both C_1 and C_2 must be connected with both C_3 and C_4 using edges from $p_1 \cup p_2$, because otherwise p_3 is a 2-cut of G. Thus, we may assume, w.l.o.g., that $E[C_1, C_3] = e_1$, $E[C_1, C_4] = e_3$, $E[C_2, C_3] = e_4$, and $E[C_2, C_4] = e_2$. Notice that this is precisely the situation depicted in (a) of Figure 5.3.

It remains to show (still being in case (2)), that there is no pair of edges $p_4 = \{e_7, e_8\}$, with $p_4 \notin \mathcal{F}$, such that $\mathcal{F} \cup \{p_4\}$ generates a collection of 4-cuts of G. So let us assume the contrary. Notice that this implies that $\{e_7, e_8\} \cap (p_1 \cup p_2 \cup p_3) = \emptyset$, because otherwise $p_4 \cup p_i$ is not a 4-element set, for some $i \in \{1, 2, 3\}$. This means that either both edges from p_4 lie entirely within $G[C_i]$, for some $i \in \{1, 2, 3, 4\}$, or that both $G[C_i]$ and $G[C_j]$ contain edges from p_4, for some $i, j \in \{1, 2, 3, 4\}$ with $i \neq j$.

Let us consider the first case first, and so let us assume w.l.o.g. that p_4 is contained within $G[C_1]$. Then $G[C_1] \setminus p_4$ consists of at most three connected components. We will show that all three cases concerning the number of connected components of $G[C_1] \setminus p_4$ lead to a contradiction. If we assume that $G[C_1] \setminus p_4$ is connected, then we contradict the fact that $p_1 \cup p_4$ is a 4-cut of G (because the endpoints of p_4 remain connected in $G \setminus (p_1 \cup p_4)$). Let us assume that $G[C_1] \setminus p_4$ consists of two connected components D_1 and D_2. Then we have $E[D_1, D_2] = p_4$. Now, if we consider all the different combinations of the incidence relation of the boundary edges of C_1 with D_1 and D_2, we will see that, in every possible case, we either contradict the fact that G is 3-edge-connected (e.g., if all the boundary edges of C_1 are incident to D_1), or the fact that p_4 must form a 4-cut of G will all of p_1, p_2, p_3 (so, e.g., if e_1 and e_3 are incident to D_1, and e_5 is incident to D_2, then we have that $p_4 \cup \{e_5\}$ is a 3-cut of G, contradicting the fact that $p_4 \cup p_3$ is a 4-cut of G). Finally, let us assume that $G[C_1] \setminus p_4$ consists of three connected components D_1, D_2 and D_3. Then w.l.o.g. we may assume that $E[D_1, D_2] = \{e_7\}$ and $E[D_2, D_3] = \{e_8\}$. But now, since C_1 has precisely three boundary edges in G, we have that, no matter the incidence relation of those boundary edges to D_1, D_2, D_3, we violate the fact that G is 3-edge-connected.

Finally, it remains to consider the case that the edges from p_4 lie in two different subgraphs $G[C_i], G[C_j]$, for some $i, j \in \{1, 2, 3, 4\}$ with $i \neq j$. Due to the symmetry of the interconnections between C_1, C_2, C_3, C_4, we may assume w.l.o.g. that e_7 is contained in $G[C_1]$ and e_8 is contained in $G[C_2]$. Observe that $G[C_1] \setminus e_7$ and $G[C_2] \setminus e_8$ cannot both be connected, because otherwise $p_1 \cup p_4$ (or $p_2 \cup p_4$, or $p_3 \cup p_4$) is not a 4-cut of G. Thus we may assume w.l.o.g. that $G[C_1] \setminus e_7$ is not connected, and let D_1, D_2 be its connected components. Then we have $E[D_1, D_2] = e_7$. But since C_1 has only three boundary edges in G, this violates the fact that G is 3-edge-connected (because either e_7 is a bridge of G, or it forms a 2-cut with one of the boundary edges of C_1 in G). Thus we have shown that $\mathcal{F}$ cannot be extended with one more pair of edges p_4 such that $\mathcal{F} \cup \{p_4\}$ generates a collection of 4-cuts of G. □

We consider the last case in Lemma 5.9 to be degenerate, because the pairs of edges p_1, p_2, p_3 are so entangled with the components induced by the three 4-cuts generated by them, that this collection of pairs of edges cannot be extended to a larger collection of pairs of edges that also generates a collection of 4-cuts. This singularity cannot occur with larger collections of pairs of edges that generate 4-cuts, because

(loosely speaking) the ability of every pair of them to provide a 4-cut forces them to produce a more organized system of 4-cuts (see Proposition 5.2). Formally, if $\mathcal{F}$ is collection of pairs of edges with $|\mathcal{F}| = 3$ that generates a collection of 4-cuts $\mathcal{C}$ such that $G \setminus \bigcup \mathcal{F}$ consists of four connected components, then $\mathcal{C}$ is a called a *degenerate* family of 4-cuts.

Corollary 5.4. *Let $\mathcal{C}$ be a degenerate family of 4-cuts. Then $\mathcal{C}$ consists of three non-essential 4-cuts.*

Proof. By definition, we have that $\mathcal{C}$ is generated by a collection of pairs of edges $\mathcal{F}$ with $|\mathcal{F}| = 3$ such that $G \setminus \bigcup \mathcal{F}$ consists of four connected components. Let $\mathcal{F} = \{\{e_1, e_2\}, \{e_3, e_4\}, \{e_5, e_6\}\}$. Then, by Lemma 5.9, we can assume w.l.o.g. that the edges in $\bigcup \mathcal{F}$ are arranged as in (a) of Figure 5.3. Thus, it is easy to see that every pair of vertices that are separated by a 4-cut from $\mathcal{C}$ can also be separated by a 3-cut. We conclude that none of the three 4-cuts in $\mathcal{C}$ is essential. $\square$

Proposition 5.2. *Let $\mathcal{C}$ be a collection of 4-cuts of G that is generated by a collection $\mathcal{F} = \{p_1, \dots, p_k\}$ of k pairs of edges, with $k \geq 4$. Then there is a partition $\{X_1, \dots, X_k\}$ of $V(G)$, such that $G[X_i]$ is connected for every $i \in \{1, \dots, k\}$, and $E[X_i, X_{i+_k 1}] = p_i$ for every $i \in \{1, \dots, k\}$. In other words, $\mathcal{C}$ is a cyclic family of 4-cuts of G.*

Proof. For every integer k' with $3 \leq k' \leq k$, we define the proposition $\Pi(k') \equiv$ "there is a partition $\{X_1, \dots, X_{k'}\}$ of $V(G)$, such that $G[X_i]$ is connected for every $i \in \{1, \dots, k'\}$, and $E[X_i, X_{i+_{k'} 1}] = p_i$ for every $i \in \{1, \dots, k'\}$". We will show inductively that $\Pi(k)$ is true.

If we take the subcollection $\{p_1, p_2, p_3\}$ of $\mathcal{F}$, then this generates a collection of 4-cuts of G; but since $k \geq 4$, this collection is not maximal w.r.t. the property of generating a collection of 4-cuts of G. Thus, by Lemma 5.9 we have that there is a partition $\{X_1, X_2, X_3\}$ of V, such that $G[X_i]$ is connected for every $i \in \{1, 2, 3\}$, and $E[X_i, X_{i+_3 1}] = p_i$ for every $i \in \{1, 2, 3\}$. This establishes $\Pi(3)$ (the base step of our induction).

Now let us suppose that $\Pi(k')$ is true, for some k' with $3 \leq k' < k$. This means that there exists a partition $\{X_1, \dots, X_{k'}\}$ of $V(G)$ such that $G[X_i]$ is connected for every $i \in \{1, \dots, k'\}$, and $E[X_i, X_{i+_{k'} 1}] = p_i$ for every $i \in \{1, \dots, k'\}$. We will show that $\Pi(k'+1)$ is also true.

Since $p_{k'+1}$ forms a 4-cut with every pair of edges in $\{p_1, \dots, p_{k'}\}$, we have that none of the edges in $p_{k'+1}$ lies in $p_1 \cup \dots \cup p_{k'}$ (because otherwise $p_{k'+1} \cup p_i$ would not be

a 4-element set, for some $i \in \{1, \ldots, k'\}$). Thus, either $p_{k'+1}$ lies entirely within $G[X_i]$, for some $i \in \{1, \ldots, k'\}$, or there are $i, j \in \{1, \ldots, k'\}$, with $i \neq j$, such that both $G[X_i]$ and $G[X_j]$ contain an edge from $p_{k'+1}$. Let us suppose, for the sake of contradiction, that the second case is true. Let $p_{k'+1} = \{e_1, e_2\}$, and let us assume, w.l.o.g., that e_1 is contained in $G[X_i]$ and e_2 is contained in $G[X_j]$. Since $k' \geq 3$, we may assume w.l.o.g. that $j \neq i -_{k'} 1$. Now, since $p_i \cup p_{k'+1}$ is a 4-cut of G and e_1 is the only edge from $p_i \cup p_{k'+1}$ that lies in $G[X_i]$, we have that e_1 is a bridge of $G[X_i]$ (because otherwise the endpoints of e_1 would remain connected in $G[X_i] \setminus (p_i \cup p_{k'+1})$). So let C_1, C_2 be the connected components of $G[X_i] \setminus e_1$. Thus, we have $E[C_1, C_2] = \{e_1\}$. By applying Lemma 5.8 on the collection of pairs of edges $\{p_1, \ldots, p_{k'}\}$, we have that $\partial(X_i) = p_i \cup p_{i -_{k'} 1}$. Notice that $\partial(C_1) = E[C_1, C_2] \sqcup E[C_1, V \setminus X_i]$ and $\partial(C_2) = E[C_1, C_2] \sqcup E[C_2, V \setminus X_i]$. Thus, since the graph is 3-edge-connected and $E[C_1, V \setminus X_i] \sqcup E[C_2, V \setminus X_i] = \partial(X_i)$ and $|\partial(X_i)| = 4$, we have that both $E[C_1, V \setminus X_i]$ and $E[C_2, V \setminus X_i]$ must contain precisely two edges from $p_i \cup p_{i -_{k'} 1}$. If $E[C_1, V \setminus X_i] = p_i$, then we have that $p_i \cup \{e_1\}$ is a 3-cut of G, contradicting the fact that $p_i \cup \{e_1, e_2\}$ is a 4-cut of G. Similarly, we can reject the case $E[C_1, V \setminus X_i] = p_{i -_{k'} 1}$. Thus, we have that both $E[C_1, V \setminus X_i]$ and $E[C_2, V \setminus X_i]$ must intersect with both p_i and $p_{i -_{k'} 1}$. But then, since $G[X_{i -_{k'} 1}]$ is connected, and e_1, e_2 do not lie in $G[X_{i -_{k'} 1}]$, we have that C_1 and C_2 remain connected in $G \setminus (p_i \cup \{e_1, e_2\})$ (because both of them remain connected with $X_{i -_{k'} 1}$ in $G \setminus (p_i \cup \{e_1, e_2\})$ through the edges from $p_{i -_{k'} 1}$). This implies that the endpoints of e_1 remain connected in $G \setminus (p_i \cup \{e_1, e_2\})$, contradicting the fact that $p_i \cup \{e_1, e_2\}$ is a 4-cut of G. Thus, we have shown that $p_{k'+1}$ lies entirely within $G[X_i]$, for some $i \in \{1, \ldots, k'\}$.

Now, since $G[X_i]$ is connected, we have that $G[X_i] \setminus p_{k'+1}$ is split into at most three connected components. It cannot be the case that $G[X_i] \setminus p_{k'+1}$ is connected, because otherwise e.g. $p_1 \cup p_{k'+1}$ is not a 4-cut of G (because the endpoints of the edges from $p_{k'+1}$ would remain connected in $G \setminus (p_1 \cup p_{k'+1})$). Now let us suppose, for the sake of contradiction, that $G[X_i] \setminus p_{k'+1}$ is split into three connected components. This implies that both edges from $p_{k'+1}$ are bridges of $G[X_i]$. By applying Lemma 5.8 on the collection of pairs of edges $\{p_1, \ldots, p_{k'}\}$, we have that the boundary of X_i in G contains exactly four edges. Then, Lemma 5.3 implies that $G[X_i]$ contains at most one bridge – a contradiction. This shows that $G[X_i] \setminus p_{k'+1}$ consists of two connected components C_1 and C_2. Then we have $E[C_1, C_2] = p_{k'+1}$.

It remains to determine the incidence relation between C_1 and C_2 and the edges from $\partial(X_i) = p_i \cup p_{i -_{k'} 1}$. Notice that $\partial(C_1) = E[C_1, C_2] \sqcup E[C_1, V \setminus X_i]$ and $\partial(C_2) =$

$E[C_1, C_2] \sqcup E[C_2, V \setminus X_i]$. Thus, since the graph is 3-edge-connected, we have that either of $E[C_1, V \setminus X_i]$ and $E[C_2, V \setminus X_i]$ must contain at least one edge from $p_i \cup p_{i-_{k'}1}$. Therefore, since $|\partial(X_i)| = 4$, we can see that either of $E[C_1, V \setminus X_i]$ and $E[C_2, V \setminus X_i]$ must contain exactly two edges from $p_i \cup p_{i-_{k'}1}$, because otherwise we contradict the fact that $p_{k'+1} \cup p_i$ and $p_{k'+1} \cup p_{i-_{k'}1}$ are 4-cuts of G (and therefore no proper subset of them can disconnect G upon removal).

Now, we either have (1) $E[C_1, X_{i+_{k'}1}] = p_i$ (and $E[C_2, X_{i-_{k'}1}] = p_{i-_{k'}1}$), or (2) $E[C_1, X_{i-_{k'}1}] = p_{i-_{k'}1}$ (and $E[C_2, X_{i+_{k'}1}] = p_i$), or (3) both $E[C_1, X_{i+_{k'}1}]$ and $E[C_2, X_{i+_{k'}1}]$ intersect with p_i (and both $E[C_1, X_{i-_{k'}1}]$ and $E[C_2, X_{i-_{k'}1}]$ intersect with $p_{i-_{k'}1}$). Let us suppose, for the sake of contradiction, that the third case is true. But then we have that $p_i \cup p_{k'+1}$ is not a 4-cut of G, since both C_1 and C_2 remain connected with $X_{i-_{k'}1}$ through the edges from $p_{i-_{k'}1}$. Thus, we may assume, w.l.o.g., that $E[C_1, X_{i+_{k'}1}] = p_i$ and $E[C_2, X_{i-_{k'}1}] = p_{i-_{k'}1}$. Now we observe that, by renaming appropriately the sets C_1, C_2 and $\{X_1, \dots, X_{k'}\} \setminus \{X_i\}$, we have that there is a partition $\{X_1, \dots, X_{k'+1}\}$ of V, such that $G[X_i]$ is connected for every $i \in \{1, \dots, k'+1\}$, and $E[X_i, X_{i+_{k'+1}1}] = p_i$ for every $i \in \{1, \dots, k'+1\}$. Thus, $\Pi(k'+1)$ is also true, and the result follows inductively. □

Corollary 5.5. *Let $\mathcal{C}$ be a collection of 4-cuts with $|\mathcal{C}| \geq 3$ that is generated by a collection of pairs of edges. Suppose that $\mathcal{C}$ contains an essential 4-cut. Then $\mathcal{C}$ is a cyclic family of 4-cuts.*

Proof. If $|\mathcal{C}| = 3$, then, since $\mathcal{C}$ contains an essential 4-cut, Corollary 5.4 implies that $\mathcal{C}$ cannot be a degenerate family of 4-cuts. Thus, Lemma 5.9 implies that $\mathcal{C}$ is a cyclic family of 4-cuts. Otherwise, if $|\mathcal{C}| > 3$, then $\mathcal{C}$ is generated by a collection of pairs of edges that has size at least four, and therefore Proposition 5.2 implies that $\mathcal{C}$ is a cyclic family of 4-cuts. □

A collection $\mathcal{C}$ of 4-cuts is called trivial if $|\mathcal{C}| = 1$.

Lemma 5.10. *Let $\mathcal{F}$ be a collection of pairs of edges of G that generates a non-trivial collection $\mathcal{C}$ of 4-cuts of G. Then, $\mathcal{F}$ is unique w.r.t. the property of generating $\mathcal{C}$.*

Proof. Let us suppose, for the sake of contradiction, that there are two distinct collections $\mathcal{F}_1$ and $\mathcal{F}_2$ of pairs of edges of G that generate $\mathcal{C}$. Then, since $\mathcal{F}_1$ and $\mathcal{F}_2$ generate the same non-trivial collection of 4-cuts, we have that $|\mathcal{F}_1| = |\mathcal{F}_2| > 2$. Now, since $\mathcal{F}_1$ and $\mathcal{F}_2$ are distinct and $|\mathcal{F}_1| = |\mathcal{F}_2|$, there is a pair of edges $\{e, e'\} \in \mathcal{F}_2 \setminus \mathcal{F}_1$.

Let $C = \{e_1, e_2, e_3, e_4\}$ be a 4-cut in $\mathcal{C}$. Since $\mathcal{F}_1$ generates C, we may assume w.l.o.g. that $\{\{e_1, e_2\}, \{e_3, e_4\}\} \subset \mathcal{F}_1$. Now let us assume, for the sake of contradiction, that $\{e_1, e_2\} \in \mathcal{F}_2$. Since $\{e_1, e_2\} \in \mathcal{F}_1$ and $\{e, e'\} \notin \mathcal{F}_1$, we have $\{e_1, e_2\} \neq \{e, e'\}$. Then, since $\{e, e'\} \in \mathcal{F}_2$, we have that $C' = \{e_1, e_2, e, e'\}$ is a 4-cut in $\mathcal{C}$. But since $\{e, e'\} \notin \mathcal{F}_1$ and $\mathcal{F}_1$ also generates C', we have that either $\{e_1, e\} \in \mathcal{F}_1$ or $\{e_1, e'\} \in \mathcal{F}_1$. (Notice that none of e, e' can be e_2, for otherwise C' would not be a 4-element set.) But then we have that either $\{e_1, e_2\} \cup \{e_1, e\}$ or $\{e_1, e_2\} \cup \{e_1, e'\}$ is a 4-cut of G, a contradiction. This shows that $\{e_1, e_2\} \notin \mathcal{F}_2$. Thus, since $\mathcal{F}_2$ generates C, we may assume w.l.o.g. that $\{\{e_1, e_3\}, \{e_2, e_4\}\} \subset \mathcal{F}_2$.

Now, since $|\mathcal{F}_1| \geq 3$, there is a pair of edges $\{e_5, e_6\} \in \mathcal{F}_1 \setminus \{\{e_1, e_2\}, \{e_3, e_4\}\}$. Then we have that $\mathcal{F}_1$ generates $\{e_1, e_2, e_5, e_6\}$, and therefore $\mathcal{F}_2$ also generates $\{e_1, e_2, e_5, e_6\}$. Then, since $\{e_1, e_2\} \notin \mathcal{F}_2$, we have that either $\{e_1, e_5\} \in \mathcal{F}_2$ or $\{e_1, e_6\} \in \mathcal{F}_2$. Notice that $e_3 \notin \{e_5, e_6\}$, because otherwise we would have that $\{e_3, e_4\} \cup \{e_5, e_6\}$ is not a 4-element set, contradicting the fact that $\mathcal{F}_1$ generates a collection of 4-cuts. This implies that $\{e_1, e_3\} \neq \{e_1, e_5\}$ and $\{e_1, e_3\} \neq \{e_1, e_6\}$. But then, since $\{e_1, e_3\} \in \mathcal{F}_2$, we have that either $\{e_1, e_3\} \cup \{e_1, e_5\}$ or $\{e_1, e_3\} \cup \{e_1, e_6\}$ is a 4-cut of G, which is absurd. We conclude that there is a unique collection of pairs of edges of G that generates $\mathcal{C}$.

□

Let $\mathcal{C}$ be a cyclic family of 4-cuts. Then, Lemma 5.10 implies that there is a unique collection $\mathcal{F} = \{p_1, \dots, p_k\}$ of pair of edges that generates $\mathcal{C}$. Now, by definition, we have that there is a partition $\{X_1, \dots, X_k\}$ of $V(G)$ such that $G[X_i]$ is connected for every $i \in \{1, \dots, k\}$, and (w.l.o.g.) $E[X_i, X_{i+_k 1}] = p_i$ for every $i \in \{1, \dots, k\}$. Thus, by Lemma 5.8 we have that the connected components of $G \setminus \bigcup \mathcal{F}$ are precisely $X_1, \dots, X_k$. Then, we call $X_1, \dots, X_k$ the *corners* of the cyclic family $\mathcal{C}$ (considered either as vertex sets, or as subgraphs of G). For every $i \in \{1, \dots, k\}$ we have that $\partial(X_i) = p_i \cup p_{i-_k 1}$. We call the 4-cuts $\partial(X_1), \dots, \partial(X_k)$ the *$\mathcal{C}$-minimal 4-cuts*.

Let C be a 4-cut of G. If there is no collection $\mathcal{F}$ of pairs of edges with $|\mathcal{F}| > 2$ that generates (a collection of 4-cuts that contains) C, then C is called an *isolated* 4-cut.

Corollary 5.6. *Let C be an essential isolated 4-cut of G. Then, C is parallel with every essential 4-cut of G.*

Proof. Let us suppose, for the sake of contradiction, that there is an essential 4-cut C' such that C and C' cross. Let $C = \{e_1, e_2, e_3, e_4\}$, and let $C' = \{f_1, f_2, f_3, f_4\}$. Then, by Corollary 5.3 we have that C and C' must cross as in Figure 5.2. By Lemma 5.3,

we have that the four corners of this figure are connected, and therefore all of them constitute 4-cuts of G. But this implies that $\{\{e_1, e_2\}, \{e_3, e_4\}, \{f_1, f_2\}, \{f_3, f_4\}\}$ is a collection of pairs of edges that generates C – in contradiction to the fact that C is an isolated 4-cut. Thus, we conclude that C does not cross with other essential 4-cuts. $\square$

5.2.3 Properties of cyclic families of 4-cuts

The following two lemmata demonstrate the importance of minimal 4-cuts. Both of them are a consequence of the structure of the sides of the minimal 4-cuts (provided by Lemma 5.8).

Lemma 5.11. *Let $\mathcal{C}$ be a cyclic family of 4-cuts, and let C and C' be two $\mathcal{C}$-minimal 4-cuts. Then C and C' are parallel.*

Proof. Since $\mathcal{C}$ is a cyclic family of 4-cuts, there is a partition $\{X_1, \dots, X_k\}$ of $V(G)$, and a collection of pairs of edges $\{p_1, \dots, p_k\}$, such that $G[X_i]$ is connected for every $i \in \{1, \dots, k\}$, $E[X_i, X_{i+_k 1}] = p_i$ for every $i \in \{1, \dots, k\}$, and $\mathcal{C} = \{p_i \cup p_j \mid i, j \in \{1, \dots, k\} \text{ with } i \neq j\}$. Then, since C and C' are $\mathcal{C}$-minimal 4-cuts, there are $i, j \in \{1, \dots, k\}$ such that $C = p_i \cup p_{i+_k 1}$ and $C' = p_j \cup p_{j+_k 1}$. Then, by Lemma 5.8 we have that the connected components of $G \setminus C$ are X_i and $V \setminus X_i$, and the connected components of $G \setminus C'$ are X_j and $V \setminus X_j$. Since $\{X_1, \dots, X_k\}$ is a partition of $V(G)$, we have that either $X_i = X_j$ or $X_i \cap X_j = \emptyset$. Thus, the 4-cuts C and C' are parallel (as an immediate consequence of the definition). $\square$

Lemma 5.12 (A non-minimal 4-cut can be replaced by minimal 4-cuts)**.** *Let $\mathcal{C}$ be a cyclic family of 4-cuts, and let C be a 4-cut in $\mathcal{C}$ that separates two vertices x and y. Then, there is a $\mathcal{C}$-minimal 4-cut C' that also separates x and y.*

Proof. Since $\mathcal{C}$ is a cyclic family of 4-cuts, there is a partition $\{X_1, \dots, X_k\}$ of $V(G)$, and a collection of pairs of edges $\{p_1, \dots, p_k\}$, such that $G[X_i]$ is connected for every $i \in \{1, \dots, k\}$, $E[X_i, X_{i+_k 1}] = p_i$ for every $i \in \{1, \dots, k\}$, and $\mathcal{C} = \{p_i \cup p_j \mid i, j \in \{1, \dots, k\} \text{ with } i \neq j\}$. Then, there are $i, j \in \{1, \dots, k\}$ with $i \neq j$, such that $C = p_i \cup p_j$. By Lemma 5.8, we have that the connected components of $G \setminus C$ are given by $X_{i+_k 1} \cup X_{i+_k 2} \cup \dots \cup X_j$ and $X_{j+_k 1} \cup X_{j+_k 2} \cup \dots \cup X_i$. Since x and y are separated by C, we have that there are $t \in \{i +_k 1, i +_k 2, \dots, j\}$ and $t' \in \{j +_k 1, j +_k 2, \dots, i\}$, such that $x \in X_t$ and $y \in X_{t'}$. Consider the 4-cut $C' = p_{t-_k 1} \cup p_t$. Then, Lemma 5.8 implies

that $C' = \partial(X_t)$, and therefore C' is a $\mathcal{C}$-minimal 4-cut (by definition). Notice that the connected components of $G \setminus C'$ are given by X_t and $V \setminus X_t$. Thus, C' separates x and y. □

Corollary 5.7. *Let $\mathcal{C}$ be a cyclic family of 4-cuts, and let $\mathcal{M}$ be the collection of the $\mathcal{C}$-minimal 4-cuts. Then $\mathcal{M}$ is a parallel family of 4-cuts with atoms$(\mathcal{M})$ = atoms$(\mathcal{C})$.*

Proof. An immediate consequence of Lemmata 5.11 and 5.12. □

5.2.4 Generating the implied 4-cuts

Let $\mathcal{C}$ be a collection of 4-cuts of G. Our goal is to construct a linear-space representation of all 4-cuts implied by $\mathcal{C}$, that will be convenient in order to essentially process all of them simultaneously and derive *atoms*$(\mathcal{C})$. We will show how to do this in linear time, by constructing a set $\{\mathcal{F}_1, \dots, \mathcal{F}_k\}$ of collections of pairs of edges, each one of which generates a collection of 4-cuts implied by $\mathcal{C}$, with the property that every 4-cut implied by $\mathcal{C}$ is generated by at least one of those collections.

Intuitively speaking, the idea is to partition every 4-cut $C \in \mathcal{C}$ into pairs of pairs of edges in all possible ways, in order to (implicitly) trace all the implicating sequences of $\mathcal{C}$ that use C. Thus, if $C = \{e_1, e_2, e_3, e_4\}$, then we produce the three partitions $\{\{e_1, e_2\}, \{e_3, e_4\}\}$, $\{\{e_1, e_3\}, \{e_2, e_4\}\}$ and $\{\{e_1, e_4\}, \{e_2, e_3\}\}$ of C into pairs of edges. Every one of those partitions has the potential to participate in an implicating sequence for a 4-cut. For example, if there is a 4-cut $C' = \{e_3, e_4, e_5, e_6\} \in \mathcal{C}$ with $C' \neq C$, then $\{e_1, e_2, e_5, e_6\}$ is a 4-cut implied by C and C', and in order to derive this implication we have to (conceptually) partition C into $\{\{e_1, e_2\}, \{e_3, e_4\}\}$ and C' into $\{\{e_3, e_4\}, \{e_5, e_6\}\}$. Thus, $\mathcal{F} = \{\{e_1, e_2\}, \{e_3, e_4\}, \{e_5, e_6\}\}$ is a collection of pairs of edges that generates a collection of 4-cuts implied by $\mathcal{C}$. Now we would like to extend this collection as much as possible, by considering the partition of another 4-cut from $\mathcal{C}$ into pairs of edges that includes one of the pairs of edges in $\mathcal{F}$. This is easy to do if we have broken every 4-cut from $\mathcal{C}$ into all its possible bipartitions of pairs of edges.

In order to implement this idea, for every 4-cut $C = \{e_1, e_2, e_3, e_4\} \in \mathcal{C}$, we produce six elements $(C, \{e_1, e_2\})$, $(C, \{e_1, e_3\})$, $(C, \{e_1, e_4\})$, $(C, \{e_2, e_3\})$, $(C, \{e_2, e_4\})$ and $(C, \{e_3, e_4\})$. Then we introduce three artificial (undirected) edges $\{(C, \{e_1, e_2\}), (C, \{e_3, e_4\})\}$, $\{(C, \{e_1, e_3\}), (C, \{e_2, e_4\})\}$ and $\{(C, \{e_1, e_4\}), (C, \{e_2, e_3\})\}$. The purpose of those edges is to maintain the information that their endpoints correspond to a specific partition of C into pairs of edges. Now suppose that there are

two 4-cuts $C, C' \in \mathcal{C}$ that participate in an implicating sequence as consecutive 4-cuts. This means that there are two elements of the form $(C, \{e, e'\})$ and $(C', \{e, e'\})$. Then we would like to have those elements connected with a new artificial edge, in order to maintain the information that the 4-cuts C, C' intersect in the pair $\{e, e'\}$. However, it would be inefficient to introduce such an edge in all those cases, because we would need $\Omega(|\mathcal{C}|^2)$ time in the worst case scenario. Instead, if $C_1, \dots, C_k$ are all the 4-cuts from $\mathcal{C}$ that contain the pair of edges $\{e, e'\}$, then we ensure that we have all elements $(C_1, \{e, e'\}), \dots, (C_k, \{e, e'\})$ in a sequence, and then we connect every consecutive pair of elements in this sequence with a new artificial edge. In total, this results in an undirected graph $\mathcal{G}$, that basically represents all the partitions of the 4-cuts from $\mathcal{C}$ into two pairs of edges, and all intersections of the 4-cuts from $\mathcal{C}$ in a pair of edges. Then we can efficiently derive this information if we simply compute the connected components of this graph. We can prove that the connected components of $\mathcal{G}$ correspond to collections of pairs of edges that generate collections of 4-cuts implied by $\mathcal{C}$ (see Proposition 5.3).

The implementation of this idea is shown in Algorithm 16. We use a total ordering of the edges of G (e.g., lexicographic order), so that the order of edges in a pair of edges is fixed. This is needed because, if C and C' are two distinct 4-cuts that contain a pair of edges $\{e, e'\}$, then we would like to have the elements $(C, \{e, e'\})$ and $(C', \{e, e'\})$ (that are generated internally by the algorithm) in a maximal sequence of elements of this form. Thus, the order of e and e' should be fixed, so that the tuples $(C, \{e, e'\})$ and $(C', \{e, e'\})$ can be recognized as having the same second component. If p is a pair of edges, then we let $\vec{p}$ denote the corresponding ordered pair of edges that respects the total ordering of $E(G)$. Whenever (e, e') denotes an ordered pair of edges, we assume that this order respects the total ordering of $E(G)$.

The remainder of this section is devoted to an exploration of the properties of the output of Algorithm 16.

Proposition 5.3. *Let $\mathcal{C}$ be a collection of 4-cuts of a graph G, and let $\mathcal{F}_1, \dots, \mathcal{F}_k$ be the output of Algorithm 16 on input $\mathcal{C}$. Then every $\mathcal{F}_i$ is a collection of pairs of edges that generates a collection of 4-cuts of G that are implied by $\mathcal{C}$. Conversely, for every 4-cut C implied by $\mathcal{C}$, there is at least one $i \in \{1, \dots, k\}$ such that C belongs to the collection of 4-cuts generated by $\mathcal{F}_i$. The running time of Algorithm 16 is $O(n+|\mathcal{C}|)$, where $n = |V(G)|$. The output of Algorithm 16 has size $O(|\mathcal{C}|)$ (i.e., $O(|\mathcal{F}_1| + \dots + |\mathcal{F}_k|) = O(|\mathcal{C}|)$).*

Algorithm 16: Return a set of collections of pairs of edges that generate in total all the 4-cuts that are implied by a collection of 4-cuts $\mathcal{C}$

1 **input**: a collection $\mathcal{C}$ of 4-cuts of G
2 **output**: a set $\mathcal{F}_1, \dots, \mathcal{F}_k$ of collections of pairs of edges that generate collections of 4-cuts of G that contain in total all the 4-cuts implied by $\mathcal{C}$
3 Let $P \leftarrow \emptyset$, $J \leftarrow \emptyset$
4 **foreach** $C = \{e_1, e_2, e_3, e_4\} \in \mathcal{C}$ **do**
5 let $p_1 \leftarrow \{e_1, e_2\}$, $p_2 \leftarrow \{e_3, e_4\}$, $p_3 \leftarrow \{e_1, e_3\}$, $p_4 \leftarrow \{e_2, e_4\}$, $p_5 \leftarrow \{e_1, e_4\}$, $p_6 \leftarrow \{e_2, e_3\}$
6 generate the elements $(C, \vec{p}_1)$, $(C, \vec{p}_2)$, $(C, \vec{p}_3)$, $(C, \vec{p}_4)$, $(C, \vec{p}_5)$, $(C, \vec{p}_6)$
7 add those elements to P
8 add to J the edges $\{(C, \vec{p}_1), (C, \vec{p}_2)\}$, $\{(C, \vec{p}_3), (C, \vec{p}_4)\}$, $\{(C, \vec{p}_5), (C, \vec{p}_6)\}$
9 **end**
10 sort the elements of P lexicographically w.r.t. their second component
11 **foreach** *pair of consecutive elements* $(C, p), (C', p)$ *of* P *with the same second component* **do**
12 add to J the edge $\{(C, p), (C', p)\}$
13 **end**
14 compute the connected components $S_1, \dots, S_k$ of the graph $\mathcal{G} = (P, J)$
15 **foreach** $i \in \{1, \dots, k\}$ **do**
16 $\mathcal{F}_i \leftarrow \{\{e, e'\} \mid \exists (C, (e, e')) \in S_i\}$ `// consider` $\mathcal{F}_i$ `as a simple set`
17 **end**
18 **return** $\mathcal{F}_1, \dots, \mathcal{F}_k$

Proof. Let $i \in \{1, \dots, k\}$. We will show that $\mathcal{F}_i$ generates a collection of 4-cuts implied by $\mathcal{C}$. Let $\mathcal{G}$ be the graph that is generated internally by the algorithm in Line 14, and let S_i be the connected component of $\mathcal{G}$ from which $\mathcal{F}_i$ is derived in Line 16. First we will show that $|\mathcal{F}_i| \geq 2$. Let $(C, (e, e'))$ be an element in S_i. Then C is a 4-cut in $\mathcal{C}$, and let $\{e'', e'''\} = C \setminus \{e, e'\}$. Due to the construction of $\mathcal{G}$, we have w.l.o.g., (i.e., by possibly changing the order of the edges), that $(C, (e'', e'''))$ is a vertex of $\mathcal{G}$. Then, there is an edge in $\mathcal{G}$ with endpoints $(C, (e, e'))$ and $(C, (e'', e'''))$ (see Line 8). This implies that $(C, (e, e'))$ and $(C, (e'', e'''))$ belong to the same connected component of $\mathcal{G}$, and therefore we have $(C, (e'', e''')) \in S_i$. This shows that $\{\{e, e'\}, \{e'', e'''\}\} \subseteq \mathcal{F}$.

Since C is a 4-element set, we have $\{e, e'\} \neq \{e'', e'''\}$. This shows that $|\mathcal{F}_i| \geq 2$.

Now let p and q be two distinct pairs of edges that are contained in $\mathcal{F}_i$. Then there are 4-cuts C and C' in $\mathcal{C}$ such that there are elements $(C, \vec{p})$ and $(C', \vec{q})$ that are contained in S_i. Then, since $(C, \vec{p})$ and $(C', \vec{q})$ are in the same connected component of $\mathcal{G}$, there is a path from $(C, \vec{p})$ to $(C', \vec{q})$ in $\mathcal{G}$ that passes from distinct vertices. This implies that there is sequence of pairs of edges $p_1, \dots, p_N$ of G, with $N \geq 2$, and a sequence $C_1, \dots, C_N$ of 4-cuts from $\mathcal{C}$, such that $(C_1, \vec{p}_1) = (C, \vec{p})$, $(C_N, \vec{p}_N) = (C', \vec{q})$, and for every $i \in \{1, \dots, N-1\}$ there is an edge of $\mathcal{G}$ with endpoints $(C_i, \vec{p}_i)$ and $(C_{i+1}, \vec{p}_{i+1})$. Since the edges of $\mathcal{G}$ are generated in Lines 8 and 12, for every $i \in \{1, \dots, N-1\}$ we have that either $C_i = C_{i+1}$ and $C_i = p_i \cup p_{i+1}$, or $C_i \cap C_{i+1} = p_i = p_{i+1}$ $(*)$.

Now we define a sequence of indexes $t(1), t(2), \dots, t(N')$, for some $N' \leq N$, as follows. First, we let $t(1)$ be the maximum index $i \geq 1$ such that $p_1 = p_2 = \cdots = p_i$. Now suppose that we have defined $t(i)$, for some $i \geq 1$, and $p_{t(i)} \neq p_N$. Then we let $t(i+1)$ be the maximum index $j > t(i)$ such that $p_{t(i)+1} = p_{t(i)+2} = \cdots = p_j$. This construction is terminated when we reach the first N' such that $p_{t(N')} = p_N$. Notice that $N' \geq 2$, since $p_1 \neq p_N$. By construction, we have $p_{t(i)+1} \neq p_{t(i)}$ and $p_{t(i+1)} = p_{t(i)+1}$, for every $i \in \{1, \dots, N'-1\}$. Thus, for every $i \in \{1, \dots, N'-1\}$, by $(*)$ we have $C_{t(i)} = p_{t(i)} \cup p_{t(i)+1}$, and therefore $C_{t(i)} = p_{t(i)} \cup p_{t(i+1)}$. Thus, $C_{t(1)}, \dots, C_{t(N'-1)}$ is an implicating sequence of $\mathcal{C}$. If $N' = 2$, then we have $C_{t(1)} = p_{t(1)} \cup p_{t(2)} = p_1 \cup p_N = p \cup q$. Thus, $p \cup q \in \mathcal{C}$. Otherwise, Lemma 5.6 implies that $p_{t(1)} \cup p_{t(N')}$ is a 4-cut implied by $\mathcal{C}$, and therefore $p_1 \cup p_N = p \cup q$ is a 4-cut implied by $\mathcal{C}$. In any case then, we have that $p \cup q$ is a 4-cut implied by $\mathcal{C}$.

Conversely, let C be a 4-cut implied by $\mathcal{C}$. This means that there is a sequence $p_1, \dots, p_{k+1}$ of pairs of edges of G, and a sequence $C_1, \dots, C_k$ of 4-cuts in $\mathcal{C}$, with $k \geq 1$, such that $C = p_1 \cup p_{k+1}$, and $C_i = p_i \cup p_{i+1}$ for every $i \in \{1, \dots, k\}$. Then, for every $i \in \{1, \dots, k\}$, there is an edge of $\mathcal{G}$ with endpoints $(C_i, \vec{p}_i)$ and $(C_i, \vec{p}_{i+1})$ (see Line 8). Furthermore, for every $i \in \{1, \dots, k-1\}$, there is path from $(C_i, \vec{p}_{i+1})$ to $(C_{i+1}, \vec{p}_{i+1})$ in $\mathcal{G}$ (due to the existence of the edges in Line 12). Thus, all pairs of the form $(C_i, \vec{p}_i)$, for $i \in \{1, \dots, k\}$, belong to the same connected component S of $\mathcal{G}$. Furthermore, $(C_k, \vec{p}_{k+1})$ also belongs to S, due to the existence of the edge with endpoints $(C_k, \vec{p}_k)$ and $(C_k, \vec{p}_{k+1})$ (see Line 8). Thus, there is a collection of pairs of edges $\mathcal{F}$ that is returned by Algorithm 16 on input $\mathcal{C}$ such that $\{p_1, \dots, p_{k+1}\} \subseteq \mathcal{F}$ (see Line 16). Then we have that $C = p_1 \cup p_{k+1}$ is generated by $\mathcal{F}$.

We can easily see that Algorithm 16 runs in $O(n+|\mathcal{C}|)$ time. For every $C \in \mathcal{C}$, we generate six elements of $O(1)$ size, and three edges of $O(1)$ size. Line 10 takes $O(n+|\mathcal{C}|)$ time if implemented with bucket sort, since the components of the tuples that we sort are edges of the graph, and so their endpoints lie in the range $\{1, \ldots, n\}$. Line 12 adds $O(|\mathcal{C}|)$ edges of $O(1)$ size. The computation of the connected components in Line 14 takes $O(|V(\mathcal{G})| + |E(\mathcal{G})|) = O(|\mathcal{C}|)$ time, and Line 16 takes $O(|V(\mathcal{G})|) = O(|\mathcal{C}|)$ time. Thus, the running time of Algorithm 16 is $O(n+|\mathcal{C}|)$. Finally, for every $i \in \{1, \ldots, k\}$, let S_i be the connected component of $\mathcal{G}$ from which $\mathcal{F}_i$ is derived (in Line 16). Then we have $O(|\mathcal{F}_1| + \cdots + |\mathcal{F}_k|) = O(|S_1| + \cdots + |S_k|) = O(|V(\mathcal{G})| = O(|\mathcal{C}|)$. The second equality is due to the fact that $S_1, \ldots, S_k$ are the connected components of $\mathcal{G}$. $\square$

Lemma 5.13. *Let $\mathcal{C}$ be a collection of 4-cuts of a graph G, and let $\mathcal{F}$ and $\mathcal{F}'$ be two distinct collections of pairs of edges that are returned by Algorithm 16 on input $\mathcal{C}$. Then $\mathcal{F} \cap \mathcal{F}' = \emptyset$.*

Proof. Let $\mathcal{F}$ and $\mathcal{F}'$ be two collections of pairs of edges returned by Algorithm 16 on input $\mathcal{C}$ with $\mathcal{F} \neq \mathcal{F}'$. Let S and S' be the connected components of the graph $\mathcal{G}$ generated in Line 14, from which $\mathcal{F}$ and $\mathcal{F}'$, respectively, are derived in Line 16. Let us assume, for the sake of contradiction, that there is a pair of edges $\{e, e'\} \in \mathcal{F} \cap \mathcal{F}'$. Then, w.l.o.g., (i.e., by possibly changing the order of the edges), there are elements $(C, (e, e')) \in S$ and $(C', (e, e')) \in S'$, such that C and C' are 4-cuts in $\mathcal{C}$. But then, due to the existence of the edges in Line 12, we have that $(C, (e, e'))$ is connected with $(C', (e, e'))$ in $\mathcal{G}$. This implies that $S = S'$, which further implies that $\mathcal{F} = \mathcal{F}'$, a contradiction. We conclude that $\mathcal{F} \cap \mathcal{F}' = \emptyset$. $\square$

Lemma 5.14. *Let $\mathcal{C}$ be a collection of 4-cuts of a graph G, and let $C = \{e_1, e_2, e_3, e_4\}$ be a 4-cut of G that is implied by $\mathcal{C}$ through the pair of edges $\{e_1, e_2\}$. Then, in the output of Algorithm 16 on input $\mathcal{C}$, there is a collection $\mathcal{F}$ of pairs of edges such that $\{\{e_1, e_2\}, \{e_3, e_4\}\} \subseteq \mathcal{F}$. Furthermore, if $C \notin \mathcal{C}$, then this inclusion is proper (i.e., $|\mathcal{F}| > 2$).*

Proof. We may assume w.l.o.g. that $e_1 < e_2$ and $e_3 < e_4$. Suppose first that $C \in \mathcal{C}$. Let $\mathcal{G}$ be the graph generated by Algorithm 16 on input $\mathcal{C}$ in Line 14. Then, the elements $(C, (e_1, e_2))$ and $(C, (e_3, e_4))$ are vertices of $\mathcal{G}$ that are connected with an edge (due to Line 8). Thus, let S be the connected component of $\mathcal{G}$ that contains $(C, (e_1, e_2))$ and $(C, (e_3, e_4))$, and let $\mathcal{F}$ be the collection of pairs of edges that is derived from S in Line 16. Then, we have $\{\{e_1, e_2\}, \{e_3, e_4\}\} \subseteq \mathcal{F}$.

Now let us suppose that $C \notin \mathcal{C}$. Since C is implied by $\mathcal{C}$ through the pair of edges $\{e_1, e_2\}$, there is a sequence $p_1, \dots, p_{k+1}$ of pairs of edges, and a sequence $C_1, \dots, C_k$ of 4-cuts from $\mathcal{C}$, such that $p_1 = \{e_1, e_2\}$, $p_{k+1} = \{e_3, e_4\}$, and $C_i = p_i \cup p_{i+1}$ for every $i \in \{1, \dots, k\}$. Then, for every $i \in \{1, \dots, k-1\}$, we have that either $C_i = C_{i+1}$ or $C_i \cap C_{i+1} = p_{i+1}$. Now let $i \in \{1, \dots, k\}$ be an index. Since $C_i = p_i \cup p_{i+1}$, we have that $(C_i, \vec{p_i})$ and $(C_i, \vec{p}_{i+1})$ are the endpoints of an edge of $\mathcal{G}$ (see Line 8). Now let $i < k$. If $C_i = C_{i+1}$, then we have $(C_i, \vec{p}_{i+1}) = (C_{i+1}, \vec{p}_{i+1})$. Otherwise, we have $C_i \cap C_{i+1} = p_{i+1}$, and therefore the elements $(C_i, \vec{p}_{i+1})$ and $(C_{i+1}, \vec{p}_{i+1})$ are vertices in the same connected component of $\mathcal{G}$ (due to the existence of the edges in Line 12). This shows that the vertices $(C_i, \vec{p_i})$, for $i \in \{1, \dots, k-1\}$, are in the same connected component S of $\mathcal{G}$. Furthermore, since there is an edge of $\mathcal{G}$ with endpoints $(C_k, \vec{p_k})$ and $(C_k, \vec{p}_{k+1})$, we have that $(C_k, \vec{p}_{k+1})$ is also in S. Now let $\mathcal{F}$ be the collection of pairs of edges that is derived from S in Line 16. Then, we have $\{p_1, \dots, p_{k+1}\} \subseteq \mathcal{F}$. Since $C \notin \mathcal{C}$ and $p_1 \cup p_2 \in \mathcal{C}$, we have $C \neq p_1 \cup p_2$. Thus, since $C = p_1 \cup p_{k+1}$, we have $p_2 \neq p_{k+1}$. Finally, since $p_1 \cup p_2$ and $p_1 \cup p_{k+1}$ are 4-cuts, we have $p_1 \neq p_2$ and $p_1 \neq p_{k+1}$. Thus, we have that p_1, p_2, p_{k+1} are three distinct pairs of edges contained in $\mathcal{F}$. ☐

Lemma 5.15. *Let $\mathcal{C}$ be a collection of 4-cuts of G, and let $\mathcal{F}$ be a collection of pairs of edges that is returned by Algorithm 16 on input $\mathcal{C}$. Suppose that $|\mathcal{F}| = 2$. Then, $\bigcup \mathcal{F} \in \mathcal{C}$.*

Proof. Let $\mathcal{F} = \{p, p'\}$. By Proposition 5.3, we have that $p \cup p'$ is a 4-cut implied by $\mathcal{C}$. Let $\mathcal{G}$ be the graph that is generated internally by the algorithm in Line 14, and let S be the connected component of $\mathcal{G}$ from which $\mathcal{F}$ is derived in Line 16. Then there are 4-cuts C and C' in $\mathcal{C}$ such that the tuples $(C, \vec{p})$ and $(C', \vec{p'})$ are in S, and we have $p \subset C$ and $p' \subset C'$. If $C = C'$, then we obviously have $p \cup p' \in \mathcal{C}$. So let us assume that $C \neq C'$.

Since the tuples $(C, \vec{p})$ and $(C', \vec{p'})$ are in the same connected component of $\mathcal{G}$, this means that there is a path P from $(C, \vec{p})$ to $(C', \vec{p'})$ in $\mathcal{G}$ that passes from distinct vertices. Let $(C'', \vec{p''})$ be the second vertex on P (where $C'' \in \mathcal{C}$ and p'' is a pair of edges in C''). Since the edges of $\mathcal{G}$ are generated in Lines 8 and 12, and since $p \neq p'$ and $C \neq C'$, we have that (1) either $C'' = C$ and $p'' \neq p$, or (2) $C'' = C'$ and $p'' \neq p'$, or (3) $C'' \neq C$ and $p'' = p$, or (4) $C'' \neq C'$ and $p'' = p'$. Notice that, between (1) and (2), we may assume w.l.o.g. (1). Also, between (3) and (4), we may assume w.l.o.g. (3).

First, let us assume that (1) is true. Due to the existence of P, we have that $(C'', \vec{p''})$

lies in S. This implies that $p'' \in \mathcal{F}$. Therefore, since $p'' \neq p$ and $\mathcal{F} = \{p, p'\}$, we have $p'' = p'$. Let us suppose, for the sake of contradiction, that $p \cup p' \neq C$. Since $(C, \vec{p})$ and $(C'', \vec{p'}) = (C, \vec{p'})$ are vertices of $\mathcal{G}$, we have $p \subset C$ and $p' \subset C$. Then, since $p \neq p'$ and $p \cup p' \neq C$, we have $|C \cap (p \cup p')| = 3$. But this contradicts Lemma 5.2. This shows that $C = p \cup p'$, and therefore $p \cup p'$ is a 4-cut in $\mathcal{C}$.

Now let us assume that (3) is true. Due to the existence of P, we have that $(C'', \vec{p''})$ lies in S. Let $q = C'' \setminus p''$. Then there is an edge of $\mathcal{G}$ with endpoints $(C'', \vec{p''})$ and $(C'', \vec{q})$ (see Line 8). Thus, $(C'', \vec{q})$ also lies in S. This implies that $q \in \mathcal{F}$. Thus, since $q \neq p'' = p$, we have $q = p'$. Therefore, we have that $C'' = p'' \cup q = p \cup p'$ is a 4-cut in $\mathcal{C}$. □

5.2.5 Isolated and quasi-isolated 4-cuts

Let $\mathcal{C}$ be a collection of 4-cuts of G, and let C be a 4-cut implied by $\mathcal{C}$. Then, it may be that there is no collection $\mathcal{F}$ of pairs of edges with $|\mathcal{F}| > 2$ that generates a collection of 4-cuts implied by $\mathcal{C}$ that includes C. In this case, we call C a *$\mathcal{C}$-isolated* 4-cut. Notice that, if $\mathcal{C}$ is a complete collection of 4-cuts of G and C is a $\mathcal{C}$-isolated 4-cut, then C is an isolated 4-cut.

The following lemma provides a necessary condition that must be satisfied by a $\mathcal{C}$-isolated 4-cut.

Lemma 5.16. *Let $\mathcal{C}$ be a collection of 4-cuts of G, and let $C = \{e_1, e_2, e_3, e_4\}$ be a $\mathcal{C}$-isolated 4-cut. Then we have $C \in \mathcal{C}$, and the 2-element collections of pairs of edges $\{\{e_1, e_2\}, \{e_3, e_4\}\}$, $\{\{e_1, e_3\}, \{e_2, e_4\}\}$ and $\{\{e_1, e_4\}, \{e_2, e_3\}\}$, are part of the output of Algorithm 16 on input $\mathcal{C}$.*

Proof. Let us suppose, for the sake of contradiction, that $C \notin \mathcal{C}$. Since C is a $\mathcal{C}$-isolated 4-cut, we have that $\mathcal{C}$ implies C. Thus, we may assume w.l.o.g. that $\mathcal{C}$ implies C through the pair of edges $\{e_1, e_2\}$. Then, Lemma 5.14 implies that there is a collection $\mathcal{F}$ of pairs of edges that is returned by Algorithm 16 on input $\mathcal{C}$ such that $\{\{e_1, e_2\}, \{e_3, e_4\}\} \subset \mathcal{F}$. Proposition 5.3 implies that $\mathcal{F}$ generates a collection $\mathcal{C}'$ of 4-cuts implied by $\mathcal{C}$. Thus, since $|\mathcal{F}| > 2$ and $C \in \mathcal{C}'$, we have a contradiction to the fact that C is $\mathcal{C}$-isolated 4-cut. This shows that $C \in \mathcal{C}$.

Now, since $C \in \mathcal{C}$, we have that $\mathcal{C}$ trivially implies C through the pair of edges $\{e_1, e_2\}$. Thus, Lemma 5.14 implies that there is a collection $\mathcal{F}$ of pairs of edges that is returned by Algorithm 16 on input $\mathcal{C}$ such that $\{\{e_1, e_2\}, \{e_3, e_4\}\} \subseteq \mathcal{F}$. Proposition 5.3

implies that $\mathcal{F}$ generates a collection $\mathcal{C}'$ of 4-cuts implied by $\mathcal{C}$. Thus, since $C \in \mathcal{C}'$, we have that $|\mathcal{F}| = 2$, because C is $\mathcal{C}$-isolated. Similarly, since $\mathcal{C}$ implies C through the pairs of edges $\{e_1, e_3\}$ and $\{e_1, e_4\}$, we have that the 2-element collections of pairs of edges $\{\{e_1, e_3\}, \{e_2, e_4\}\}$ and $\{\{e_1, e_4\}, \{e_2, e_3\}\}$ are part of the output of Algorithm 16 on input $\mathcal{C}$. □

We note that the condition provided by Lemma 5.16 is only necessary, but not sufficient. In other words, it may be that there is a 4-cut $C = \{e_1, e_2, e_3, e_4\}$ such that the collections of pairs of edges $\{\{e_1, e_2\}, \{e_3, e_4\}\}$, $\{\{e_1, e_3\}, \{e_2, e_4\}\}$ and $\{\{e_1, e_4\}, \{e_2, e_3\}\}$, are part of the output of Algorithm 16 on input $\mathcal{C}$, but C is not a $\mathcal{C}$-isolated 4-cut. In this case, we call C a *quasi $\mathcal{C}$-isolated* 4-cut.

Corollary 5.8. *Let $\mathcal{C}$ be a collection of 4-cuts of G, and let C be a quasi $\mathcal{C}$-isolated 4-cut. Then $C \in \mathcal{C}$.*

Proof. Let $C = \{e_1, e_2, e_3, e_4\}$. Since C is quasi $\mathcal{C}$-isolated, we have that the three collections of pairs of edges $\{\{e_1, e_2\}, \{e_3, e_4\}\}$, $\{\{e_1, e_3\}, \{e_2, e_4\}\}$ and $\{\{e_1, e_4\}, \{e_2, e_3\}\}$, are part of the output of Algorithm 16 on input $\mathcal{C}$. Thus, Lemma 5.15 implies that $C \in \mathcal{C}$. □

The following lemma, which concerns essential quasi $\mathcal{C}$-isolated 4-cuts, will be very useful in computing all the essential $\mathcal{C}$-isolated 4-cuts, because it provides a criterion with which we can distinguish the $\mathcal{C}$-isolated 4-cuts from the quasi $\mathcal{C}$-isolated 4-cuts (see Corollary 5.9).

Lemma 5.17 (An essential quasi-isolated 4-cut shares a pair of edges with a minimal 4-cut). *Let $\mathcal{C}$ be a collection of 4-cuts of G, and let C be an essential quasi $\mathcal{C}$-isolated 4-cut. Then, there is a pair of edges $p = C \cap C'$, where C' is an essential $\mathcal{C}'$-minimal 4-cut, where $\mathcal{C}'$ is a cyclic family of 4-cuts that is generated by a collection $\mathcal{F}'$ of pairs of edges with $|\mathcal{F}'| \geq 3$ that is returned by Algorithm 16 on input $\mathcal{C}$.*

Proof. Let $C = \{e_1, e_2, e_3, e_4\}$. Since C is a quasi $\mathcal{C}$-isolated 4-cut, we have that C is not $\mathcal{C}$-isolated. This means that there is a collection $\mathcal{F}$ of pairs of edges with $|\mathcal{F}| > 2$ that generates a collection $\widetilde{\mathcal{C}}$ of 4-cuts that are implied by $\mathcal{C}$ such that $C \in \widetilde{\mathcal{C}}$. Thus, we may assume w.l.o.g. that $\{\{e_1, e_2\}, \{e_3, e_4\}, \{x, y\}\} \subseteq \mathcal{F}$, where $\{x, y\}$ is a pair of edges with $\{x, y\} \notin \{\{e_1, e_2\}, \{e_3, e_4\}\}$. Since C is an essential 4-cut, by Corollary 5.5 we have that $\widetilde{\mathcal{C}}$ is a cyclic family of 4-cuts. This implies that there is a partition

$\{X_1, X_2, X_3\}$ of $V(G)$ such that the subgraphs $G[X_1]$, $G[X_2]$ and $G[X_3]$ are connected, and $E[X_1, X_2] = \{e_1, e_2\}$, $E[X_2, X_3] = \{e_3, e_4\}$, $E[X_3, X_1] = \{x, y\}$. (See Figure 5.6.) Since C is an essential 4-cut and the connected components of $G \setminus C$ are X_2 and $X_1 \cup X_3$, we have that there is a pair u, v of 4-edge-connected vertices such that $u \in X_2$ and $v \in X_1 \cup X_3$. We may assume w.l.o.g. that $v \in X_1$.

Let $C' = \{e_1, e_2, x, y\}$. Then we have that C' is a 4-cut implied by $\mathcal{C}$. Let us suppose, for the sake of contradiction, that $\mathcal{C}$ implies C' through the pair of edges $\{e_1, e_2\}$. Then, Lemma 5.14 implies that there is a collection $\mathcal{F}'$ of pairs of edges that is returned by Algorithm 16 on input $\mathcal{C}$ such that $\{\{e_1, e_2\}, \{x, y\}\} \subseteq \mathcal{F}'$. Since C is a quasi $\mathcal{C}$-isolated 4-cut, by definition we have that the collection of pairs of edges $\mathcal{F}'' = \{\{e_1, e_2\}, \{e_3, e_4\}\}$ is returned by Algorithm 16 on input $\mathcal{C}$. Since $\{x, y\} \neq \{e_3, e_4\}$, we have that $\mathcal{F}' \neq \mathcal{F}''$. Therefore, Lemma 5.13 implies that $\mathcal{F}' \cap \mathcal{F}'' = \emptyset$, in contradiction to the fact that $\{e_1, e_2\} \in \mathcal{F}' \cap \mathcal{F}''$. Thus, we have that $\mathcal{C}$ does not imply C' through the pair of edges $\{e_1, e_2\}$. Thus, we may assume w.l.o.g. that $\mathcal{C}$ implies C' through the pair of edges $\{e_1, x\}$.

Notice that, since $\mathcal{C}$ does not imply C' through the pair of edges $\{e_1, e_2\} \subset C'$, we have that $C' \notin \mathcal{C}$. Thus, since $\mathcal{C}$ implies C' through the pair of edges $\{e_1, x\}$, by Lemma 5.14 we have that there is a collection $\mathcal{F}'$ of pairs of edges that is returned by Algorithm 16 on input $\mathcal{C}$ such that $\{\{e_1, x\}, \{e_2, y\}\} \subset \mathcal{F}'$. By Proposition 5.3, we have that $\mathcal{F}'$ generates a collection $\mathcal{C}'$ of 4-cuts that are implied by $\mathcal{C}$. We have that the connected components of $G \setminus C'$ are X_1 and $X_2 \cup X_3$. Since $u \in X_2$ and $v \in X_1$, and u, v are 4-edge-connected vertices, this implies that C' is an essential 4-cut. Thus, since $|\mathcal{F}'| \geq 3$, by Corollary 5.5 we have that $\mathcal{C}'$ is a cyclic family of 4-cuts. We will prove that $X_2 \cup X_3$ is a corner of $\mathcal{C}'$, and therefore C' is $\mathcal{C}'$-minimal 4-cut.

So let us suppose, for the sake of contradiction, that $X_2 \cup X_3$ is not a corner of $\mathcal{C}'$. Since $X_2 \cup X_3$ is one of the connected components of $G \setminus C'$, this implies that there is a pair of edges $\{z, w\} \in \mathcal{F}'$, such that $\{z, w\} \subset E(G[X_2 \cup X_3])$. Then, since $\{e_1, x\} \in \mathcal{F}'$ and $\mathcal{F}'$ generates 4-cuts of G, we have that $C'' = \{e_1, x, z, w\}$ is a 4-cut of G. Let $G' = G \setminus \{z, w\}$. Since $\{z, w\} \subset E(G[X_2 \cup X_3])$, we have that $G'[X_1]$ is connected. Thus, it cannot be that either $G'[X_2]$ or $G'[X_3]$ is connected, because otherwise the endpoints of e_1 or x, respectively, remain connected in $G' \setminus \{e_1, x\}$, in contradiction to the fact that C'' is a 4-cut of G. Thus, we have that one of z, w is a bridge of $G[X_2]$, and the other is a bridge of $G[X_3]$. Thus, we may assume w.l.o.g. that z is a bridge of $G[X_2]$, and let Y_1 and Y_2 be the connected components of $G[X_2] \setminus z$. Then we have

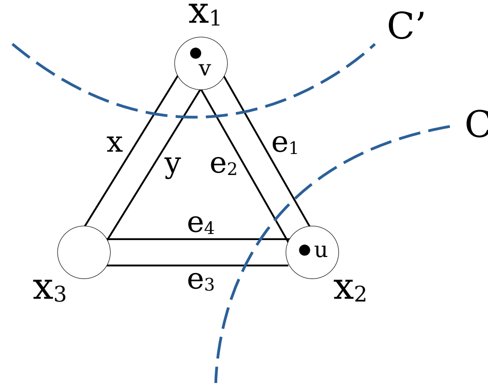


Figure 5.6: A depiction of the situation analyzed in Lemma 5.17.

that $E[Y_1, Y_2] = \{z\}$. Since $|\partial(X_2)| = 4$, by Lemma 5.3 we have that $|E[Y_1, V \setminus X_2]| = 2$ and $|E[Y_2, V \setminus X_2]| = 2$. Since $E[Y_1, Y_2] = \{z\}$, this implies that $|\partial(Y_1)| = |\partial(Y_2)| = 3$. But then we have that u is not 4-edge-connected with v, since either $\partial(Y_1)$ or $\partial(Y_2)$ (depending on whether Y_1 or Y_2 contains u, respectively) is a 3-cut that separates u from v – a contradiction.

Thus, we have shown that $X_2 \cup X_3$ is a corner of $\mathcal{C}'$. Therefore, since $X_2 \cup X_3$ is one of the connected components of $G \setminus C'$, we have that C' is a $\mathcal{C}'$-minimal 4-cut. Furthermore, since $v \in X_1$ and $u \in X_2$, we have that C' is an essential 4-cut. Finally, we have that $\{e_1, e_2\} = C \cap C'$, and $\mathcal{C}'$ is generated by $\mathcal{F}'$, where $\mathcal{F}'$ is one of the collections of pairs of edges that are returned by Algorithm 16 on input $\mathcal{C}$. Thus, the proof is complete. □

Corollary 5.9. *Let $\mathcal{C}$ be a complete collection of 4-cuts of G, and let C be an essential 4-cut of G. Let $\mathcal{F}_1, \ldots, \mathcal{F}_k$ be the collections of pairs of edges that are returned by Algorithm 16 on input $\mathcal{C}$, and let $\mathcal{C}_1, \ldots, \mathcal{C}_k$ be the collections of 4-cuts that they generate, respectively. Then, C is a quasi $\mathcal{C}$-isolated 4-cut if and only if:*

(1) $C \in \mathcal{C}$.

(2) All three partitions of C into pairs of edges are contained in $\{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$.

(3) There is a pair of edges p in C such that $p = C \cap C'$, where C' is an essential $\mathcal{C}_i$-minimal 4-cut, for some $i \in \{1, \ldots, k\}$.

Proof. ($\Rightarrow$) We have $C \in \mathcal{C}$ by Corollary 5.8. (2) is an immediate consequence of the definition of quasi $\mathcal{C}$-isolated 4-cuts. (3) is ensured by Lemma 5.17, since C is

essential.

($\Leftarrow$) Since $C \cap C' = p$, we have that $p' = C' \setminus C$ and $q = C \setminus C'$ are two distinct pairs of edges. Thus, we have $C = p \cup q$ and $C' = p \cup p'$. Then, by Lemma 5.5 we have that $C'' = q \cup p'$ is a 4-cut of G. Thus, since $\mathcal{C}$ is a complete collection of 4-cuts of G, we have that $\{p, p', q\}$ is a collection of pairs of edges that generates 4-cuts implied by $\mathcal{C}$, including C. This shows that C is not a $\mathcal{C}$-isolated 4-cut. Thus, since (2) is satisfied, by definition we have that C is a quasi $\mathcal{C}$-isolated 4-cut.

$\square$

5.2.6 Some additional properties satisfied by the output of Algorithm 16

Lemma 5.18. *Let $\mathcal{C}$ be a collection of 4-cuts of G, and let $\mathcal{F}$ and $\mathcal{F}'$ be two distinct collections of pairs of edges that are returned by Algorithm 16 on input $\mathcal{C}$. Let p and p' be two pairs of edges such that $p \in \mathcal{F}$ and $p' \in \mathcal{F}'$. Then, $p \cup p'$ (if it is a 4-cut of G) is not implied by $\mathcal{C}$ through the pair of edges p.*

Proof. Let $\mathcal{G}$ be the graph that is generated internally by Algorithm 16 on input $\mathcal{C}$ (in Line 14). Since $\mathcal{F}$ and $\mathcal{F}'$ are returned by Algorithm 16 on input $\mathcal{C}$, there are connected components S and S' of $\mathcal{G}$, such that $\mathcal{F}$ is derived from S, and $\mathcal{F}'$ is derived from S' (in Line 16). Since $p \in \mathcal{F}$ and $p' \in \mathcal{F}'$, there are 4-cuts C and C' in $\mathcal{C}$ such that $(C, \vec{p}) \in S$ and $(C', \vec{p'}) \in S'$.

Let us assume, for the sake of contradiction, that $C'' = p \cup p'$ is a 4-cut of G that is implied by $\mathcal{C}$ through the pair of edges p. (We note that it is not even necessary that C'' is a 4-element set, but our assumption implies that.) This means that there is a sequence $p_1, \dots, p_{k+1}$ of pairs of edges, and a sequence $C_1, \dots, C_k$ of 4-cuts from $\mathcal{C}$, such that $p_1 = p$, $p_{k+1} = p'$, and $C_i = p_i \cup p_{i+1}$ for every $i \in \{1, \dots, k\}$. Then, for every $i \in \{1, \dots, k-1\}$, Lemma 5.2 implies that either $C_i = C_{i+1}$ or $C_i \cap C_{i+1} = p_{i+1}$. Now let $i \in \{1, \dots, k\}$ be an index. Since $C_i = p_i \cup p_{i+1}$, we have that $(C_i, \vec{p_i})$ and $(C_i, \vec{p}_{i+1})$ are the endpoints of an edge of $\mathcal{G}$ (see Line 8). Now let $i < k$. If $C_i = C_{i+1}$, then we have $(C_i, \vec{p}_{i+1}) = (C_{i+1}, \vec{p}_{i+1})$. Otherwise, we have $C_i \cap C_{i+1} = p_{i+1}$, and therefore the elements $(C_i, \vec{p}_{i+1})$ and $(C_{i+1}, \vec{p}_{i+1})$ are vertices in the same connected component of $\mathcal{G}$ (due to the existence of the edges in Line 12). This shows that the vertices $(C_i, \vec{p_i})$, for $i \in \{1, \dots, k-1\}$, are in the same connected component S'' of $\mathcal{G}$. Furthermore, since $C_k = p_k \cup p_{k+1}$, there is an edge of $\mathcal{G}$ with endpoints $(C_k, \vec{p_k})$ and $(C_k, \vec{p}_{k+1})$. Thus,

we have that $(C_k, \vec{p}_{k+1})$ is also in S''. Now let $\mathcal{F}''$ be the collection of pairs of edges that is derived from S'' in Line 16. Then, we have $\{p_1, \dots, p_{k+1}\} \subseteq \mathcal{F}''$. Since $p_1 = p$ and $p_{k+1} = p'$, this implies that $\mathcal{F}'' \cap \mathcal{F} \neq \emptyset$ and $\mathcal{F}'' \cap \mathcal{F}' \neq \emptyset$. But this contradicts Lemma 5.13. We conclude that C'' (if it is a 4-cut of G) is not implied by $\mathcal{C}$ through the pair of edges p. □

In order to appreciate the following lemma, we need to discuss a subtle point that concerns the way in which cyclic families of 4-cuts are implied by collections of 4-cuts. Suppose that we have a collection $\mathcal{C}$ of 4-cuts that implies the cyclic family $\mathcal{C}'$ of 4-cuts that is generated by the collection of pairs of edges $\mathcal{F} = \{p_1, p_2, p_3\}$. Then, $\mathcal{C}'$ consists of the 4-cuts $\{p_1 \cup p_2, p_1 \cup p_3, p_2 \cup p_3\}$. However, it is not necessary that $\mathcal{C}$ implies, say, $p_1 \cup p_2$ through the pair of edges p_1. In other words, it is not necessary that $\mathcal{C}$ implies the 4-cuts in $\mathcal{C}'$ through the pairs of edges from which they are generated. An example for that is given in Figure 5.7. Moreover, the same is true even if $\mathcal{C}'$ is generated by a collection of four pairs of edges, as we can see in Figure 5.8. However, if $\mathcal{C}'$ is generated by a collection $\mathcal{F}$ of six or more pairs of edges, then there are some pairs of pairs of edges in $\mathcal{F}$ that have “distance” at least three (there is no need to define precisely this term, but we refer to Figure 5.9 for an intuitive understanding of it). The 4-cuts that are formed by the union of such pairs of edges have the property that they are implied by $\mathcal{C}$ through them. In particular, if $\mathcal{F}$ consists of six pairs of edges, then there are three pairs of pairs of edges in $\mathcal{F}$ that are “antipodal” (see Figure 5.9). In this case, the following lemma establishes our claim. The intuitive idea behind Lemma 5.19 is that every 4-cut that is formed by the union of two pairs of edges that have distance at least three, has the property that the pairs of edges that form it are not entangled with other edges in forming 4-cuts in a way that would interfere with the straightforward way through which we would expect $\mathcal{C}$ to imply it.

Lemma 5.19 (Antipodal pairs of edges in a hexagonal family of 4-cuts). *Let $\mathcal{C}$ be a collection of 4-cuts of G, and let $\{p_1, \dots, p_6\}$ be a collection of pairs of edges that generates a cyclic family $\mathcal{C}'$ of 4-cuts that is implied by $\mathcal{C}$. We may assume w.l.o.g. that there is a partition $\{X_1, \dots, X_6\}$ of $V(G)$ such that $G[X_i]$ is connected for every $i \in \{1, \dots, 6\}$, and $E[X_i, X_{i+_6 1}] = p_i$ for every $i \in \{1, \dots, 6\}$. (See Figure 5.9). Then, $p_1 \cup p_4$ is implied by $\mathcal{C}$ through the pair of edges p_1.*

Proof. Let us suppose, for the sake of contradiction, that $C = p_1 \cup p_4$ is not implied by $\mathcal{C}$ through the pair of edges p_1. In particular, this implies that $C \notin \mathcal{C}$. Let $p_1 = \{e_1, e_2\}$

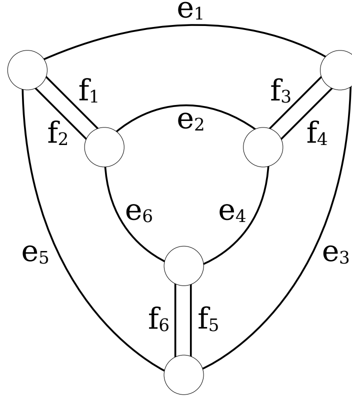


Figure 5.7: This is a 3-edge-connected graph with 4-cuts $C_1 = \{e_1, e_2, e_3, e_4\}$, $C_2 = \{e_1, e_2, e_5, e_5\}$, $C_3 = \{e_3, e_4, e_5, e_6\}$, $D_1 = \{e_1, e_5, f_1, f_2\}$, $D_2 = \{e_2, e_6, f_1, f_2\}$, $E_1 = \{e_1, e_3, f_3, f_4\}$, $E_2 = \{e_2, e_4, f_3, f_4\}$, $F_1 = \{e_3, e_5, f_5, f_6\}$ and $F_2 = \{e_4, e_6, f_5, f_6\}$. It is easy to see that $\mathcal{C} = \{D_1, D_2, E_1, E_2, F_1, F_2\}$ is a collection of 4-cuts that implies all 4-cuts of this graph. In particular, $\mathcal{C}$ implies the cyclic family of 4-cuts $\{C_1, C_2, C_3\}$, which is generated by the collection of pairs of edges $\{\{e_1, e_2\}, \{e_3, e_4\}, \{e_5, e_6\}\}$. However, notice that C_1 is not implied by $\mathcal{C}$ through the pair of edges $\{e_1, e_2\}$ (it is only implied by $\mathcal{C}$ through the pair of edges $\{e_1, e_3\}$ or $\{e_2, e_4\}$, with the implicating sequence $E_1 = \{e_1, e_3, f_3, f_4\}$, $E_2 = \{f_3, f_4, e_2, e_4\}$).

and let $p_4 = \{e_3, e_4\}$. Since C is implied by $\mathcal{C}$, but not through the pair of edges p_1, we may assume w.l.o.g. that C is implied by $\mathcal{C}$ through the pair of edges $\{e_1, e_3\}$. Then, by Lemma 5.14 we have that there is a collection $\mathcal{F}$ of pairs of edges that is returned by Algorithm 16 on input $\mathcal{C}$, such that $\{\{e_1, e_3\}, \{e_2, e_4\}\} \subset \mathcal{F}$. So let $\{f_1, f_2\}$ be a pair of edges in $\mathcal{F} \setminus \{\{e_1, e_3\}, \{e_2, e_4\}\}$. Since $\mathcal{F}$ is returned by Algorithm 16 on input $\mathcal{C}$, Proposition 5.3 implies that $C' = \{e_1, e_3, f_1, f_2\}$ is a 4-cut implied by $\mathcal{C}$.

Let $G' = G \setminus \{e_1, e_3\}$. Then the subgraphs $G'[X_1]$, $G'[X_2]$, $G'[X_4]$ and $G'[X_5]$ remain connected, and we have $E_{G'}[X_1, X_2] = \{e_2\}$ and $E_{G'}[X_4, X_5] = \{e_4\}$. Thus, it cannot be that both $(G' \setminus \{f_1, f_2\})[X_1]$ and $(G' \setminus \{f_1, f_2\})[X_2]$ are connected, or that both $(G' \setminus \{f_1, f_2\})[X_4]$ and $(G' \setminus \{f_1, f_2\})[X_5]$ are connected, because otherwise the endpoints of e_1 or e_3, respectively, would remain connected in $G \setminus \{e_1, e_3, f_1, f_2\}$ – in contradiction to the fact that C' is a 4-cut of G. Thus, we have that one of f_1, f_2 is a bridge of either $G'[X_1]$ or $G'[X_2]$, and the other is a bridge of either $G'[X_4]$ or $G'[X_5]$. Thus, we may assume w.l.o.g. (considering the symmetry of Figure 5.9), that f_1 is a bridge of $G'[X_1] = G[X_1]$ (and therefore f_2 lies in either $G[X_4]$ or $G[X_5]$). Let Y_1 and Y_2 be the

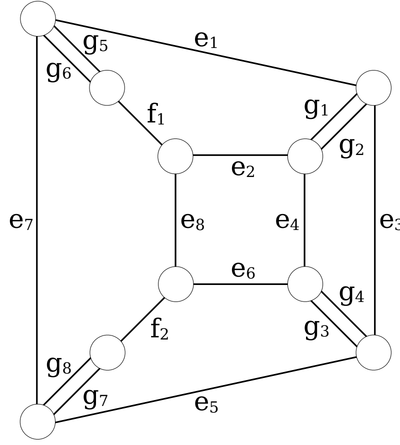


Figure 5.8: This is a 3-edge-connected graph with 4-cuts $C_1 = \{e_1, e_2, e_3, e_4\}$, $C_2 = \{e_1, e_2, e_5, e_6\}$, $C_3 = \{e_1, e_2, e_7, e_8\}$, $C_4 = \{e_3, e_4, e_5, e_6\}$, $C_5 = \{e_3, e_4, e_7, e_8\}$, $C_6 = \{e_5, e_6, e_7, e_8\}$, $D_1 = \{e_1, e_3, g_1, g_2\}$, $D_2 = \{e_2, e_4, g_1, g_2\}$, $E_1 = \{e_3, e_5, g_3, g_4\}$, $E_2 = \{e_4, e_6, g_3, g_4\}$, $F_1 = \{e_1, e_5, f_1, f_2\}$, $F_2 = \{e_2, e_6, f_1, f_2\}$, $G_1 = \{e_1, e_7, g_5, g_6\}$, $G_2 = \{e_2, e_8, g_5, g_6\}$, $H_1 = \{e_5, e_7, g_7, g_8\}$ and $H_2 = \{e_6, e_8, g_7, g_8\}$. Notice that C_1 is implied by $\{D_1, D_2\}$, C_2 is implied by $\{F_1, F_2\}$, C_3 is implied by $\{G_1, G_2\}$, C_4 is implied by $\{E_1, E_2\}$, and C_6 is implied by $\{H_1, H_2\}$. Thus, we have that $\mathcal{C} = \{C_5, D_1, D_2, E_1, E_2, F_1, F_2, G_1, G_2, H_1, H_2\}$ is a collection of 4-cuts that implies all 4-cuts of this graph. In particular, $\mathcal{C}$ implies the cyclic family of 4-cuts $\mathcal{C}' = \{C_1, C_2, C_3, C_4, C_5, C_6\}$, which is generated by the collection of pairs of edges $\{\{e_1, e_2\}, \{e_3, e_4\}, \{e_5, e_6\}, \{e_7, e_8\}\}$. However, notice that $C_2 = \{e_1, e_2, e_5, e_6\}$ is not implied by $\mathcal{C}$ through the pair of edges $\{e_1, e_2\}$, and the pairs of edges $\{e_1, e_2\}$ and $\{e_5, e_6\}$ have distance 2 in $\mathcal{C}'$.

connected components of $G'[X_1] \setminus f_1$. Since $\partial(X_1) = p_1 \cup p_6$, Lemma 5.3 implies that $|E[Y_1, V \setminus X_1]| = 2$ and $|E[Y_2, V \setminus X_1]| = 2$. Now there are three possibilities to consider: either (1) $E[Y_1, V \setminus X_1] = p_1$ (and $E[Y_2, V \setminus X_1] = p_6$), or (2) $E[Y_1, V \setminus X_1] = p_6$ (and $E[Y_2, V \setminus X_1] = p_1$), or (3) both $E[Y_1, V \setminus X_1]$ and $E[Y_2, V \setminus X_1]$ intersect with both p_1 and p_6.

Let us suppose that (1) is true. Then we have that $E[Y_1, X_2] = \{e_1, e_2\}$. Since neither of e_1, e_3, f_1, f_2 lies in $G[X_2]$, we have that $(G \setminus C')[X_2]$ remains connected, and $E_{G \setminus C'}[Y_1, X_2] = \{e_2\}$. Thus, the endpoints of e_1 remain connected in $G \setminus C'$, in contradiction to the fact that C' is a 4-cut of G. Thus, case (1) is rejected. With the analogous argument, we can reject case (2). Thus, only case (3) can be true. This implies that neither of $E[Y_1, X_6]$ and $E[Y_2, X_6]$ is empty (because each contains one

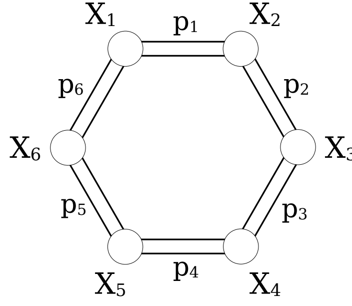


Figure 5.9: A cyclic family of 4-cuts $\mathcal{C}_6$ generated by the collection of pairs of edges $\{p_1, p_2, p_3, p_4, p_5, p_6\}$. If i and j are two indices in $\{1, \dots, 6\}$, then we say that the pairs of edges p_i and p_j have distance $min\{(i-j+6)mod6, (j-i+6)mod6\}$ in $\mathcal{C}_6$. Thus, $\{p_1, p_4\}$, $\{p_2, p_5\}$ and $\{p_3, p_6\}$ are the only pairs of pairs of edges that have distance 3 in $\mathcal{C}_6$. Notice that these pairs of pairs of edges are antipodal in this figure. The point of Lemma 5.19 is that if there is a collection $\mathcal{C}$ of 4-cuts that implies $\mathcal{C}_6$, then the 4-cuts $p_1 \cup p_4$, $p_2 \cup p_5$ and $p_3 \cup p_6$ are implied by $\mathcal{C}$ through the pairs of edges p_1, p_2 and p_3, respectively. In general, it is not necessary that this is the case for 4-cuts that are generated by pairs of pairs of edges that have distance less than 3. This is demonstrated in the previous figures, 5.7 and 5.8.

edge from p_6). But then we have that Y_1 and Y_2 (and therefore the endpoints of f_1) remain connected in $G \setminus C'$ (because $(G \setminus C')[X_6]$ is connected, and neither of $E[Y_1, X_6]$ and $E[Y_2, X_6]$ intersects with C'), in contradiction to the fact that C' is a 4-cut of G. Thus, we conclude that our initial supposition cannot be true, and therefore $p_1 \cup p_4$ is implied by $\mathcal{C}$ through the pair of edges p_1. □

Lemma 5.20 (Minimal essential 4-cuts do not cross). *Let $\mathcal{C}$ be a complete collection of 4-cuts of G, and let $\mathcal{F}_1$ and $\mathcal{F}_2$ be two distinct collections of pairs of edges with $|\mathcal{F}_1| > 2$ and $|\mathcal{F}_2| > 2$ that are returned by Algorithm 16 on input $\mathcal{C}$. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be the collections of 4-cuts that are generated by $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively. Let C_1 be an essential $\mathcal{C}_1$-minimal 4-cut, and let C_2 be an essential $\mathcal{C}_2$-minimal 4-cut. Then, C_1 and C_2 are parallel.*

Proof. Let $C_1 = \{e_1, e_2, e_3, e_4\}$ and let $C_2 = \{f_1, f_2, f_3, f_4\}$. Let us suppose, for the sake of contradiction, that C_1 and C_2 cross. Since C_1 and C_2 are essential 4-cuts, by Corollary 5.3 we may assume w.l.o.g. that C_1 and C_2 cross as in Figure 5.2. Notice that, by Lemma 5.3, we have that the four corners of Figure 5.2 are connected subgraphs of G. Thus, the boundaries of those corners are 4-cuts of G, and therefore these are 4-cuts implied by $\mathcal{C}$ (because $\mathcal{C}$ implies all 4-cuts of G). Now let X be the

connected component of $G \setminus C_1$ that contains f_1 and f_2, and let Y be the connected component of $G \setminus C_1$ that contains f_3 and f_4 (see Figure 5.10(a)). Since C_1 is $\mathcal{C}_1$-minimal, we have that either X or Y is a corner of $\mathcal{C}_1$. So let us assume, w.l.o.g., that X is a corner of $\mathcal{C}_1$. Similarly, let X' be the connected component of $G \setminus C_2$ that contains e_1 and e_2, and let Y' be the connected component of $G \setminus C_2$ that contains e_3 and e_4. Since C_2 is $\mathcal{C}_2$-minimal, we have that either X' or Y' is a corner of $\mathcal{C}_2$. Due to the symmetry of Figure 5.10(a), we may assume w.l.o.g. that X' is a corner of $\mathcal{C}_2$. (I.e., although we have assumed that X is a corner of $\mathcal{C}_1$, there is no loss of generality in assuming that X' is a corner of $\mathcal{C}_2$.)

Now, since $C_1 \in \mathcal{C}_1$ and $\mathcal{C}_1$ is generated by $\mathcal{F}_1$, there are three possibilities to consider: either (1) $\{\{e_1, e_2\}, \{e_3, e_4\}\} \subset \mathcal{F}_1$, or (2) $\{\{e_1, e_3\}, \{e_2, e_4\}\} \subset \mathcal{F}_1$, or (3) $\{\{e_1, e_4\}, \{e_2, e_3\}\} \subset \mathcal{F}_1$. Let us consider case (2) first. Since C_1 is an essential 4-cut and $|\mathcal{C}_1| \geq 3$, by Corollary 5.5 we have that $\mathcal{C}_1$ is a cyclic family of 4-cuts. Thus, we may consider the neighboring corners X_1 and X_2 of X in $\mathcal{C}_1$ such that $E[X, X_1] = \{e_1, e_3\}$ and $E[X, X_2] = \{e_2, e_4\}$. Let Y_1 and Y_2 be the connected components of $G[X] \setminus \{f_1, f_2\}$. Then, since we are in the situation depicted in Figure 5.10(b), we have that both $E[Y_1, V \setminus X]$ and $E[Y_2, V \setminus X]$ intersect with both $\{e_1, e_3\}$ and $\{e_2, e_4\}$. Thus we may assume, w.l.o.g., that $E[Y_1, X_1] = \{e_1\}$, $E[Y_1, X_2] = \{e_2\}$, $E[Y_2, X_1] = \{e_3\}$ and $E[Y_2, X_2] = \{e_4\}$ (see Figure 5.10(b)). Then, notice that it cannot be the case that either $(G \setminus C_2)[X_1]$ or $(G \setminus C_2)[X_2]$ is connected, because otherwise the endpoints of f_1 and f_2 would remain connected in $G \setminus C_2$, in contradiction to the fact that C_2 is a 4-cut of G. Thus, we have that either f_3 is a bridge of $G[X_1]$ and f_4 is a bridge of $G[X_2]$, or reversely. So let us assume, w.l.o.g., that f_3 is a bridge of $G[X_1]$ and f_4 is a bridge of $G[X_2]$. Let Z_1 and Z_1' be the connected components of $G[X_1] \setminus f_3$, and let Z_2 and Z_2' be the connected components of $G[X_2] \setminus f_4$. Then we have $E[Z_1, Z_1'] = \{f_3\}$ and $E[Z_2, Z_2'] = \{f_4\}$.

Now consider the neighboring corner X_1' of X_1 in $\mathcal{C}_1$ that is different from X. We claim that both $E[Z_1, X_1']$ and $E[Z_1', X_1']$ are non-empty. To see this, suppose the contrary. Then we may assume w.l.o.g. that $E[Z_1, X_1'] = \emptyset$. Since the graph is 3-edge-connected, we have $|\partial(Z_1)| \geq 3$. Thus, since $\partial(Z_1) = E[Z_1, Z_1'] \cup E[Z_1, X_1'] \cup E[Z_1, X]$, we have that $E[Z_1, X]$ contains at least two edges (because $E[Z_1, Z_1'] = \{f_3\}$). Thus, $E[Z_1, X]$ consists of $\{e_1, e_3\}$. But this implies that $\partial(Z_1) = \{e_1, e_3, f_3\}$ is a 3-cut of G, which is impossible (see Figure 5.10(a)). This shows that both $E[Z_1, X_1']$ and $E[Z_1', X_1']$ are non-empty. Similarly, if we let X_2' denote the neighboring corner of X_2 in $\mathcal{C}_1$ that

is different from X, then we have that both $E[Z_2, X'_2]$ and $E[Z'_2, X'_2]$ are non-empty.

Then, if $(G \setminus C_2)[X'_1]$ is connected, we have that Z_1 and Z'_1 (and therefore the endpoints of f_3) remain connected in $G \setminus C_2$, in contradiction to the fact that C_2 is a 4-cut of G. Thus, we have that $(G \setminus C_2)[X'_1]$ is disconnected. This implies that an edge from C_2 is in $G[X'_1]$, and the only candidate is f_4. Thus, we have that $X'_1 = X_2$ (and $X'_2 = X_1$), and so we have a situation like that depicted in Figure 5.10(c). But then we have that $C_1 = \{e_1, e_2, e_3, e_4\}$ is a non-essential 4-cut, because the connected components of $G \setminus C_1$ are $Y_1 \cup Y_2$ and $Z_1 \cup Z'_1 \cup Z_2 \cup Z'_2$, and there is no pair of vertices in those components that are 4-edge-connected (because $|\partial(Z_1)| = |\partial(Z'_1)| = |\partial(Z_2)| = |\partial(Z'_2)| = 3$). Thus, case (2) cannot be true. With the analogous argument we can see that case (3) also cannot be true. (To see this, just switch the labels of edges e_3 and e_4.) Thus, only case (1) is true. Similarly, if we consider the three possibilities for the collection $\mathcal{F}_2$ – i.e., either $\{\{f_1, f_2\}, \{f_3, f_4\}\} \subset \mathcal{F}_2$, or $\{\{f_1, f_3\}, \{f_2, f_4\}\} \subset \mathcal{F}_2$, or $\{\{f_1, f_4\}, \{f_2, f_3\}\} \subset \mathcal{F}_2$ –, then we can see that only $\{\{f_1, f_2\}, \{f_3, f_4\}\} \subset \mathcal{F}_2$ can be true.

Now consider a pair of edges $\{e_5, e_6\} \in \mathcal{F}_1 \setminus \{\{e_1, e_2\}, \{e_3, e_4\}\}$, and a pair of edges $\{f_5, f_6\} \in \mathcal{F}_2 \setminus \{\{f_1, f_2\}, \{f_3, f_4\}\}$. Then, Proposition 5.3 implies that $\{e_1, e_2, e_5, e_6\}$ and $\{f_1, f_2, f_5, f_6\}$ are 4-cuts implied by $\mathcal{C}$. Therefore, a repeated application of Lemma 5.5 implies that $\mathcal{F}_6 = \{\{e_1, e_2\}, \{e_3, e_4\}, \{e_5, e_6\}, \{f_1, f_2\}, \{f_3, f_4\}, \{f_5, f_6\}\}$ is a collection of pairs of edges that generates a collection $\mathcal{C}_6$ of 4-cuts. Lemma 5.13 implies that $\mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset$, and therefore $|\mathcal{F}_6| = 6$. Then, Proposition 5.2 implies that $\mathcal{C}_6$ is a cyclic family of 4-cuts.

For every $i \in \{1, 2, 3\}$, let $p_i = \{e_{2i-1}, e_{2i}\}$ and let $q_i = \{f_{2i-1}, f_{2i}\}$. Now we will demonstrate that there are $i \in \{1, 2, 3\}$ and $j \in \{1, 2, 3\}$, such that p_i and q_j are antipodal pairs of edges in $\mathcal{C}_6$. Let A, B, C, D be the corners of the square of the crossing 4-cuts C_1 and C_2, as shown in Figure 5.11. Since $G[X]$ is a corner of $\mathcal{C}_1$, we have that the pair of edges $\{e_5, e_6\}$ lies in $G[Y] = B \cup C$. Then, since the collection of pairs of edges $\{\{e_1, e_2\}, \{e_3, e_4\}, \{f_1, f_2\}, \{f_3, f_4\}\}$ generates a cyclic family of 4-cuts, and can be extended into the collection $\mathcal{F}_6$, we have that the pair of edges $\{e_5, e_6\}$ lies entirely within B or C. Similarly, since $G[X']$ is a corner of $\mathcal{C}_2$, we can infer that the pair of edges $\{f_5, f_6\}$ lies entirely within C or D. Thus, all the possible configurations for the pairs of edges in $\mathcal{F}_6$ on the hexagon of the corners of $\mathcal{C}_6$ are shown in Figure 5.11. There, we can see that, in either case, there are $i \in \{1, 2, 3\}$ and $j \in \{1, 2, 3\}$, such that p_i and q_j are antipodal pairs of edges in $\mathcal{F}_6$.

Now let $C = p_i \cup q_j$. Then, by Lemma 5.19 we have that C is implied by $\mathcal{C}$ through the pair of edges p_i. But Lemma 5.18 implies that $\mathcal{C}$ does not imply C through the pair of edges p_i (because $p_i \in \mathcal{F}_1$ and $q_j \in \mathcal{F}_2$), a contradiction. Thus, our initial assumption cannot be true, and therefore C_1 and C_2 do not cross.

□

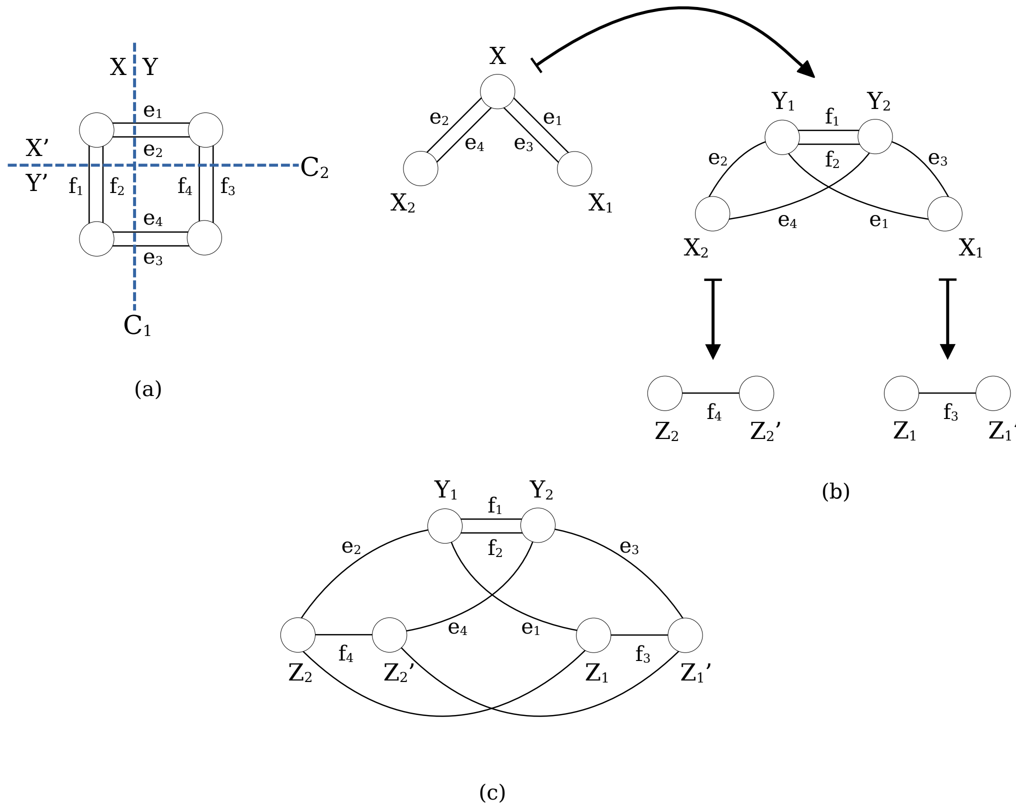


Figure 5.10: Companion figures to Lemma 5.20. (a) The square of the crossing 4-cuts C_1 and C_2. The sides of C_1 are X and Y, and the sides of C_2 are X' and Y'. (b) We assume that X is the corner of C_1 in $\mathcal{C}_1$, and $\{\{e_1, e_3\}, \{e_2, e_4\}\} \subset \mathcal{F}_1$. X_1 and X_2 are the neighboring corners of X in $\mathcal{C}_1$. Then, we have w.l.o.g. that f_3 is a bridge of $G[X_1]$ and f_4 is a bridge of $G[X_2]$. (c) We infer this situation, which contradicts the essentiality of C_1.

We note that the condition of essentiality of both 4-cuts C_1 and C_2 in Lemma 5.20 cannot be removed without destroying the inference of the lemma. This is demonstrated in Figure 5.12.

Lemma 5.21 (The essential quasi-isolated 4-cuts are replaceable). *Let $\mathcal{C}$ be a collection of 4-cuts of G, and let C be an essential quasi $\mathcal{C}$-isolated 4-cut. Let x and y be two vertices*

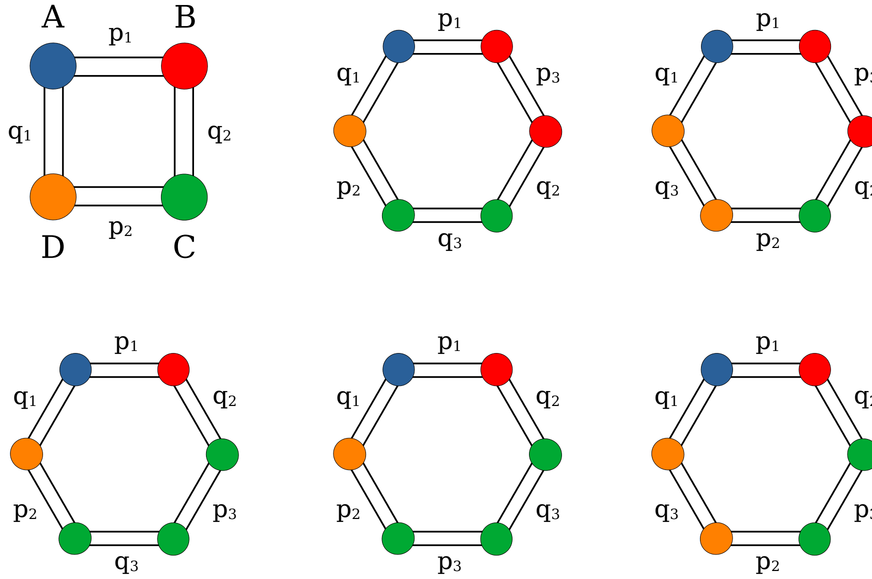


Figure 5.11: $\mathcal{F} = \{p_1, p_2, q_1, q_2\}$ is a collection of pairs of edges that generates a cyclic family of 4-cuts $\mathcal{C}$ with corners A, B, C and D. We consider all the different ways in which $\mathcal{C}$ can be expanded into a cyclic family of 4-cuts $\mathcal{C}_6$, where $\mathcal{C}_6$ is generated by the collection of pairs of edges $\mathcal{F} \cup \{p_3, q_3\}$, under the restriction that $p_3 \in B \cup C$ and $q_3 \in C \cup D$. The colors of the corners of $\mathcal{C}_6$ correspond to the colors of the corners of the square that got expanded. Notice that, in either case, we have that there are $i, j \in \{1, 2, 3\}$ such that p_i and q_j are antipodal pairs of edges in $\mathcal{C}_6$.

that are separated by C. Then, there is a collection of pairs of edges $\mathcal{F}$ with $|\mathcal{F}| > 2$ that is returned by Algorithm 16 on input $\mathcal{C}$, such that $\mathcal{F}$ generates a 4-cut that separates x and y.

Proof. Let $C = \{e_1, e_2, e_3, e_4\}$. Since C is quasi $\mathcal{C}$-isolated, we may assume w.l.o.g. that there is a pair of edges $\{e, e'\} \notin \{\{e_1, e_2\}, \{e_3, e_4\}\}$ such that $\mathcal{F}' = \{\{e_1, e_2\}, \{e_3, e_4\}, \{e, e'\}\}$ generates a collection $\mathcal{C}'$ of 4-cuts implied by $\mathcal{C}$. Since C is an essential 4-cut, by Corollary 5.4 we have that $\mathcal{C}'$ cannot be a degenerate family of 4-cuts. Therefore, by Lemma 5.9 we have that $\mathcal{C}'$ is a cyclic family of 4-cuts. Thus, there is a partition $\{X_1, X_2, X\}$ of $V(G)$, such that $E[X, X_1] = \{e_1, e_2\}$, $E[X, X_2] = \{e_3, e_4\}$ and $E[X_1, X_2] = \{e, e'\}$ (see Figure 5.13). Notice that the connected components of $G \setminus C$ are X and $X_1 \cup X_2$. Thus, since x, y are separated by C, we may assume w.l.o.g. that $x \in X$ and $y \in X_1$.

Let $C' = \{e_1, e_2, e, e'\}$. Since C' is generated by $\mathcal{F}'$, we have that C' is a 4-cut implied by $\mathcal{C}$. Let us suppose, for the sake of contradiction, that C' is implied by $\mathcal{C}$ through the pair of edges $\{e_1, e_2\}$. Then, Lemma 5.14 implies that there is a collection $\mathcal{F}$ of pairs of edges that is returned by Algorithm 16 on input $\mathcal{C}$ such that $\{\{e_1, e_2\}, \{e, e'\}\} \subseteq \mathcal{F}$. Since C is a quasi $\mathcal{C}$-isolated 4-cut, we have that the collection of pairs of edges $\mathcal{F}'' = \{\{e_1, e_2\}, \{e_3, e_4\}\}$ is returned by Algorithm 16 on input $\mathcal{C}$. Since $\{e, e'\} \neq \{e_3, e_4\}$, we have that $\mathcal{F} \neq \mathcal{F}''$. But then, Lemma 5.13 implies that $\mathcal{F} \cap \mathcal{F}'' = \emptyset$, in contradiction to the fact that $\mathcal{F} \cap \mathcal{F}'' = \{e_1, e_2\}$. Thus, we have shown that C' is not implied by $\mathcal{C}$ through the pair of edges $\{e_1, e_2\}$. This further implies that $C' \notin \mathcal{C}$.

Since C' is nonetheless implied by $\mathcal{C}$, we may assume w.l.o.g. that C' is implied by $\mathcal{C}$ through the pair of edges $\{e_1, e\}$. Then, Lemma 5.14 implies that there is a collection $\mathcal{F}$ of pairs of edges with $\{\{e_1, e\}, \{e_2, e'\}\} \subset \mathcal{F}$ that is returned by Algorithm 16 on input $\mathcal{C}$. Thus we have that $|\mathcal{F}| > 2$, and $\mathcal{F}$ generates C'. Notice that the 4-cut $C' = \{e_1, e_2, e, e'\}$ separates x and y. Thus, the proof is complete. □

5.3 Using a DFS-tree for some problems concerning 4-cuts

In this section we present some applications of identifying 4-cuts on a DFS-tree. First, there is a linear-time preprocessing, after which we can report the r-size of any 4-cut in constant time (Lemma 5.22). Second, there is a linear-time preprocessing, after which we can check the essentiality of any 4-cut in constant time (Proposition 5.4). And third, given a parallel family of 4-cuts $\mathcal{C}$, we can compute in linear time the atoms of $\mathcal{C}$ (Proposition 5.5), as well as an oracle that can answer queries of the form "given two vertices x and y, return a 4-cut from $\mathcal{C}$ that separates x and y, or determine that no such 4-cut exists", in constant time (Corollary 5.10).

Let G be a 3-edge-connected graph, and let r be a vertex of G. Consider the following problems. Given a 4-cut C of G (as an edge-set), what is the size of the side of C that does not contain r? Also, which endpoints of the edges in C lie in the connected component of $G \setminus C$ that contains r? We will show how we can answer those questions in constant time, provided that we have computed a DFS-tree of G. Given a 4-cut C of G, the number of vertices of the part of C that does not contain r is called the *r-size* of C.

So let T be a DFS-tree of G with start vertex (root) r [63]. We identify the vertices

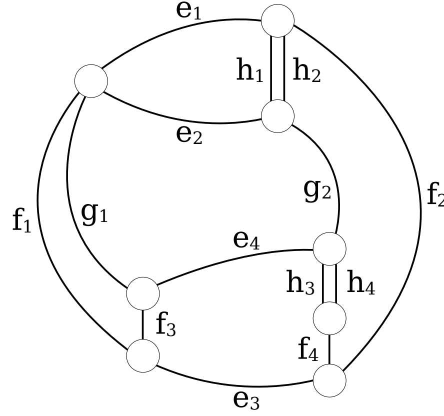


Figure 5.12: This is a 3-edge-connected graph with 4-cuts: $C_1 = \{e_1, e_2, e_3, e_4\}$, $C_2 = \{e_1, e_2, f_1, g_1\}$, $C_3 = \{e_1, e_2, f_2, g_2\}$, $C_4 = \{e_3, e_4, f_1, g_1\}$, $C_5 = \{e_3, e_4, f_2, g_2\}$, $C_6 = \{f_1, f_2, g_1, g_2\}$, $C_7 = \{f_3, f_4, g_1, g_2\}$, $C_8 = \{f_1, f_2, f_3, f_4\}$, $C_9 = \{e_1, f_2, h_1, h_2\}$, $C_{10} = \{e_2, g_2, h_1, h_2\}$, $C_{11} = \{f_2, e_3, h_3, h_4\}$ and $C_{12} = \{g_2, e_4, h_3, h_4\}$. We have that C_3 is implied by $\{C_9, C_{10}\}$, C_4 is implied by $\{C_1, C_2\}$, C_5 is implied by $\{C_{11}, C_{12}\}$, and C_6 is implied by $\{C_7, C_8\}$. Thus, $\mathcal{C} = \{C_1, C_2, C_7, C_8, C_9, C_{10}, C_{11}, C_{12}\}$ is a complete collection of 4-cuts of this graph. If we apply Algorithm 16 on $\mathcal{C}$, we will get as a result the collections of pairs of edges $\mathcal{F}_1 = \{\{e_1, e_2\}, \{f_1, g_1\}, \{e_3, e_4\}\}$, $\mathcal{F}_2 = \{\{f_1, f_2\}, \{g_1, g_2\}, \{f_3, f_4\}\}$, $\mathcal{F}_3 = \{\{e_1, f_2\}, \{e_2, g_2\}, \{h_1, h_2\}\}$, and $\mathcal{F}_4 = \{\{f_2, e_3\}, \{g_2, e_4\}, \{h_3, h_4\}\}$. Notice that $\mathcal{F}_1$ and $\mathcal{F}_2$ generate the cyclic families of 4-cuts $\mathcal{C}_1 = \{C_1, C_2, C_4\}$ and $\mathcal{C}_2 = \{C_6, C_7, C_8\}$, respectively. Since $|\mathcal{C}_1| = 3$, we have that every 4-cut in $\mathcal{C}_1$ is $\mathcal{C}_1$-minimal. Similarly, every 4-cut contained in $\mathcal{C}_2$ is $\mathcal{C}_2$-minimal. Thus, C_1 is a $\mathcal{C}_1$-minimal 4-cut, and C_8 is a $\mathcal{C}_2$-minimal 4-cut. However, we have that C_1 and C_8 cross. Notice that C_8 is not an essential 4-cut (because one of its sides consists of the endpoints of e_3, both of which have degree 3, and therefore they are not 4-edge-connected with any other vertex of the graph). On the other hand, C_1 is an essential 4-cut (because e.g. the endpoints of e_1 are 4-edge-connected). Thus, the condition of essentiality of both 4-cuts in the statement of Lemma 5.20 cannot be removed.

of G with their order of visit by the DFS. Thus, $r = 1$, and the last vertex visited by G is n. For a vertex $v \neq r$ of G, we let $p(v)$ denote the parent of v on T. (Thus, v is a child of $p(v)$.) For every two vertices u and v, we let $T[u, v]$ denote the simple tree-path from u to v. A vertex v is called an ancestor of u, if v lies on the tree-path $T[r, u]$. (Equivalently, u is a descendant of v.) The set of all descendants of v is denoted as

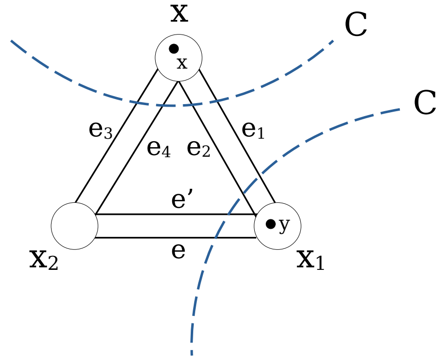


Figure 5.13: A depiction of the situation analyzed in Lemma 5.21.

$T(v)$. (In particular, we have $v \in T(v)$.) The number of descendants of v is denoted as $ND(v)$. In other words, $ND(v) = |T(v)|$. The ND values can be computed easily during the DFS, because they satisfy the recursive formula $ND(v) = ND(c_1)+\cdots+ND(c_k)+1$, where $c_1, \ldots, c_k$ are the children of v. We can use the ND values in order to check the ancestry relation in constant time. Specifically, given two vertices u and v, we have that u is a descendant of v if and only if $v \leq u \leq v + ND(v) - 1$. Equivalently, we have $T(v) = \{v, v+1, \ldots, v+ND(v)-1\}$.

5.3.1 Computing the r-size of 4-cuts

Let C be a 4-cut of G. We will show how to answer each of the questions above in constant time. To do this, we first consider the connected components of $T \setminus C$. These are determined by the tree-edges in C. Notice that C must contain at least one tree-edge (because otherwise $G \setminus C$ remains connected through the tree-edges from T). We distinguish the following cases.

First, let us consider the case that C contains only one tree-edge $(u, p(u))$. Then the connected components of $T \setminus C$ are $T(u)$ and $T(r) \setminus T(u)$. Thus, the r-size of C is $ND(u)$. Furthermore, for each non-tree edge $(x, y) \in C$, we can easily determine in constant time which of x and y lies in $T(u)$, and which lies in $T(r) \setminus T(u)$.

Now let us consider the case that C contains exactly two tree-edges $(u, p(u))$ and $(v, p(v))$. By Lemma 3.14 in Section 3.2 we have that one of u and v must be an ancestor of the other. Thus, we may assume w.l.o.g. that v is a proper ancestor of u. Then the connected components of $T\setminus C$ are given by $T(u)$, $T(v)\setminus T(u)$, and $T(r)\setminus T(v)$. Thus, the connected components of $G \setminus C$ are given by the union of two of those

subtrees, plus the other subtree. Now we are guided by the fact that the endpoints of the edges in C lie in different connected components of $G \setminus C$. Thus, $T(u) \cup (T(r) \setminus T(v))$ lies in a distinct connected component of $G \setminus C$ than $T(v) \setminus T(v)$. (Because $u \in T(u)$ whereas $p(u) \in T(v) \setminus T(v)$, and $v \in T(v) \setminus T(u)$ whereas $p(v) \in T(r) \setminus T(v)$.) Therefore, the connected components of $G \setminus C$ are given by $T(u) \cup (T(r) \setminus T(v))$ and $T(v) \setminus T(u)$. This implies that the r-size of C is $ND(v) - ND(u)$. Furthermore, it is easy to determine which endpoints of the edges in C lie in which connected components of $G \setminus C$.

Now let us consider the case that C contains exactly three tree-edges $(u, p(u))$, $(v, p(v))$ and $(w, p(w))$. Then, by Lemma 3.14 in Section 3.2 we have that one of u, v and w must be an ancestor of the other two. Thus, we may assume w.l.o.g. that w is an ancestor of both u and v. Now there are two cases to consider: either u and v are not related as ancestor and descendant, or one of u and v is an ancestor of the other. Let us consider the first case first. Then the connected components of $T \setminus C$ are given by $T(u)$, $T(v)$, $T(w) \setminus (T(u) \cup T(v))$ and $T(r) \setminus T(w)$. Then, in order to determine the connected components of $G \setminus C$, we are guided by the property that the endpoints of the edges in C lie in different connected components of $G \setminus C$. Thus, it is not difficult to see that the connected components of $G \setminus C$ are given by $T(u) \cup T(v) \cup (T(r) \setminus T(w))$ and $T(w) \setminus (T(u) \cup T(v))$. Thus, the r-size of C is $ND(w) - ND(u) - ND(v)$. Also, given an edge from C, it is easy to determine which endpoints of the edges in C lie in which connected component of $G \setminus C$. Now let us consider the case that one of u and v is an ancestor of the other. We may assume w.l.o.g. that v is an ancestor of u. Then we can see as previously that the connected components of $G \setminus C$ are $T(u) \cup (T(w) \setminus T(v))$ and $(T(v) \setminus T(u)) \cup (T(r) \setminus T(w))$. Thus, the r-size of C is $ND(u) + ND(w) - ND(v)$. Furthermore, it is easy to determine which endpoints of the edges in C lie in the connected component of $G \setminus C$ that contains r.

In the case that C consists of four tree-edges we follow the same arguments as previously. In each case, the connected components of $G \setminus C$ are given by unions and differences of five subtrees of T. Thus, given any vertex x, we can check in constant time whether x belongs to the connected component of $G \setminus C$ that contains r. Furthermore, we can easily compute the r-size of C as previously.

The results of this section are summarized in the following.

Lemma 5.22. *Let G be a 3-edge-connected graph, and let r be a vertex of G. Then there is a linear-time preprocessing of G, such that we can answer queries of the form "given a 4-cut C of G, determine the r-size of C" and "given a 4-cut C of G, determine the endpoints of*

the edges in C that lie in the connected component of $G \setminus C$ that contains r", in constant time.

5.3.2 Checking the essentiality of 4-cuts

Here we provide an oracle for performing essentiality checks for 4-cuts of a 3-edge-connected graph G. Specifically, after a linear-time preprocessing of G, if we are given a 4-cut C of G (as an edge-set), we can determine in constant time whether C is an essential 4-cut of G.

Proposition 5.4. *Let G be a 3-edge-connected graph. We can preprocess G in linear time, so that we can perform essentiality checks for 4-cuts of G in constant time.*

Proof. First, we compute the 4-edge-connected components of G. This can be done in linear time (see [36] or [50]). Then, for every 4-edge-connected component S of G, we connect all vertices of S in a path, by introducing artificial edges in G. Specifically, if $x_1, \dots, x_k$ are the vertices in S, then we introduce the artificial edges $(x_1, x_2), (x_2, x_3), \dots, (x_{k-1}, x_k)$ into G. This takes $O(n)$ time in total, where n is the number of vertices of G. Let G' be the resulting graph. (Thus, G' is given by G, plus the artificial edges we have introduced.) Then we perform the linear-time preprocessing described in Proposition 6.1 on G', so that we can answer connectivity queries for pairs of vertices of G', in the presence of at most four edge-failures, in constant time.

Now let C be a 4-cut of G. Let (x, y) be any edge in C. We claim that C is an essential 4-cut of G if and only if x and y are connected in $G' \setminus C$. To see this, let us assume first that C is an essential 4-cut of G. This means that there are two vertices u and v that are 4-edge-connected and lie in different connected components of $G \setminus C$. Then, by construction of G', we have that u and v are connected in G'. This implies that the two connected components of $G \setminus C$ are connected in $G' \setminus C$. (To be precise: if X and Y are the connected components of $G \setminus C$, then there is at least one edge between X and Y in $G' \setminus C$.) Thus, $G' \setminus C$ is connected, and therefore x and y are connected in $G' \setminus C$. Conversely, suppose that x and y are connected in $G' \setminus C$. Since C is a 4-cut of G, we have that x and y are disconnected in $G \setminus C$. Let X and Y be the connected components of $G \setminus C$. We may assume w.l.o.g. that $x \in X$ and $y \in Y$. Then, since x and y are connected in $G' \setminus C$, there is a path P from x to y in $G' \setminus C$. Since this path starts from a vertex in X and ends in a vertex in Y, it must use an edge (u, v) such that $u \in X$ and $v \in Y$. The edge (u, v) does not exist in G (because

otherwise $G \setminus C$ would be connected). Thus, it is one of the artificial edges that we have introduced. This means that u and v belong to the same 4-edge-connected component of G. Thus, C separates a pair of 4-edge-connected vertices of G, and therefore it is an essential 4-cut of G.

Thus, we can determine if C is an essential 4-cut of G, by simply checking whether the endpoints of an edge in C are connected in $G' \setminus C$. Since C is a 4-element set, we can perform this check in constant time, due to the preprocessing of G' according to Proposition 6.1. □

5.3.3 Computing the atoms of a parallel family of 4-cuts

Let $\mathcal{C}$ be a parallel family of 4-cuts. This implies that $\mathcal{C}$ has size $O(n)$ (see, e.g., [24]). Our goal is to show how to compute efficiently the collection *atoms*($\mathcal{C}$). That is, we want to compute the partition $\mathcal{P}$ with the property that two vertices are separated by a set in $\mathcal{P}$ if and only if they are separated by a 4-cut in $\mathcal{C}$. Here we follow the idea in [36], that computes the 4-edge connected components of a 3-edge-connected graph given its collection of 3-cuts. [36] essentialy provides an algorithm to compute *atom*($\mathcal{C}_{3cuts}$). On a high level, the idea is to break the graph into two components according to every 4-cut $C \in \mathcal{C}$. Specifically, let $C = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$ be a 4-cut in $\mathcal{C}$. Then, let X and Y be the two connected components of $G \setminus C$, and assume w.l.o.g. that $\{x_1, x_2, x_3, x_4\} \subseteq X$ and $\{y_1, y_2, y_3, y_4\} \subseteq Y$. Then we attach an auxiliary vertex y to X, and the edges $(x_1, y), (x_2, y), (x_3, y), (x_4, y)$. Similarly, we attach an auxiliary vertex x to Y, and the edges $(x, y_1), (x, y_2), (x, y_3), (x, y_4)$. The purpose of those auxiliary vertices and edges is to simulate for each part X and Y the existence of the other part, while maintaining the same connections. Let G' denote the resulting graph; we call this the result of splitting G according to C. Notice that $V(G') = V(G) \sqcup \{x, y\}$. The non-auxiliary vertices of G' (that is, the vertices in $V(G)$) are called ordinary. Now, if C is the unique 4-cut in $\mathcal{C}$, then *atoms*($\mathcal{C}$) is given by the connected components of G'. More precisely, we compute the two connected components X' and Y' of G', and we keep the collection of the ordinary vertices from each component (thus, we get X and Y). If there are more 4-cuts in $\mathcal{C}$, then we keep doing the same process, this time splitting G'. Due to the parallelicity of $\mathcal{C}$, we have that a 4-cut $C' \in \mathcal{C}$ with $C' \neq C$ lies entirely within a connected component of G'. And due to the construction of G', we have that (the edge-set corresponding to) C' is a 4-cut of G'. Thus, it makes

sense to split G' according to its 4-cut C'. When no more splittings are possible (i.e., when we have used every 4-cut in $\mathcal{C}$ for a splitting), then we compute the connected components of the final graph, and we collect the subsets of the ordinary vertices from each connected component. Thus, we get *atoms*($\mathcal{C}$).

In order to prove the correctness of the above procedure, we need to formalize the concept of splitting a graph according to a 4-cut. A key-concept that we will use throughout is the quotient map to a split graph.

Definition 5.7 (Splitting a graph according to a 4-cut). Let G be a connected graph, let $C = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$ be a 4-cut of G, and let X and Y be the two sides of C. We may assume w.l.o.g. that $\{x_1, x_2, x_3, x_4\} \subseteq X$ and $\{y_1, y_2, y_3, y_4\} \subseteq Y$. Then we define the two split graphs G_X and G_Y of G according to C as follows. We introduce two auxiliary vertices x_C and y_C (that simulate the parts X and Y, respectively). Then the vertex set of G_X is $X \cup \{y_C\}$, and the edge set of G_X is $E(G[X]) \cup \{(x_1, y_C), (x_2, y_C), (x_3, y_C), (x_4, y_C)\}$. Similarly, the vertex set of G_Y is $Y \cup \{x_C\}$, and the edge set of G_Y is $E(G[Y]) \cup \{(x_C, y_1), (x_C, y_2), (x_C, y_3), (x_C, y_4)\}$. (See Figure 5.14 for an example.)

X is called the set of the ordinary vertices of G_X, and Y is called the set of the ordinary vertices of G_Y. Notice that G_X and G_Y is the quotient graph that is formed from G by shriking Y and X, respectively, into a single node. Then we also define the quotient maps q_X and q_Y from $V(G)$ to $V(G_X)$ and $V(G_Y)$, respectively. q_X coincides with the identity map on X, and it sends Y onto y_C. (In other words, $q_X(v) = v$ for every $v \in X$, and $q_X(v) = y_C$ for every $v \in Y$.) Similarly, q_Y coincides with the identity map on Y, and it sends X onto x_C. (In other words, $q_Y(v) = v$ for every $v \in Y$, and $q_Y(v) = x_C$ for every $v \in X$.) These maps induce a natural correspondence between edges of G and edges of G_X and G_Y. Specifically, for every edge (u, v) of G, we let $q_X((u, v)) := (q_X(u), q_X(v))$ and $q_Y((u, v)) := (q_Y(u), q_Y(v))$.[4]

The following lemma shows that the operation of splitting a graph according to a 4-cut maintains families of parallel 4-cuts inside the split graphs.

[4]More precisely, since G is a multigraph, every edge $e = (u, v)$ of G has a unique edge-identifier i. Thus, we can consider this edge as a triple (u, v, i). Then q_X maps (u, v, i) into $(q_X(u), q_X(v), i)$, so that, if $q_X(u) \neq q_X(v)$, then $q_X(e)$ is a unique edge of G_X (i.e., it is not the image of any other edge of G through q_X). However, in order to keep our notation and our arguments simple, we will drop this consideration, and we will keep considering the edges of G as pairs of vertices.

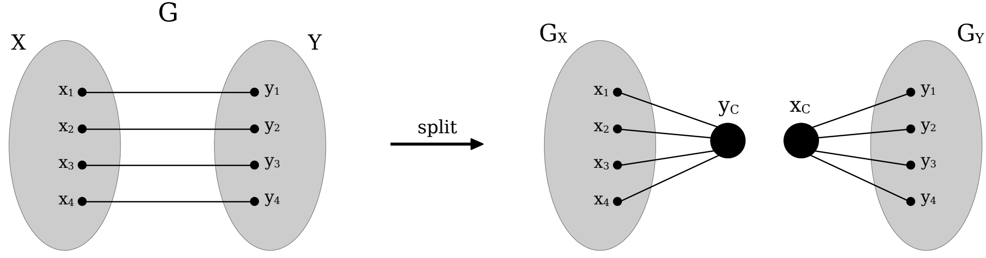


Figure 5.14: Splitting a graph G according to a 4-cut $C = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$ with sides X and Y. We introduce two new auxiliary vertices x_C and y_C, that simulate the parts X and Y, respectively.

Lemma 5.23. *Let C be a 4-cut of G, let X and Y be the two sides of C, and let (G_X, q_X) and (G_Y, q_Y) be the corresponding split graphs of G according to C, together with the respective quotient maps. Then we have the following.*

(a) Let C' be a 4-cut of G, distinct from C, that is parallel with C. Then one of $q_X(C')$ and $q_Y(C')$ contains at least one self-loop, and the other is a set of four edges.

(b) Let C' be as in (a), and suppose that $q_X(C')$ is a set of four edges (of G_X). Then $q_X(C')$ is a 4-cut of G_X. Let X' and Y' be the sides of $q_X(C')$ in G_X. Then $q_X^{-1}(X')$ and $q_X^{-1}(Y')$ are the two sides of C' in G.

(c) Let C_1 and C_2 be two 4-cuts of G, distinct from C, that are parallel with C and among themselves. Suppose that $q_X(C_1)$ and $q_X(C_2)$ are 4-cuts of G_X. Then $q_X(C_1)$ and $q_X(C_2)$ are two distinct parallel 4-cuts of G_X.

Proof. By Definition 5.7, there are two auxiliary vertices x_C and y_C, such that $V(G_X) = X \cup \{y_C\}$, $V(G_Y) = Y \cup \{x_C\}$, q_X coincides with the identity map on X and $q_X(Y) = \{y_C\}$, and q_Y coincides with the identity map on Y and $q_Y(X) = \{x_C\}$.

(a) Let X' and Y' be the two sides of C' in G. Then, since C and C' are parallel, we have that one of X' and Y' lies entirely within X or Y. Thus, we may assume w.l.o.g. that $X' \subset X$. (We have strict inclusion, because $C' \neq C$.) Let (x, y) be an edge in C'. Then we may assume w.l.o.g. that $x \in X'$ and $y \in Y'$. Since $X' \subset X$, we have $q_X(x) = x$. If $y \in X$, then we have $q_X(y) = y$. Otherwise, we have $q_X(y) = y_C$. Thus, in either case we have $q_X(x) \neq q_X(y)$, and therefore $q_X((x, y))$ is an edge of G_X. This shows that $q_X(C')$ is a set of four edges of G_X.

On the other hand, let $C' = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$, and let us assume w.l.o.g. that $\{x_1, x_2, x_3, x_4\} \subseteq X'$ and $\{y_1, y_2, y_3, y_4\} \subseteq Y'$. Then, since $X' \subset X$, for every $i \in \{1, 2, 3, 4\}$ we have $q_Y(x_i) = x_C$. Since $E_G[X, Y] = C$ and $C \neq C'$, there must be an $i \in \{1, 2, 3, 4\}$ such that $(x_i, y_i) \notin E_G[X, Y]$. Thus, since $x_i \in X$, we have $y_i \notin Y$, and therefore $y_i \in X$. This implies that $q_Y(y_i) = x_C$, and therefore $q_Y((x_i, y_i))$ is a self-loop.

(b) Let X' and Y' be the two sides of C' in G. Then, since C and C' are parallel, we have that one of X' and Y' lies entirely within X or Y. Since $q_X(C')$ is a set of four edges of G_X, by (a) we have that $q_Y(C')$ contains at least one self-loop. Then, by following the argument of (a), we have that it cannot be that one of X' and Y' lies entirely within Y (because then we would have that $q_Y(C')$ consists of four edges of G_Y). Thus, one of X' and Y' lies entirely within X. Then, we may assume w.l.o.g. that $X' \subset X$. This implies that $Y \subset Y'$.

Now we will establish a correspondence between paths in G and paths in G_X, that satisfies some useful properties. Let P be a path in G with endpoints x and y. We define a path $\widetilde{P}$ in G_X as follows. First, suppose that both x and y are in X. If P uses edges only from $G[X]$, then $\widetilde{P} = P$. Otherwise, let $v_1, (v_1, v_2), v_2, \ldots, (v_{k-1}, v_k), v_k$ be a maximal segment of P such that $v_1 \in X$, $\{v_2, \ldots, v_{k-1}\} \subseteq Y$, and $v_k \in X$. (Notice that, since P starts from X and ends in X, there must exist such a maximal segment of P, and it has $k \geq 3$.) Then we replace this segment with $v_1, (v_1, y_C), y_C, (y_C, v_k), v_k$. We repeat this process until we arrive at a sequence $\widetilde{P}$ of alternating vertices and edges that does not use vertices from Y. Then we can see that $\widetilde{P}$ is a path in G_X from x to y. Furthermore, $\widetilde{P}$ has the following properties. First, every occurrence of a vertex from X in P is maintained in $\widetilde{P}$. Second, every maximal segment of occurrences of vertices from Y in P is replaced by a single occurrence of y_C. Third, every occurrence of an edge (z, w) such that not both z and w are in Y, is replaced by an occurrence of $(q_X(z), q_X(w))$. And fourth, all the vertices and edges used by $\widetilde{P}$ are essentially given by the previous three properties.

Now suppose that one of x and y lies in X, and the other lies in Y. Then let us suppose that $x \in X$ and $y \in Y$ (in the other case, we have a similar definition and properties). Then, let $w_1, (w_1, w_2), w_2, \ldots, (w_{l-1}, w_l), w_l$ be the final segment of P that satisfies $w_1 \in X$ and $\{w_2, \ldots, w_l\} \subseteq Y$. (Notice that, since P starts from X and ends in Y, this is indeed the form of the final part of P, and it has $l \geq 2$.) Then we replace this segment with $w_1, (w_1, y_C), y_C$. Then, we perform the substitutions that we

described previously for segments of P of the form $v_1, (v_1, v_2), v_2, \ldots, (v_{k-1}, v_k), v_k$ that are maximal w.r.t. $v_1 \in X$, $\{v_2, \ldots, v_{k-1}\} \subseteq Y$, and $v_k \in X$. Let $\widetilde{P}$ be the result after we have applied all those substitutions. Then we can see that $\widetilde{P}$ is a path in G_X from x to y_C. Furthermore, $\widetilde{P}$ satisfies the four properties that we described previously (that essentially define $\widetilde{P}$).

Now let $Y'' = q_X(Y')$. (Thus, we have $Y'' = (Y' \cap X) \cup \{y_C\}$.) Notice that $q_X^{-1}(X') = X'$ and $q_X^{-1}(Y'') = Y'$. We will show that $q_X(C')$ is a 4-cut of G_X with sides X' and Y''. First, notice that $\{X', Y''\}$ constitutes a partition of $V(G_X)$, and $E_{G_X}[X', Y''] = q_X(C')$. Thus, it is sufficient to show that both X' and Y'' induce a connected subgraph of $G_X \setminus q_X(C')$. We will derive this result as a consequence of the correspondence between paths in G and paths in G_X.

So let x and y be two vertices in X'. Then, since X' is a connected component of $G \setminus C'$, there is a path P from x to y in $G \setminus C'$ that uses vertices only from X'. Thus, $\widetilde{P}$ is a path from x to y that avoids the edges from $q_X(C')$ and uses vertices only from X'. This shows that X' induces a connected subgraph of $G_X \setminus q_X(C')$.

Now let x and y be two vertices in Y''. Let us assume first that none of x and y is y_C. (Thus, we have that both x and y are in X.) Since Y' is a connected component of $G \setminus C'$, there is a path P from x to y in $G \setminus C'$ that uses vertices only from Y'. Then $\widetilde{P}$ is a path from x to y that uses vertices only from $q_X(Y')$ and avoids the edges from $q_X(C')$. Now let us assume that one of x and y is y_C, and the other is not. Then we may assume w.l.o.g. that $y = y_C$ and $x \in Y'$. Since x is a vertex in $V(G_X) \setminus \{y_C\}$, notice that $x \in X$. Now let y_0 be a vertex in Y. Then, since Y' is a connected component of $G \setminus C'$ that contains Y, there is a path P from x to y_0 in $G \setminus C'$ that uses vertices only from Y'. Then $\widetilde{P}$ is a path from x to y_C that uses vertices only from $q(Y')$ and avoids the edges from $q_X(C')$. Thus, in either case we have that x and y are connected in $G_X \setminus q_X(C')$ through a path that uses vertices only from Y''. This shows that Y'' induces a connected subgraph of $G_X \setminus q_X(C')$. Thus, we have that $q_X(C')$ is a 4-cut of G_X, with sides X' and Y''. Since we have $q_X^{-1}(X') = X'$ and $q_X^{-1}(Y'') = Y'$, this completes the proof.

(c) Let X_1 and Y_1 be the sides of $q_X(C_1)$ in G_X, and let X_2 and Y_2 be the sides of $q_X(C_2)$ in G_X. Then, by (b) we have that $q_X^{-1}(X_1)$ and $q_X^{-1}(Y_1)$ are the two sides of C_1 in G, and $q_X^{-1}(X_2)$ and $q_X^{-1}(Y_2)$ are the two sides of C_2 in G. Thus, since C_1 and C_2 are distinct, we have $\{q_X^{-1}(X_1), q_X^{-1}(Y_1)\} \neq \{q_X^{-1}(X_2), q_X^{-1}(Y_2)\}$, and therefore $\{X_1, Y_1\} \neq \{X_2, Y_2\}$. This means that $q_X(C_1)$ and $q_X(C_2)$ are distinct 4-cuts of G_X. Since C_1

and C_2 are parallel 4-cuts of G, at least one of the intersections $q_X^{-1}(X_1) \cap q_X^{-1}(X_2)$, $q_X^{-1}(X_1) \cap q_X^{-1}(Y_2)$, $q_X^{-1}(Y_1) \cap q_X^{-1}(X_2)$, $q_X^{-1}(Y_1) \cap q_X^{-1}(Y_2)$ is empty, and therefore at least one of the inverse images $q_X^{-1}(X_1 \cap X_2)$, $q_X^{-1}(X_1 \cap Y_2)$, $q_X^{-1}(Y_1 \cap X_2)$, $q_X^{-1}(Y_1 \cap Y_2)$ is empty. Since $q_X : G \to G_X$ is a surjective map, this implies that at least one of the intersections $X_1 \cap X_2$, $X_1 \cap Y_2$, $Y_1 \cap X_2$, $Y_1 \cap Y_2$ is empty. This means that the 4-cuts $q_X(C_1)$ and $q_X(C_2)$ are parallel. □

Lemma 5.23 implies that if we have a parallel family $\mathcal{C}$ of 4-cuts, then we can successively partition the graph according to all 4-cuts in $\mathcal{C}$. This is made precise in the following.

Definition 5.8 (Splitting a graph according to a parallel family of 4-cuts). Let G be a connected graph, let $\mathcal{C}$ be a parallel family of 4-cuts of G, and let C be a 4-cut in $\mathcal{C}$. Let X and Y be the sides of C in G, and let (G_X, q_X) and (G_Y, q_Y) be the corresponding split graphs, together with the respective quotient maps. Then, Lemma 5.23 implies that q_X maps some of the 4-cuts from $\mathcal{C} \setminus \{C\}$ into a collection $\mathcal{C}_X$ of parallel 4-cuts of G_X, and q_Y maps the remaining 4-cuts from $\mathcal{C} \setminus \{C\}$ into a collection $\mathcal{C}_Y$ of parallel 4-cuts of G_Y. Then we can repeat the same process into G_X and G_Y, with the collections of 4-cuts $\mathcal{C}_X$ and $\mathcal{C}_Y$, respectively. Let $G_1, \dots, G_k$ be the final split graphs that we get, after we have completed this process. (We note that $k = |\mathcal{C}|+1$.) For every $i \in \{1, \dots, k\}$, we denote $V(G_i) \cap V(G)$ as G_i^o, and we call it the set of the ordinary vertices of G_i.

Every split graph G_i, for $i \in \{1, \dots, k\}$, comes together with the respective quotient map $q_i : V(G) \to V(G_i)$, that is formed by the repeated composition of the quotient maps that we used in order to arrive at G_i. More precisely, let G' be one of the split graphs in $\{G_1, \dots, G_k\}$. Then there is a sequence $C_1, \dots, C_t$ of 4-cuts from $\mathcal{C}$, for a $t \geq 1$, and a sequence $G'_0, \dots, G'_t$ of graphs, such that $G'_0 = G$, $G'_t = G'$, and G'_i is derived from the splitting of G'_{i-1} according to C_i, for every $i \in \{1, \dots, t\}$. Then, according to Definition 5.7, we get a quotient map $q'_i : V(G'_{i-1}) \to V(G'_i)$ for every $i \in \{1, \dots, t\}$, that corresponds to the splitting of G'_{i-1} into G'_i according to C_i. Then the composition $q'_t \circ \dots \circ q'_1$ is the quotient map from $V(G)$ to $V(G')$.

Notice that the split graphs that we get in Definition 5.8 from a parallel family $\mathcal{C}$ of 4-cuts depend on the order in which we use the 4-cuts from $\mathcal{C}$ in order to perform the splittings. However, this order is irrelevant if we only care about deriving the atoms of $\mathcal{C}$, as shown in the following.

Lemma 5.24. *Let G be a connected graph, let $\mathcal{C}$ be a parallel family of 4-cuts of G, and let $G_1, \dots, G_k$ be the split graphs that we get from G by splitting it according to the 4-cuts from $\mathcal{C}$ (in any order). Then atoms($\mathcal{C}$) $= \{G_1^o, \dots, G_k^o\}$.*

Proof. First, we note that when we use a 4-cut in order to split a graph G into two graphs G_X and G_Y, we have that $\{G_X^o, G_Y^o\}$ is a partition of $V(G)$. Thus, since the collection of graphs $\{G_1, \dots, G_k\}$ is formed by repeated splittings of G, we have that $\{G_1^o, \dots, G_k^o\}$ is a partition of $V(G)$.

Now let x and y be two vertices of G that belong to different sets in *atoms*($\mathcal{C}$). Then there is a 4-cut $C \in \mathcal{C}$ that separates x and y. Thus, we may consider the first 4-cut $C \in \mathcal{C}$ that we used for the splittings and has the property that it separates x and y. Then there is a sequence $C_1, \dots, C_t$ of 4-cuts from $\mathcal{C}$, with $t \geq 0$, that were successively used in order to split G, until we arrived at a split graph G' with the property $x, y \in V(G')$, and it was time to split G' using C. (We allow $t = 0$, because this corresponds to the case that C is the first 4-cut from $\mathcal{C}$ that was used in order to split G.) Let G_X and G_Y be the two split graphs that we get by splitting G' according to C (more precisely: according to the image of C in G' through the quotient map). Then, since C is a 4-cut of G that separates x and y, as a consequence of Lemma 5.23 we have that (the image of) C separates x and y in G'. Thus, w.l.o.g., we may assume that x is a vertex of G_X, but not of G_Y, and y is a vertex of G_Y, but not of G_X. Then, we have that the graph in $G_1, \dots, G_k$ that contains x is either G_X, or it is derived by further splitting G_X. Similarly, the graph in $G_1, \dots, G_k$ that contains y is either G_Y, or it is derived by further splitting G_Y. This implies that x and y belong to different sets from $\{G_1^o, \dots, G_k^o\}$.

Conversely, let x and y be two vertices that belong to different sets from $\{G_1^o, \dots, G_k^o\}$. Thus, there is a sequence $C_1, \dots, C_t$ of 4-cuts from $\mathcal{C}$, with $t \geq 1$, that led to the separation of x and y into different split graphs. More precisely, there is a sequence $C_1, \dots, C_t$ of 4-cuts from $\mathcal{C}$, with $t \geq 1$, and a sequence of graphs $G_0, G_1, \dots, G_t$, such that: (1) $G_0 = G$, (2) G_i was derived from the splitting of G_{i-1} according to C_i, for every $i \in \{1, \dots, t\}$, and (3) G_{t-1} contains both x and y, but G_t contains only one of x and y. Due to (3), we may assume w.l.o.g. that $x \in V(G_t)$ and $y \notin V(G_t)$. We will show that C_t separates x and y in G. Let q_i be the quotient map from $V(G_{i-1})$ to $V(G_i)$, for every $i \in \{1, \dots, t\}$, and let q_0 be the identity map on $V(G)$. Thus, by (2) we have that $q_{i-1}(\dots q_0(C_i) \dots)$ is a 4-cut of G_{i-1}, for every $i \in \{1, \dots, t\}$. Let $C'_t = q_{t-1}(\dots q_0(C_t) \dots)$, and let X be the side of C'_t in G_{t-1} from which G_t is derived.

Then, by a repeated application of Lemma 5.23 we get that $X' = q_1^{-1}(\dots q_{t-1}^{-1}(X)\dots)$ is one of the sides of C_t in G. Then, since $x \in X$, we have $x \in X'$. (Because the inverses of the quotient mappings maintain the vertices from $V(G)$.) Since $y \notin V(G_t)$, we have $y \notin X$. We claim that $y \notin X'$. To see this, assume the contrary. Then, since all graphs $G_0, \dots, G_{t-1}$ contain y, we have that the only vertex $z \in V(G_{t-1})$ that satisfies $q_1^{-1}(\dots q_{t-1}^{-1}(z)\dots) = y$ is $z = y$. But then, since $y \in X' = q_1^{-1}(\dots q_{t-1}^{-1}(X)\dots)$, this implies that $y \in X$, a contradiction. This shows that $y \notin X'$. We conclude that C_t separates x and y in G, and therefore x and y belong to different sets in *atoms*($\mathcal{C}$). $\square$

Thus, in order to compute the atoms of $\mathcal{C}$, it is sufficient to split G according to the 4-cuts in $\mathcal{C}$, and then collect the sets of ordinary vertices of the split graphs. In order to implement this idea efficiently, as in [36] we have to take care of three things. First, given a 4-cut C in $\mathcal{C}$, we need to know the distribution of the endpoints of the edges of C in the connected components of $G \setminus C$. We want to achieve this without explicitly computing the connected components of $G \setminus C$. Second, for every 4-cut that we process, we have to be able to determine the split graph that contains it. And third, since every splitting removes the edges of a 4-cut and substitutes them with new auxiliary edges, now given a new 4-cut C for splitting, we must know if some of its edges correspond to auxiliary edges of the split graph (so that we have to remove its auxiliary counterparts, and not the original edges of C). We solve these problems by locating the 4-cuts from $\mathcal{C}$ on a DFS-tree rooted at r, and by processing them in increasing order w.r.t. their r-size. By locating a 4-cut C on the DFS-tree we can determine easily in constant time how the endpoints of the edges of C are separated in the connected components of $G \setminus C$, according to Lemma 5.22. And by processing the 4-cuts from $\mathcal{C}$ in increasing order w.r.t. their r-size, we can be certain that whenever we process a 4-cut, this essentially lies within the split graph that contains r, as shown in the following.

Lemma 5.25. *Let G be a connected graph, let r be a vertex of G, and let $\mathcal{C}$ be a parallel family of 4-cuts of G. Let C be a 4-cut of G that is not in $\mathcal{C}$, such that C is parallel with every 4-cut in $\mathcal{C}$, and the r-size of C is at least as great as the maximum r-size of all 4-cuts in $\mathcal{C}$. Suppose that we split G according to the 4-cuts in $\mathcal{C}$, and let (G', q') be the split graph that contains r, together with the respective quotient map. Then $q'(C)$ is a 4-cut of G'.*

Proof. Let $(G_1, q_1), \dots, (G_k, q_k)$ be all the split graphs, together with the respective quotient maps, that we get after splitting G according to the 4-cuts in $\mathcal{C}$. (We note

that q_i is a map from G to G_i, for every $i \in \{1,\dots,k\}$.) Then it is a consequence of Lemma 5.23 that there is a unique $(G',q') \in \{(G_1,q_1),\dots,(G_k,q_k)\}$ such that $q'(C)$ is a 4-cut of G' (because, in all other cases, we have that the image of C through a quotient map contains at least one self-loop). By construction of the split graphs, we have that $\{V(G_i) \cap V(G) \mid i \in \{1,\dots,k\}\}$ is a partition of $V(G)$. Thus, there is only one split graph that contains r. We will show that G' is the graph that contains r. To do this, we will use induction on the number of splittings that we had to perform in order to reach the split graph in which C is mapped as a 4-cut.

As the base step of our induction, we can consider the case that no splittings took place at all. In this case, we let the "quotient" map q' be the identity map on $V(G)$. Then, it is obviously true that $q'(C)$ is a 4-cut of G, and G is the "split" graph that contains r. Now let us assume that we have performed t consecutive splittings, for $t \geq 0$, which resulted in a graph G_0 with quotient map q_0, with the property that $r \in V(G_0)$ and $q_0(C)$ is a 4-cut of G_0. Now suppose that we split G_0 once more according to a 4-cut $C' \in \mathcal{C}$. Thus, we have that $q_0(C')$ is a 4-cut of G_0, and let X and Y be the two sides of $q_0(C')$. Let (G_X,q_X) and (G_Y,q_Y) be the resulting split graphs, together with the respective quotient maps. Then there are two auxiliary vertices $x_{C'}$ and $y_{C'}$, such that $V(G_X) = X \cup \{y_{C'}\}$, $V(G_Y) = Y \cup \{x_{C'}\}$, $q_X(Y) = \{y_{C'}\}$ and $q_Y(X) = \{x_{C'}\}$. We may assume w.l.o.g. that $r \in X$. Thus, we have $q_Y(r) = x_{C'}$.

Since $q_0(C)$ is a 4-cut of G_0 that is parallel with $q_0(C')$, by Lemma 5.23 we have one of $q_X(q_0(C))$ and $q_Y(q_0(C))$ contains a self-loop, and the other is a set of four edges. So let us suppose, for the sake of contradiction, that $q_X(q_0(C))$ contains a self-loop. Then, Lemma 5.23 implies that $q_Y(q_0(C))$ is a 4-cut of G_Y. Let X' and Y' be the two sides of $q_Y(q_0(C))$ (in G_Y). Then, Lemma 5.23 implies that $q_Y^{-1}(X')$ and $q_Y^{-1}(Y')$ are the two sides of $q_0(C)$ (in G_0). We may assume w.l.o.g. that $q_Y^{-1}(X')$ is the side of $q_0(C)$ that contains r. This implies that $x_{C'} \in X'$, and therefore $q_Y^{-1}(X')$ contains X. Since C and C' are distinct 4-cuts of G, Lemma 5.23 implies that $q_0(C)$ and $q_0(C')$ are distinct 4-cuts of G_0. Thus, we cannot have $q_Y^{-1}(X') = X$, and therefore we have that $q_Y^{-1}(X')$ contains X as a proper subset. This implies that $q_Y^{-1}(Y')$ is a proper subset of Y. By Lemma 5.23 we have that $q_0^{-1}(X)$ and $q_0^{-1}(Y)$ are the two sides of C' (in G). Thus, since $r \in X$, we have that $r \in q_0^{-1}(X)$, and therefore the r-size of C' is $|q_0^{-1}(Y)|$. Similarly, by Lemma 5.23 we have that $q_0^{-1}(q_Y^{-1}(X'))$ and $q_0^{-1}(q_Y^{-1}(Y'))$ are the two sides of C (in G). Thus, since $r \in q_Y^{-1}(X')$, we have that $r \in q_0^{-1}(q_Y^{-1}(X'))$, and therefore the r-size of C is $|q_0^{-1}(q_Y^{-1}(Y'))|$. Since $q_Y^{-1}(Y')$ is a proper subset of

Y, we have that $q_0^{-1}(q_Y^{-1}(Y'))$ is a proper subset of $q_0^{-1}(Y)$ (because q_0 is a surjective function). This implies that $|q_0^{-1}(q_Y^{-1}(Y'))| < |q_0^{-1}(Y)|$, in contradiction to the fact that the r-size of C is at least as great as the r-size of C'. This shows that $q_X(q_0(C))$ does not contain a self-loop. Thus, by Lemma 5.23 we have that $q_X(q_0(C))$ is a 4-cut of G_X. This shows that, after $t+1$ splittings, C is mapped as a 4-cut to the split graph that contains r. Thus, the lemma follows inductively. □

Thus, if we process the 4-cuts from $\mathcal{C}$ in increasing order w.r.t. their r-size, then we can be certain that every 4-cut $C \in \mathcal{C}$ that we process lies within the split graph G' that contains r. Therefore, it is sufficient to maintain only the quotient map to G'. We use v' to denote the image of a vertex $v \in V(G)$ to G', and we let C' denote the translation of C within G' through the quotient map. Thus, whenever we process a 4-cut $C \in \mathcal{C}$, we translate the endpoints of every edge $(x,y) \in C$ to their corresponding vertices x' and y' in G'. Then we delete (x',y') from G', and we substitute it with two auxiliary edges (x',y_C) and (x_C,y') (i.e., we create one copy of (x',y') for each of the connected components of $G' \setminus C$). Let (x_C,y') be the copy of (x',y') that is contained in the connected component of $G' \setminus C'$ that contains r. Then we update the pointer of x to G' as $x' \leftarrow x_C$.

The procedure for computing *atoms*($\mathcal{C}$) is shown in Algorithm 17. The proof of correctness and linear complexity is given in Proposition 5.5. As a corollary of this method for computing *atoms*($\mathcal{C}$), we can construct in linear time a data structure that we can use in order to answer queries of the form "given two vertices x and y, determine whether x and y are separated by a 4-cut in $\mathcal{C}$, and, if yes, report a 4-cut in $\mathcal{C}$ that separates them" in constant time. This is proved in Corollary 5.10. We note that these results are essentially independent of the fact that we consider 4-cuts, and so they generalize to any parallel family of cuts (of various cardinalities), provided that there is a fixed upped bound on their number of edges.

Proposition 5.5. *Algorithm 17 correctly computes the atoms of a parallel family of 4-cuts $\mathcal{C}$. Furthermore, it has a linear-time implementation.*

Proof. By Lemma 5.24, it is sufficient to split the graph according to the 4-cuts in $\mathcal{C}$, and then return the sets of ordinary vertices of the split graphs. Since the vertex-sets of the split graphs are pairwise disjoint, we can consider the collection of all of them as a single graph G'. Thus, we can equivalently compute the connected components

Algorithm 17: Compute $atoms(\mathcal{C})$ of a parallel family of 4-cuts $\mathcal{C}$

```
1  sort the 4-cuts in C in increasing order w.r.t. their r-size
2  foreach vertex v do
       // initialize the pointers of the vertices to the split graph that
          contains r
3      set v' ← v
4  end
5  let G' ← G // we maintain throughout the collection of the split graphs
       as a single graph G'
6  foreach C ∈ C do
7      let C = {(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)}
8      determine the endpoints of the edges in C that are in the connected
         component of G \ C that contains r; let those endpoints be y_1, y_2, y_3, y_4
9      remove the edges (x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3), (x'_4, y'_4) from G'
10     insert two new vertices x_C and y_C to G'
11     insert the edges (x'_1, y_C), (x'_2, y_C), (x'_3, y_C), (x'_4, y_C) and
         (x_C, y'_1), (x_C, y'_2), (x_C, y'_3), (x_C, y'_4) to G'
12     set x'_1 ← x_C, x'_2 ← x_C, x'_3 ← x_C, x'_4 ← x_C
13 end
14 compute the connected components S_1, ..., S_k of G'
15 foreach i ∈ {1, ..., k} do
16     let S'_i be the set of the ordinary vertices in S_i
17 end
18 return S'_1, ..., S'_k
```

of G', and then return the sets of ordinary vertices of its connected components (see Lines 14 and 16).

Let r be a vertex of G. Then Lemma 5.25 implies that if we use the 4-cuts from $\mathcal{C}$ in increasing order w.r.t. their r-size for the splittings, then, every time we pick a 4-cut for the splitting, this is mapped as a 4-cut into the split graph that contains r. Thus, it is sufficient to maintain throughout only the quotient map from $V(G)$ to the split graph that contains r. We denote this as v', for every vertex $v \in V(G)$.

Now let $C = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$ be a 4-cut in $\mathcal{C}$, and let us as-

sume w.l.o.g. that y_1, y_2, y_3, y_4 are the endpoints of the edges in C that lie in the connected component of $G \setminus C$ that contains r. Then we have that $C' = \{(x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3), (x'_4, y'_4)\}$ is a 4-cut of the split graph that contains r, and Lemma 5.23(b) implies that y'_1, y'_2, y'_3, y'_4 are the endpoints of the edges in C' that lie in the same side of C' as r. Now, in order to perform the splitting induced by C, we introduce two new auxiliary vertices x_C and y_C to G', we delete the edges of C' from G', we introduce the new edges $(x'_1, y_C), (x'_2, y_C), (x'_3, y_C), (x'_4, y_C)$ and $(x_C, y'_1), (x_C, y'_2), (x_C, y'_3), (x_C, y'_4)$ to G', and we update the quotient map to the split graph that contains r as $x'_1 \leftarrow x_C, x'_2 \leftarrow x_C, x'_3 \leftarrow x_C, x'_4 \leftarrow x_C$. (The images of y_1, y_2, y_3, y_4 into the split graph that contains r have not changed.) These are precisely the operations that take place during the processing of every 4-cut $C \in \mathcal{C}$ during the course of the **for** loop in Line 6. (We assume that the **for** loop in Line 6 processes the 4-cuts in $\mathcal{C}$ in increasing order w.r.t. their r-size, according to the sorting that took place in Line 1.)

Now it remains to argue about the complexity of Algorithm 17. First, after the linear-time preprocessing described in Lemma 5.22, we have that the r-sizes of the 4-cuts in $\mathcal{C}$ can be computed in $O(|\mathcal{C}|)$ time in total. Since $\mathcal{C}$ is a parallel family of 4-cuts of G, it contains $O(n)$ 4-cuts (see, e.g., [24]). Then, we can sort the 4-cuts in $\mathcal{C}$ in increasing order w.r.t. their r-size using bucket-sort. Thus, Line 1 can be performed in linear time. After the preprocessing described in Lemma 5.22, we can also compute in constant time, for every 4-cut $C \in \mathcal{C}$, the endpoints of the edges in C that lie in the same connected component of $G \setminus C$ that contains r. Thus, Line 8 incurs total cost $O(n)$. In order to perform efficiently the deletions and insertions of edges in Lines 9 and 11, respectively, we just process them in an off-line manner. Thus, we first collect every `insert_edge`(x, y) query as a triple $(x, y, +)$, and we collect every `delete_edge`(x, y) query as $(x, y, -)$. Furthermore, we also collect every edge (x, y) of the original graph G as a triple $(x, y, +)$; this corresponds to the initialization in Line 5. Then we sort all those triples lexicographically (giving, e.g., priority to $+$). Since their total number is $O(m + |\mathcal{C}|) = O(m + n)$, we can perform this sorting in $O(m + n)$ time with bucket-sort. Let L be the resulting list of triples. Then, for every maximal segment of L that consists of triples whose first two components coincide – and so these correspond to insertions and deletions of the same edge (x, y) –, we just determine whether the number of "pluses" dominates the number of "minuses" for (x, y). If these values are equal, then we do not include the edge (x, y) in the final

graph G'. Otherwise, we create as many copies of (x,y) for G', as is the difference between the number of pluses and the number of minuses for (x,y). Thus, we can create G' in linear time in total. Therefore, the computation in Line 14 takes $O(m+n)$ time. Then, the **for** loop in Line 15 takes $O(|V(G')|) = O(n)$ time, because we can easily check whether a vertex of G' is auxiliary (by maintaining a bit that signifies it). We conclude that Algorithm 17 has a linear-time implementation. □

Corollary 5.10. *Let $\mathcal{C}$ be a parallel family of 4-cuts of G. Then we can construct in linear time a data structure of $O(n)$ size that we can use in order to answer queries of the form "given two vertices x and y, determine if x and y are separated by a 4-cut from $\mathcal{C}$, and, if yes, report a 4-cut from $\mathcal{C}$ that separates them" in constant time.*

Proof. This is an easy consequence of the method that we use in order to compute the atoms of $\mathcal{C}$. First, by using Algorithm 17, we can compute the split graphs according to $\mathcal{C}$ (w.r.t. an ordering of the 4-cuts from $\mathcal{C}$ in increasing order w.r.t. their r-size). By Proposition 5.5, all these split graphs can be computed in linear time in total. Since the split graphs are pairwise vertex-disjoint, we can consider the collection of them as a graph G' (whose connected components are the split graphs). Let C be a 4-cut from $\mathcal{C}$, and let x_C and y_C be the auxiliary vertices that were introduced due to the splitting according to C. Then, we insert an artificial edge (x_C, y_C) to G'. We perform this for all 4-cuts from $\mathcal{C}$, and let G'' denote the resulting graph. (Notice that $|V(G'')| = O(n)$ and $|E(G'')| = O(m)$.) Then, for every 4-cut $C \in \mathcal{C}$, we have that (x_C, y_C) is a bridge of G'', whose sides correspond to the connected components of $G \setminus C$ (if we keep the ordinary vertices from each side). Notice that the 2-edge-connected components of G'' are in a bijective correspondence with the split graphs of G. Thus, our construction is as follows. First, we compute the tree T of the 2-edge-connected components of G''. Thus, the nodes of T are the split graphs of G, and the edges of T are the new artificial edges of the form (x_C, y_C) that we have created, for every 4-cut $C \in \mathcal{C}$. Finally, we perform a DFS on T, and we keep the parent pointers p, the parent edges, the values *ND* for all vertices, and a pointer from every parent edge to the 4-cut from $\mathcal{C}$ that corresponds to it, and reversely. It is easy to see that this whole construction can be completed in linear time and takes $O(n)$ space.

Now, given two vertices x and y, and a query that asks for a 4-cut from $\mathcal{C}$ that separates x and y, we first retrieve the nodes u and v of T that contain x and y, respectively. If these coincide, then there is no 4-cut from $\mathcal{C}$ that separates x and y.

Otherwise, we first check whether u and v are related as ancestor and descendant. If that is the case, let us assume w.l.o.g. that u is a descendant of v. Then, the 4-cut from $\mathcal{C}$ that corresponds to the parent edge $(u, p(u))$ is a 4-cut that separates x and y. Otherwise, if u and v are not related as ancestor and descendant, we may assume w.l.o.g. that u is not the root of T. Then, the 4-cut from $\mathcal{C}$ that corresponds to the parent edge $(u, p(u))$ is a 4-cut that separates x and y. Thus, we can see that in $O(1)$ time we can determine a 4-cut from $\mathcal{C}$ that separates x and y, or report that no such 4-cut exists.

□

As a concluding remark, we note that Proposition 5.5 and Corollary 5.10 hold for general parallel families of cuts (even of mixed cardinalities), provided that there is a fixed upper bound on their cardinality. This is because, in all the arguments in this section, we did not rely in an essential way on the fact that we consider collections of 4-cuts. Even Lemma 5.22, that concerns the computation of the r-size, and the determining of the endpoints of the edges of a cut that lie in the same connected component as r, holds for general collections of cuts (of bounded cardinality). This is an easy combinatorial problem, that relies on the fact that, when we traverse a tree-path, every time that we cross an edge that participates in a cut, we move to the other side of the cut.

5.4 Computing the 5-edge-connected components

5.4.1 Overview

In this section we present the linear-time algorithm for computing the 5-edge-connected components of a 3-edge-connected graph G. On a high level, the idea is to collect enough 4-cuts of G so that: (1) the collection of those 4-cuts is sufficient to provide the partition of G into its 5-edge-connected components, and (2) there is a linear-time algorithm for computing the partition of $V(G)$ induced by this collection.

Solving (1) and (2) simultaneously is a very complex problem. First of all, computing a collection of 4-cuts of G that has $O(|V(G)|)$ size and is enough in order to provide the 5-edge-connected components, is in itself a highly non-trivial task. This is because the number of all 4-cuts of G can be as high as $\Omega(|V(G)|^2)$. Thus, we

must discover a structure in G that allows us to select enough 4-cuts for our purpose. This seems very demanding, because, to the best of our knowledge, no linear-time algorithm exists even for checking the existence of a 4-cut. Furthermore, even if we had such a collection of 4-cuts, we are faced with problem (2), which is basically to compute the atoms induced by this collection. We do not know how to do this in linear time for general collections of 4-cuts. However, if the collection is a parallel family of 4-cuts, then Proposition 5.5 establishes that we can compute its atoms in linear time.

Our solution to both (1) and (2) is as follows. First, we solve (1) indirectly, by computing a complete collection $\mathcal{C}$ of 4-cuts of G that has size $O(|V(G)|)$. We note that it is not even clear why such a collection should exist. A large part of this work is devoted to establishing Theorem 5.3, which guarantees the existence of such a collection, and also that it can be computed in linear time. Now, provided with $\mathcal{C}$, we basically have a compact representation of all 4-cuts of G. However, we cannot expand all the implicating sequences of $\mathcal{C}$ in order to derive all 4-cuts of G, as this could demand $\Omega(|V(G)|^2)$ time. Instead, we apply Algorithm 16 on $\mathcal{C}$, which implicitly expands all the implicating sequences of $\mathcal{C}$, and packs them into a set $\{\mathcal{F}_1, \dots, \mathcal{F}_k\}$ of collections of pairs of edges, that have the property that (i) they generate 4-cuts implied by $\mathcal{C}$, and (ii) every 4-cut implied by $\mathcal{C}$ is generated by at least one such collection.

Now, a tempting idea would be to compute all $\mathcal{C}_i$-minimal 4-cuts, for every $i \in \{1, \dots, k\}$, where $\mathcal{C}_i$ is the collection of 4-cuts generated by $\mathcal{F}_i$. This is because the collection of all $\mathcal{C}_i$-minimal 4-cuts is a parallel family of 4-cuts, that provides the same atoms as $\mathcal{C}_i$ (Corollary 5.7). According to Proposition 5.6, we can indeed compute the $\mathcal{C}_i$-minimal 4-cuts, for all $i \in \{1, \dots, k\}$ such that $\mathcal{C}_i$ is a cyclic family of 4-cuts, in linear time in total. However, we cannot use Algorithm 17 in order to compute the atoms provided by the $\mathcal{C}_i$-minimal 4-cuts, for every $i \in \{1, \dots, k\}$ separately, because this would take $O(k|V(G)|)$ time in total, and k can be as large as $\Omega(|V(G)|)$. On the other hand, we cannot compute the atoms provided by all $\mathcal{C}_i$-minimal 4-cuts, for all $i \in \{1, \dots, k\}$ simultaneously, because there is no guaranteee that this is a parallel family of 4-cuts. Thus, we have to carefully select enough $\mathcal{C}_i$-minimal 4-cuts, for every $i \in \{1, \dots, k\}$, so that (1) and (2) are satisfied simultaneously. This is still a challenging task. A first step towards resolving it is to keep only the essential $\mathcal{C}_i$-minimal 4-cuts, for every $i \in \{1, \dots, k\}$. This is enough in order to provide the

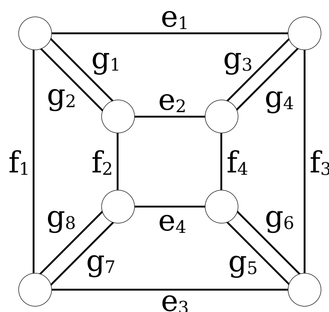


Figure 5.15: This is a 3-edge-connected graph with 4-cuts $C_1 = \{e_1, e_2, e_3, e_4\}$, $C_2 = \{f_1, f_2, f_3, f_4\}$, $D_1 = \{e_1, f_1, g_1, g_2\}$, $D_2 = \{e_2, f_2, g_1, g_2\}$, $D_3 = \{e_1, f_1, e_2, f_2\}$, $E_1 = \{e_1, f_3, g_3, g_4\}$, $E_2 = \{e_2, f_4, g_3, g_4\}$, $E_3 = \{e_1, f_3, e_2, f_4\}$, $F_1 = \{e_3, f_3, g_5, g_6\}$, $F_2 = \{e_4, f_4, g_5, g_6\}$, $F_3 = \{e_3, f_3, e_4, f_4\}$, $G_1 = \{e_3, f_1, g_7, g_8\}$, $G_2 = \{e_4, f_2, g_7, g_8\}$ and $G_3 = \{e_3, f_1, e_4, f_2\}$. Thus, it is not difficult to see that $\mathcal{C} = \{C_1, C_2, D_1, D_2, E_1, E_2, F_1, F_2, G_1, G_2\}$ is a complete collection of 4-cuts of this graph. If we apply Algorithm 16 on $\mathcal{C}$, we will get as a result the collections of pairs of edges $\mathcal{F}_1 = \{\{e_1, e_2\}, \{e_3, e_4\}\}$, $\mathcal{F}_2 = \{\{e_1, e_3\}, \{e_2, e_4\}\}$, $\mathcal{F}_3 = \{\{e_1, e_4\}, \{e_2, e_3\}\}$, $\mathcal{F}_4 = \{\{f_1, f_2\}, \{f_3, f_4\}\}$, $\mathcal{F}_5 = \{\{f_1, f_3\}, \{f_2, f_4\}\}$, $\mathcal{F}_6 = \{\{f_1, f_4\}, \{f_2, f_3\}\}$, $\mathcal{F}_7 = \{\{e_1, f_1\}, \{e_2, f_2\}, \{g_1, g_2\}\}$, $\mathcal{F}_8 = \{\{e_1, f_3\}, \{e_2, f_4\}, \{g_3, g_4\}\}$, $\mathcal{F}_9 = \{\{e_3, f_3\}, \{e_4, f_4\}, \{g_5, g_6\}\}$ and $\mathcal{F}_{10} = \{e_3, f_1\}, \{e_4, f_2\}, \{g_7, g_8\}\}$. Notice that $\{\{e_1, e_2\}, \{e_3, e_4\}, \{f_1, f_2\}, \{f_3, f_4\}\}$ is a collection of pairs of edges that generates a cyclic family of 4-cuts of this graph, that includes C_1 and C_2. Thus, C_1 and C_2 are not isolated 4-cuts. Therefore, since all three partitions into pairs of edges of C_1 and C_2 are returned by Algorithm 16 on input $\mathcal{C}$, we have that C_1 and C_2 are quasi $\mathcal{C}$-isolated 4-cuts. Notice that C_1 and C_2 are essential and cross.

5-edge-connected components. Furthermore, according to Lemma 5.20, this provides a parallel family of 4-cuts. However, so far we have overlooked the fact that there may be some collections of pairs of edges in $\{\mathcal{F}_1, \dots, \mathcal{F}_k\}$ that have size 2, and yet can be expanded into larger collections of pairs of edges that generate 4-cuts implied by $\mathcal{C}$. Then, the 4-cuts that are generated by such collections may cross with other such 4-cuts. Furthermore, this can be true even if those 4-cuts are essential. (See Figure 5.15 for an example.) In this case Lemma 5.21 is very useful, because it establishes that we can drop from our consideration those 4-cuts. Thus, it is sufficient to consider only the essential $\mathcal{C}$-isolated 4-cuts. By Corollary 5.6, these have the property that they are parallel with every other essential 4-cut.

This is the general idea for computing the 5-edge-connected components in linear

time. Our result is summarized in Proposition 5.9. In order to establish this proposition, we need to provide efficient algorithms for computing the minimal 4-cuts and the essential isolated 4-cuts. We perform these tasks in Sections 5.4.2 and 5.4.3, respectively. In Section 5.4.4 we describe the procedure for achieving both (1) and (2), given a complete collection of 4-cuts.

Throughout this chapter, we assume that G is a 3-edge-connected graph, and all graph-related elements refer to G. Also, we assume that we have performed the linear-time preprocessings that are described in Lemma 5.22 and Proposition 5.4. (These are for reporting the r-size and for testing the essentiality of 4-cuts in constant time.)

5.4.2 Computing the minimal 4-cuts

Let $\mathcal{C}$ be a collection of 4-cuts of G, and let $\mathcal{F}$ be a collection of pairs of edges that is returned by Algorithm 16 on input $\mathcal{C}$. By Proposition 5.3, we have that $\mathcal{F}$ generates a collection $\mathcal{C}'$ of 4-cuts implied by $\mathcal{C}$. Suppose that $\mathcal{C}'$ is a cyclic family of 4-cuts. Then we want to find all the $\mathcal{C}'$-minimal 4-cuts. Let us recall precisely what this means. Let $\mathcal{F} = \{p_1, \dots, p_k\}$, where $k \geq 3$. Then, since $\mathcal{F}$ generates a cyclic family of 4-cuts, we may assume w.l.o.g. that there is a partition $\{X_1, \dots, X_k\}$ of $V(G)$ such that $G[X_i]$ is connected for every $i \in \{1, \dots, k\}$, and $E[X_i, X_{i+_k 1}] = p_i$ for every $i \in \{1, \dots, k\}$. Then the collection of all $\mathcal{C}'$-minimal 4-cuts is $\{p_i \cup p_{i+_k 1} \mid i \in \{1, \dots, k\}\}$. Of course, the problem is that we do not receive the collection $\mathcal{F}$ in such an orderly fashion, and the number of all 4-cuts generated by $\mathcal{F}$ is $\Theta(k^2)$, which can be as large as $\Omega(n^2)$.

We propose the following method to compute the $\mathcal{C}'$-minimal 4-cuts. Suppose that we have fixed a vertex r, and assume w.l.o.g. that $r \in X_1$. If we knew at least one of the two pairs of edges in $\mathcal{F}$ that are incident to X_1 (i.e., either p_1 or p_k), then it would be easy to find all the $\mathcal{C}'$-minimal 4-cuts. Specifically, suppose that we know that p_1 is incident to the corner of $\mathcal{C}'$ that contains r. Then we have that the 4-cuts $p_1 \cup p_2, p_1 \cup p_3, \dots, p_1 \cup p_k$ are sorted in increasing order w.r.t. their r-size. Thus, if we have sorted the collection of 4-cuts $\{p_1 \cup p_i \mid i \in \{2, \dots, k\}\}$ in increasing order w.r.t. the r-size, then we can retrieve the sequence $p_1, p_2, p_3, \dots, p_k$, which is enough to provide the $\mathcal{C}'$-minimal 4-cuts (i.e., by collecting the union of every pair of consecutive elements in this sequence, plus $p_k \cup p_1$).

Thus, the problem is how to identify one of the two pairs of edges in $\mathcal{F}$ that are incident to the corner of $\mathcal{C}'$ that contains r. To do this, we start with any pair of edges

$p \in \mathcal{F}$. Then, it is easy to see that the 4-cut $p \cup p_i$ in $\{p \cup p_i \mid i \in \{1, \ldots, k\}$ and $p_i \neq p\}$ with the *maximum* r-size has the property that p_i is a pair of edges incident to X_1 (i.e., i is either 1 or k).

If we put those two ideas together, then we can see that Algorithm 18 computes, simultaneously, all the $\mathcal{C}_1$-, ..., $\mathcal{C}_t$-minimal 4-cuts, for every set of collections of pairs of edges $\mathcal{F}_1, \ldots, \mathcal{F}_t$ that generate the cyclic families of 4-cuts $\mathcal{C}_1, \ldots, \mathcal{C}_t$, respectively. The analysis of Algorithm 18, as well as its proof of correctness, is given in Proposition 5.6.

Proposition 5.6. *Let $\mathcal{F}_1, \ldots, \mathcal{F}_t$ be a set of collections of pairs of edges that generate the cyclic families of 4-cuts $\mathcal{C}_1, \ldots, \mathcal{C}_t$. Then, the output of Algorithm 18 on input $\mathcal{F}_1, \ldots, \mathcal{F}_t$ is the collection of all $\mathcal{C}_1$-, ..., $\mathcal{C}_t$-minimal 4-cuts. The running time of Algorithm 18 is $O(n + |\mathcal{F}_1| + \cdots + |\mathcal{F}_t|)$.*

Proof. Let $\mathcal{F} = \{p_1, \ldots, p_k\}$ be one of the collections of pairs of edges in $\{\mathcal{F}_1, \ldots, \mathcal{F}_t\}$. Since $\mathcal{F}$ generates a cyclic family $\mathcal{C}$ of 4-cuts, we may assume w.l.o.g. that there is a partition $\{X_1, \ldots, X_k\}$ of $V(G)$, such that $G[X_i]$ is connected for every $i \in \{1, \ldots, k\}$, and $E[X_i, X_{i+_k 1}] = p_i$ for every $i \in \{1, \ldots, k\}$. Then, the $\mathcal{C}$-minimal 4-cuts are given by $\{p_i \cup p_{i+_k 1} \mid i \in \{1, \ldots, k\}\}$. Let r be any fixed vertex, and let us assume w.l.o.g. that $r \in X_1$.

Let p be any pair of edges in $\mathcal{F}$. Thus, there is an $i \in \{1, \ldots, k\}$ such that $p = p_i$. Then, for every $j \in \{1, \ldots, i-1\}$, the two sides of the 4-cut $p_i \cup p_j$ are given by $X_{j+1} \cup \cdots \cup X_i$ and $X_{i+_k 1} \cup \cdots \cup X_j$. Thus, the r-size of $p_i \cup p_j$ is given by $|X_{j+1}| + \cdots + |X_i|$. Also, for every $j \in \{i+1, \ldots, k\}$, the two sides of the 4-cut $p_i \cup p_j$ are given by $X_{i+1} \cup \cdots \cup X_j$ and $X_{j+_k 1} \cup \cdots \cup X_i$. Thus, the r-size of $p_i \cup p_j$ is given by $|X_{i+1}| + \cdots + |X_j|$. Thus, if $j \in \{1, \ldots, i-1\}$, then the r-size of $p_i \cup p_j$ is maximized for $j = 1$. And if $j \in \{i+1, \ldots, k\}$, then the r-size of $p_i \cup p_j$ is maximized for $j = k$. This shows that, if we consider the collection of 4-cuts of the form $\{p_i \cup p_j \mid j \in \{1, \ldots, k\} \setminus \{i\}\}$, then the 4-cut with the maximum r-size in this collection is either $p_i \cup p_1$ or $p_i \cup p_k$. In either case, then, we receive one of the two pairs of edges in $\mathcal{F}$ that are incident to the corner of $\mathcal{C}$ that contains r. This shows that the **for** loop in Line 3 correctly computes a pair of edges $q_i \in \mathcal{F}_i$ that is incident to the corner of $\mathcal{C}_i$ that contains r, for every $i \in \{1, \ldots, t\}$.

Now consider again the collection of pairs of edges $\mathcal{F}$. Suppose that we have determined that p_1 is one of the pairs of edges in $\mathcal{F}$ that is incident to X_1. Then, for every $i \in \{2, \ldots, k\}$, the two sides of $p_1 \cup p_i$ are $X_2 \cup \cdots \cup X_i$ and $X_{i+_k 1} \cup \cdots \cup X_1$. Thus,

Algorithm 18: Compute all the $\mathcal{C}_1$-, ... , $\mathcal{C}_t$-minimal 4-cuts, for a given set of collections of pairs of edges $\mathcal{F}_1, \dots, \mathcal{F}_t$ that generate the cyclic families of 4-cuts $\mathcal{C}_1, \dots, \mathcal{C}_t$, respectively

1. let r be any fixed vertex
2. let q_i be a *null* pointer to a pair of edges in $\mathcal{F}_i$, for every $i \in \{1, \dots, t\}$
3. **foreach** $i \in \{1, \dots, t\}$ **do**
   - // find a pair of edges $q_i \in \mathcal{F}_i$ that is incident to the corner of $\mathcal{C}_i$ that contains r
4.   let p be any pair of edges in $\mathcal{F}_i$
5.   let $max \leftarrow 0$
6.   **foreach** *pair of edges* q *in* $\mathcal{F}_i \setminus \{p\}$ **do**
7.   let *size* be the r-size of the 4-cut $p \cup q$
8.   **if** $size > max$ **then**
9. $q_i \leftarrow q$
10. $size \leftarrow max$
11.   **end**
12.   **end**
13. **end**
14. initialize $\mathcal{P} \leftarrow \emptyset$
15. **foreach** $i \in \{1, \dots, t\}$ **do**
16.   **foreach** $p \in \mathcal{F}_i \setminus \{q_i\}$ **do**
17.   insert the pair $(q_i \cup p, i)$ into $\mathcal{P}$
18.   **end**
19. **end**
20. sort $\mathcal{P}$ in increasing order w.r.t. the r-size of the first component of its elements
21. initialize $\mathcal{M} \leftarrow \emptyset$
22. initialize $p_i \leftarrow q_i$, for every $i \in \{1, \dots, t\}$
23. **foreach** *pair* $(q_i \cup p, i) \in \mathcal{P}$ **do**
24.   insert the 4-cut $p_i \cup p$ into $\mathcal{M}$
25.   set $p_i \leftarrow p$
26. **end**
27. **foreach** $i \in \{1, \dots, t\}$ **do** insert the 4-cut $p_i \cup q_i$ into $\mathcal{M}$
28. **return** $\mathcal{M}$

the r-size of $p_1 \cup p_i$ is $|X_2| + \cdots + |X_i|$. This shows that the 4-cuts $p_1 \cup p_2, \ldots, p_1 \cup p_k$ are sorted in increasing order w.r.t. their r-size. Notice that, if we knew the sequence of pairs of edges $p_1, \ldots, p_k$ then we could collect all $\mathcal{C}$-minimal 4-cuts, by forming the union of every two consecutive pairs of edges in this sequence, plus $p_k \cup p_1$. Thus, the idea is to collect the 4-cuts of the form $\{p_1 \cup p_i \mid i \in \{2, \ldots, k\}\}$, sort them in increasing order w.r.t. their r-size, and then gather the sequence $p_2, \ldots, p_k$ by taking the difference from p_1. (The argument is similar if the pair of edges that was determined to be incident to X_1 is p_k.)

In order to sort the collection $\{p_1 \cup p_i \mid i \in \{2, \ldots, k\}\}$, we use bucket-sort. This takes $O(n)$ time. However, we cannot apply this procedure separately for every collection $\mathcal{F}_i$, because otherwise we will need $\Omega(kn)$ time, and k can be as large as $\Omega(n)$. Thus, we have to collect the 4-cuts from all those collections in a set $\mathcal{P}$, and sort them simultaneously with bucket-sort. In order to retrieve the information for every 4-cut in $\mathcal{P}$ what is the collection of pairs of edges from which it was generated, we index every 4-cut that we put in $\mathcal{P}$ with the index of the collection of pairs of edges from which it was generated. This is implemented in Lines 14 to 20. Since we perform the sorting with bucket-sort, this takes $O(|\mathcal{F}_1| + \cdots + |\mathcal{F}_t| + n)$ time in total. Finally, Lines 21 to 27 implement the idea that we explained above in order to collect all $\mathcal{C}_i$-minimal 4-cuts, for every $i \in \{1, \ldots, t\}$.

It should be clear that the expression $O(|\mathcal{F}_1| + \cdots + |\mathcal{F}_t| + n)$ dominates the total running time. We only note that, given any 4-cut, we can easily compute its r-size in $O(1)$ time, according to the ancestry relation of the tree-edges that are contained in it. This was explained in Section 5.3.1 (see Lemma 5.22). □

5.4.3 Computing the essential isolated 4-cuts

Let $\mathcal{C}$ be a complete collection of 4-cuts of G. We will provide an algorithm that computes all the essential $\mathcal{C}$-isolated 4-cuts. In other words, we want to compute all the essential 4-cuts $C \in \mathcal{C}$ that have the property that there is no collection $\mathcal{F}$ of pairs of edges with $|\mathcal{F}| > 2$ that generates a collection $\mathcal{C}'$ of 4-cuts implied by $\mathcal{C}$ such that $C \in \mathcal{C}'$. By Lemma 5.16, we have that every $\mathcal{C}$-isolated 4-cut C has the property that the three different partitions of C into pairs of edges are returned by Algorithm 16 on input $\mathcal{C}$ $(*)$. However, property $(*)$ is also satisfied by the *quasi* $\mathcal{C}$-isolated 4-cuts. Therefore, given that a 4-cut $C \in \mathcal{C}$ satisfies property $(*)$, we have to be able to

determine whether C is $\mathcal{C}$-isolated or quasi $\mathcal{C}$-isolated.

Since we actually care only about the *essential* 4-cuts, Lemma 5.17 is very useful in this situation. Because Lemma 5.17 shows that every essential quasi $\mathcal{C}$-isolated 4-cut shares a pair of edges with an essential $\mathcal{C}'$-minimal 4-cut, where $\mathcal{C}'$ is a cyclic family of 4-cuts that is generated by a collection of pairs of edges that is returned by Algorithm 16 on input $\mathcal{C}$. The reason that this property is very useful, is that the number of minimal 4-cuts that are extracted from the collections of pairs of edges that are returned by Algorithm 16 is bounded by $O(n)$ (if $|\mathcal{C}| = O(n)$), and therefore the search space for intersections of quasi $\mathcal{C}$-isolated 4-cuts with other 4-cuts implied by $\mathcal{C}$ is conveniently small.

Thus, our strategy for computing the essential $\mathcal{C}$-isolated 4-cuts can be summarized as follows. First, we collect the output $\mathcal{F}_1, \dots, \mathcal{F}_k$ of Algorithm 16 on input $\mathcal{C}$. Then, we find all 4-cuts $C \in \mathcal{C}$ that satisfy property $(*)$: i.e., we collect all 4-cuts $C \in \mathcal{C}$ such that all three partitions of C into pairs of edges are included in $\mathcal{F}_1, \dots, \mathcal{F}_k$. Then, among those 4-cuts, we keep only the essential. Let $\widetilde{\mathcal{C}}$ be the resulting collection. Thus, we have that all the essential $\mathcal{C}$-isolated 4-cuts are contained in $\widetilde{\mathcal{C}}$. However, the problem is that $\widetilde{\mathcal{C}}$ may also contain some quasi $\mathcal{C}$-isolated 4-cuts, which we have to identify in order to discard. For this purpose, we rely on Corollary 5.9 in the following way. We apply Algorithm 18, in order to compute the collection $\mathcal{M}$ of all the essential $\mathcal{C}'$-minimal 4-cuts, for every cyclic family of 4-cuts $\mathcal{C}'$ for which there exists an $i \in \{1, \dots, k\}$ such that $\mathcal{C}'$ is generated by $\mathcal{F}_i$. Then, for every 4-cut $C \in \widetilde{\mathcal{C}}$, and every 4-cut $C' \in \mathcal{M}$, we have to check whether C and C' share a pair of edges.

To perform this efficiently, for every 4-cut $C \in \mathcal{M}$, and every pair of edges $p \subset C$, we create a pair $(p, *)$. Also, for every 4-cut $C \in \widetilde{\mathcal{C}}$, and every pair of edges $p \subset C$, we create a pair (p, C). Now, we collect all those pairs $(p, *)$ and (p, C) into a collection $\mathcal{P}$, which we sort in lexicographic order, giving priority to $*$. Then, we simply check, for every pair (p, C) in $\mathcal{P}$, whether it is preceded by a pair of the form $(p, *)$. If that is the case, then we know that C (which is a 4-cut in $\widetilde{\mathcal{C}}$) shares a pair of edges with a $\mathcal{C}'$-minimal 4-cut, where $\mathcal{C}'$ is the cyclic family of 4-cuts that is generated by some $\mathcal{F}_i$, for an $i \in \{1, \dots, k\}$. Thus, Corollary 5.9 implies that C is a quasi $\mathcal{C}$-isolated 4-cut. Conversely, we can prove that, if C is an essential quasi $\mathcal{C}$-isolated 4-cut, then there is a pair of edges $p \subset C$ such that (p, C) is preceded by $(p, *)$ in $\mathcal{P}$. Thus, we can collect all the 4-cuts in $\widetilde{\mathcal{C}}$ that are provably not quasi $\mathcal{C}$-isolated: these are precisely the essential $\mathcal{C}$-isolated 4-cuts. This procedure is shown in Algorithm 19. Its correctness

is established in Proposition 5.7.

Algorithm 19: Compute all the essential $\mathcal{C}$-isolated 4-cuts, where $\mathcal{C}$ is a complete collection of 4-cuts of G

1 compute the collections of pairs of edges $\mathcal{F}_1, \dots, \mathcal{F}_k$ that are returned by Algorithm 16 on input $\mathcal{C}$
2 initialize a counter *Count*$(C) \leftarrow 0$, for every $C \in \mathcal{C}$
3 **foreach** $i \in \{1, \dots, k\}$ **do**
4   **if** $|\mathcal{F}_i| = 2$ **then**
5   let C be the 4-cut in $\mathcal{C}$ from which $\mathcal{F}_i$ is derived
6   set *Count*$(C) \leftarrow$ *Count*$(C) + 1$
7   **end**
8 **end**
9 initialize an empty collection $\widetilde{\mathcal{C}}$
10 **foreach** $C \in \mathcal{C}$ **do**
11   **if** *Count*$(C) = 3$ ***and*** *C is an essential 4-cut of G* **then**
12   insert C into $\widetilde{\mathcal{C}}$
13   **end**
14 **end**
15 compute the collection $\mathcal{M}$ of all the essential $\mathcal{C}_i$-minimal 4-cuts, for every $i \in \{1, \dots, k\}$ such that $\mathcal{F}_i$ generates a cyclic family $\mathcal{C}_i$ of 4-cuts
16 initialize an empty collection $\mathcal{P}$
17 **foreach** $C \in \mathcal{M}$ **do**
18   **foreach** *ordered pair of edges p in C* **do**
19   insert a pair $(p, *)$ into $\mathcal{P}$
20   **end**
21 **end**
22 **foreach** $C \in \widetilde{\mathcal{C}}$ **do**
23   **foreach** *ordered pair of edges p in C* **do**
24   insert a pair (p, C) into $\mathcal{P}$
25   **end**
26 **end**
27 sort $\mathcal{P}$ in lexicographic order, giving priority to $*$
28 initialize an empty collection $\mathcal{Q}$
29 **foreach** *pair* $(p, C) \in \mathcal{P}$ **do**
30   **if** *the predecessor of (p, C) in $\mathcal{P}$ is $(p, *)$* **then**
31   insert C into $\mathcal{Q}$
32   **end**
33 **end**
34 **return** $\widetilde{\mathcal{C}} \setminus \mathcal{Q}$

Proposition 5.7. *Let $\mathcal{C}$ be a complete collection of 4-cuts of G. Then, Algorithm 19 correctly computes the collection of all the essential $\mathcal{C}$-isolated 4-cuts. The running time of Algorithm 19 is $O(n + |\mathcal{C}|)$.*

Proof. Let $\mathcal{F}_1, \ldots, \mathcal{F}_k$ be the output of Algorithm 16 on input $\mathcal{C}$. By Lemma 5.16, we have that every $\mathcal{C}$-isolated 4-cut C has the property that $C \in \mathcal{C}$ and all three partitions of C into pairs of edges are contained in the set $\{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$ $(*)$. Thus, the first step is to find all $C \in \mathcal{C}$ that have this property. To do this, we first find all $\mathcal{F}_i$, for $i \in \{1, \ldots, k\}$, that satisfy $|\mathcal{F}_i| = 2$. If for an $i \in \{1, \ldots, k\}$ we have $|\mathcal{F}_i| = 2$, then by Lemma 5.15 we have that $C = \bigcup \mathcal{F}_i \in \mathcal{C}$. Thus, for the 4-cut C, we increase the counter *Count*(C) by one (in Line 6), signifying that we have found one more partition of C into pairs of edges within $\{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$. Thus, if for a 4-cut $C \in \mathcal{C}$ we have *Count*$(C) = 3$, then we know that all partitions of C into pairs of edges are contained in $\{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$, and so we insert this 4-cut into $\widetilde{\mathcal{C}}$ if it is essential (Line 12). The purpose of $\widetilde{\mathcal{C}}$ is precisely to contain all the essential 4-cuts $C \in \mathcal{C}$ that satisfy property $(*)$.

Let us provide a simple extension to Algorithm 16, in order to maintain the information that $\mathcal{F}_i$ generates precisely the 4-cut C, whenever $|\mathcal{F}_i| = 2$, for $i \in \{1, \ldots, k\}$. (This is needed in Line 5.) Let $C = \{e_1, e_2, e_3, e_4\} \in \mathcal{C}$, and let us assume w.l.o.g. that $\mathcal{F}_i = \{\{e_1, e_2\}, \{e_3, e_4\}\}$. Let us also assume, w.l.o.g., that $e_1 < e_2$ and $e_3 < e_4$. Then we have that $\mathcal{F}_i$ is derived (in Line 16 of Algorithm 16) from a connected component S of the graph $\mathcal{G}$ that is generated internally by Algorithm 16 in Line 14. Thus, we have that S contains at least two elements $(C', (e_1, e_2))$ and $(C'', (e_3, e_4))$, for some 4-cuts $C', C'' \in \mathcal{C}$. Since $C \in \mathcal{C}$, Algorithm 16 also generates the elements $(C, (e_1, e_2))$ and $(C, (e_3, e_4))$. By construction of $\mathcal{G}$, we have that $(C, (e_1, e_2))$ is connected with $(C', (e_1, e_2))$ in $\mathcal{G}$, and $(C, (e_3, e_4))$ is connected with $(C'', (e_3, e_4))$ in $\mathcal{G}$. Let us suppose, for the sake of contradiction, that $C' \neq C$. Then, let $\{x, y\}$ be the pair of edges such that $C' = \{e_1, e_2, x, y\}$, and let us assume w.l.o.g. that $x < y$. Then, we have that Algorithm 16 generates the element $(C', (x, y))$, and this is connected with $(C', (e_1, e_2))$ (see Line 8). Thus, $(C', (x, y))$ is also in S, and therefore $\mathcal{F}_i$ must also contain $\{x, y\}$ (see Line 16). Since $C' \neq C$, we have that $\{x, y\} \neq \{e_3, e_4\}$. And since C' is a 4-cut, it is a 4-element set, and therefore $\{x, y\} \neq \{e_1, e_2\}$. This implies that $|\mathcal{F}_i| \geq 3$, a contradiction. Thus, we have that $C' = C$. Similarly, we have that $C'' = C$. Thus, we can see that the only elements of S are $(C, (e_1, e_2))$ and $(C, (e_3, e_4))$. Thus, whenever a connected component S of $\mathcal{G}$ contains precisely two elements of the form

$(C,(e_1,e_2))$ and $(C,(e_3,e_4))$, then we can simply associate with the collection of pairs of edges that is derived from S (in Line 16) the information that it generates C.

Thus, when we reach Line 15, we can be certain that $\widetilde{\mathcal{C}}$ contains precisely all the essential 4-cuts $C \in \mathcal{C}$ that satisfy property $(*)$. Now, among all the 4-cuts in $\widetilde{\mathcal{C}}$, we have to identify and discard those that are quasi $\mathcal{C}$-isolated. Then, by definition, we will be left with the (essential) $\mathcal{C}$-isolated 4-cuts. Let $\mathcal{M}$ be the collection of all the essential $\mathcal{C}_i$-minimal 4-cuts, for all $i \in \{1,\dots,k\}$ such that $\mathcal{F}_i$ generates a cyclic family $\mathcal{C}_i$. Now, according to Corollary 5.9, in order to determine whether a 4-cut in $\widetilde{\mathcal{C}}$ is quasi $\mathcal{C}$-isolated, it is sufficient to check whether it intersects with a 4-cut in $\mathcal{M}$ in a pair of edges. Thus, the idea is basically to break every 4-cut in $\mathcal{M}$ into all different combinations of (ordered) pairs of edges, and then check, for every 4-cut $C \in \widetilde{\mathcal{C}}$, whether C contains any of those pairs. If that is the case, then by Corollary 5.9 we have that C is quasi $\mathcal{C}$-isolated; otherwise, by Corollary 5.9 we have that C is $\mathcal{C}$-isolated.

In order to perform this checking efficiently, we collect every pair of edges p that is contained in a 4-cut in $\mathcal{M}$, and then we form a pair $(p,*)$. We demand that those pairs are ordered (according to any total ordering of the edges). Thus, for every 4-cut C in $\mathcal{M}$, we have that C generates six pairs of the form $(p,*)$. The symbol $*$ is simply to signify that p is contained in a 4-cut in $\mathcal{M}$. Now, we basically do the same for every 4-cut C in $\widetilde{\mathcal{C}}$: for every pair of edges p in C, we create a pair (p,C). Notice that here we maintain the information, what is the 4-cut in $\widetilde{\mathcal{C}}$ from which (p,C) is derived. Now, we collect all those pairs in a collection $\mathcal{P}$, which we sort lexicographically (with bucket-sort), giving priority to $*$. Thus, if there is a 4-cut $C \in \widetilde{\mathcal{C}}$ that contains a pair of edges p which is shared by a 4-cut in $\mathcal{M}$, then we have that the element (p,C) is preceded by an element $(p,*)$ in $\mathcal{P}$. Moreover, we have that $(p,*)$ is precisely the predecessor of (p,C) in $\mathcal{P}$. To see this, suppose the contrary. Then, we have that there is a 4-cut $C' \in \widetilde{\mathcal{C}}$ with $C' \neq C$ that contains the pair of edges p. Let $q = C' \setminus p$. Since $C' \in \widetilde{\mathcal{C}}$, we have that the collection of pairs of edges $\{p,q\}$ is contained in $\{\mathcal{F}_1,\dots,\mathcal{F}_k\}$. Let $p' = C \setminus p$. Then, since $C \in \widetilde{\mathcal{C}}$, we have that the collection of pairs of edges $\{p,p'\}$ is contained in $\{\mathcal{F}_1,\dots,\mathcal{F}_k\}$. Since $C' \neq C$, we have that $\{p,q\} \neq \{p,p'\}$. Therefore, Lemma 5.13 implies that $\{p,q\} \cap \{p,p'\} = \emptyset$, a contradiction. Thus, we have that there is no 4-cut $C' \in \widetilde{\mathcal{C}}$ with $C' \neq C$ that contains the pair of edges p. Therefore, the predecessor of (p,C) in $\mathcal{P}$ is precisely $(p,*)$. (We note that this is because we consider the *ordered* pairs of edges that are produced by the pairs of edges that are contained

in the 4-cuts. Otherwise, $(p,*)$ could be the predecessor of the predecessor of (p,C).) Thus, we can infer that C shares a pair of edges with an essential C_i-minimal 4-cut, for some $i\in\{1,\dots,k\}$. Thus, in Line 31 we insert C into the collection $\mathcal{Q}$ (which is to contain all the essential quasi $\mathcal{C}$-isolated 4-cuts). Conversely, if C does not have this property, then obviously the predecessor of (p,C) in $\mathcal{P}$ cannot have the form $(p,*)$, for any pair of edges $p\subset C$. Thus, by Corollary 5.9 we infer that C is not a quasi $\mathcal{C}$-isolated 4-cut, and therefore, since $C\in\widetilde{\mathcal{C}}$, it must be an essential $\mathcal{C}$-isolated 4-cut. Thus, when we reach Line 34, $\mathcal{Q}$ contains precisely all the essential quasi $\mathcal{C}$-isolated 4-cuts, and therefore we only have to return $\widetilde{\mathcal{C}}\setminus\mathcal{Q}$, because this is now the collection of all the essential $\mathcal{C}$-isolated 4-cuts.

To conclude the proof of correctness, we have to provide a method to compute the collection $\mathcal{M}$. To do this, we perform for every $i\in\{1,\dots,k\}$ the following check. First of all, by Proposition 5.3 we can be certain that $\mathcal{F}_i$ generates a collection of 4-cuts of G. Now, if $|\mathcal{F}_i|=2$, then i is ignored. If $|\mathcal{F}_i|>3$, then Proposition 5.2 implies that $\mathcal{F}_i$ generates a cyclic family of 4-cuts, and so we keep i. If $|\mathcal{F}_i|=3$, then we compute the three 4-cuts C_1, C_2 and C_3, that are generated by $\mathcal{F}_i$. If either of C_1,C_2,C_3 is essential, then Corollary 5.5 implies that $\mathcal{F}_i$ generates a cyclic family of 4-cuts, and so we keep i. Otherwise, either $\mathcal{F}_i$ does not generate a cyclic family of 4-cuts, or, if it does, all of its minimal 4-cuts are non-essential, and so we can ignore i. Now, let I be the collection of all the indices that we have collected. Then, for every $i\in I$, we have that $\mathcal{F}_i$ generates a cyclic family of 4-cuts. Thus, we can apply Algorithm 18 on the collection $\{\mathcal{F}_i\mid i\in I\}$, which will produce the collection $\mathcal{M}'$ of all $\mathcal{C}_i$-minimal 4-cuts, where $\mathcal{C}_i$ is the cyclic family of 4-cuts generated by $\mathcal{F}_i$, for $i\in I$. Then, we only keep from $\mathcal{M}'$ the essential 4-cuts, and so we compute the collectiom $\mathcal{M}$.

Now let us provide the time-bounds for the non-trivial steps of Algorithm 19. First, by Proposition 5.3 we have that Line 1 takes $O(n+|\mathcal{C}|)$ time. Furthermore, Proposition 5.3 ensures that $|\mathcal{F}_1|+\cdots+|\mathcal{F}_k|=O(|\mathcal{C}|)$. In order to check the essentiality in Line 11, we rely on the assumption we have made at the beginning of this section: that is, we have completed the linear-time preprocessing of the graph that is described in Proposition 5.4, so that we can check the essentiality of any 4-cut in $O(1)$ time. Thus, the **for** loop in Line 10 takes $O(|\mathcal{C}|)$ time in total. By Proposition 5.6, we can compute the collection $\mathcal{M}'$ of all the $\mathcal{C}_i$-minimal 4-cuts, for every $i\in I$, in $O(n+\sum_{i\in I}|\mathcal{F}_i|)=O(n+|\mathcal{F}_1|+\cdots+|\mathcal{F}_k|)=O(n+|\mathcal{C}|)$ time. Notice that the size of $\mathcal{M}'$ is $O(|\mathcal{F}_1|+\cdots+|\mathcal{F}_k|)=O(|\mathcal{C}|)$, and there are $O(|\mathcal{C}|)$ essentiality checks that are involved

in the computation of $\mathcal{M}$. Finally, notice that the size of $\mathcal{P}$ is $O(|\mathcal{C}|)$, and therefore Line 27 takes $O(n+|\mathcal{C}|)$ time if we perform the sorting with bucket-sort. Also, we can compute the set difference $\widetilde{\mathcal{C}} \setminus \mathcal{Q}$ in Line 34 with bucket-sort. Thus, the total running time of Algorithm 19 is $O(n+|\mathcal{C}|)$. □

5.4.4 Computing enough 4-cuts in order to derive the 5-edge-connected components

Let $\mathcal{C}$ be a complete collection of 4-cuts of G. Then, Algorithm 20 shows how we can extract a parallel collection $\mathcal{C}'$ of 4-cuts from $\mathcal{C}$, that contains enough 4-cuts in order to separate all pairs of vertices x, y with $\lambda(x,y) = 4$. The proof of correctness is given in Proposition 5.8.

Algorithm 20: Generate a parallel collection of 4-cuts from a complete collection $\mathcal{C}$ of 4-cuts, that contains enough 4-cuts in order to separate all pairs of vertices x, y with $\lambda(x,y) = 4$

---

1 compute the collections of pairs of edges $\mathcal{F}_1, \dots, \mathcal{F}_k$ that are returned by Algorithm 16 on input $\mathcal{C}$
2 compute the collection $\mathcal{M}$ of all the essential $\mathcal{C}_i$-minimal 4-cuts, where $\mathcal{C}_i$ is the cyclic family of 4-cuts that is generated by $\mathcal{F}_i$, for some $i \in \{1, \dots, k\}$
3 compute the collection $\mathcal{ISO}$ of the essential $\mathcal{C}$-isolated 4-cuts
4 **return** $\mathcal{M} \cup \mathcal{ISO}$

Proposition 5.8. *Let $\mathcal{C}$ be a complete collection of 4-cuts of G. Then, the output of Algorithm 20 on input $\mathcal{C}$ is a parallel family $\mathcal{C}'$ of 4-cuts such that: for every pair of vertices x, y of G with $\lambda(x,y) = 4$, there is a 4-cut in $\mathcal{C}'$ that separates x and y. The running time of Algorithm 20 is $O(n+|\mathcal{C}|)$.*

Proof. Let $\mathcal{F}_1, \dots, \mathcal{F}_k$ be the collections of pairs of edges that are returned by Algorithm 16 on input $\mathcal{C}$. Then, by Proposition 5.3 we have that $\mathcal{F}_i$ generates a collection of 4-cuts implied by $\mathcal{C}$, for every $i \in \{1, \dots, k\}$. Thus, let $\mathcal{C}_i$ be the collection of 4-cuts generated by $\mathcal{F}_i$, for every $i \in \{1, \dots, k\}$. By Proposition 5.3, we also have that every 4-cut implied by $\mathcal{C}$ is contained in $\mathcal{C}_i$, for some $i \in \{1, \dots, k\}$. Thus, since $\mathcal{C}$ is a complete collection of 4-cuts, we have that every 4-cut of G is contained in some $\mathcal{C}_i$, for an $i \in \{1, \dots, k\}$. Let $\mathcal{C}'$ be the output of Algorithm 20 on input $\mathcal{C}$. Let also $\mathcal{M}$

and $\mathcal{ISO}$ be the collections that are constructed in Lines 2 and 3, respectively, on input $\mathcal{C}$. Notice that every 4-cut in $\mathcal{ISO}$ is an essential *isolated* 4-cut, because $\mathcal{C}$ is a complete collection of 4-cuts (and therefore it implies all the 4-cuts of G).

Now let x, y be a pair of vertices of G with $\lambda(x, y) = 4$. This means that there is a 4-cut C of G that separates x and y. By definition, C is an essential 4-cut. Since C is a 4-cut of G, there is an $i \in \{1, \dots, k\}$ such that $C \in \mathcal{C}_i$. Thus, we can distinguish two cases: either (1) there is an $i \in \{1, \dots, k\}$ such that $C \in \mathcal{C}_i$ and $|\mathcal{F}_i| > 2$, or (2) for every $i \in \{1, \dots, k\}$ such that $C \in \mathcal{C}_i$ we have that $|\mathcal{F}_i| = 2$. Let us consider case (1) first. Thus, there is an $i \in \{1, \dots, k\}$ such that $C \in \mathcal{C}_i$ and $|\mathcal{F}_i| > 2$. Then, since C is an essential 4-cut in $\mathcal{C}_i$, by Corollary 5.5 we have that $\mathcal{C}_i$ is a cyclic family of 4-cuts. Then, since C separates x and y and $\lambda(x, y) = 4$, by Lemma 5.12 we have that there is an essential $\mathcal{C}_i$-minimal 4-cut C' that separates x and y. Then, we have that $C' \in \mathcal{M}$, and therefore $C' \in \mathcal{C}'$.

Now let us consider case (2). Thus, we have that, for every $i \in \{1, \dots, k\}$ such that $C \in \mathcal{C}_i$, we have $|\mathcal{F}_i| = 2$. Let us consider an $i \in \{1, \dots, k\}$ such that $C \in \mathcal{C}_i$ (we have already shown that such an i exists). Then, we have that $|\mathcal{F}_i| = 2$. Thus, Lemma 5.15 implies that $C \in \mathcal{C}$. Then, by Lemma 5.14 we have that every partition $\mathcal{F}'$ of C into pairs of edges is contained in some $\mathcal{F}_j$, for $j \in \{1, \dots, k\}$. Then we have that $C \in \mathcal{C}_j$, and therefore $|\mathcal{F}_j| = 2$. This implies that $\mathcal{F}' = \mathcal{F}_j$. This shows that all partitions of C into pairs of edges are contained in the output of Algorithm 16 on input $\mathcal{C}$. Thus, we can distinguish two cases: either (2.1) C is a $\mathcal{C}$-isolated 4-cut, or (2.2) C is a quasi $\mathcal{C}$-isolated 4-cut. In case (2.1), we have that $C \in \mathcal{ISO}$ (because C is essential). In case (2.2), we can evoke Lemma 5.21: this implies that there is a $t \in \{1, \dots, k\}$ such that $|\mathcal{F}_t| > 2$ and $\mathcal{C}_t$ contains a 4-cut C' that separates x and y. By definition, we have that C' is an essential 4-cut. Thus, Corollary 5.5 implies that $\mathcal{C}_t$ is a cyclic family of 4-cuts. Therefore, since $C' \in \mathcal{C}_t$ separates x and y (which are 4-edge-connected), Lemma 5.12 implies that there is an essential $\mathcal{C}_t$-minimal 4-cut C'' that separates x and y. Thus, we have that $C'' \in \mathcal{M}$, and therefore $C'' \in \mathcal{C}'$. Thus, we have shown that, for every pair of vertices x, y of G with $\lambda(x, y) = 4$, there is a 4-cut in $\mathcal{C}'$ that separates x and y.

Now we will show that $\mathcal{C}'$ is a parallel collection of 4-cuts. Let C, C' be two distinct 4-cuts in $\mathcal{C}'$. If at least one of C, C' is in $\mathcal{ISO}$, then it is an essential isolated 4-cut, and so Corollary 5.6 implies that it is parallel with every other essential 4-cut. Thus, let us assume that both C and C' are in $\mathcal{M}$. Then there are $i, j \in \{1, \dots, k\}$ such that C is a $\mathcal{C}_i$-minimal 4-cut, and C' is a $\mathcal{C}_j$-minimal 4-cut. If $i = j$, then Lemma 5.11 implies

that C and C' are parallel 4-cuts. Otherwise, since both C and C' are essential 4-cuts, Lemma 5.20 implies that C and C' are parallel. Thus, we have shown that $\mathcal{C}'$ is a parallel collection of 4-cuts.

Finally, let us consider the running time of Algorithm 20. By Proposition 5.3, we have that Line 1 takes time $O(n + |\mathcal{C}|)$, and the output $\mathcal{F}_1, \ldots, \mathcal{F}_k$ has size $O(|\mathcal{F}_1| + \cdots + |\mathcal{F}_k|) = O(|\mathcal{C}|)$. Then, we can implement Line 2 with the same idea as Line 15 of Algorithm 19 (see the proof of Proposition 5.7). This will take $O(n + |\mathcal{C}|)$ time, provided that we have made the linear-time preprocessing on G that is described in Proposition 5.4, in order to be able to perform essentiality checks in $O(1)$ worst-case time per 4-cut. Finally, by Proposition 5.7, we have that Line 3 takes time $O(n+|\mathcal{C}|)$. We conclude that the running time of Algorithm 20 is $O(n + |\mathcal{C}|)$.

□

5.4.5 The algorithm

The full algorithm for computing the 5-edge-connected components of a 3-edge-connected graph in linear time is shown in Algorithm 21. The proof of correctness is given in Proposition 5.9.

Algorithm 21: Compute the 5-edge-connected components of a 3-edge-connected graph G

1 compute the partition $\mathcal{P}_4$ of the 4-edge-connected components of G
2 compute a complete collection $\mathcal{C}$ of 4-cuts of G
3 compute the output $\mathcal{C}'$ of Algorithm 20 on input $\mathcal{C}$
4 compute the partition $\mathcal{P}_5 = \mathit{atoms}(\mathcal{C}')$
5 **return** $\mathcal{P}_4$ refined by $\mathcal{P}_5$

Proposition 5.9. *Algorithm 21 correctly computes the 5-edge-connected components of a 3-edge-connected graph. Furthermore, it has a linear-time implementation.*

Proof. Let $\mathcal{P}_4$, $\mathcal{C}$, $\mathcal{C}'$, and $\mathcal{P}_5$, be as defined in Lines 1, 2, 3, and 4, respectively. Let $\mathcal{P}$ be the output of Algorithm 21. Notice that $\mathcal{P}$ is a partition of the vertex set of G. We will show that, for every pair of vertices x, y of G, we have $\lambda(x, y) < 5$ if and only if x and y are separated by $\mathcal{P}$. So let x, y be a pair of vertices of G with $\lambda(x, y) < 5$. Since G is 3-edge-connected, we have that either $\lambda(x, y) = 3$, or $\lambda(x, y) = 4$. If $\lambda(x, y) = 3$,

then x and y lie in different 4-edge-connected components of G. Thus, x and y are separated by $\mathcal{P}_4$, and therefore they are separated by $\mathcal{P}$, since $\mathcal{P}$ is a refinement of $\mathcal{P}_4$. Now let us assume that $\lambda(x,y)=4$. Then, Proposition 5.8 implies that there is a 4-cut $C\in\mathcal{C}'$ that separates x and y. Thus, x and y are separated by $\mathcal{P}_5 = \mathit{atoms}(\mathcal{C}')$, and therefore they are separated by $\mathcal{P}$, since $\mathcal{P}$ is a refinement of $\mathcal{P}_5$. Thus, for every pair of vertices x,y of G with $\lambda(x,y)<5$, we have that x and y are separated by $\mathcal{P}$. Conversely, every pair of vertices that are separated by $\mathcal{P}$, are separated by either $\mathcal{P}_4$ or $\mathcal{P}_5$, and therefore they are separated by either a 3-cut or a 4-cut of G, and therefore they are not 5-edge-connected. This shows that $\mathcal{P}$ is the collection of the 5-edge-connected components of G.

By previous work, we know that Line 1 can be implemented in linear time (see [36] or [50]). By Theorem 5.3, we have that a complete collection $\mathcal{C}$ of 4-cuts of G with size $O(n)$ can be computed in linear time (in Line 2). Thus, by Proposition 5.8 we have that the output $\mathcal{C}'$ of Algorithm 20 on input $\mathcal{C}$ can be computed in $O(n+|\mathcal{C}|)=O(n)$ time. Furthermore, by Proposition 5.8 we have that $\mathcal{C}'$ is a parallel family of 4-cuts. Thus, by Proposition 5.5 we have that the computation of $\mathit{atoms}(\mathcal{C}')$ can be performed in $O(n)$ time. Finally, the common refinement of $\mathcal{P}_4$ and $\mathcal{P}_5$ can be computed in $O(n)$ time with bucket-sort, since these are partitions of $V(G)$. We conclude that Algorithm 21 has a linear-time implementation. □

We note that this method for computing the 5-edge-connected components is also useful for constructing a data structure that has the same functionality as a partial Gomory-Hu tree that retains all connectivities up to 5 [41]. Specifically, we have the following.

Corollary 5.11. *Given a 3-edge-connected graph G with n vertices, there is a linear-time preprocessing of G that constructs a data structure of size $O(n)$, such that, given two 4-edge-connected vertices x and y of G, we can determine in $O(1)$ time a 4-cut of G that separates x and y, or report that no such 4-cut exists.*

Proof. This is a consequence of the fact that Algorithm 21 computes a parallel family $\mathcal{C}$ of 4-cuts of G, with the property that every two 4-edge-connected vertices that are separated by a 4-cut of G, are also separated by a 4-cut from $\mathcal{C}$. Then, we can apply Corollary 5.10 on $\mathcal{C}$. □

5.5 Computing a complete collection of 4-cuts

The purpose of this chapter is to provide a summary of the methods that we use in order to establish Theorem 5.3. Let G be a 3-edge-connected graph. The idea is to classify the 4-cuts of G on a DFS-tree, in order to make it easy for us to compute enough of them efficiently. This results in several algorithms, each of which specializes in computing a specific type of 4-cuts. In this section we provide our classification of 4-cuts, down to all the subcases, and we provide an overview of the methods that we use in order to handle each case. We also provide figures with detailed captions, that we consider an organic part of our exposition. The complete analysis, the proofs and the algorithms, are given in the chapters that follow (in Sections 5.6, 5.7 and 5.8). We conclude with a technical result (in Section 5.5.4), that we will need in the following chapters. Throughout this chapter we assume familiarity with the DFS-based concepts that we defined in Section 3.1.

5.5.1 A typology of 4-cuts on a DFS-tree

Let r be a vertex of G, and let T be a DFS-tree of G rooted at r. Initially, we classify the 4-cuts of G according to the number of tree-edges that they contain. Thus, we distinguish Type-1, Type-2, Type-3 and Type-4 4-cuts, depending on whether they contain one, two, three or four tree-edges, respectively. Notice that there are no 4-cuts that consist entirely of non-tree edges, because the removal of any set of non-tree edges is insufficient to disconnect the graph, due to the existence of T.

We found that it is very difficult to compute enough Type-4 4-cuts directly. Thus, we use an idea from [50], in order to reduce the case of those 4-cuts to the previous cases. Specifically, we first establish the following result.

Proposition 5.10. *Let G be a 3-edge-connected graph with n vertices, and let T be a DFS-tree of G. Then there is a linear-time algorithm that computes a collection $\mathcal{C}$ of 4-cuts of G, that has size $O(n)$ and implies the collection of all 4-cuts of G that contain at least one back-edge w.r.t. T.*

We establish Proposition 5.10 by essentially following the framework of classification and the techniques of [36] for computing all 3-cuts of a 3-edge-connected graph (although we have to extend the concepts and techniques significantly). However, for Type-4 4-cuts, it seems extremely complicated to apply the framework of [36]. Thus,

here we rely on the reduction used by [50]. In more detail, [50] also provided a linear-time algorithm for computing all 3-cuts of a 3-edge connected graph, by using techniques very similar to [36] for the case where there is at least one back-edge in the 3-cut. Contrary to [36], however, they do not deal directly with the 3-cuts that consist of three tree-edges. Instead, they show how to reduce the case of 3-cuts that consist of three tree-edges to the previous cases. They do this through a "contraction" technique that works as follows. First, we remove all the tree-edges from G, and we compute the connected components of the resulting graph. Then we shrink every connected component into a single node, and we re-insert the tree-edges that join different nodes. This results in a graph Q that is 3-edge-connected; its k-cuts, for $k \geq 3$, coincide with those of G that consist only of tree-edges, and its number of edges is at most $2/3$ that of G. Thus, we can reduce the computation of k-cuts to Q. We believe that it is important to formalize and prove this result.

Definition 5.9 (Contracted Graph). Let G be a connected graph, and let T be a spanning tree of G. Let $C_1, \ldots, C_k$ be the connected components of $G \setminus E(T)$. Now we have a function $q : V(G) \rightarrow \{C_1, \ldots, C_k\}$ (the quotient map), that maps every vertex of G to the connected component of $G \setminus E(T)$ that contains it. This function induces naturally a map between edges of G, and edges between the connected components of $G \setminus E(T)$. Specifically, given an edge $e = (x, y)$ of G, we let $q(e) = (q(x), q(y))$.[5] Then we define the contracted (quotient) graph Q as follows. The vertex set of Q is $\{C_1, \ldots, C_k\}$, and the edge set of Q consists of all edges of the form $q(e)$, where e is a tree-edge of T that connects two different connected components of $G \setminus E(T)$.

Lemma 5.26 (Implicit in [50]). *Let $k \geq 3$ be an integer, let G be a k-edge-connected graph with n vertices and m edges, let T be a spanning tree of G, let Q be the resulting contracted graph of the connected components of $G \setminus E(T)$, and let q be the corresponding quotient map. Then Q is k-edge-connected and it has at most $2m/k$ edges. Furthermore, let $k' \geq k$ be a positive integer. Then, a k'-element subset C of $E(T)$ is a k'-cut of G if and only if $q(C)$ is a k'-cut of Q.*

Proof. It is easy to bound the number of edges of Q. First, since G is k-edge-connected, every vertex of G has degree at least k. The sum of the degrees of all vertices is $2m$.

[5] To be more precise, the image of $e = (x, y)$ through q should be an edge $q(e)$ (possibly a self-loop) with endpoints $q(x)$ and $q(y)$, associated with a unique identifier that signifies that $q(e)$ is derived from e. This is because another edge $e' = (x', y')$ of G may also satisfy that $(q(x), q(y)) = (q(x'), q(y'))$, but we want to distinguish between $q(e)$ and $q(e')$.

Thus, we have $kn \leq 2m$, and therefore $n \leq 2m/k$. By construction, Q has at most $|E(T)| = n - 1$ edges. Thus, the number of edges of Q is bounded by $2m/k$.

The remaining part of the lemma follows essentially from a correspondence between paths of G and paths of Q. Specifically, let P be a path from x to y in G. Then there is a contracted path $\widetilde{P}$ from $q(x)$ to $q(y)$ in Q with the property that, for every tree-edge e used by P such that e connects two different connected components of $G \setminus E(T)$, there is an instance of $q(e)$ in $\widetilde{P}$. Furthermore, these are all the edges that appear in $\widetilde{P}$. We note that $\widetilde{P}$ is formed by contracting every part of P that lies entirely within a connected component z of $G \setminus E(T)$ into z (viewed as a vertex of Q). Conversely, for every path P' in Q, there is a path P in G such that $\widetilde{P} = P'$ (which is formed basically by expanding every vertex z that is used by P' into a path within z).

This explains why Q is k-edge-connected. To see this, let C be a set of less than k edges of Q, and let u and v be two vertices of Q. (Recall that, by definition, we have that u and v are connected components of $G \setminus E(T)$.) Then let x be a vertex of G in u, and let y be a vertex of G in v. Then, since G is k-edge-connected, we have that $G \setminus q^{-1}(C)$ is connected, and therefore there is a path P from x to y in $G \setminus q^{-1}(C)$. Then, $\widetilde{P}$ is a path from $q(x)$ to $q(y)$ in $Q \setminus C$, and therefore u and v are connected in $Q \setminus C$. This shows that no set of less than k edges of Q is sufficient to disconnect Q upon removal. This means that Q is k-edge-connected.

Now let $k' \geq k$ be an integer, and let C be a k'-cut of G that consists only of tree-edges. We will show that $q(C)$ is k-cut of Q. First, we have to show that $q(C)$ is well-defined (i.e., it is a set of edges of Q). So let e be an edge in C. Then, since C is a k'-cut of G, we have that the endpoints of e lie in different connected components of $G \setminus C$. Therefore, since $C \subseteq E(T)$, we have that the endpoints of e lie in different connected components of $G \setminus E(T)$. This shows that $q(e)$ is an edge of Q. Now we will show that $Q \setminus q(C)$ is disconnected. So let us suppose, for the sake of contradiction, that $Q \setminus q(C)$ is connected. Since C is a k'-cut of G, we have that $G \setminus C$ is disconnected. Thus, there are vertices x and y of $G \setminus C$ that lie in different connected components of $G \setminus C$. Since $Q \setminus q(C)$ is connected, there is a path P' from $q(x)$ to $q(y)$ in $Q \setminus q(C)$. Then there is a path P in G such that $\widetilde{P} = P'$. This implies that P avoids all the edges from C, and therefore it is a path in $G \setminus C$. Furthermore, the start of P is in the connected component of $G \setminus E(T)$ that contains x, and the end of P is in the connected component of $G \setminus E(T)$ that contains y. There is a path P_x in $G \setminus E(T)$

from x to the start of P. Similarly, there is a path P_y in $G \setminus E(T)$ from the end of P to y. Now, since $C \subseteq E(T)$, the concatenation $P_x + P + P_y$ is a path in $G \setminus C$ from x to y. But this contradicts the fact that x and y are disconnected in $G \setminus C$. This shows that $Q \setminus q(C)$ is disconnected. Finally, let C' be a proper subset of C. We will show that $Q \setminus q(C')$ is connected. So let u and v be two vertices of Q. Then there is a vertex x of G in u, and there is a vertex y of G in v. Since C is a k-cut of G, we have that $G \setminus C'$ is connected. Thus, there is path P from x to y in $G \setminus C'$. Then, $\widetilde{P}$ is a path from $q(x)$ to $q(y)$ in $Q \setminus q(C')$. This shows that u and v are connected in $Q \setminus C'$. Due to the generality of u and v in Q, this shows that $Q \setminus C'$ is connected. Thus, we have that $q(C)$ is a k-cut of Q.

Conversely, let C be a k'-element subset of $E(T)$ such that $q(C)$ is a k'-cut of Q. We will show that C is a k'-cut of G. First, we will show that $G \setminus C$ is disconnected. So let us suppose, for the sake of contradiction, that $G \setminus C$ is connected. Since $q(C)$ is a k'-cut of Q, we have that $Q \setminus q(C)$ is disconnected. Thus, there are vertices u and v of Q such that u and v are disconnected in $Q \setminus q(C)$. Now let x be a vertex in u, and let y be a vertex in v. Then, since $G \setminus C$ is connected, there is a path P from x to y in $G \setminus C$. But then $\widetilde{P}$ is a path from $q(x)$ to $q(y)$ in $Q \setminus q(C)$, contradicting the fact that u and v are not connected in $Q \setminus q(C)$. This shows that $G \setminus C$ is disconnected. Now let C' be a proper subset of C. We will show that $G \setminus C'$ is connected. So let x and y be two vertices of G. Since $q(C)$ is a k'-cut of Q, we have that $Q \setminus q(C')$ is connected. Thus, there is a path P' from $q(x)$ to $q(y)$ in $Q \setminus q(C')$. Then, there is a path P in G such that $\widetilde{P} = P'$. This implies that P is a path in $G \setminus C'$. Furthermore, P starts from a vertex in the connected component of $G \setminus E(T)$ that contains x, and ends in a vertex in the connected component of $G \setminus E(T)$ that contains y. Then there is a path P_x in $G \setminus E(T)$ from x to the start of P. Furthermore, there is a path P_y in $G \setminus E(T)$ from the end of P to y. Then, since $C' \subseteq E(T)$, the concatenation $P_x + P + P_y$ is a path from x to y in $G \setminus C'$. Due to the generality of x and y in G, this shows that $G \setminus C'$ is connected. We conclude that C is a k'-cut of G. □

Now, given Proposition 5.10, we show how to derive Theorem 5.3 by a repeated application of Lemma 5.26.

Theorem 5.3. *Let G be a 3-edge-connected graph with n vertices. There is a linear-time algorithm that computes a complete collection of 4-cuts of G with size $O(n)$.*

Proof. Let us assume that G has at least two vertices, because otherwise there is

nothing to show. We define a sequence of graphs $G_0, G_1, G_2, \dots$ as follows. First, $G_0 = G$. Now suppose that G_i is defined, for some $i \geq 0$, and that it has at least two vertices. Let T_i be an arbitrary DFS-tree of G_i. Then we let G_{i+1} be the contracted graph of G_i w.r.t. T_i, and we let q_i be the corresponding quotient map (see Definition 5.9). Let N be the largest index such that G_N has at least two vertices. Since G_0 is 3-edge-connected, Lemma 5.26 implies that $G_0, G_1, \dots, G_N$ is a sequence of 3-edge-connected graphs. Let $n_i = |V(G_i)|$ and $m_i = |E(G_i)|$, for every $i \in \{0, \dots, N\}$. By construction, we have $m_1 \leq n - 1$. Then, Lemma 5.26 implies that $m_i \leq n(\frac{2}{3})^{i-1}$, for every $i \in \{1, \dots, N\}$. Since G_i is a 3-edge-connected graph, for every $i \in \{0, \dots, N\}$, we have $n_i \leq m_i$. This implies that $n_i \leq n(\frac{2}{3})^{i-1}$, for every $i \in \{1, \dots, N\}$.

Now, for every $i \in \{0, \dots, N\}$, we apply Proposition 5.10 in order to derive, in $O(m_i + n_i)$ time, a collection $\mathcal{C}_i$ of 4-cuts of G_i, that has size $O(n_i)$ and implies all the 4-cuts of G_i that contain at least one back-edge w.r.t. T_i. Notice that this whole process takes time $O(m_0+n_0)+\dots+O(m_N+n_N) = O(m_0+\dots+m_N) = O(m+n\sum_{i=1}^{N}(\frac{2}{3})^{i-1}) = O(m+n)$. Let i be an index in $\{0, \dots, N-1\}$. By Lemma 5.26 we have that $q_i^{-1}(\mathcal{C}_{i+1})$ is a collection of 4-cuts of G_i. Furthermore, by repeated application of Lemma 5.26 we have that $\mathcal{C}'_{i+1} = q_0^{-1}(q_1^{-1}(\dots q_i^{-1}(\mathcal{C}_{i+1})\dots))$ is a collection of 4-cuts of G.

Now let $\mathcal{C}$ be the collection $\mathcal{C}_0 \cup \mathcal{C}'_1 \cup \dots \cup \mathcal{C}'_N$. Notice that, for every $i \in \{1, \dots, N\}$, we can construct $\mathcal{C}_i$ in time $O(i|\mathcal{C}_i|) = O(ni(\frac{2}{3})^{i-1})$. Thus, the collection $\mathcal{C}$ can be constructed in time $O(n\sum_{i=1}^{N} i(\frac{2}{3})^{i-1}) = O(n)$. We have that $\mathcal{C}$ is a collection of 4-cuts of G. We claim that every 4-cut of G is implied by $\mathcal{C}$.

So let C be a 4-cut of G. If C contains at least one back-edge w.r.t. T_0, then by construction of $\mathcal{C}_0$ (due to Proposition 5.10) we have that $\mathcal{C}_0$ implies C. Therefore, $\mathcal{C}$ also implies C (since $\mathcal{C}_0 \subseteq \mathcal{C}$). Otherwise, suppose that C consists only of tree-edges from T_0. Then, by Lemma 5.26 we have that $q_0(C)$ is a 4-cut of G_1. Now, if $q_0(C)$ contains at least one back-edge w.r.t. T_1, then by construction of $\mathcal{C}_1$ (due to Proposition 5.10) we have that $\mathcal{C}_1$ implies $q_0(C)$. This means that there is a sequence $C_1, \dots, C_k$ of 4-cuts from $\mathcal{C}_1$, and a sequence $p_1, \dots, p_{k+1}$ of pairs of edges of G_1, such that $C_i = p_i \cup p_{i+1}$ for every $i \in \{1, \dots, k\}$, and $p_1 \cup p_{k+1} = q_0(C)$ (Definition 5.4). Now consider the sequence $q_0^{-1}(C_1), \dots, q_0^{-1}(C_k)$. Then this is a sequence of 4-cuts from $\mathcal{C}'_1$. Furthermore, we have $q_0^{-1}(C_i) = q_0^{-1}(p_i) \cup q_0^{-1}(p_{i+1})$ for every $i \in \{1, \dots, k\}$, and $C = q_0^{-1}(q_0(C)) = q_0^{-1}(p_1 \cup p_{k+1}) = q_0^{-1}(p_1) \cup q_0^{-1}(p_{k+1})$. This shows that $q_0^{-1}(C_1), \dots, q_0^{-1}(C_k)$ is an implicating sequence of $\mathcal{C}'_1$ that demonstrates that C is implied from $\mathcal{C}'_1$. Thus, $\mathcal{C}$ implies C (since $\mathcal{C}'_1 \subseteq \mathcal{C}$).

Otherwise, suppose that $q_0(C)$ consists only of tree-edges from T_1. Then, let t be the maximum index in $\{0,\dots,N\}$ such that $q_{t'}(q_{t'-1}(\dots q_0(C)\dots))$ consists only of tree-edges from $T_{t'+1}$, for every $t' \in \{0,\dots,t\}$. Then, Lemma 5.26 implies that $C' = q_t(q_{t-1}(\dots q_0(C)\dots))$ is a 4-cut of G_{t+1}. Furthermore, since C' consists only of tree-edges from T_{t+1}, Lemma 5.26 implies that $q_{t+1}(C')$ is a 4-cut of G_{t+2}. Due to the maximality of t, we have that $q_{t+1}(C')$ must contain at least one back-edge w.r.t. T_{t+2}. Then, by construction of $\mathcal{C}_{t+2}$ (due to Proposition 5.10), we have that $\mathcal{C}_{t+2}$ implies $q_{t+1}(C')$. This means that there is a sequence $C_1,\dots,C_k$ of 4-cuts from $\mathcal{C}_{t+2}$, and a sequence $p_1,\dots,p_{k+1}$ of pairs of edges of G_{t+2}, such that $C_i = p_i \cup p_{i+1}$ for every $i \in \{1,\dots,k\}$, and $p_1 \cup p_{k+1} = q_{t+1}(C')$. Now consider the sequence $q_0^{-1}(q_1^{-1}(\dots q_{t+1}^{-1}(C_1)\dots)),\dots,q_0^{-1}(q_1^{-1}(\dots q_{t+1}^{-1}(C_k)\dots))$. Then, it is not difficult to see that this is an implicating sequence of $\mathcal{C}'_{t+2}$, that demonstrates that $q_0^{-1}(q_1^{-1}(\dots q_{t+1}^{-1}(q_{t+1}(C'))\dots)) = C$ is implied from $\mathcal{C}'_{t+2}$. Thus, $\mathcal{C}$ implies C (since $\mathcal{C}'_{t+2} \subseteq \mathcal{C}$). □

The purpose of everything that follows is to establish Proposition 5.10. The case of Type-1 4-cuts is the easiest one. So let C be a Type-1 4-cut of G, let $(u,p(u))$ be the tree-edge that is contained in C, and let e_1,e_2,e_3 be the back-edges that are contained in C. Then, by removing C from G, we have that each of the subtrees $T(u)$ and $T(r)\setminus T(u)$ of T remains connected (see Figure 5.16). Thus, these are the two connected components of $G\setminus C$, and therefore the back-edges in C are all the non-tree edges that connect $T(u)$ with $T(r)\setminus T(u)$. Notice that these are precisely the back-edges in $B(u)$. Thus, we have $B(u)=\{e_1,e_2,e_3\}$. Therefore, it is easy to identify all Type-1 4-cuts: we only have to check, for every vertex $u \neq r$, whether $\mathit{bcount}(u)=3$, and, if yes, we mark $\{(u,p(u)),e_1,e_2,e_3\}$ as a 4-cut, where e_1,e_2,e_3 are the three back-edges that leap over u. In order to find e_1, e_2 and e_3, it is sufficient to maintain three distinct back-edges from $B(u)$, for every vertex $u \neq r$. The low_1, low_2 and low_3 edges of u are sufficient for this purpose. By Proposition 3.2, we can have those edges computed for all vertices $\neq r$, in linear time in total. Thus, all Type-1 4-cuts can be computed in linear time in total. Notice that every one of them corresponds to a unique vertex $u \neq r$. Thus, the total number of Type-1 4-cuts is $O(n)$.

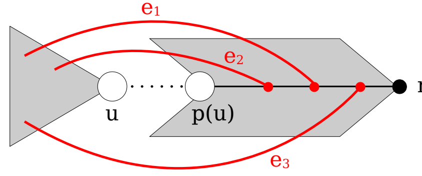


Figure 5.16: A Type-1 4-cut of the form $\{(u, p(u)), e_1, e_2, e_3\}$, where e_1, e_2, e_3 are back-edges. In this case, we have $B(u) = \{e_1, e_2, e_3\}$.

5.5.2 Type-2 4-cuts

Now we consider the case of Type-2 4-cuts. Let C be a Type-2 4-cut, and let $(u, p(u))$ and $(v, p(v))$ be the tree-edges that are contained in C. Then Lemma 3.14 implies that u and v are related as ancestor and descendant. So let us assume w.l.o.g. that u is a descendant of v. Then, Lemma 5.28 shows that there are three distinct cases to consider (see Figure 5.17): either (1) $B(v) = B(u) \sqcup \{e_1, e_2\}$, or (2) $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$, or (3) $B(u) = B(v) \sqcup \{e_1, e_2\}$, where e_1 and e_2 are the back-edges in C. We call the 4-cuts in those cases Type-$2i$, Type-$2ii$, and Type-$2iii$, respectively.

All Type-$2i$ and Type-$2iii$ 4-cuts can be computed explicitly in linear time, and their total number is $O(n)$. On the other hand, the number of Type-$2ii$ 4-cuts can be as high as $\Omega(n^2)$, and so we cannot compute all of them in linear time. Instead, we compute only a collection of $O(n)$ Type-$2ii$ 4-cuts, so that the rest of them are implied from this collection. The Type-$2ii$ 4-cuts are particularly interesting, because their existence is basically the reason that we can have $\Omega(n^2)$ 4-cuts of Type-2 and Type-3. More precisely, we show that every Type-3 4-cut that we have not explicitly computed, is implied by the collection of Type-3 4-cuts that we have computed, plus that of the Type-$2ii$ 4-cuts that we have computed.

First, let us consider the Type-$2i$ 4-cuts. So let $\{(u, p(u)), (v, p(v)), e_1, e_2\}$ be a 4-cut such that u is a descendant of v and $B(v) = B(u) \sqcup \{e_1, e_2\}$. Then, Lemma 5.30 shows that there are basically three different cases for the back-edges e_1 and e_2. That is, e_1 and e_1 are either (1) the first and second leftmost edges of v, or (2) the first leftmost and rightmost edges of v, or (3) the first and the second rightmost edges of v. In either case, by Lemma 5.31 we have that u is uniquely determined by v and e_1, e_2: that is, u is the lowest proper descendant of v that has $M(u) = M(B(v) \setminus \{e_1, e_2\})$.

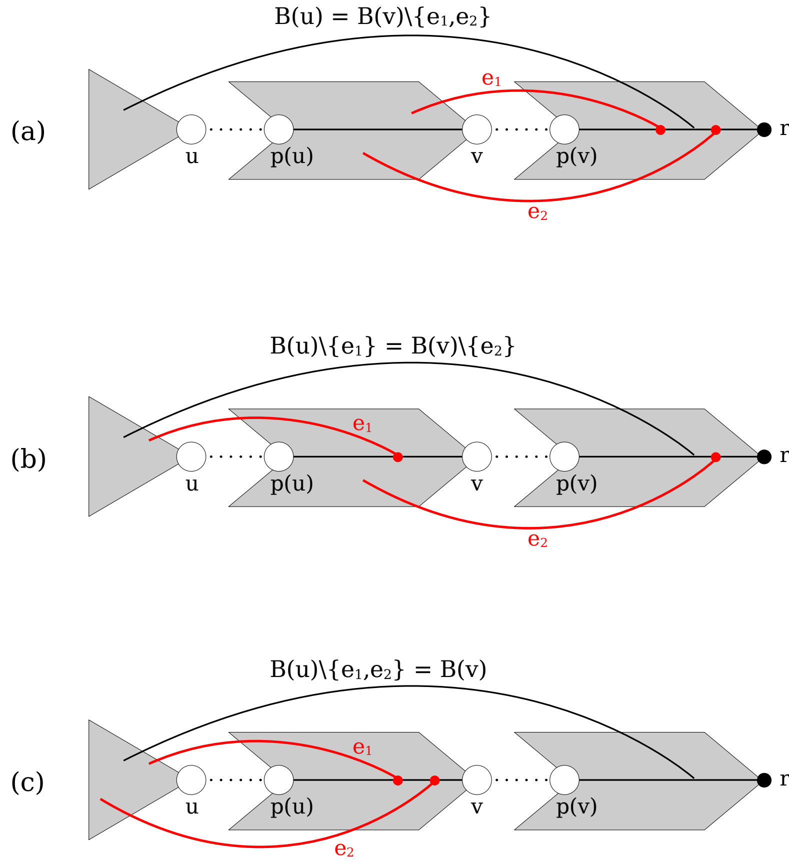


Figure 5.17: All different cases for Type-2 4-cut of the form $\{(u, p(u)), (v, p(v)), e_1, e_2\}$, where u is a descendant of v. In (a) we have $B(v) = B(u) \sqcup \{e_1, e_2\}$. In (b) we have $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$. In (c) we have $B(v) \sqcup \{e_1, e_2\} = B(u)$.

Thus, the idea is basically to compute all three different values $M(B(v) \setminus \{e_1, e_2\})$, for all different cases (1), (2) and (3). Then, we can precisely determine u in every one of those cases, and then we apply Lemma 5.29 in order to verify that we indeed have a 4-cut. Thus, by Proposition 5.11 we have that we can compute all Type-$2i$ 4-cuts in linear time. Notice that their number is $O(n)$.

Now let us consider the Type-$2iii$ 4-cuts. So let $\{(u, p(u)), (v, p(v)), e_1, e_2\}$ be a 4-cut such that u is a descendant of v and $B(u) = B(v) \sqcup \{e_1, e_2\}$. Then Lemma 5.39 shows that e_1 and e_2 are completely determined by u: i.e., these are the $high_1$ and $high_2$ edges of u. Then, by Lemma 5.40 we have that v is either the greatest or the second-greatest proper ancestor of u with $M(v) = M(B(u) \setminus \{e_1, e_2\})$. This shows that the number of Type-$2iii$ 4-cuts is $O(n)$. By Proposition 5.13, we can compute all of

them in linear time in total.

Finally, let us consider the Type-$2ii$ 4-cuts. So let $\{(u,p(u)),(v,p(v)),e_1,e_2\}$ be a 4-cut such that u is a descendant of v and $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$. Then, Lemma 5.34 shows that e_1 is the *high*$_1$ edge of u, and e_2 is either the first leftmost or the first rightmost edge of v. Thus, there are two different cases to consider for the back-edge in $B(v) \setminus B(u)$. Let us assume that we have fixed a case for the back-edge $e \in B(v) \setminus B(u)$ (e.g., let e be the first leftmost edge of v). As we can see in Figure 5.18, the number of proper descendants u of v with the property that $B(v) \sqcup \{e'\} = B(u) \sqcup \{e\}$ can be $\Omega(n)$, and this can be true for $\Omega(n)$ vertices v. Thus, the idea is to properly select one such vertex u, for every vertex v, for every one of the two choices for the back-edge e. Thus, we compute $O(n)$ Type-$2ii$ 4-cuts in total. Furthermore, we can show that one such selection, for every v and e, is enough to produce a collection of Type-$2ii$ 4-cuts that implies all Type-$2ii$ 4-cuts. We denote the vertex u that we select as *lowestU*(v,e). As its name suggests, this is the lowest u that has the property that u is a proper descendant of v and there is a back-edge e' such that $B(v) \sqcup \{e'\} = B(u) \sqcup \{e\}$. The reason for selecting this vertex, is that it is convenient to compute. Specifically, for technical reasons we distinguish two cases: either $M(u) = M(B(u) \setminus \{e_{high}(u)\})$, or $M(u) \neq M(B(u) \setminus \{e_{high}(u)\})$. In the first case, by Lemma 5.37 we have that u is either the lowest or the second-lowest proper descendant of v such that $M(u) = M(B(v) \setminus \{e\})$. In the second case, by Lemma 5.38 we have that u is the lowest proper descendant of v such that $M(u) \neq M(B(u) \setminus \{e_{high}(u)\}) = M(B(v) \setminus \{e\})$. With this information, Proposition 5.12 establishes that we can compute in linear time a collection $\mathcal{C}$ of $O(n)$ Type-$2ii$ 4-cuts that implies all Type-$2ii$ 4-cuts. More precisely, every Type-$2ii$ 4-cut of the form $\{(u,p(u)),(v,p(v)),e_1,e_2\}$, where $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$, is implied by $\mathcal{C}$ through $\{(u,p(u)),e_1\}$ (or equivalently, through $\{(v,p(v)),e_2\}$). This is an important property that allows us also to compute a collection of $O(n)$ Type-3 4-cuts, that, together with $\mathcal{C}$, implies all Type-3 4-cuts.

5.5.3 Type-3 4-cuts

Let C be a Type-3 4-cut, and let $(u,p(u))$, $(v,p(v))$ and $(w,p(w))$ be the tree-edges in C. We may assume w.l.o.g. that $u > v > w$. Then, Lemma 3.14 implies that w is a common ancestor of u and v. Thus, we distinguish two cases: either (1) u and v are

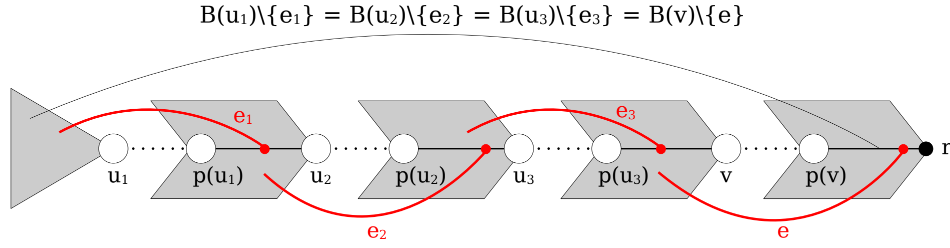


Figure 5.18: With this figure we can see why there can be $\Omega(n^2)$ Type-2ii 4-cuts in a graph with n vertices. Any of the pairs of edges $\{(v,p(v)),e\}$, $\{(u_1,p(u_1)),e_1\}$, $\{(u_2,p(u_2)),e_2\}$ and $\{(u_3,p(u_3)),e_3\}$ forms a 4-cut with any of the rest. For example, $\{(u_1,p(u_1)),(v,p(v)),e_1,e\}$ and $\{(u_2,p(u_2)),(u_3,p(u_3)),e_2,e_3\}$ are two 4-cuts in this figure.

not related as ancestor and descendant, or (2) u and v are related as ancestor and descendant. In case (1), we call C a Type-3α 4-cut. In case (2), we call C a Type-3β 4-cut. Both of these cases are much more involved than the case of Type-2 4-cuts, but the case of Type-3β 4-cuts is the most challenging.

5.5.3.1 Type-3α 4-cuts

Let $C = \{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ be a Type-3α 4-cut, where w is a common ancestor of u and v. Then Lemma 5.41 implies that either (i) $e \in B(u) \cup B(v)$ and $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$, or (ii) $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ (see Figure 5.19). In case (i), C is called a Type-3αi 4-cut. In case (ii), C is called a Type-3αii 4-cut. We treat those cases differently.

Type-3αi 4-cuts

Let $C = \{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ be a Type-3αi 4-cut, where w is a common ancestor of u and v. Then we have $e \in B(u) \cup B(v)$ and $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$. We may assume w.l.o.g. that $e \in B(u)$. Then Lemma 5.43 implies that $e = e_{high}(u)$. By Lemma 5.42 we have that one of u and v is a descendant of the *low1* child of $M(w)$, and the other is a descendant of the *low2* child of $M(w)$. Either of those cases can be true, regardless of whether e is in $B(u)$ or $B(v)$. So let us assume that u is a descendant of the *low1* child c_1 of $M(w)$, and v is a descendant of the *low2* child c_2 of $M(w)$. Then Lemma 5.44 implies that v is the lowest proper descendant of w with

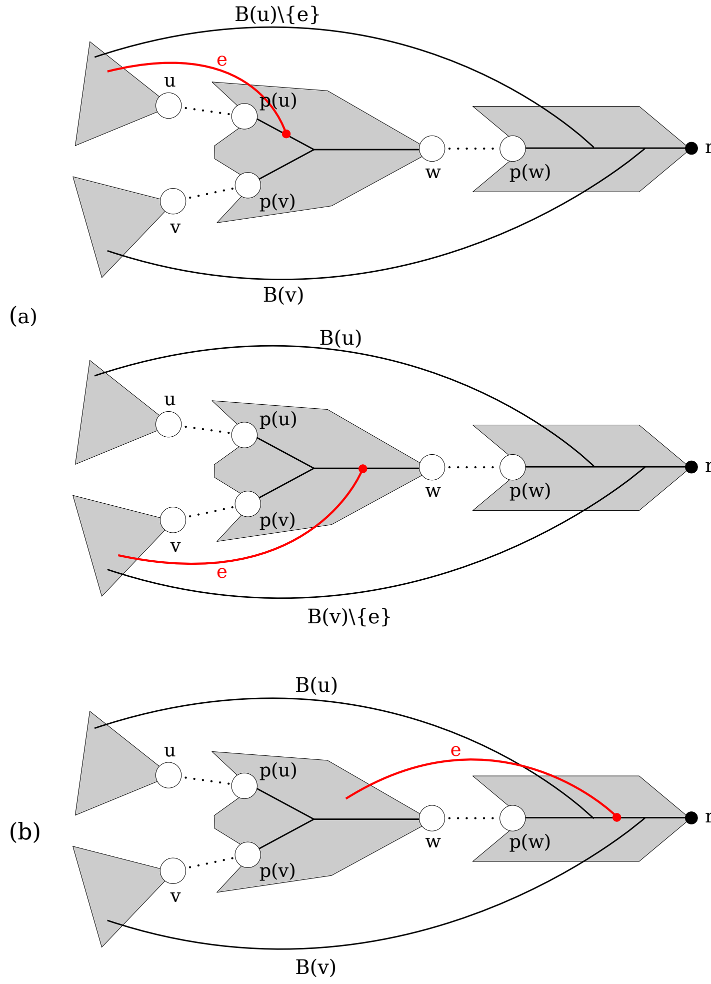


Figure 5.19: The two different cases of a Type-3α 4-cut $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$. In (a) we have $e \in B(u) \cup B(v)$ and $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$. Although this implies that e is either in $B(u)$ or in $B(v)$, these cases are symmetric. In (b) we have $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$.

$M(v) = M(w, c_2)$.

Concerning the higher endpoint of e, we distinguish two cases, depending on whether $M(u) = M(B(u) \setminus \{e_{high}(u)\})$ or $M(u) \neq M(B(u) \setminus \{e_{high}(u)\})$. In the first case, we can compute all such 4-cuts in linear time, because Lemma 5.45 implies that u is either the lowest or the second-lowest proper descendant of w such that $M(u) = M(w, c_1)$. Thus, given w, we have that v is completely determined, and there are only two options for u. This implies that the number of those 4-cuts is $O(n)$, and by Proposition 5.14 we can compute all of them in linear time in total.

In the case where $M(u) \neq M(B(u) \setminus \{e_{high}(u)\})$, the number of 4-cuts can be $\Omega(n^2)$, as shown in Figure 5.20. This is because, although for fixed w we have that v is

determined, there may be $\Omega(n)$ options for u, and this may be true for $\Omega(n)$ vertices w. Then the idea is basically the same as that for computing the Type-$2ii$ 4-cuts: we only select a proper u for every w. Specifically, we select the lowest proper descendant u of w such that $M(u) \neq M(B(u) \setminus \{e_{high}(u)\}) = M(w, c_1)$. It turns out that this is sufficient, in the sense that the 4-cuts of this type that we compute, are able to imply, together with the collection of Type-$2ii$ 4-cuts that we have computed, all 4-cuts of this type. This result is given in Proposition 5.15.

After this procedure, we may simply reverse the roles of u and v. Thus, we assume that the descendant u of w that has $e \in B(u)$ is a descendant of the *low2* child of $M(w)$, and v is a descendant of the *low1* child of $M(w)$. Then we follow a similar procedure to compute all 4-cuts of this type. The arguments are essentially the same.

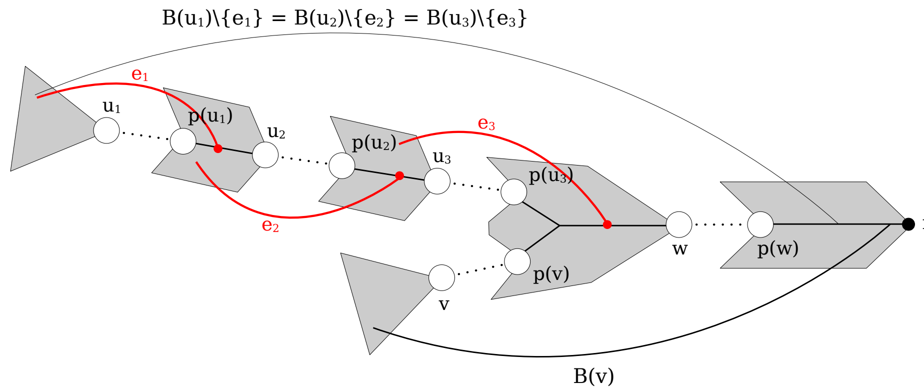


Figure 5.20: With this figure we can see why the number of Type-$3\alpha i$ 4-cuts can be $\Omega(n^2)$. For a particular w, there may be a sequence $(u_1, v), (u_2, v), \dots$ of $\Omega(n)$ pairs of vertices, and a corresponding sequence of back-edges $e_1, e_2, \dots$, such that $e_i \in B(u_i)$ and $B(w) = (B(u_i) \setminus \{e_i\}) \sqcup B(v)$, for every $i = 1, 2, \dots$ – and this can be true for $\Omega(n)$ vertices w. In this example, we have that $C_i = \{(u_i, p(u_i)), (v, p(v)), (w, p(w)), e_i\}$ is a 4-cut, for every $i \in \{1, 2, 3\}$. We have $e_i = e_{high}(u_i)$ for every $i \in \{1, 2, 3\}$, $M(u_2) \in T(u_2) \setminus T(u_1)$, $M(u_3) \in T(u_3) \setminus T(u_2)$, and $M(B(u_3)\setminus\{e_3\}) = M(B(u_2)\setminus\{e_2\}) = M(B(u_1)\setminus\{e_1\}) \in T(u_1)$. This implies that $M(u_3) \neq M(B(u_3) \setminus \{e_{high}(u_3)\})$ and $M(u_2) \neq M(B(u_2) \setminus \{e_{high}(u_2)\})$. Notice that it is enough to have computed the collection $\mathcal{C} = \{\{(u_1, p(u_1)), (u_2, p(u_2)), e_1, e_2\}, \{(u_2, p(u_2)), (u_3, p(u_3)), e_2, e_3\}\}$ of Type-$2ii$ 4-cuts, and the 4-cut C_3. Then, C_1 and C_2 are implied from $\mathcal{C} \cup \{C_3\}$.

Type-$3\alpha ii$ 4-cuts

Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-$3\alpha ii$ 4-cut, where w is a common

ancestor of u and v. Then we have $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$. Let $e = (x, y)$. Then there are various cases to consider, according to the relation of x with u and v. Specifically, by Lemma 5.50 we have the following cases (see Figure 5.21 for cases (1)-(3), and Figure 5.22 for case (4)).

(1) x is an ancestor of both u and v.

(2) x is an ancestor of u, but not an ancestor of v.

(3) x is an ancestor of v, but not an ancestor of u.

(4) x is neither an ancestor of u nor an ancestor of v.

In any case, Lemma 5.51 implies that u and v are uniquely determined by w, and therefore the number of all Type-$3\alpha ii$ 4-cuts is $O(n)$. Furthermore, in any case we have $y = l(x)$.

Now, in case (1), by Lemma 5.53 we have that $x = M(w)$, one of u and v is a descendant of the *low1* child c_1 of $\widetilde{M}(w)$, and the other is a descendant of the *low2* child c_2 of $\widetilde{M}(w)$. Since these cases are symmetric, we may assume w.l.o.g. that u is a descendant of c_1 and v is a descendant of c_2. Then Lemma 5.53 implies that $M(u) = M(w, c_1)$ and $M(v) = M(w, c_2)$. Then, by Lemma 5.51 we can determine precisely u and v. Thus, by Proposition 5.16 we can compute all those 4-cuts in linear time in total.

Notice that cases (2) and (3) are essentially the same (to see this, just switch the labels of u and v). So let us consider case (2). Then by Lemma 5.50 we have that one of x and v is a descendant of the *low1* child c_1 of $M(w)$, and the other is a descendant of the *low2* child c_2 of $M(w)$. Thus, w.l.o.g. we may assume that x is a descendant of c_1 and v is a descendant of c_2. Then Lemma 5.54 implies that $x = M(w, c_1)$, $M(u) = M(w, c_1')$ and $M(v) = M(w, c_2)$, where c_1' is the *low1* child of $M(w, c_1)$. Then, by Lemma 5.51 we can determine precisely u and v. Thus, by Proposition 5.17 we can compute all those 4-cuts in linear time in total.

Now let us consider case (4). According to Lemma 5.50, this case is further subdivided into the following two cases (see Figure 5.22).

(4.1) Two of $\{u, v, x\}$ are descendants of the *low1* child of $M(w)$ and the other is a descendant of the *low2* child of $M(w)$, or reversely: two of $\{u, v, x\}$ are descendants of the *low2* child of $M(w)$ and the other is a descendant of the *low1* child of $M(w)$.

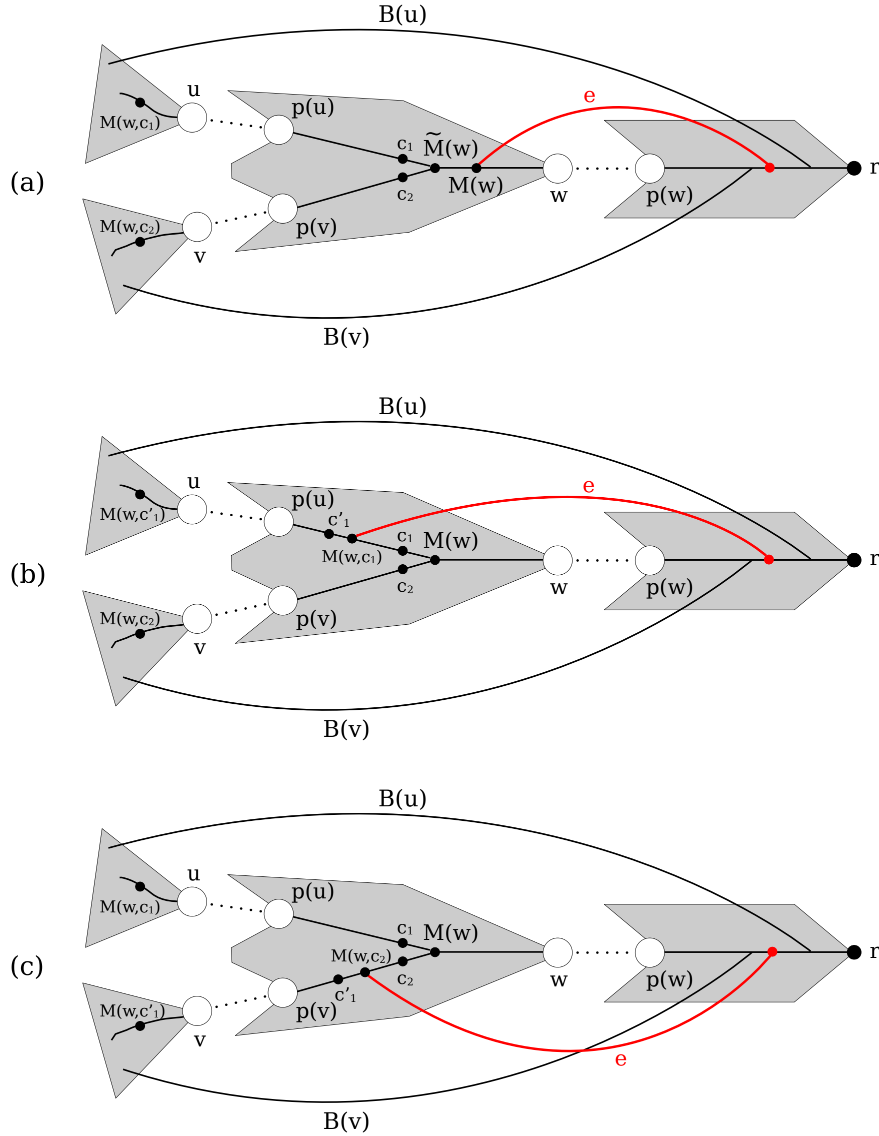


Figure 5.21: (a)-(c) correspond to cases (1)-(3) of Lemma 5.50. These are the cases in which the higher endpoint of e is related as ancestor and descendant with either u or v. In (a), the higher endpoint of e is an ancestor of both u and v. c_1 and c_2 are the *low1* and *low2* children of $\widetilde{M}(w)$ (not necessarily in that order). We have $M(u) = M(w, c_1)$ and $M(v) = M(w, c_2)$. In (b), the higher endpoint of e is an ancestor of u, but not of v. c_1 and c_2 are the *low1* and *low2* children of $M(w)$ (not necessarily in that order). We have $M(u) = M(w, c'_1)$ and $M(v) = M(w, c_2)$, where c'_1 is the *low1* child of $M(w, c_1)$. In (c), the higher endpoint of e is an ancestor of v, but not of u. We note that cases (b) and (c) are essentially equivalent; to see this, just switch the labels of u and v.

(4.2) There is a permutation σ of $\{1, 2, 3\}$ such that u is a descendant of the *low*σ_1 child of $M(w)$, v is a descendant of the *low*σ_2 child of $M(w)$, and x is a

descendant of the *low*σ_3 child of $M(w)$.

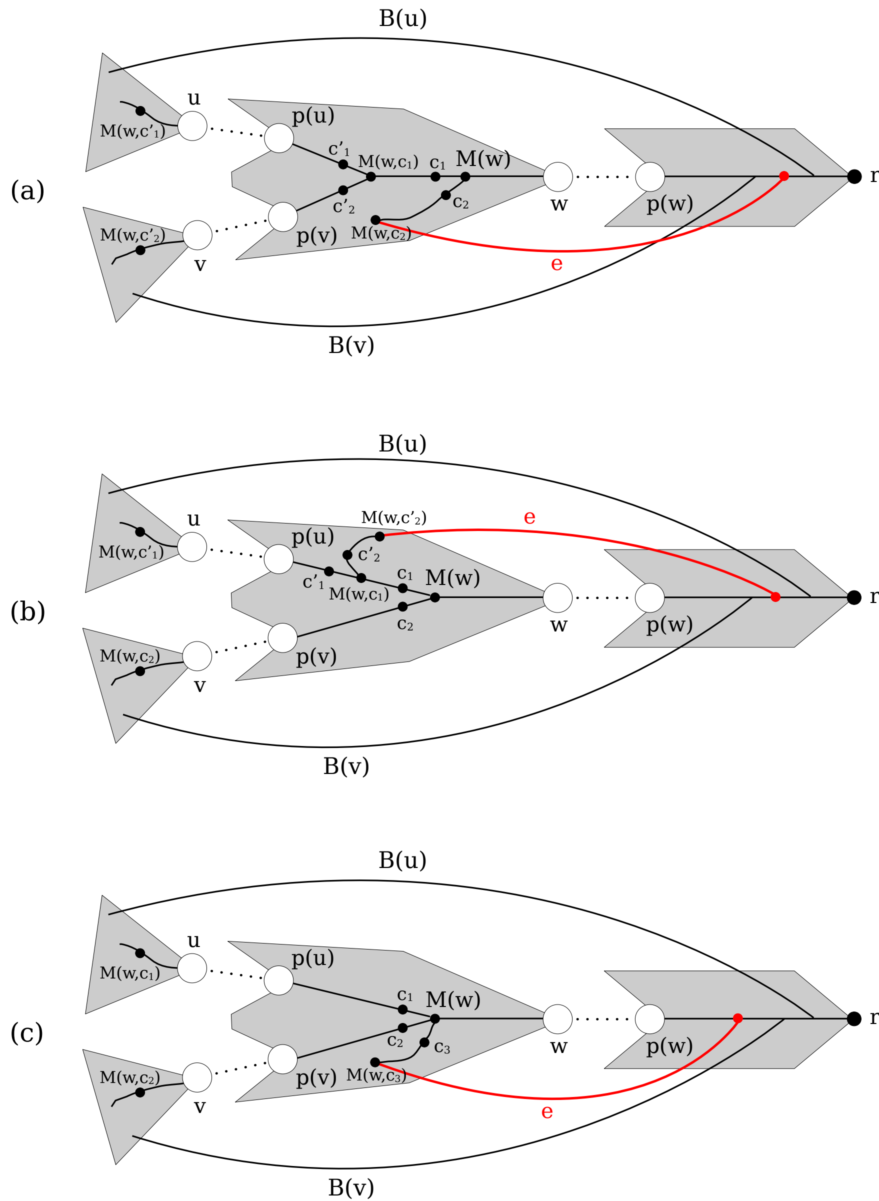


Figure 5.22: (a) and (b) correspond to case (4.1) of Lemma 5.50, and (c) corresponds to case (4.2) of Lemma 5.50. These are the cases in which the higher endpoint of e is not related as ancestor and descendant with u and v. In (a), we get all the different possibilities for case $(4.1.1)$ of Lemma 5.55 by swapping the labels c_1', c_2'. In (b), we get all the different possibilities for cases $(4.1.2)$ and $(4.1.3)$ of Lemma 5.55 by swapping the labels c_1, c_2 and c_1', c_2'. In (c), we get all the different possibilities for case (4.2) of Lemma 5.50 by permuting the labels c_1, c_2 and c_3.

Let us consider case (4.1) first. Let c_1 be the *low1* child of $M(w)$, and let c_2 be the

low2 child of $M(w)$. Let us assume that two of $\{u, v, x\}$ are descendants of c_1, and the other is a descendant of c_2. (The reverse case is treated similarly.) Now let c'_1 be the *low1* child of $M(w, c_1)$, and let c'_2 be the *low2* child of $M(w, c_1)$. Then Lemma 5.55 implies that we have the following three subcases.

(4.1.1) u and v are descendants of c_1, and x is a descendant of c_2.

(4.1.2) u and x are descendants of c_1, and v is a descendant of c_2.

(4.1.3) v and x are descendants of c_1, and u is a descendant of c_2.

In case (4.1.1) we have $M(u) = M(w, c'_1)$ and $M(v) = M(w, c'_2)$ (or reversely), and $x = M(w, c_2)$. In case (4.1.2) we have $M(u) = M(w, c'_1)$ and $x = M(w, c'_2)$ (or reversely), and $M(v) = M(w, c_2)$. And in case (4.1.3) we have $M(v) = M(w, c'_1)$ and $x = M(w, c'_2)$ (or reversely), and $M(u) = M(w, c_2)$.

Thus, we have to consider all the different possibilities (which are $O(1)$ in total), in order to find all 4-cuts of this type. In either case, by Lemma 5.51 we can determine precisely u and v. Thus, by Proposition 5.18 we can compute all those 4-cuts in linear time in total.

Finally, let us consider case (4.2). Let c_1, c_2 and c_3 be the *low1*, *low2* and *low3* children of $M(w)$, respectively. Thus, there is a permutation σ of $\{1, 2, 3\}$ such that u is a descendant of $c_{\sigma(1)}$, v is a descendant of $c_{\sigma(2)}$ and x is a descendant of $c_{\sigma(3)}$. By Lemma 5.56 we have that $M(u) = M(w, c_{\sigma(1)})$, $M(v) = M(w, c_{\sigma(2)})$ and $x = M(w, c_{\sigma(3)})$. Thus, we consider all the different combinations for σ, and in each case we can determine precisely u and v by Lemma 5.51. Proposition 5.19 establishes that we can compute all those 4-cuts in linear time in total.

5.5.3.2 Type-3β 4-cuts

Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3β 4-cut of G, where u is a descendant of v, and v is a descendant of w. Then Lemma 5.2 implies that e is the unique back-edge with the property that $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ is a 4-cut of G. Thus, we say that (u, v, w) *induces* the 4-cut C. Whenever we say that a triple of vertices (u, v, w) induces a 4-cut, we always assume that u is a proper descendant of v, and v is a proper descendant of w.

Since C is a Type-3β 4-cut, Lemma 5.57 implies that there are four distinct cases to consider (see Figure 5.23):

(1) $e \in B(u) \cap B(v) \cap B(w)$ and $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$.

(2) $e \in B(w)$, $e \notin B(v) \cup B(u)$, and $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$.

(3) $e \in B(u)$, $e \notin B(v) \cup B(w)$, and $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$.

(4) $e \in B(v)$ and $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$.

For technical reasons, we make a distinction into Type-$3\beta i$ and Type-$3\beta ii$ 4-cuts.

In case (1) we have $B(w) \setminus \{e\} \subset B(v) \setminus \{e\}$, and therefore $M(B(w) \setminus \{e\})$ is a descendant of $M(B(v) \setminus \{e\})$. If $M(B(w) \setminus \{e\}) \neq M(B(v) \setminus \{e\})$, then we say that C is a Type-$3\beta i$ 4-cut. Otherwise, we say that C is a Type-$3\beta ii$ 4-cut.

In case (2) we have $B(w) \setminus \{e\} \subset B(v)$, and therefore $M(B(w) \setminus \{e\})$ is a descendant of $M(v)$. If $M(B(w) \setminus \{e\}) \neq M(v)$, then we say that C is a Type-$3\beta i$ 4-cut. Otherwise, we say that C is a Type-$3\beta ii$ 4-cut.

In case (3) we have $B(w) \subset B(v)$, and therefore $M(w)$ is a descendant of $M(v)$. If $M(w) \neq M(v)$, then we say that C is a Type-$3\beta i$ 4-cut. Otherwise, we say that C is a Type-$3\beta ii$ 4-cut.

In case (4) we have $B(w) \subset B(v) \setminus \{e\}$, and therefore $M(w)$ is a descendant of $M(B(v) \setminus \{e\})$. If $M(w) \neq M(B(v) \setminus \{e\})$, then we say that C is a Type-$3\beta i$ 4-cut. Otherwise, we say that C is a Type-$3\beta ii$ 4-cut.

In each of those cases, the Type-$3\beta i$ 4-cuts are easier to compute than the respective Type-$3\beta ii$ 4-cuts. This is because we have more information in order to determine (some possible values of) u and w given v. More specifically, given v, we have that one of u and w is completely determined, and only the other may vary, but only in a very orderly manner. For Type-$3\beta ii$ 4-cuts, many possible combinations of pairs u and w may exist, given v, and this makes things much more complicated.

When we consider case (4) of Lemma 5.57, in either Type-$3\beta i$ or Type-$3\beta ii$ 4-cuts, we perceive a distinct difficulty (which is significantly more involved for Type-$3\beta ii$ 4-cuts). This is because the back-edge e leaps over v, which is "between" u and w (i.e., v is an ancestor of u, but also a descendant of w). This forces us to distinguish several subcases, by considering the different possibilities for the higher or the lower endpoint of e. (Whereas, in the previous cases, we have some predetermined options for e, which then allow us to compute either u or w.) In some of those subcases, we cannot even identify beforehand the endpoints of e, and we can only retrieve them after having first computed both u and w (see Lemma 5.67).

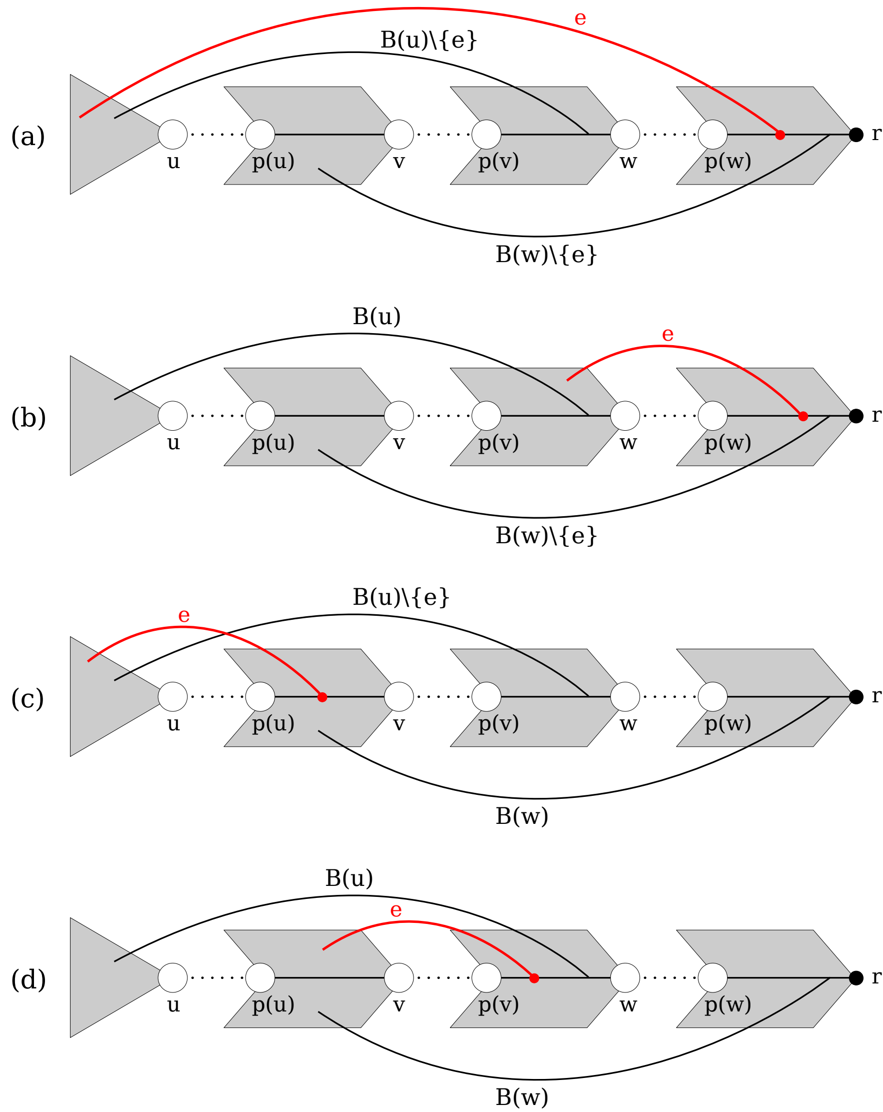


Figure 5.23: All different cases of Type-3β 4-cuts. (a)-(d) correspond to cases (1)-(4) of Lemma 5.57. In (a) we have $e \in B(u) \cap B(v) \cap B(w)$ and $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$. In (b) we have $e \in B(w)$, $e \notin B(v) \cup B(u)$, and $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$. In (c) we have $e \in B(u)$, $e \notin B(v) \cup B(w)$, and $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$. In (d) we have $e \in B(v)$ and $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$.

Type-$3\beta i$ 4-cuts

First, let us consider case (1) of Lemma 5.57. Then by Lemma 5.58 we have that u is the lowest proper descendant of v with $M(u) = M(v, c)$, where c is either the *low1* or the *low2* child of $M(v)$. Furthermore, we have that e is the *low* edge

of u, and $M(w) = M(v)$. Also, w satisfies $bcount(w) = bcount(v) - bcount(u) + 1$. Thus, since $M(w) = M(v)$, we have that w is completely determined by this property (because the vertices with the same M point have distinct *bcount*). Then, notice that the number of those 4-cuts is $O(n)$ (because u and w are completely determined by v). Proposition 5.20 shows that we can compute all of them in linear time in total.

Now let us consider case (2) of Lemma 5.57. The number of those 4-cuts can be $\Omega(n^2)$, as shown in Figure 5.24, and so we only compute a subcollection of them, that, together with the collection of Type-$2ii$ 4-cuts that we have computed, implies all 4-cuts of this kind. Specifically, let c_1 and c_2 be the *low1* and *low2* child of $M(v)$, respectively. Then by Lemma 5.60 we have that u is the lowest proper descendant of v such that $M(u) = M(v, c_2)$. Also, we have that w is an ancestor of $low(u)$ and $M(w) \neq M(B(w) \setminus \{e\}) = M(v, c_1)$. Then, Lemma 5.61 implies that it is sufficient to have computed the greatest ancestor w' of $low(u)$ for which there is a back-edge $e' \in B(w')$ such that $M(w') \neq M(B(w') \setminus \{e'\}) = M(v, c_1)$. Thus, for every vertex v, there is only a specific pair of vertices u and w that we have to check, and so the number of 4-cuts that we will collect is $O(n)$. Proposition 5.21 establishes that this collection, plus that of the Type-$2ii$ 4-cuts that we have computed, is enough in order to imply all 4-cuts of this kind. Furthermore, we can compute all of them in linear time in total.

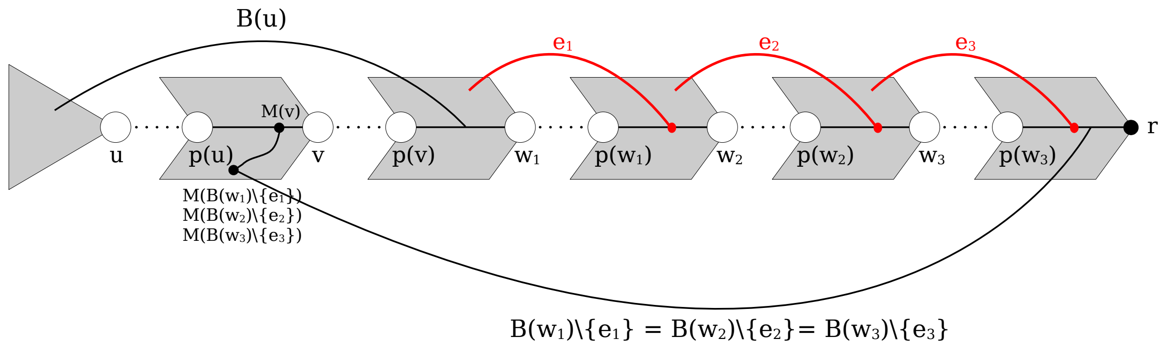


Figure 5.24: Here we have that $\{(u, p(u)), (v, p(v)), (w_i, p(w_i)), e_i\}$ is a 4-cut, for every $i \in \{1, 2, 3\}$. This example shows why the number of Type-$3\beta i$ 4-cuts that satisfy (2) of Lemma 5.57 can be $\Omega(n^2)$. For a particular v, we can have $\Omega(n)$ vertices w such that $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$, for a vertex u and a back-edge e, and this can be true for $\Omega(n)$ vertices v. However, notice that it is sufficient to have computed only $\{(u, p(u)), (v, p(v)), (w_1, p(w_1)), e_1\}$ and the Type-$2ii$ 4-cuts $\{(w_1, p(w_1)), (w_2, p(w_2)), e_1, e_2\}$ and $\{(w_2, p(w_2)), (w_3, p(w_3)), e_2, e_3\}$, because the remaining 4-cuts are implied from this selection.

Now let us consider case (3) of Lemma 5.57. Let c_1 and c_2 be the *low1* and the *low2* child of $M(v)$, respectively. Then by Lemma 5.63 we have that w is the greatest proper ancestor of v such that $M(w) = M(v, c_1)$. Here we distinguish two cases for u, depending on whether $M(u) = M(B(u) \setminus \{e\})$, or $M(u) \neq M(B(u) \setminus \{e\})$. In any case, by Lemma 5.63 we have that e is the *high* edge of u. Now, in the first case, we can compute all such 4-cuts explicitly. This is because by Lemma 5.63 we have that $M(B(u) \setminus \{e_{high}(u)\}) = M(v, c_2)$, and by Lemma 5.64 we have that u is either the lowest or the second-lowest proper descendant of v with this property. Thus, there are only $O(n)$ 4-cuts in this case (because, given v, there is only one candidate w, and at most two candidates u). In the case $M(u) \neq M(B(u) \setminus \{e\})$, the number of 4-cuts can be $\Omega(n^2)$, as shown in Figure 5.25. However, it is sufficient to consider only the lowest proper descendant u' of v that satisfies $M(u') \neq M(B(u') \setminus \{e_{high}(u')\}) = M(v, c_2)$, according to Lemma 5.65. Thus, in any case, we compute $O(n)$ 4-cuts in total, and Proposition 5.22 establishes that these, together with the collection of Type-$2ii$ 4-cuts that we have computed, are enough in order to imply all 4-cuts of this kind. Furthermore, this computation can be performed in linear time in total.

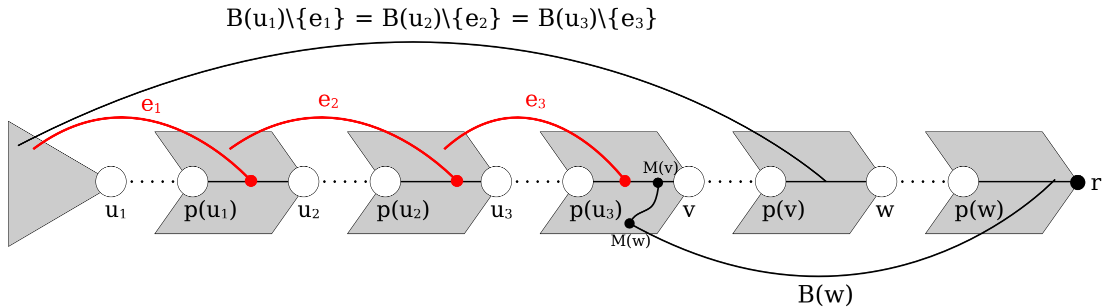


Figure 5.25: Here we have that $\{(u_i, p(u_i)), (v, p(v)), (w, p(w)), e_i\}$ is a 4-cut, for every $i \in \{1, 2, 3\}$. This example shows why the number of Type-$3\beta i$ 4-cuts that satisfy (3) of Lemma 5.57 can be $\Omega(n^2)$. For a particular v, we can have $\Omega(n)$ vertices u such that $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$, for a vertex w and a back-edge e, and this can be true for $\Omega(n)$ vertices v. However, notice that it is sufficient to have computed only $\{(u_3, p(u_3)), (v, p(v)), (w, p(w)), e_3\}$ and the Type-$2ii$ 4-cuts $\{(u_1, p(u_1)), (u_2, p(u_2)), e_1, e_2\}$ and $\{(u_2, p(u_2)), (u_3, p(u_3)), e_2, e_3\}$, because the remaining 4-cuts are implied from this selection. Notice that $e_i = e_{high}(u_i)$, for $i \in \{1, 2, 3\}$, $M(u_2) \in T(u_2) \setminus T(u_1)$, $M(u_3) \in T(u_3) \setminus T(u_2)$, and $M(B(u_1) \setminus \{e_1\}) = M(B(u_2) \setminus \{e_2\}) = M(B(u_3) \setminus \{e_3\})$.

Finally, let us consider case (4) of Lemma 5.57. This branches into several subcases. First, we distinguish two cases, depending on whether $M(v) \neq M(B(v) \setminus \{e\})$ or

$M(v) = M(B(v) \setminus \{e\})$. In the first case, by Lemma 3.9 we have that e is either $e_L(v)$ or $e_R(v)$. Then, by Lemma 5.68, we know precisely u and w: that is, u is the lowest proper descendant of v such that $M(u) = M(v, c_2)$, and w is the greatest proper ancestor of v such that $M(w) = M(v, c_1)$, where c_1 and c_2 are the *low1* and *low2* children of $M(B(v) \setminus \{e\})$, respectively. Thus, the number of all 4-cuts of this kind is $O(n)$, and Proposition 5.23 shows that we can compute all of them in linear time in total. The case that $M(v) = M(B(v) \setminus \{e\})$ is more involved, because we cannot immediately determine the back-edge e. Thus, we first determine u and w according to Lemma 5.69 and Lemma 5.70, by considering all the different cases of Lemma 5.69. Notice that the total number of pairs of u and w that we check are $O(1)$ for a given v. Then e can be determined by Lemma 5.67. Thus, the number of all such 4-cuts is $O(n)$, and Proposition 5.24 shows that we can compute all of them in linear time in total.

Type-$3\beta ii$ 4-cuts

In the case of Type-$3\beta ii$ 4-cuts, given a vertex v, there may be many pairs of u and w such that (u, v, w) induces a 4-cut, for any of the cases (1)-(4) of Lemma 5.57. For all those cases, we follow a common strategy. First, we define a set $U(v)$, for every vertex v, that contains some candidates u with the property that there may exist a w such that (u, v, w) induces a 4-cut of the type we consider. Then, for every $u \in U(v)$, we determine a w (if it exists) such that (u, v, w) induces a 4-cut. We make sure that the selection of 4-cuts that we have computed in each case is enough to imply, together with the collection of Type-$2ii$ 4-cuts that we have computed, all 4-cuts of the kind that we consider. Since the general strategy is the same, there are a lot of similarities in all those cases on a high level. In particular, the sets $U(v)$ that we define in each particular case have similar definitions, satisfy similar properties, and can be computed with similar methods. However, each particular case presents unique challenges, and demands special care in order to ensure correctness. Thus, we distinguish between Type-$3\beta ii$-1, Type-$3\beta ii$-2, Type-$3\beta ii$-3 and Type-$3\beta ii$-4 4-cuts, depending on whether they satisfy (1), (2), (3) or (4) of Lemma 5.57, respectively.

Type-$3\beta ii$-1 4-cuts

For the case of Type-$3\beta ii$-1 4-cuts, we define the set $U_1(v)$, for every vertex $v \neq r$, as a segment of $H(\mathit{high}_1(v))$ that consists of the proper descendants of v that have low enough *low* point in order to be possible to participate in a triple of vertices (u, v, w) that induces a Type-$3\beta ii$-1 4-cut. The sets $U_1(v)$ have total size $O(n)$, and they have the property that, if there is a w such that (u, v, w) induces a Type-$3\beta ii$-1 4-cut, then $u \in U_1(v)$ (Lemma 5.73). By Lemma 5.76, we can compute all sets $U_1(v)$ in linear time in total. Given v and $u \in U_1(v)$, by Lemma 5.78 we have that there is at most one w such that (u, v, w) induces a Type-$3\beta ii$-1 4-cut: i.e., w is the greatest proper ancestor of v with $M(w) = M(v)$ and $w \leq \mathit{low}_2(u)$. Thus, the number of all Type-$3\beta ii$-1 4-cuts is $O(n)$. Proposition 5.25 shows that we can compute all of them in linear time in total.

Type-$3\beta ii$-2 4-cuts

According to Lemma 5.79, if a triple of vertices (u, v, w) induces a Type-3βii-2 4-cut, then the back-edge e of this 4-cut is either $e_L(w)$ or $e_R(w)$. Here we discuss the case where $e = e_L(w)$. The arguments for the case $e = e_R(w)$ are similar. For convenience, we distinguish two cases, depending on whether $L_1(w)$ is a descendant of $\mathit{high}(v)$.

First, we consider the case that $L_1(w)$ is not a descendant of $\mathit{high}(v)$. The number of 4-cuts of this kind can be $\Omega(n^2)$, and so we do not compute all of them explicitly. Instead, we compute a subcollection of $O(n)$ of them with the property that, together with the collection of Type-$2ii$ 4-cuts that we have computed, it implies all 4-cuts of this kind. To do this, we define two sets, $W(v)$ and $U_2(v)$, for every vertex $v \neq r$. The set $W(v)$ contains all candidates w with the property that there may exist a u such that (u, v, w) induces a Type-$3\beta ii$-2 4-cut, and $U_2(v)$ contains all candidates u with the property that there may exist a w such that (u, v, w) induces a Type-$3\beta ii$-2 4-cut (see Lemma 5.84). We do not explicitly compute the sets $W(v)$, but only the greatest and the lowest vertices that are contained in them (denoted as $\mathit{firstW}(v)$ and $\mathit{lastW}(v)$, respectively). The sets $U_2(v)$ have total size $O(n)$, and by Lemma 5.88 we can compute all of them in linear time in total. Then, given v and $u \in U_2(v)$, it is sufficient to compute the greatest w such that (u, v, w) induces a Type-$3\beta ii$-2 4-cut, according to Lemma 5.85. Thus, we compute a collection of $O(n)$ 4-cuts of this kind, which, together with the collection of Type-$2ii$ 4-cuts that we have computed, implies

all 4-cuts of this kind. This result is summarized in Proposition 5.26.

Now we consider the case where $L_1(w)$ is a descendant of $high(v)$. By Lemma 5.90 we have that w is uniquely determined by u and v. Lemma 5.90 motivates the definition of the sets $\widetilde{W}(v)$, that contain all possible candidates w with the property that there may exist a u such that (u, v, w) induces a 4-cut of this kind. By Lemma 5.91 we have that the total size of those sets is $O(n)$. By Lemma 5.93, we can compute all of them in linear time in total. Then, by Lemma 5.94 we have that, if (u, v, w) induces a 4-cut of this kind, then u belongs to $S(v)$, and $bcount(u) = bcount(v) - bcount(w) + 1$. Here we can exploit the fact that all vertices in $S(v)$ have different *bcount* (see Lemma 5.95). Thus, for every $w \in \widetilde{W}(v)$, we have that u is uniquely determined by the properties $u \in S(v)$ and $bcount(u) = bcount(v) - bcount(w) + 1$. Then, Proposition 5.27 establishes that we can compute all such 4-cuts in linear time in total. Notice that their total number is bounded by $O(n)$.

Type-$3\beta ii$-3 4-cuts

For the case of Type-$3\beta ii$-3 4-cuts, we define the set $U_3(v)$, for every vertex $v \neq r$, as a segment of $\widetilde{H}(high(v))$ that consists of proper descendants of v that have low enough *low* point in order to be possible to participate in a triple of vertices (u, v, w) that induces a Type-$3\beta ii$-3 4-cut. The set $U_3(v)$ does not contain all possible candidates u with the property that there may exist a w such that (u, v, w) induces a Type-$3\beta ii$-3 4-cut. However, by Lemma 5.97 we have that, if (u, v, w) is a triple of vertices that induces a Type-$3\beta ii$-3 4-cut, then $(\tilde{u}, v, w)$ also has this property, where $\tilde{u}$ is the greatest vertex in $U_3(v)$. This is very useful, because the total number of triples (u, v, w) that induce a Type-$3\beta ii$-3 4-cut can be $\Omega(n)$, and this can be true for $\Omega(n)$ vertices v. (Thus, the actual number of Type-$3\beta ii$-3 4-cuts can be $\Omega(n^2)$.) The sets $U_3(v)$ have total size $O(n)$, and Lemma 5.99 establishes that we can compute all of them in linear time in total. Given v and $u \in U_3(v)$, by Lemma 5.100 we have that there is at most one w such that (u, v, w) induces a Type-$3\beta ii$-3 4-cut: i.e., w is the greatest proper ancestor of v such that $M(w) = M(v)$ and $w \leq low(u)$. Thus, the number of all Type-$3\beta ii$-3 4-cuts that we compute is $O(n)$. Proposition 5.28 shows that we can compute this selection in linear time in total, and this has the property that, together with the collection of Type-$2ii$ 4-cuts that we have computed, it implies all Type-$3\beta ii$-3 4-cuts.

Type-$3\beta ii$-4 4-cuts

In the case of Type-$3\beta ii$-4 4-cuts we distinguish four different subcases, depending on the location of the endpoints of the back-edge e. Specifically, we consider the cases:

1. $M(B(v)\setminus\{e\}) \neq M(v)$ and $\mathit{high}_1(v) > \mathit{high}(u)$.
2. $M(B(v)\setminus\{e\}) \neq M(v)$ and $\mathit{high}_1(v) = \mathit{high}(u)$.
3. $M(B(v)\setminus\{e\}) = M(v)$ and $\mathit{high}_1(v) > \mathit{high}(u)$.
4. $M(B(v)\setminus\{e\}) = M(v)$ and $\mathit{high}_1(v) = \mathit{high}(u)$.

In Case 1 we know precisely the back-edge e: i.e., we have $e = e_{\mathit{high}}(v)$ (due to $\mathit{high}_1(v) > \mathit{high}(u)$, as a consequence of Lemma 5.101). Here there may be several vertices v for which there is a specific pair of vertices u and w such that (u,v,w) induces a 4-cut of this kind. That is, we may have two distinct v and v' such that (u,v,w) and (u,v',w) induce a 4-cut of this kind. Here we define a collection of vertices $V(v)$, for every vertex v, that has the property that, if there is a triple of vertices (u,v,w) that induces a 4-cut of this kind, then, for every $v' \in V(v)$, we have that (u,v',w) also induces a 4-cut of this kind (see Lemma 5.110). Furthermore, if two vertices v and v' belong to the same such collection, then $\{(v,p(v)),(v',p(v')),e_{\mathit{high}}(v),e_{\mathit{high}}(v')\}$ is a Type-$2ii$ 4-cut (as a consequence of Lemma 5.105). This is very useful in order to establish that it is sufficient to have computed only a selection of $O(n)$ size of those 4-cuts, so that the rest of them are implied from this selection, plus the collection of Type-$2ii$ 4-cuts that we have computed.

Now the idea is to pick one representative vertex from every one of the collections of vertices V; thus, we form a collection of vertices $\mathcal{V}$. Then, for every vertex $v \in \mathcal{V}$, we construct a set $U_4^1(v)$, that is a subset of $\widetilde{S}_2(v)$, and consists of proper descendants u of v that have low enough *low* point in order to be possible to participate in a triple of the form (u,v,w) that induces a 4-cut of the kind that we consider (see Lemma 5.110). The sets $U_4^1(v)$ have total size $O(n)$, and by Lemma 5.109 we can compute all of them in linear time in total (for the specific selection of representatives $\mathcal{V}$). Then, given $v \in \mathcal{V}$ and $u \in U_4^1(v)$, Lemma 5.111 implies that there is at most one w such that (u,v,w) induces a 4-cut of the kind that we consider. Thus, we compute a selection

of $O(n)$ 4-cuts of this kind. Proposition 5.29 establishes that we can compute this selection in linear time in total, and this is enough in order to imply, together with the collection of Type-$2ii$ 4-cuts that we have computed, all 4-cuts of this kind.

In Case 2, we have that either $e = e_L(v)$ or $e = e_R(v)$, as a consequence of $M(B(v) \setminus \{e\}) \neq M(v)$ (see Lemma 3.9). Then we only consider the case that $e = e_L(v)$, because the other case is treated with similar arguments and methods. Then we define the set $U_4^2(v)$, for every vertex v that has the potential to participate in a triple of vertices (u, v, w) that induces a 4-cut of this kind. The sets $U_4^2(v)$, for all vertices v for which they are defined, have total size $O(n)$, and Lemma 5.114 shows that we can compute all of them in linear time in total. Then Lemma 5.116 shows that if we have a triple of vertices (u, v, w) that induces a 4-cut of the kind that we consider, then $u \in U_4^2(v)$. By Lemma 5.117 we have that w is uniquely determined by u and v. Then Proposition 5.30 shows that we can compute all 4-cuts of this kind, in linear time in total.

In Case 3 we again know precisely the back-edge e, due to $high_1(v) > high(u)$. Then we follow the same idea as in Case 2 (by properly defining the sets $U_4^3(v)$), and Proposition 5.31 establishes that we can compute all 4-cuts of this kind in linear time in total.

In Case 4, the conditions $M(B(v) \setminus \{e\}) = M(v)$ and $high_1(v) = high(u)$ are not sufficient in order to determine the endpoints of e. However, we can follow the same idea as previously if we assume that the lower endpoint of e is distinct from $high(u)$. In this case, we define the sets $U_4^4(v)$ appropriately, as those segments of $S(v)$ that contain enough vertices u that have the potential to participate in a triple (u, v, w) that induces a 4-cut of this kind. The total size of all sets $U_4^4(v)$ is $O(n)$, and by Lemma 5.126 we can compute all of them in linear time in total. Then, by Lemma 5.127 we have that if (u, v, w) is a triple of vertices that induces a 4-cut of the kind that we consider, then either $u \in U_4^4(v)$, or u is the predecessor of the greatest vertex of $U_4^4(v)$ in $S(v)$. Furthermore, Lemma 5.128 shows that w is either the greatest or the second-greatest proper ancestor of v such that $M(w) = M(v)$ and $w \leq low(u)$. Thus, the total number of triples that we have to check is $O(n)$. Proposition 5.32 establishes that we can compute all 4-cuts of this kind in linear time in total.

Finally, it remains to consider the case where the lower endpoint of e coincides with $high(u)$. Since we are in Case (4) of Lemma 5.57, we have $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. This implies that $e \notin B(u)$, and therefore $e \neq e_{high}(u)$. This means that e and $e_{high}(u)$

are two distinct back-edges that have the same lower endpoint. If we could eliminate this possibility, then we could revert to any of the previous cases of 4-cuts that we have considered. The idea is precisely that: we compute a "4-cut equivalent" graph, in which there is a DFS-tree with the property that no two back-edges that correspond to edges of the original graph can have the same lower endpoint. To do this, we split every vertex z that has at least two incoming back-edges of the form (x, z) and (y, z), into two vertices z_1 and z_2 that are connected with five multiple edges (z_1, z_2). We make z_2 the parent of z_1, and z_1 inherits the back-edge (x, z) (in the form (x, z_1)), whereas z_2 inherits the remaining incoming back-edges to z. We continue this process until no more such splittings are possible. We show that the 4-cuts of the original graph are in a bijective correspondence with the 4-cuts of the resulting graph, and we show how to construct it in linear time. Thus, it is sufficient to perform one more pass on the resulting graph, of all the algorithms that we have developed for computing 4-cuts. (Or we may perform this computation directly on the resulting graph from the start.) Then we translate the computed 4-cuts to those of the original graph. We conclude with a post-processing step, that eliminates repetitions of 4-cuts.

5.5.4 Min-max vertex queries

As we saw in the preceding section, in order to implement the algorithms in the following chapters we need an oracle for answering min-max vertex queries. Specifically, at various places we have to answer queries of the form "find the lowest (resp., the greatest) vertex that is greater (resp., lower) than a particular vertex, and belongs to a specific collection of vertices". More precisely, we are given a collection $W_1, \dots, W_k$ of pairwise disjoint sets of vertices, and a set of N queries of the form $q(i, z) \equiv$ "given an index $i \in \{1, \dots, k\}$ and a vertex z, find the greatest (resp., the lowest) vertex $w \in W_i$ such that $w \leq z$ (resp., $w \geq z$)". We can answer all those queries simultaneously, in $O(n + N)$ time in total. The idea is to sort the vertices in the sets $W_1, \dots, W_k$ properly – in increasing or decreasing order –, depending on whether the queries ask for the greatest or the lowest vertex, respectively. Then we also sort the queries in the same order (w.r.t. the vertices that appear in them). Now we can basically answer all the queries for which the answer lies in a specific collection W, independently of the others, by simply traversing the list W and the list of the queries whose answer lies in W. The precise procedure is shown in Algorithm 22, and the explication as well

as the proof of correctness is given in Lemma 5.27. It is straightforward to modify Algorithm 22 appropriately, so that we get an algorithm for answering the reverse type of queries (i.e., those that ask for the lowest vertex that is greater than another vertex), with or without equality.

Lemma 5.27. *Let $W_1, \ldots, W_k$ be a collection of pairwise disjoint sets of vertices. Let $q(i_1, z_1), \ldots, q(i_N, z_N)$ be a set of queries of the form $q(i, z) \equiv$" given an index $i \in \{1, \ldots, k\}$ and a vertex z, find the greatest vertex $w \in W_i$ such that $w \leq z$". Then, Algorithm 22 answers all those queries in $O(n + N)$ time.*

Proof. The idea is basically to collect all queries of the form $q(i, \cdot)$, for every $i \in \{1, \ldots, k\}$, and then process them simultaneously with the list W_i. More precicely, we first sort all W_i, for $i \in \{1, \ldots, k\}$, in decreasing order. Since the sets of vertices $W_1, \ldots, W_k$ are pairwise disjoint, we have that $|W_1| + \cdots + |W_k| = O(n)$. Thus, all these sortings can be performed in $O(n)$ time in total with bucket-sort. Then, for every $i \in \{1, \ldots, k\}$, we collect in a list $Q(i)$ all tuples of the form (z, t), such that $i_t = i$ and $z_t = z$. In other words, if the t-th query $q(i_t, z_t)$ has $i_t = i$, then $Q(i)$ contains the entry (z_t, t). Then we sort the lists $Q(i)$ in decreasing order w.r.t. the first component of the tuples that are contained in them. This can be done in $O(n + N)$ time in total with bucket-sort.

Now, in order to answer a query $q(i, z)$, we have to traverse the list W_i, until we reach the first $w \in W_i$ that has $w \leq z$ (since W_i is sorted in decreasing order). If we performed this process for every individual query, we would possibly make an excessive amount of steps in total, because each time we would process the list W_i from the beginning. Thus, the idea in sorting the queries too, is that we can pick up the search from the last entry of W_i that we accessed. Therefore, we process the entries in $Q(i)$ in order, for every index $i \in \{1, \ldots, k\}$. Since the first components of the tuples in $Q(i)$ are vertices in decreasing order, it is sufficient to start the search in W_i from the last entry that we accessed. The second component of every tuple (z, t) in $Q(i)$ is a pointer to the corresponding query that is being answered (i.e., this corresponds to the t-th query, $q(i_t, z_t)$).

It is easy to see that this is the procedure that is implemented by Algorithm 22. The **for** loop in Line 13 takes $O(n + N)$ time in total, because it traverses the entire list of the queries, and possibly the entire lists $W_1, \ldots, W_k$. Thus, Algorithm 22 runs in $O(n + N)$ time. □

Algorithm 22: Given a collection $W_1, \ldots, W_k$ of pairwise disjoint sets of vertices, answer a set of queries $q(i_1, z_1), \ldots, q(i_N, z_N)$, where $q(i_t, z_t)$, for every $t \in \{1, \ldots, N\}$, asks for the greatest vertex $w \in W_{i_t}$ such that $w \leq z_t$.

```
   // W_1, ..., W_k is a collection of pairwise disjoint sets of vertices
1  foreach i ∈ {1, ..., k} do
2  │  sort W_i in decreasing order
3  end
4  foreach i ∈ {1, ..., k} do
5  │  initialize Q(i) ← ∅
6  end
7  foreach t ∈ {1, ..., N} do
8  │  insert a tuple (z_t, t) into Q(i_t)
9  end
   // Q(i) contains a tuple of the form (z, t) if and only if the query q(i_t, z_t) has
      i_t = i and z_t = z. The first component of a tuple in Q(i) stores the vertex of
      the respective query, and the second component stores a pointer to the query.
      The information, that we have to search in the set W_i for the answer to this
      query, is given precisely by the index i of this bucket of tuples
10 foreach i ∈ {1, ..., k} do
11 │  sort Q(i) in decreasing order w.r.t. the first component of its elements
12 end
13 foreach i ∈ {1, ..., k} do
14 │  let w be the first element of W_i
15 │  let p be the first element of Q(i)
16 │  while p ≠ ⊥ do
17 │  │  let p = (z, t)
18 │  │  while w ≠ ⊥ and w > z do
19 │  │  │  w ← next_{W_i}(w)
20 │  │  end
21 │  │  the answer to q(i_t, z_t) is w
22 │  │  p ← next_{Q(i)}(p)
23 │  end
24 end
```

5.6 Computing Type-2 4-cuts

Throughout this section, we assume that G is a 3-edge-connected graph with n vertices and m edges. All graph-related elements (e.g., vertices, edges, cuts, etc.) refer to G. Furthermore, we assume that we have computed a DFS-tree T of G rooted at a vertex r.

Lemma 5.28. *Let u, v be two vertices such that v is a proper ancestor of u with $v \neq r$. Then there exist two distinct back-edges e_1, e_2 such that $\{(u, p(u)), (v, p(v)), e_1, e_2\}$ is a 4-cut if and only if either (1) $B(v) = B(u) \sqcup \{e_1, e_2\}$, or (2) $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$, or (3) $B(u) = B(v) \sqcup \{e_1, e_2\}$.*

Proof. ($\Rightarrow$) Let $C = \{(u, p(u)), (v, p(v)), e_1, e_2\}$. First we will show that e_1 and e_2 are back-edges in $B(u) \cup B(v)$. So let us suppose the contrary. Then we may assume w.l.o.g. that $e_1 \notin B(u) \cup B(v)$. Let $e_1 = (x, y)$. Then $e_1 \notin B(u) \cup B(v)$ means that neither u nor v lies on the tree-path $T[x, y)$. This implies that the tree-path $T[x, y)$ remains intact in $G \setminus C$. But then we have that the endpoints of e_1 remain connected in $G \setminus C$, in contradiction to the fact that C is a 4-cut of G. This shows that $e_1 \in B(u) \cup B(v)$. Similarly, we can show that $e_2 \in B(u) \cup B(v)$.

Since C is a 4-cut of G, we have that $G' = G \setminus \{(u, p(u)), (v, p(v))\}$ is connected. We define the three parts $X = T(u)$, $Y = T(v) \setminus T(u)$, and $Z = T(r) \setminus T(v)$. Notice that these parts remain connected in G'.

Suppose first that both e_1 and e_2 leap over v. Let us suppose, for the sake of contradiction, that there is a back-edge $e = (x, y)$ from X to Y. Then e leaps over u, but not over v. Thus, $e \notin \{e_1, e_2\}$. But then we have that u is connected with $p(u)$ in $G' \setminus \{e_1, e_2\}$ through the path $T[u, x], (x, y), T[y, p(u)]$, contradicting the fact that C is a 4-cut of G. This means that there is no back-edge in $B(u) \setminus B(v)$, and therefore $B(u) \subseteq B(v)$. Now let us suppose, for the sake of contradiction, that there is a back-edge $e = (x, y)$ that leaps over v, does not leap over u, and is distinct from e_1 and e_2. Since e leaps over v, but not over u, we have that it is a back-edge from Y to Z. But then, since $e \notin \{e_1, e_2\}$, we have that v is connected with $p(v)$ in $G' \setminus \{e_1, e_2\}$ through the path $T[v, x], (x, y), T[y, p(v)]$, contradicting the fact that C is a 4-cut of G. This means that e_1 and e_2 are the only edges in $B(v) \setminus B(u)$, and so we have $B(v) = B(u) \cup \{e_1, e_2\}$. Notice that it cannot be that both e_1 and e_2 are in $B(u)$, because otherwise we have $B(v) = B(u)$, in contradiction to the fact that G is 3-edge-connected. Now let us suppose, for the sake of contradiction, that one of e_1, e_2

is in $B(u)$. We may assume w.l.o.g. that e_1 is in $B(u)$. Since C is a 4-cut of G, we have that $G' \setminus e_2$ is connected. In particular, v is connected with $p(v)$ in $G' \setminus e_2$. There are two possible ways in which this can be true: either (i) there is a back-edge from Y to Z in $G' \setminus e_2$, or (ii) there is a back-edge from Y to X, and a back-edge from X to Z, in $G' \setminus e_2$. Case (i) is rejected, because the only back-edge that leaps over v but not over u is e_2. Case (ii) is rejected, because $B(u) \subset B(v)$ (and so there is no back-edge from X to Y). Since neither of (i) and (ii) can be true, we have arrived at a contradiction. Thus, we have that neither of e_1, e_2 can be in $B(u)$, and so it is correct to write $B(v) = B(u) \sqcup \{e_1, e_2\}$.

Now let us suppose that only one of e_1, e_2 leaps over v. Then we may assume w.l.o.g. that $e_1 \notin B(v)$ and $e_2 \in B(v)$. Since each one of e_1, e_2 leaps over either u or v, we have that $e_1 \in B(u)$. Since $e_1 \in B(u) \setminus B(v)$, we have that e_1 is back-edge from X to Y. Now let us suppose, for the sake of contradiction, that there is also another back-edge $e = (x, y)$ from X to Y (i.e., $e \neq e_1$). Notice that, since $e_2 \in B(v)$, it is impossible that $e = e_2$. But now we have that u remains connected with $p(u)$ in $G' \setminus \{e_1, e_2\}$ through the path $T[u, x], (x, y), T[y, p(u)]$, contradicting the fact that C is a 4-cut of G. Thus we have that e_1 is the only back-edge from X to Y. Since $e_2 \in B(v)$, we have that e_2 is either a back-edge from X to Z, or a back-edge from Y to Z. Let us suppose, for the sake of contradiction, that e_2 is a back-edge from X to Z. Since C is a 4-cut of G, we have that $G' \setminus e_1$ is connected. In particular, u is connected with $p(u)$ in $G' \setminus e_1$. There are two possible ways for this to be true: either (i) there is a back-edge from X to Y in $G' \setminus e_1$, or (ii) there is a back-edge from X to Z, and a back-edge from Z to Y in $G' \setminus e_1$. Case (i) is rejected, since e_1 is the only back-edge from X to Y. Thus, (ii) implies that there is a back-edge $e = (x, y)$ from Y to Z in $G' \setminus e_1$. Since e_2 is a back-edge from X to Z, we have that $e \notin \{e_1, e_2\}$. But then v is connected with $p(v)$ in $G' \setminus \{e_1, e_2\}$ through the path $T[v, x], (x, y), T[y, p(v)]$, contradicting the fact that C is a 4-cut of G. Thus we have that e_2 is not a back-edge from X to Z, and therefore it is a back-edge from Y to Z. Similarly, we can show that e_2 is the unique back-edge from Y to Z. Thus far we have that $e_1 \in B(u) \setminus B(v)$ and $e_2 \in B(v) \setminus B(u)$, and both e_1 and e_2 are unique with this property. Now, since e_1 is the only back-edge in $B(u) \setminus B(v)$, we have that $B(u) \setminus \{e_1\} \subseteq B(v)$. And since e_2 is the only back-edge in $B(v) \setminus B(u)$, we have that $B(v) \setminus \{e_2\} \subseteq B(u)$. Now let e be a back-edge in $B(u) \sqcup \{e_2\}$. Then, either $e = e_1$, or $e \in B(v)$. Thus we get $B(u) \sqcup \{e_2\} \subseteq B(v) \sqcup \{e_1\}$. Similarly, we get the reverse inclusion, and so we have $B(u) \sqcup \{e_2\} = B(v) \sqcup \{e_1\}$.

Finally, let us suppose that neither of e_1, e_2 leaps over v. Then, since each one of e_1, e_2 leaps over either u or v, we have that $\{e_1, e_2\} \subseteq B(u)$. Notice that both e_1 and e_2 are back-edges from X to Y. Now we can argue as above, in order to establish that e_1 and e_2 are the only back-edges from X to Y. Thus, the remaining back-edges in $B(u)$ must also be in $B(v)$. Again, arguing as above, we can establish that there is no back-edge from Y to Z. Thus, all the back-edges in $B(v)$ are also in $B(u)$. Thus we have $B(u) \setminus \{e_1, e_2\} \subseteq B(v)$, and $B(v) \subseteq B(u)$. Therefore, since $\{e_1, e_2\} \subseteq B(u) \setminus B(v)$, we have that $B(v) \sqcup \{e_1, e_2\} = B(u)$.

($\Leftarrow$) Let $C = \{(u, p(u)), (v, p(v)), e_1, e_2\}$. Since the graph is 3-edge-connected, we have that $G' = G \setminus \{(u, p(u)), (v, p(v))\}$ is connected. We define the three parts $X = T(u)$, $Y = T(v) \setminus T(u)$, and $Z = T(r) \setminus T(v)$. Notice that these parts are connected in G'. Now it is easy to see that either of (1), (2), or (3), implies that C is a 4-cut of G: (1) means that e_1 and e_2 are the only back-edges from Y to Z, and there are no back-edges from X to Y; (2) means that e_1 is the only back-edge from X to Y, and e_2 is the only back-edge from Y to Z; and (3) means that e_1 and e_2 are the only back-edges from X to Y, and there are no back-edges from Y to Z. In either case, we can see that Y becomes disconnected from the rest of the graph in $G \setminus C$, but $G \setminus C'$ remains connected for every proper subset C' of C. □

Based on Lemma 5.28, we distinguish three different cases for Type-2 4-cuts of the form $\{(u, p(u)), (v, p(v)), e_1, e_2\}$, where v is ancestor of u and e_1, e_2 are back-edges: either (1) $B(v) = B(u) \sqcup \{e_1, e_2\}$, or (2) $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$, or (3) $B(u) = B(v) \sqcup \{e_1, e_2\}$. We show how to find all 4-cuts in cases (1) and (3) in linear time, in Sections 5.6.1 and 5.6.3, respectively. The 4-cuts in case (2) cannot be computed in linear time, since there can be $\Omega(n^2)$ of them. Instead, we calculate only a specific selection of $O(n)$ of them, so that the rest of them are implied from this selection. We show how we can handle this case in Section 5.6.2.

5.6.1 The case $B(v) = B(u) \sqcup \{e_1, e_2\}$

Lemma 5.29. *Let u and v be two vertices such that v is a proper ancestor of u with $v \neq r$. Then there exist two back-edges e_1 and e_2 such that $B(v) = B(u) \sqcup \{e_1, e_2\}$ if and only if $bcount(v) = bcount(u) + 2$ and $high_1(u) < v$.*

Proof. ($\Rightarrow$) $bcount(v) = bcount(u) + 2$ is an immediate consequence of $B(v) = B(u) \sqcup \{e_1, e_2\}$. Now let (x, y) be a back-edge in $B(u)$. Then $B(v) = B(u) \sqcup \{e_1, e_2\}$ implies

that (x, y) is a back-edge in $B(v)$, and therefore y is a proper ancestor of v, and therefore $y < v$. Due to the generality of $(x, y) \in B(u)$, this implies that $high_1(u) < v$.

($\Leftarrow$) Since u is a common descendant of v and $high_1(u)$, we have that v and $high_1(u)$ are related as ancestor and descendant. Thus, $high_1(u) < v$ implies that $high_1(u)$ is a proper ancestor of v. Now let (x, y) be a back-edge in $B(u)$. Then x is a descendant of u, and therefore a descendant of v. Furthermore, y is an ancestor of $high_1(u)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$, and therefore we have $B(u) \subseteq B(v)$. Now $bcount(v) = bcount(u) + 2$ implies that $|B(v) \setminus B(u)| = 2$, and so there are two back-edges e_1, e_2 such that $B(v) = B(u) \sqcup \{e_1, e_2\}$. $\square$

Lemma 5.30. *Let u and v be two vertices such that v is a proper ancestor of u with $v \neq r$, and there exist two back-edges $e_1 = (x_1, y_1)$ and $e_2 = (x_2, y_2)$ such that $B(v) = B(u) \sqcup \{e_1, e_2\}$. Then, neither of x_1, x_2 is a descendant of u, and one of the following is true:*

(1) $L_1(v) = L_2(v)$, $x_1 = x_2 = L_1(v)$ *and* $\{y_1, y_2\} = \{l_1(L_1(v)), l_2(L_1(v))\}$

(2) $L_1(v) \neq L_2(v)$, $\{x_1, x_2\} = \{L_1(v), L_2(v)\}$ *and* $\{y_1, y_2\} = \{l_1(L_1(v)), l_1(L_2(v))\}$

(3) $\{x_1, x_2\} = \{L_1(v), R_1(v)\}$ *and* $\{y_1, y_2\} = \{l_1(L_1(v)), l_1(R_1(v))\}$

(4) $R_1(v) = R_2(v)$, $x_1 = x_2 = R_1(v)$ *and* $\{y_1, y_2\} = \{l_1(R_1(v)), l_2(R_1(v))\}$

(5) $R_1(v) \neq R_2(v)$, $\{x_1, x_2\} = \{R_1(v), R_2(v)\}$ *and* $\{y_1, y_2\} = \{l_1(R_1(v)), l_1(R_2(v))\}$

Proof. Let us suppose, for the sake of contradiction, that at least one of x_1, x_2 is a descendant of u. We may assume w.l.o.g. that x_1 is a descendant of u. Then, since $(x_1, y_1) \in B(v)$, we have that y_1 is a proper ancestor of v, and therefore it is a proper ancestor of u. But this implies that $(x_1, y_1) \in B(u)$, a contradiction. Thus we have that neither of x_1, x_2 is a descendant of u.

Now let $(x'_1, y'_1), \dots, (x'_k, y'_k)$ be the back-edges in $B(v)$ sorted in increasing order w.r.t. their higher endpoint. (Notice that $L_1(v) = x'_1$, $L_2(v) = x'_2$, $R_1(v) = x'_k$, and $R_2(v) = x'_{k-1}$.) Let $i, j \in \{1, \dots, k\}$ be two indices such that $i \leq j$. Suppose that x'_i and x'_j are descendants of u. Since we have $x'_i \leq \dots \leq x'_j$, this implies that all $x'_i, \dots, x'_j$ are descendants of u. Furthermore, since $(x'_i, y'_i), \dots, (x'_j, y'_j)$ are back-edges in $B(v)$, we have that $y'_i, \dots, y'_j$ are proper ancestors of v, and therefore they are proper ancestors of u. This shows that all the back-edges $(x'_i, y'_i), \dots, (x'_j, y'_j)$ are in $B(u)$. Now, since $B(v) = B(u) \sqcup \{e_1, e_2\}$, we have that only two back-edges in $B(v)$ are not in $B(u)$.

Therefore, e_1 and e_2 can either be (i) (x'_1, y'_1) and (x'_2, y'_2), or (ii) (x'_1, y'_1) and (x'_k, y'_k), or (iii) (x'_{k-1}, y'_{k-1}) and (x'_k, y'_k). Thus we get that (1)-(5) is an exhaustive list for the different combinations for x_1 and x_2.

Suppose first that $L_1(v) = L_2(v)$ and $x_1 = x_2 = L_1(v)$. Let us suppose, for the sake of contradiction, that $l_3(L_1(v)) < v$. Then this means that there are at least three different back-edges $(L_1(v), z_1)$, $(L_1(v), z_2)$, $(L_1(v), z_3)$ in $B(v)$. Since $x_1 = L_1(v)$ is not a descendant of u, we have that these three back-edges are in $B(v) \setminus B(u)$. But this contradicts $B(v) = B(u) \sqcup \{e_1, e_2\}$. Thus we have that $l_3(L_1(v)) \geq v$. Since $e_1, e_2 \in B(v)$, we have that $y_1 < v$ and $y_2 < v$. Thus we get $\{y_1, y_2\} = \{l_1(L_1(v)), l_2(L_1(v))\}$.

Now suppose that $L_1(v) \neq L_2(v)$ and $\{x_1, x_2\} = \{L_1(v), L_2(v)\}$. We may assume w.l.o.g. that $x_1 = L_1(v)$ and $x_2 = L_2(v)$. Let us suppose, for the sake of contradiction, that $l_2(L_1(v)) < v$. This implies that there are at least two different back-edges $(L_1(v), z_1)$ and $(L_1(v), z_2)$ in $B(v)$. Since $x_1 = L_1(v)$ is not a descendant of u, we have that there are at least three back-edges in $B(v) \setminus B(u)$ (these being $(L_1(v), z_1)$, $(L_1(v), z_2)$, and $(L_2(v), y_2)$), a contradiction. Thus we have $l_2(L_1(v)) \geq v$, and so $y_1 = l_1(L_1(v))$, since $(x_1, y_1) \in B(v)$. Similarly, we can show that $y_2 = l_1(L_2(v))$.

With similar arguments we get the results for y_1 and y_2 for the cases $\{x_1, x_2\} = \{L_1(v), R_1(v)\}$ and $\{x_1, x_2\} = \{R_1(v), R_2(v)\}$ (whether $R_1(v) = R_2(v)$ or $R_1(v) \neq R_2(v)$). ☐

Lemma 5.31. *Let u and v be two vertices such that v is a proper ancestor of u with $v \neq r$, and there exist two back-edges e_1 and e_2 such that $B(v) = B(u) \sqcup \{e_1, e_2\}$. Then u is the lowest proper descendant of v that has $M(u) = M(B(v) \setminus \{e_1, e_2\})$.*

Proof. First we observe that $M(u) = M(B(v) \setminus \{e_1, e_2\})$, as an immediate consequence of $B(u) = B(v) \setminus \{e_1, e_2\}$. Thus we may consider the lowest proper descendant u' of v that has $M(u') = M(B(v) \setminus \{e_1, e_2\})$. Let us suppose, for the sake of contradiction, that $u' \neq u$. Then, since $M(u') = M(u)$ and u' is lower than u, we have that u' is a proper ancestor of u, and Lemma 3.2 implies that $B(u') \subseteq B(u)$. Since the graph is 3-edge-connected, this can be strengthened to $B(u') \subset B(u)$. Thus there is a back-edge $(x, y) \in B(u) \setminus B(u')$. Then, we have that x is a descendant of u, and therefore a descendant of u'. Furthermore, $B(v) = B(u) \sqcup \{e_1, e_2\}$ implies that $(x, y) \in B(v)$, and therefore y is a proper ancestor of v. But then y is also a proper ancestor of u', and so $(x, y) \in B(u')$, a contradiction. We conclude that u is the lowest proper descendant of v with $M(u) = M(B(v) \setminus \{e_1, e_2\})$. ☐

Now the idea to find all 4-cuts of the form $\{(u,p(u)),(v,p(v)),e_1,e_2\}$, where v is a proper ancestor of u with $B(v) = B(u) \sqcup \{e_1,e_2\}$, is the following. According to Lemma 5.30, there are three different cases for the back-edges e_1 and e_2: either $\{e_1,e_2\} = \{e_{L1}(v),e_{L2}(v)\}$, or $\{e_1,e_2\} = \{e_{L1}(v),e_{R1}(v)\}$, or $\{e_1,e_2\} = \{e_{R1}(v),e_{R2}(v)\}$. We consider all these cases in turn. For every one of those cases, we seek the lowest proper descendant u of v that satisfies $M(u) = M(B(v) \setminus \{e_1,e_2\})$, according to Lemma 5.31. Then we can check whether we have a 4-cut using the criterion provided by Lemma 5.29. This procedure is shown in Algorithm 23. The proof of correctness is given in Proposition 5.11.

Algorithm 23: Compute all 4-cuts of the form $\{(u,p(u)),(v,p(v)),e_1,e_2\}$, where v is an ancestor of u and $B(v) = B(u) \sqcup \{e_1,e_2\}$

1 **foreach** *vertex* $v \neq r$ **do**
2 compute $M_{LL}(v) \leftarrow M(B(v) \setminus \{e_{L1}(v),e_{L2}(v)\})$
3 compute $M_{LR}(v) \leftarrow M(B(v) \setminus \{e_{L1}(v),e_{R1}(v)\})$
4 compute $M_{RR}(v) \leftarrow M(B(v) \setminus \{e_{R1}(v),e_{R2}(v)\})$
5 **end**
6 let u be the lowest proper descendant of v such that $M(u) = M_{LL}(v)$
7 **if** $bcount(v) = bcount(u) + 2$ ***and*** $high_1(u) < v$ **then**
8 mark $\{(u,p(u)),(v,p(v)),e_{L1}(v),e_{L2}(v)\}$ as a 4-cut
9 **end**
10 let u be the lowest proper descendant of v such that $M(u) = M_{LR}(v)$
11 **if** $bcount(v) = bcount(u) + 2$ ***and*** $high_1(u) < v$ **then**
12 mark $\{(u,p(u)),(v,p(v)),e_{L1}(v),e_{R1}(v)\}$ as a 4-cut
13 **end**
14 let u be the lowest proper descendant of v such that $M(u) = M_{RR}(v)$
15 **if** $bcount(v) = bcount(u) + 2$ ***and*** $high_1(u) < v$ **then**
16 mark $\{(u,p(u)),(v,p(v)),e_{R1}(v),e_{R2}(v)\}$ as a 4-cut
17 **end**

Proposition 5.11. *Algorithm 23 correctly computes all 4-cuts of the form $\{(u,p(u)),(v,p(v)),e_1,e_2\}$, where v is an ancestor of u and $B(v) = B(u) \sqcup \{e_1,e_2\}$. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u,p(u)),(v,p(v)),e_1,e_2\}$ be a 4-cut such that u is a descen-

dant of v and $B(v) = B(u) \sqcup \{e_1, e_2\}$. Then, Lemma 5.30 implies that either $\{e_1, e_2\} = \{e_{L1}(v), e_{L2}(v)\}$, or $\{e_1, e_2\} = \{e_{L1}(v), e_{R1}(v)\}$, or $\{e_1, e_2\} = \{e_{R1}(v), e_{R2}(v)\}$. By Lemma 5.31, we have that u is the lowest proper descendant of v such that $M(u) = M(B(v) \setminus \{e_1, e_2\})$. Lemma 5.29 implies that $bcount(v) = bcount(u) + 2$ and $high_1(u) < v$. Thus, we can see that C will be marked in Line 8, or 12, or 16.

Conversely, suppose that a 4-element set $C = \{(u, p(u)), (v, p(v)), e'_1, e'_2\}$ is marked by Algorithm 23 in Line 8, or 12, or 16. In either case, we have that u is a proper descendant of v such that $bcount(v) = bcount(u)+2$ and $high_1(u) < v$. Thus, Lemma 5.29 implies that there are two back-edges e_1 and e_2 such that $B(v) = B(u) \sqcup \{e_1, e_2\}$. Lemma 5.30 implies that the higher endpoints of e_1 and e_2 are not descendants of u. Thus, if S is a subset of $B(v)$ that contains either e_1 or e_2, then we have that $M(S) \neq M(u)$. To see this, suppose the contrary. Then, w.l.o.g. we may assume that $e_1 = (x, y) \in S$, and $M(S) = M(u)$. Then we have that x is a descendant of $M(S)$, and therefore a descendant of $M(u)$, and therefore a descendant of u. Furthermore, since $e_1 \in B(v)$, we have that y is a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$, a contradiction. Thus, we have that $M(S) \neq M(u)$. Now, since $B(v) = B(u) \sqcup \{e_1, e_2\}$, Lemma 5.30 implies that either $\{e_1, e_2\} = \{e_{L1}(v), e_{L2}(v)\}$, or $\{e_1, e_2\} = \{e_{L1}(v), e_{R1}(v)\}$, or $\{e_1, e_2\} = \{e_{R1}(v), e_{R2}(v)\}$. Let us assume that $\{e_1, e_2\} = \{e_{L1}(v), e_{L2}(v)\}$ (the other cases are similar). Then, if C is marked in Line 8, we have that $\{e'_1, e'_2\} = \{e_1, e_2\}$, and so it is correct to mark C as a 4-cut. Now, if $\{e_{L1}(v), e_{R1}(v)\} \neq \{e_1, e_2\}$, then we have that $M(u) \neq M_{LR}(v)$ (because $B(v) \setminus \{e_{L1}(v), e_{R1}(v)\}$ contains either e_1 or e_2), and therefore C is not marked in Line 12. Similarly, if $\{e_{R1}(v), e_{R2}(v)\} \neq \{e_1, e_2\}$, then we have that $M(u) \neq M_{RR}(v)$, and therefore C is not marked in Line 16. This shows that C is indeed a 4-cut of G.

Now we will show that Algorithm 23 runs in linear time. By Proposition 3.6, we have that the values $M(B(v) \setminus \{e_{L1}(v), e_{L2}(v)\})$, $M(B(v) \setminus \{e_{L1}(v), e_{R1}(v)\})$ and $M(B(v) \setminus \{e_{R1}(v), e_{R2}(v)\})$ can be computed in linear time in total, for all vertices $v \neq r$. Thus, the **for** loop in Line 1 can be performed in linear time. In order to compute the u in Lines 6, 10 and 14, we use Algorithm 22. Specifically, let us discuss the implementation of Line 6 (the argument for Lines 10 and 14 is similar). Then, for every vertex $v \neq r$, we generate a query $q(M^{-1}(M_{LL}(v)), v)$. This query will return the lowest vertex u with $M(u) = M_{LL}(v)$ that is greater than v. Notice that $M(u) = M_{LL}(v)$ implies that $M(u)$ is a common descendant of u and v, and therefore u and v are related as ancestor and descendant. Thus, u is the lowest proper descendant of v

that satisfies $M(u) = M_{LL}(v)$. Then, since the number of all those queries is $O(n)$, Algorithm 22 can answer them in $O(n)$ time in total, according to Lemma 5.27. This shows that Algorithm 23 runs in linear time. $\square$

5.6.2 The case $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$

Let $\{(u,p(u)),(v,p(v)),e_1,e_2\}$ be a Type-2 4-cut such that $e_1 \in B(u)$ and $e_2 \in B(v)$. Then, by Lemma 5.28 we have that $B(u) \sqcup \{e_2\} = B(v) \sqcup \{e_1\}$, and we call this a Type-2ii 4-cut. Our goal in this section is to prove that we can compute enough such 4-cuts in linear time, so that the rest of them are implied from the collection we have computed. For a precise statement of our result, see Proposition 5.12. The Type-2ii 4-cuts are the most significant Type-2 4-cuts, because their existence is the reason why we may have a quadratic number of 4-cuts overall. This will become clear in the following sections, where we will see that there are some subtypes of 4-cuts whose number can be quadratic, but they are involved in an implicating sequence with Type-2ii 4-cuts, and so we can compute in linear time a subcollection of them that implies them all.

The following two lemmata establish conditions under which we have a Type-2ii 4-cut.

Lemma 5.32. *Let u and v be two vertices $\neq r$ such that u is a proper descendant of v with bcount(u) = bcount(v). Suppose that there is a back-edge $e \in B(v)$ such that $M(B(u) \setminus \{e_{high}(u)\}) = M(B(v) \setminus \{e\})$. Then $B(v) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e\}$.*

Proof. Let (x,y) be a back-edge in $B(v) \setminus \{e\}$. Then x is a descendant of $M(B(v) \setminus \{e\})$, and therefore a descendant of $M(B(u) \setminus \{e_{high}(u)\})$, and therefore a descendant of u. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x,y) \in B(u)$. Due to the generality of $(x,y) \in B(v) \setminus \{e\}$, this implies that $B(v) \setminus \{e\} \subseteq B(u)$. Since $e \in B(v)$ and *bcount*(u) = *bcount*(v), this implies that there is a back-edge $e' \in B(u)$ such that $B(v) \setminus \{e\} = B(u) \setminus \{e'\}$. If we assume that $e \in B(u)$, then we have $e' = e$, and therefore $B(v) = B(u)$, contradicting the fact that the graph is 3-edge-connected. Similarly, we have $e' \notin B(v)$. Thus, $B(v) \setminus \{e\} = B(u) \setminus \{e'\}$ implies that $B(v) \sqcup \{e'\} = B(u) \sqcup \{e\}$, and e' is the unique back-edge in $B(u) \setminus B(v)$.

Let us suppose, for the sake of contradiction, that $e' \neq e_{high}(u)$. Since e' is the unique back-edge in $B(u) \setminus B(v)$, this implies that $e_{high}(u) \in B(v)$. Therefore, *high*(u) is a proper ancestor of v. Now let (x,y) be a back-edge in $B(u)$. Then we have that x

is a descendant of u, and therefore a descendant of v. Furthermore, y is an ancestor of *high*(u), and therefore a proper ancestor of v. This shows that $(x,y) \in B(v)$. Due to the generality of $(x,y) \in B(u)$, this implies that $B(u) \subseteq B(v)$. But this contradicts the fact that there is a back-edge in $B(u) \setminus B(v)$. This shows that $e' = e_{high}(u)$. Therefore, we have $B(v) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e\}$. $\square$

Lemma 5.33. *Let u and v be two vertices such that v is a proper ancestor of u with $v \neq r$. Then there exist two distinct back-edges e_1 and e_2 such that $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$ if and only if bcount$(v) =$ bcount(u) and high$_2(u) < v$.*

Proof. ($\Rightarrow$) *bcount*$(v) =$ *bcount*(u) is an immediate consequence of $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$. Now let $(x_1,y_1),\dots,(x_k,y_k)$ be the back-edges in $B(u)$ sorted in decreasing order w.r.t. their lower endpoint. (Thus, we have *high*$_i(u) = y_i$, for every $i \in \{1,\dots,k\}$.) Let $i \in \{1,\dots,k\}$. If $(x_i,y_i) \in B(v)$, then y_i is a proper ancestor of v. This implies that every y_j with $j \in \{i,\dots,k\}$ is also a proper ancestor of v, which implies that $(x_j,y_j) \in B(v)$ (since all of $x_1,\dots,x_k$ are descendants of v). Thus, we cannot have $(x_1,y_1) \in B(v)$, for otherwise we would have $B(u) \subseteq B(v)$, in contradiction to $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$ and $e_1 \neq e_2$. Since $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$ implies that only one back-edge from $B(u)$ is not in $B(v)$, we thus have that $(x_2,y_2) \in B(v)$, and so *high*$_2(u) < v$.

($\Leftarrow$) Let $(x_1,y_1),\dots,(x_k,y_k)$ be the back-edges in $B(u)$ sorted in decreasing order w.r.t. their lower endpoint. Then *high*$_2(u) < v$ implies that $(x_2,y_2) \in B(v)$. Therefore, with the same argument that we used above we can infer that $\{(x_2,y_2),\dots,(x_k,y_k)\} \subseteq B(v)$. If we had that $(x_1,y_1) \in B(v)$, then *bcount*$(v) =$ *bcount*(u) would imply that $B(v) = B(u)$, in contradiction to the fact that the graph is 3-edge-connected. Thus we have that $e_1 = (x_1,y_1)$ is the only back-edge in $B(u)$ that is not in $B(v)$. Then, *bcount*$(v) =$ *bcount*(u) implies that there must be exactly one back-edge e_2 in $B(v)$ that is not in $B(u)$. Thus we get $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$. $\square$

The following lemma characterizes the back-edges that participate in a Type-$2ii$ 4-cut.

Lemma 5.34. *Let u and v be two vertices such that v is a proper ancestor of u with $v \neq r$, and there exist two distinct back-edges e_1 and e_2 such that $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$. Then we have $e_1 = ($highD$_1(u),$ high$_1(u))$ and either (1) $e_2 = (L_1(v), l_1(L_1(v)))$, or (2) $e_2 = (R_1(v), l_1(R_1(v)))$.*

Proof. The fact that $e_1 = ($*highD*$_1(u),$ *high*$_1(u))$ has essentially been proved in the proof of Lemma 5.33. Now let $(x_1,y_1),\dots,(x_k,y_k)$ be the back-edges in $B(v)$ sorted in

increasing order w.r.t. their higher endpoint. (Then we have $x_1 = L_1(v)$ and $x_k = R_1(v)$.) Let $i, j \in \{1, \ldots, k\}$ be two indices such that $i \leq j$. If both (x_i, y_i) and (x_j, y_j) are in $B(u)$, then we have that both x_i and x_j are descendants of u. This implies that all of $x_i, \ldots, x_j$ are descendants of u, and therefore all the back-edges $(x_i, y_i), \ldots, (x_j, y_j)$ are in $B(u)$ (since all of $y_1, \ldots, y_k$ are proper ancestors of u). Since $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$ implies that exactly one back-edge in $B(v)$ is not in $B(u)$ (and that is e_2), we thus have that the higher endpoint of e_2 is either $L_1(v)$ or $R_1(v)$.

Let us consider the case that $e_2 = (L_1(v), y)$, for some vertex y. Let us suppose, for the sake of contradiction, that $l_2(L_1(v)) < v$. Then there exist at least two back-edges of the form $(L_1(v), z_1)$ and $(L_1(v), z_2)$ that are in $B(v)$. Since $e_2 \notin B(u)$, we have that $L_1(v)$ cannot be a descendant of u (for otherwise, since y is a proper ancestor of v, we would have that $e_2 \in B(u)$). Thus we have that none of $(L_1(v), z_1)$ and $(L_1(v), z_2)$ is in $B(u)$, in contradiction to the fact that there is only one back-edge in $B(v) \setminus B(u)$. This shows that $l_2(L_1(v)) \geq v$. Since $e_2 \in B(v)$, we have $y < v$, and so y must be $l_1(L_1(v))$. The case that $e_2 = (R_1(v), z)$, for some vertez z, is treated with a similar argument. ☐

Let u and v be two vertices such that u is a proper descendant of v with $B(v) \sqcup \{e\} = B(u) \sqcup \{e'\}$, where e and e' are two distinct back-edges. This implies that $B(v) \setminus \{e'\} = B(u) \setminus \{e\}$, and therefore $M(B(v) \setminus \{e'\}) = M(B(u) \setminus \{e\})$. By Lemma 5.34, we have $e = e_{high}(u)$. Furthermore, $B(v) \sqcup \{e\} = B(u) \sqcup \{e'\}$ implies that $bcount(v) = bcount(u)$. Then, we let $lowestU(v, e')$ denote the lowest proper descendant u' of v such that $e_{high}(u') \notin B(v)$, $e' \notin B(u')$, $M(B(u') \setminus \{e_{high}(u')\}) = M(B(v) \setminus \{e'\})$, and $bcount(u') = bcount(v)$.

Lemma 5.35. *Let u and v be two vertices such that $u = lowestU(v, e)$, where e is a back-edge in $B(v)$. Then $B(v) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e\}$.*

Proof. By definition, we have that u is a proper descendant of v such that $e_{high}(u) \notin B(v)$, $e \notin B(u)$, $M(B(u) \setminus \{e_{high}(u)\}) = M(B(v) \setminus \{e\})$, and $bcount(u) = bcount(v)$. Let (x, y) be a back-edge in $B(v) \setminus \{e\}$. Then x is a descendant of $M(B(v) \setminus \{e\}) = M(B(u) \setminus \{e_{high}(u)\})$, and therefore a descendant of u. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Due to the generality of $(x, y) \in B(v) \setminus \{e\}$, this implies that $B(v) \setminus \{e\} \subseteq B(u)$. Since $e_{high}(u) \notin B(v)$, this can be strengthened to $B(v) \setminus \{e\} \subseteq B(u) \setminus \{e_{high}(u)\}$. Since $bcount(v) = bcount(u)$,

this implies that $B(v) \setminus \{e\} = B(u) \setminus \{e_{high}(u)\}$. Since $e_{high}(u) \notin B(v)$ and $e \notin B(u)$, this implies that $B(v) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e\}$. □

Lemma 5.36. *Let u and v be two vertices such that u is a proper descendant of v with $B(v) \sqcup \{e\} = B(u) \sqcup \{e'\}$, where e and e' are two distinct back-edges. Let $u' = lowestU(v, e')$. If $u' \neq u$, then $B(u') \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e_{high}(u')\}$.*

Proof. Since $B(v) \sqcup \{e\} = B(u) \sqcup \{e'\}$, we have $B(v) \setminus \{e'\} = B(u) \setminus \{e\}$, and therefore $M(B(v) \setminus \{e'\}) = M(B(u) \setminus \{e\})$. By Lemma 5.34, we have $e = e_{high}(u)$. Furthermore, $B(v) \sqcup \{e\} = B(u) \sqcup \{e'\}$ implies that $bcount(v) = bcount(u)$. Thus, it makes sense to consider the lowest proper descendant u' of v such that $e_{high}(u') \notin B(v)$, $e' \notin B(u')$, $M(B(u') \setminus \{e_{high}(u')\}) = M(B(v) \setminus \{e'\})$, and $bcount(u') = bcount(v)$. By definition, we have $u' = lowestU(v, e')$.

Let us assume that $u' \neq u$. Lemma 5.35 implies that $B(v) \sqcup \{e_{high}(u')\} = B(u') \sqcup \{e'\}$. From this we infer that $B(v) \setminus \{e'\} = B(u') \setminus \{e_{high}(u')\}$. Since $B(v) \setminus \{e'\} = B(u) \setminus \{e_{high}(u)\}$, this implies that $B(u') \setminus \{e_{high}(u')\} = B(u) \setminus \{e_{high}(u)\}$. If we assume that $e_{high}(u') \in B(u)$, then we get $e_{high}(u) = e_{high}(u')$ and $B(u') = B(u)$, in contradiction to the fact that the graph is 3-edge-connected. Similarly, we cannot have $e_{high}(u) \in B(u')$. Thus, $B(u') \setminus \{e_{high}(u')\} = B(u) \setminus \{e_{high}(u)\}$ implies that $B(u') \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e_{high}(u')\}$. □

The following two lemmata show how we can determine the vertex $u = lowestU(v, e)$. They correspond to two distinct cases, depending on whether $M(u) = M(B(u) \setminus \{e_{high}(u)\})$, or $M(u) \neq M(B(u) \setminus \{e_{high}(u)\})$.

Lemma 5.37. *Let u and v be two vertices such that $u = lowestU(v, e)$, where e is a back-edge in $B(v)$. Suppose that $M(u) = M(B(u) \setminus \{e_{high}(u)\})$. Then u is either the lowest or the second-lowest proper descendant of v such that $M(u) = M(B(v) \setminus \{e\})$.*

Proof. Let $z = M(B(v) \setminus \{e\})$. Since $u = lowestU(v, e)$, we have that $M(B(u) \setminus \{e_{high}(u)\}) = z$. Since $M(u) = M(B(u) \setminus \{e_{high}(u)\})$, this implies that $M(u) = z$. Now let us suppose, for the sake of contradiction, that u is neither the lowest nor the second-lowest proper descendant of v such that $M(u) = z$. This means that there are two proper descendants u' and u'' of v, such that $u > u' > u''$ and $M(u') = M(u'') = z$. Then we have that z is a common descendant of u, u' and u'', and therefore u, u' and u'' are related as ancestor and descendant. Thus, $u > u' > u''$ implies that u is a proper descendant of u', and u' is a proper descendant of u''. Then, since

$M(u) = M(u') = M(u'')$, Lemma 3.2 implies that $B(u'') \subseteq B(u') \subseteq B(u)$. Since the graph is 3-edge-connected, this can be strengthened to $B(u'') \subset B(u') \subset B(u)$. Thus, there is a back-edge $(x, y) \in B(u') \setminus B(u'')$, and a back-edge $(x', y') \in B(u) \setminus B(u')$. Since $(x, y) \in B(u')$ and $B(u') \subset B(u)$, we have that $(x, y) \in B(u)$. And since $(x', y') \notin B(u')$ and $(x, y) \in B(u')$, we have $(x, y) \neq (x', y')$. Thus, (x, y) and (x', y') are two distinct back-edges in $B(u)$. We have that x is a descendant of u, and therefore a descendant of u''. Thus, y cannot be a proper ancestor of v, because otherwise it is a proper ancestor of u'', and therefore $(x, y) \in B(u'')$. This shows that $(x, y) \notin B(v)$. Similarly, x' is a descendant of u, and therefore a descendant of u'. Thus, y' cannot be a proper ancestor of v, because otherwise it is a proper ancestor of u', and therefore $(x', y') \in B(u')$. This shows that $(x', y') \notin B(v)$. Since $u = \mathit{lowestU}(v, e)$, Lemma 5.35 implies that $B(v) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e\}$. This implies that there is only one back-edge in $B(u) \setminus B(v)$. But this contradicts the fact that (x, y) and (x', y') are two distinct back-edges in $B(u) \setminus B(v)$. Thus, we conclude that u is either the lowest or the second-lowest proper descendant of v such that $M(u) = M(B(v) \setminus \{e\})$. □

Lemma 5.38. *Let u and v be two vertices such that $u = \mathit{lowestU}(v, e)$, where e is a back-edge in $B(v)$. Suppose that $M(u) \neq M(B(u) \setminus \{e_{high}(u)\})$. Then u is the lowest proper descendant of v such that $M(u) \neq M(B(u) \setminus \{e_{high}(u)\}) = M(B(v) \setminus \{e\})$.*

Proof. Let $z = M(B(v) \setminus \{e\})$. Since $u = \mathit{lowestU}(v, e)$, we have that $M(B(u) \setminus \{e_{high}(u)\}) = z$. Now let us suppose, for the sake of contradiction, that u is not the lowest proper descendant of v such that $M(u) \neq M(B(u) \setminus \{e_{high}(u)\}) = z$. This means that there is a proper descendant u' of v, such that $u > u'$ and $M(u') \neq M(B(u') \setminus \{e_{high}(u')\}) = z$. Then we have that z is a common descendant of u and u', and therefore u and u' are related as ancestor and descendant. Thus, $u > u'$ implies that u is a proper descendant of u'.

Let us suppose, for the sake of contradiction, that $e_{high}(u) \in B(u')$. This implies that $\mathit{high}(u)$ is a proper ancestor of u'. Let (x, y) be a back-edge in $B(u)$. Then x is a descendant of u, and therefore a descendant of u'. Furthermore, y is an ancestor of $\mathit{high}(u)$, and therefore a proper ancestor of u'. This shows that $(x, y) \in B(u')$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(u')$. Since $M(u) \neq M(B(u) \setminus \{e_{high}(u)\}) = z$, we have that the higher endpoint of $e_{high}(u)$ is not a descendant of z. Similarly, since $M(u') \neq M(B(u') \setminus \{e_{high}(u')\}) = z$, we have that the higher endpoint of $e_{high}(u')$ is not a descendant of z. Furthermore, $e_{high}(u')$ is the only back-edge in $B(u')$

with this property. Since $e_{high}(u) \in B(u')$, this implies that $e_{high}(u') = e_{high}(u)$. Now let (x, y) be a back-edge in $B(u')$. Then, if $(x, y) = e_{high}(u')$, we have that $(x, y) = e_{high}(u) \in B(u)$. Otherwise, we have that x is a descendant of z, and therefore a descendant of u. Furthermore, y is a proper ancestor of u', and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Due to the generality of $(x, y) \in B(u')$, this implies that $B(u') \subseteq B(u)$. Thus, we have $B(u') = B(u)$, in contradiction to the fact that the graph is 3-edge-connected. This shows that $e_{high}(u) \notin B(u')$.

Let us suppose, for the sake of contradiction, that $e_{high}(u') \in B(u)$. Since $M(u') \neq M(B(u') \setminus \{e_{high}(u')\}) = z$, we have that the higher endpoint of $e_{high}(u')$ is not a descendant of z. Since $M(u) \neq M(B(u) \setminus \{e_{high}(u)\}) = z$, we have that $e_{high}(u)$ is the only back-edge in $B(u)$ with the property that its higher endpoint is not a descendant of z. Thus, since $e_{high}(u') \in B(u)$, we have that $e_{high}(u) = e_{high}(u')$. But this implies that $e_{high}(u) \in B(u')$, a contradiction. Thus, we have $e_{high}(u') \notin B(u)$.

Let us suppose, for the sake of contradiction, that $e_{high}(u') \in B(v)$. This implies that $high(u')$ is a proper ancestor of v. Since $u = lowestU(v, e)$, by Lemma 5.35 we have that $B(v) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e\}$. This implies that e is the only back-edge in $B(v)$ that is not in $B(u)$. Since $e_{high}(u') \notin B(u)$ and $e_{high}(u') \in B(v)$, this implies that $e = e_{high}(u')$. Now let (x, y) be a back-edge in $B(u')$. Then, x is a descendant of u', and therefore a descendant of v. Furthermore, y is an ancestor of $high(u')$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u')$, this implies that $B(u') \subseteq B(v)$. Conversely, let (x, y) be a back-edge in $B(v)$. If $(x, y) = e$, then $e = e_{high}(u') \in B(u')$. Otherwise, x is a descendant of $M(B(v) \setminus \{e\}) = z$, and therefore a descendant of u'. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of u'. This shows that $(x, y) \in B(u')$. Due to the generality of $(x, y) \in B(v)$, this implies that $B(v) \subseteq B(u')$. Thus, we have $B(u') = B(v)$, in contradiction to the fact that the graph is 3-edge-connected. This shows that $e_{high}(u') \notin B(v)$.

Let us suppose, for the sake of contradiction, that $e \in B(u')$. Since $u = lowestU(v, e)$, by Lemma 5.35 we have that $B(v) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e\}$. This implies that $e \notin B(u)$. The lower endpoint of e is a proper ancestor of v, and therefore a proper ancestor of u. Thus, since $e \notin B(u)$, we have that the higher endpoint of e is not a descendant of u. This implies that the higher endpoint of e is not a descendant of z either. Since $M(u') \neq M(B(u') \setminus \{e_{high}(u')\}) = z$, we have that $e_{high}(u')$ is the only back-edge in $B(u')$ whose higher endpoint is not a descendant of z. Since $e \in B(u')$, this shows

that $e_{high}(u') = e$. But this implies that $e_{high}(u') \in B(v)$, a contradiction. Thus, we have $e \notin B(u')$.

Now let (x, y) be a back-edge in $B(u') \setminus \{e_{high}(u')\}$. Then x is a descendant of z, and therefore a descendant of u. Furthermore, y is a proper ancestor of u', and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Due to the generality of $(x, y) \in B(u') \setminus \{e_{high}(u')\}$, this implies that $B(u') \setminus \{e_{high}(u')\} \subseteq B(u)$. And since $e_{high}(u) \notin B(u')$, this can be strengthened to $B(u') \setminus \{e_{high}(u')\} \subseteq B(u) \setminus \{e_{high}(u)\}$. Since $u = lowestU(v, e)$, by Lemma 5.35 we have that $B(v) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e\}$. This implies that $B(u) \setminus \{e_{high}(u)\} = B(v) \setminus \{e\}$. Thus, we have $B(u') \setminus \{e_{high}(u')\} \subseteq B(v) \setminus \{e\}$. Conversely, let (x, y) be a back-edge in $B(v) \setminus \{e\}$. Then x is a descendant of z, and therefore a descendant of u'. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of u'. This shows that $(x, y) \in B(u')$. Due to the generality of $(x, y) \in B(v) \setminus \{e\}$, this implies that $B(v) \setminus \{e\} \subseteq B(u')$. And since $e_{high}(u') \notin B(v)$, this can be strengthened to $B(v) \setminus \{e\} \subseteq B(u') \setminus \{e_{high}(u')\}$. Thus, we have $B(v) \setminus \{e\} = B(u') \setminus \{e_{high}(u')\}$. This implies that $bcount(v) = bcount(u')$. Thus, we have that u' is a proper descendant of v such that $e_{high}(u') \notin B(v)$, $e \notin B(u')$, $M(B(u') \setminus \{e_{high}(u')\}) = M(B(v) \setminus \{e\})$, and $bcount(v) = bcount(u')$. But since u' is lower than u, we have a contradiction to the minimality of $u = lowestU(v, e)$.

Thus, we conclude that u is the lowest proper descendant of v such that $M(u) \neq M(B(u) \setminus \{e_{high}(u)\}) = M(B(v) \setminus \{e\})$. □

Proposition 5.12. *Algorithm 24 computes a collection $\mathcal{C}$ of Type-$2ii$ 4-cuts, such that every Type-$2ii$ 4-cut of the form $\{(u, p(u)), (v, p(v)), e_1, e_2\}$, where $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$, is implied by $\mathcal{C}$ through the pair of edges $\{(u, p(u)), e_1\}$. Furthermore, Algorithm 24 has a linear-time implementation.*

Proof. First, we observe that all 4-element sets marked by Algorithm 24 are Type-$2ii$ 4-cuts. To see this, notice that the markings take place in Lines 16, 20, 24, 28, 32, or 36. In Lines 16 and 28, we have that $u = u_{highL}(v)$ or $u = u_{highR}(v)$, respectively, and so u is a proper descendant of v such that $M(B(u) \setminus \{e_{high}(u)\}) = M(B(v) \setminus \{e\})$, where $e = e_L(v)$ or $e = e_R(v)$, respectively, and $bcount(u) = bcount(v)$. Thus, Lemma 5.32 implies that $B(v) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e\}$, and therefore Lemma 5.28 implies that $\{(u, p(u)), (v, p(v)), e_{high}(u), e\}$ is a Type-$2ii$ 4-cut. In Lines 20, 24, 32, and 36, we have that $u = u_L(v)$, $u = prevM(u_L(v))$, $u = u_R(v)$, or $u = prevM(u_R(v))$, respectively, and so u is also a proper descendant of v. Furthermore, since the conditions in Lines 19,

Algorithm 24: Compute a collection of Type-$2ii$ 4-cuts, which implies all Type-$2ii$ 4-cuts

1 **foreach** *vertex* $v \neq r$ **do**
2 compute $M_L(v) = M(B(v) \setminus \{e_L(v)\})$ and $M_R(v) = M(B(v) \setminus \{e_R(v)\})$
3 **end**
4 **foreach** *vertex* $u \neq r$ **do**
5 compute $M_{high}(u) = M(B(u) \setminus \{e_{high}(u)\})$
6 **end**
7 **foreach** *vertex* $v \neq r$ **do**
8 compute the lowest proper descendant u of v with $M(u) \neq M_{high}(u) = M_L(v)$; denote this vertex as $u_{highL}(v)$
9 compute the lowest proper descendant u of v with $M(u) \neq M_{high}(u) = M_R(v)$; denote this vertex as $u_{highR}(v)$
10 compute the lowest proper descendant u of v with $M(u) = M_L(v)$; denote this vertex as $u_L(v)$
11 compute the lowest proper descendant u of v with $M(u) = M_R(v)$; denote this vertex as $u_R(v)$
12 **end**
13 **foreach** *vertex* $v \neq r$ **do**
14 let $u \leftarrow u_{highL}(v)$
15 **if** $bcount(u) = bcount(v)$ **then**
16 mark $\{(u,p(u)),(v,p(v)),e_{high}(u),e_L(v)\}$ as a Type-$2ii$ 4-cut
17 **end**
18 let $u \leftarrow u_L(v)$
19 **if** $M_{high}(u) = M_L(v)$ ***and*** $bcount(u) = bcount(v)$ **then**
20 mark $\{(u,p(u)),(v,p(v)),e_{high}(u),e_L(v)\}$ as a Type-$2ii$ 4-cut
21 **end**
22 let $u \leftarrow prevM(u)$
23 **if** $M_{high}(u) = M_L(v)$ ***and*** $bcount(u) = bcount(v)$ **then**
24 mark $\{(u,p(u)),(v,p(v)),e_{high}(u),e_L(v)\}$ as a Type-$2ii$ 4-cut
25 **end**
26 let $u \leftarrow u_{highR}(v)$
27 **if** $bcount(u) = bcount(v)$ **then**
28 mark $\{(u,p(u)),(v,p(v)),e_{high}(u),e_R(v)\}$ as a Type-$2ii$ 4-cut
29 **end**
30 let $u \leftarrow u_R(v)$
31 **if** $M_{high}(u) = M_R(v)$ ***and*** $bcount(u) = bcount(v)$ **then**
32 mark $\{(u,p(u)),(v,p(v)),e_{high}(u),e_R(v)\}$ as a Type-$2ii$ 4-cut
33 **end**
34 let $u \leftarrow prevM(u)$
35 **if** $M_{high}(u) = M_R(v)$ ***and*** $bcount(u) = bcount(v)$ **then**
36 mark $\{(u,p(u)),(v,p(v)),e_{high}(u),e_R(v)\}$ as a Type-$2ii$ 4-cut
37 **end**
38 **end**

23, 31, or 35, respectively, are satisfied, we have that $M(B(u) \setminus \{e_{high}(u)\}) = M(B(v) \setminus \{e\})$, where $e = e_L(v)$ or $e = e_R(v)$, and $bcount(u) = bcount(v)$. Thus, Lemma 5.32 implies that $B(v) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e\}$, and therefore Lemma 5.28 implies that $\{(u, p(u)), (v, p(v)), e_{high}(u), e\}$ is a Type-$2ii$ 4-cut. Thus, the collection $\mathcal{C}$ of 4-element sets marked by Algorithm 24 is a collection of Type-$2ii$ 4-cuts.

Now let $\{(u, p(u)), (v, p(v)), e_1, e_2\}$ be a Type-$2ii$ 4-cut. Then we may assume w.l.o.g. that u is a proper descendant of v, and $B(v) \sqcup \{e_1\} = B(u) \sqcup \{e_2\}$. This implies that $B(u) \setminus \{e_1\} = B(v) \setminus \{e_2\}$, and therefore $M(B(u) \setminus \{e_1\}) = M(B(v) \setminus \{e_2\})$. Furthermore, we have $e_1 \neq e_2$, and $B(u) \sqcup \{e_2\} = B(v) \sqcup \{e_1\}$ implies that $bcount(v) = bcount(u)$. By Lemma 5.34 we have that $e_1 = e_{high}(u)$. Thus, it makes sense to consider the lowest proper descendant u' of v such that $e_{high}(u') \notin B(v)$, $e_2 \notin B(u')$, $M(B(u') \setminus \{e_{high}(u')\}) = M(B(v) \setminus \{e_2\})$ and $bcount(u') = bcount(v)$. In other words, we have that $u' = lowestU(v, e_2)$ is defined. Then, Lemma 5.35 implies that $\{(u', p(u')), (v, p(v)), e_{high}(u'), e_2\}$ is also a Type-$2ii$ 4-cut. We will show that $\{(u', p(u')), (v, p(v)), e_{high}(u'), e_2\}$ is marked by Algorithm 24.

By Lemma 5.34 we have that either $e_2 = e_L(v)$, or $e_2 = e_R(v)$. Let us assume that $e_2 = e_L(v)$ (the argument for $e_2 = e_R(v)$ is similar). If $M(u') \neq M(B(u') \setminus \{e_{high}(u')\})$, then Lemma 5.38 implies that u' is the lowest proper descendant of v such that $M(u') \neq M(B(u') \setminus \{e_{high}(u')\}) = M(B(v) \setminus \{e_L(v)\})$. Thus, we have $u' = u_{highL}(v)$ (see Line 8), and therefore $\{(u', p(u')), (v, p(v)), e_{high}(u'), e_L(v)\}$ will be marked in Line 16. On the other hand, if $M(u') = M(B(u') \setminus \{e_{high}(u')\})$, then Lemma 5.37 implies that u' is either the lowest or the second-lowest proper descendant of v such that $M(u') = M(B(v) \setminus \{e_L(v)\})$. Thus, we have that either $u' = u_L(v)$ or $u' = prevM(u_L(v))$ (see Line 10), and therefore $\{(u', p(u')), (v, p(v)), e_{high}(u'), e_L(v)\}$ will be marked in either Line 20 or Line 24.

Let us rephrase our result so far in a more succinct notation. let $v \neq r$ be a vertex and let $\tilde{e}$ be a back-edge in $B(v)$, such that there is a Type-$2ii$ 4-cut of the form $\{(u, p(u)), (v, p(v)), e, \tilde{e}\}$, where u is a proper descendant of v. Then, we may consider the lowest proper descendant u' of v such that there is a Type-$2ii$ 4-cut of the form $\{(u', p(u')), (v, p(v)), e', \tilde{e}\}$. Let $C(v, \tilde{e})$ denote $\{(u', p(u')), (v, p(v)), e', \tilde{e}\}$. Then, we have basically shown that $C(v, \tilde{e}) \in \mathcal{C}$. We denote u' as $U(v, \tilde{e})$. Notice that, by Lemma 5.34, we have $e' = e_{high}(u')$.

Now, let $C = \{(u, p(u)), (v, p(v)), e, \tilde{e}\}$ be a Type-$2ii$ 4-cut such that u is a proper descendant of v and $B(u) \sqcup \{\tilde{e}\} = B(v) \sqcup \{e\}$. If $C \in \mathcal{C}$, then we have that $\mathcal{C}$ implies

C through any partition of C into pairs of edges. So let us suppose that $C \notin \mathcal{C}$. By Lemma 5.34, we have $e = e_{high}(u)$. Now consider the 4-cut $C_1 = C(v, \tilde{e}) \in \mathcal{C}$, and let $u_1 = U(v, \tilde{e})$. Then we have $C_1 = \{(u_1, p(u_1)), (v, p(v)), e_{high}(u_1), \tilde{e}\}$ and $B(u_1) \sqcup \{\tilde{e}\} = B(v) \sqcup \{e_{high}(u_1)\}$. Since $C \notin \mathcal{C}$, we have $u \neq u_1$. Thus, Lemma 5.36 implies that $B(u) \sqcup \{e_{high}(u_1)\} = B(u_1) \sqcup \{e_{high}(u)\}$. Notice that u and u_1 are related as ancestor and descendant. (To see this, consider any back-edge $(x, y) \in B(u) \setminus \{e_{high}(u)\} = B(u_1) \setminus \{e_{high}(u_1)\}$. Then we have that x is a common descendant of u and u_1, and therefore u and u_1 are related as ancestor and descendant.) Therefore, Lemma 5.28 implies that $C'_1 = \{(u, p(u)), (u_1, p(u_1)), e, e_{high}(u_1)\}$ is a Type-$2ii$ 4-cut. If $C'_1 \in \mathcal{C}$, then we have that C is implied by $\mathcal{C}$ through the pair of edges $\{(u, p(u)), e\}$, since $\{C_1, C'_1\} \subseteq \mathcal{C}$. So let us suppose that $C'_1 \notin \mathcal{C}$.

Due to the mimimality of u_1, we have $u_1 < u$. Thus, u is a proper descendant of u_1. Therefore, we can consider the 4-cut $C_2 = C(u_1, e_{high}(u_1))$, and let $u_2 = U(u_1, e_{high}(u_1))$. Then we have $C_2 = \{(u_2, p(u_2)), (u_1, p(u_1)), e_{high}(u_2), e_{high}(u_1)\}$ and $B(u_2) \sqcup \{e_{high}(u_1)\} = B(u_1) \sqcup \{e_{high}(u_2)\}$. Since $C_1 \notin \mathcal{C}$, we have $u \neq u_2$. Thus, Lemma 5.36 implies that $B(u) \sqcup \{e_{high}(u_2)\} = B(u_2) \sqcup \{e_{high}(u)\}$. Therefore, Lemma 5.28 implies that $C'_2 = \{(u, p(u)), (u_2, p(u_2)), e, e_{high}(u_2)\}$ is a Type-$2ii$ 4-cut. If $C'_2 \in \mathcal{C}$, then C is implied by $\mathcal{C}$ through the pair of edges $\{(u, p(u)), e\}$, since $\{C_1, C_2, C'_2\} \subseteq \mathcal{C}$. Otherwise, we can proceed in the same manner, and eventually this process must terminate, because we consider a proper descendant u_1 of v, then a proper descendant u_2 of u_1, and so on. Termination implies that we will have arrived at a sequence of 4-cuts $C_1, C_2, \dots, C_k, C'_k$ in $\mathcal{C}$, such that $C_1 = \{(u_1, p(u_1)), (v, p(v)), e_{high}(u_1), \tilde{e}\}$, $C_i = \{(u_i, p(u_i)), (u_{i-1}, p(u_{i-1})), e_{high}(u_i), e_{high}(u_{i-1})\}$ for every $i \in \{2, \dots, k\}$, and $C'_k = \{(u, p(u)), (u_k, p(u_k)), e, e_{high}(u_k)\}$. Thus, C is implied by $\mathcal{C}$ through the pair of edges $\{(u, p(u)), e\}$.

It remains to establish the linear complexity of Algorithm 24. First, Proposition 3.4 implies that we can compute the edges $e_L(v)$ and $e_R(v)$, for all vertices $v \neq r$, in linear time in total. Also, Proposition 3.3 implies that we can compute the edges $e_{high}(u)$, for all vertices $u \neq r$, in linear time in total. Then, by Proposition 3.6 we can compute the values $M(B(v) \setminus \{e_L(v)\})$ and $M(B(v) \setminus \{e_R(v)\})$, for all vertices $v \neq r$, in linear time in total. Thus, the **for** loop in Line 1 can be performed in linear time. Similarly, the **for** loop in Line 4 can also be performed in linear time. In order to compute the vertices $u_{highL}(v)$ and $u_{highR}(v)$ in Lines 8 and 9, respectively, we use Algorithm 22. Specifically, for every vertex x, let $M^{-1}_{high}(x)$ denote the set of all vertices $u \neq r$ such

that $M(u) \neq M(B(u) \setminus \{e_{high}(u)\}) = x$. Then, if x and y are two distinct vertices, we have that the sets $M^{-1}_{high}(x)$ and $M^{-1}_{high}(y)$ are disjoint. Now let $v \neq r$ be a vertex, and let $x = M(B(v) \setminus \{e_L(v)\})$. Then we generate a query $q(M^{-1}_{high}(x), v)$. This is to return the lowest vertex u such that $u \in M^{-1}_{high}(x)$ and $u > v$. This implies that $M(B(u) \setminus \{e_{high}(u)\}) = x = M(B(v) \setminus \{e_L(v)\})$, and therefore we have that x is a common descendant of u and v, and therefore u and v are related as ancestor and descendant. Then, $u > v$ implies that u is a proper descendant of v. Thus, we have that u is the lowest proper descendant of v such that $u \in M^{-1}_{high}(x)$, and therefore $u = u_{highL}(v)$. Since the number of all those queries is $O(n)$, Algorithm 22 can answer all of them in linear time in total, according to Lemma 5.27. Similarly, we can compute all vertices $u_{highR}(v)$, for $v \neq r$, in linear time in total. Also, we can similarly compute the vertices $u_L(v)$ and $u_R(v)$ in Lines 10 and 11, respectively, in linear time in total. We conclude that Algorithm 25 runs in linear time. □

5.6.3 The case $B(u) = B(v) \sqcup \{e_1, e_2\}$

Lemma 5.39. *Let u and v be two vertices such that v is a proper ancestor of u with $v \neq r$. Then there exist two distinct back-edges e_1, e_2 such that $B(u) = B(v) \sqcup \{e_1, e_2\}$ if and only if: $bcount(u) = bcount(v)+2$ and $M(B(u) \setminus \{e_1, e_2\}) = M(v)$, where $e_1 = (highD_1(u), high_1(u))$ and $e_2 = (highD_2(u), high_2(u))$ (or reversely).*

Proof. (⇒) $bcount(u) = bcount(v) + 2$ and $M(B(u) \setminus \{e_1, e_2\}) = M(v)$ are immediate consequences of $B(u) = B(v) \sqcup \{e_1, e_2\}$. Now let $(x_1, y_1), \dots, (x_k, y_k)$ be all the back-edges in $B(u)$ sorted in decreasing order w.r.t. their lower endpoint. (We note that $(x_i, y_i) = (highD_i(u), high_i(u))$, for every $i \in \{1, \dots, k\}$.) Let $i \in \{1, \dots, k\}$ be an index such that $(x_i, y_i) \in B(v)$. Then y_i is a proper ancestor of v, and therefore $y_i < v$. This implies that every y_j, for $j \in \{i, \dots, k\}$, has $y_j < v$. Since y_j is a proper ancestor of u and u is a descendant of v, this implies that y_j is a proper ancestor of v. Thus we have that all back-edges $(x_i, y_i), \dots, (x_k, y_k)$ are in $B(v)$, since all $x_1, \dots, x_k$ are descendants of v. Since $B(v) = B(u) \setminus \{e_1, e_2\}$, we have that exactly two back-edges in $B(u)$ are not in $B(v)$ (i.e., e_1 and e_2). Thus we have $\{e_1, e_2\} = \{(x_1, y_1), (x_2, y_2)\}$.

(⇐) Let $e_1 = (highD_1(u), high_1(u))$ and $e_2 = (highD_2(u), high_2(u))$. We have that $M(B(u) \setminus \{e_1, e_2\})$ is a descendant of $M(u)$, and therefore $M(v)$ is a descendant of $M(u)$. Thus, since v is an ancestor of u, by Lemma 3.2 we have that $B(v) \subseteq B(u)$. Since $bcount(u) = bcount(v) + 2$, this implies that there exist two back-edges

$e'_1, e'_2 \in B(u) \setminus B(v)$ such that $B(u) = B(v) \sqcup \{e'_1, e'_2\}$. By the $\Rightarrow$ direction we have $\{e'_1, e'_2\} = \{e_1, e_2\}$. $\square$

Lemma 5.40. *Let u and v be two vertices such that v is a proper ancestor of u and $B(u) = B(v) \sqcup \{e_1, e_2\}$, for two back-edges e_1, e_2. Then v is either the greatest or the second-greatest proper ancestor of u with $M(v) = M(B(u) \setminus \{e_1, e_2\})$.*

Proof. First, by Lemma 5.39 we have that $M(v) = M(B(u) \setminus \{e_1, e_2\})$. Thus, we may consider the greatest proper ancestor v' of u with $M(v') = M(B(u) \setminus \{e_1, e_2\})$. If $v' = v$, then we are done. Otherwise, let us suppose, for the sake of contradiction, that $v' \neq v$ and v is not the second-greatest proper ancestor of u with $M(v) = M(B(u) \setminus \{e_1, e_2\})$. Then there is a proper descendant v'' of v that is a proper ancestor of v', such that $M(v'') = M(B(u) \setminus \{e_1, e_2\})$. Since $M(v') = M(v'') = M(v)$, by Lemma 3.2 we have that $B(v) \subseteq B(v'') \subseteq B(v')$. This can be strengthened to $B(v) \subset B(v'') \subset B(v')$, since the graph is 3-edge-connected. This implies that there is a back-edge $e \in B(v'') \setminus B(v)$ and a back-edge $e' \in B(v') \setminus B(v'')$. Then neither of e and e' is in $B(v)$, but both of them are in $B(v')$.

Now let (x, y) be a back-edge in $B(v')$. Then we have that y is a proper ancestor of v', and therefore a proper ancestor of u. Furthermore, x is a descendant of $M(v')$, and therefore it is a descendant of $M(B(u) \setminus \{e_1, e_2\})$ (since $M(v) = M(B(u) \setminus \{e_1, e_2\})$), and therefore a descendant of u. This shows that (x, y) is in $B(u)$, and thus we have $B(v') \subseteq B(u)$. In particular, we have that both e and e' are in $B(u)$. But since none of them is in $B(v)$, by $B(u) = B(v) \sqcup \{e_1, e_2\}$ we have that $\{e, e'\} = \{e_1, e_2\}$. Now let (x, y) be a back-edge in $B(u)$. If $(x, y) = e_1$ or $(x, y) = e_2$, then $(x, y) \in B(v')$. Otherwise, $B(u) = B(v) \sqcup \{e_1, e_2\}$ implies that $(x, y) \in B(v)$, and therefore $B(v) \subseteq B(v'') \subseteq B(v')$ implies that $(x, y) \in B(v')$. This shows that $B(u) \subseteq B(v')$. Thus we have $B(v') = B(u)$, in contradiction to the fact that the graph is 3-edge-connected. Thus, we have that v is the second-greatest proper ancestor of u with $M(v) = M(B(u) \setminus \{e_1, e_2\})$. $\square$

Now we can describe the method to compute all 4-cuts of the form $\{(u, p(u)), (v, p(v)), e_1, e_2\}$, where v is an ancestor of u and $B(u) = B(v) \sqcup \{e_1, e_2\}$. The idea is to find, for every vertex u, a good candidate proper ancestor v of u that may provide such a 4-cut. According to Lemma 5.40, v must be either the greatest or the second-greatest proper ancestor of u that satisfies $M(v) = M(B(u) \setminus \{e_1, e_2\})$, where e_i is the $high_i$-edge of u, for $i \in \{1, 2\}$. Then, if such a v exists, we can simply apply Lemma 5.39 in order to check whether u and v satisfy $B(u) = B(v) \sqcup \{e_1, e_2\}$.

This procedure is implemented in Algorithm 25. The proof of correctness and linear complexity is given in Proposition 5.13.

Algorithm 25: Compute all 4-cuts of the form $\{(u,p(u)),(v,p(v)),e_1,e_2\}$, where v is an ancestor of u and $B(u) = B(v) \sqcup \{e_1,e_2\}$

```
1  foreach vertex u ≠ r do
2  |   let e_i(u) ← (highD_i(u), high_i(u)), for i ∈ {1,2}
3  |   compute M(B(u) \ {e_1(u), e_2(u)})
4  end
5  foreach vertex u ≠ r do
6  |   let v be the greatest proper ancestor of u such that
   |     M(v) = M(B(u) \ {e_1(u), e_2(u)})
7  |   if bcount(u) = bcount(v) + 2 then
8  |   |   mark {(u,p(u)), (v,p(v)), e_1(u), e_2(u)} as a 4-cut
9  |   end
10 |   let v ← prevM(v)
11 |   if v is an ancestor of u and bcount(u) = bcount(v) + 2 then
12 |   |   mark {(u,p(u)), (v,p(v)), e_1(u), e_2(u)} as a 4-cut
13 |   end
14 end
```

Proposition 5.13. *Algorithm 25 computes all 4-cuts of the form $\{(u,p(u)),(v,p(v)),e_1,e_2\}$, where v is an ancestor of u and $B(u) = B(v) \sqcup \{e_1,e_2\}$. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u,p(u)),(v,p(v)),e_1,e_2\}$ be a 4-cut such that u is a descendant of v and $B(u) = B(v) \sqcup \{e_1,e_2\}$. Then Lemma 5.39 implies that $\{e_1,e_2\} = \{e_{high1}(u), e_{high2}(u)\}$ and $bcount(u) = bcount(v) + 2$. Furthermore, Lemma 5.40 implies that v is either the greatest or the second-greatest proper ancestor of u such that $M(v) = M(B(u) \setminus \{e_1,e_2\})$. Thus, it is clear that, if v is the greatest proper ancestor of u with $M(v) = M(B(u) \setminus \{e_1,e_2\})$, then C will be marked in Line 8. Otherwise, we have that v is the predecessor of v' in $M^{-1}(M(v))$, where v' is the greatest proper ancestor of u such that $M(v') = M(B(u) \setminus \{e_1(u),e_2(u)\})$. Thus, C will be marked in Line 12.

Conversely, let $C = \{(u,p(u)),(v,p(v)),e_1,e_2\}$ be a 4-element set that is marked

in Line 8 or 12. In either case, we have that v is a proper ancestor of u such that $M(v) = M(B(u)\setminus\{e_{high1}(u), e_{high2}(u)\})$ and $bcount(u) = bcount(v)+2$. Thus, Lemma 5.39 implies that $B(u) = B(v) \sqcup \{e_{high1}(u), e_{high2}(u)\}$, and therefore C is correctly marked as a 4-cut.

Now we will show that Algorithm 25 runs in linear time. By Proposition 3.6 we have that the values $M(B(u) \setminus \{e_{high1}(u), e_{high2}(u)\})$ can be computed in linear time in total, for all vertices $u \neq r$. Thus, the **for** loop in Line 1 is performed in linear time. In order to compute the vertex v in Line 6, we use Algorithm 22. Specifically, let $u \neq r$ be a vertex, and let $x = M(B(u) \setminus \{e_{high1}(u), e_{high2}(u)\})$. Then we generate a query $q(M^{-1}(x), u)$. This returns the greatest vertex v such that $M(v) = x$ and $v < u$. Notice that, since $M(v) = x$, we have that $M(v)$ is a common descendant of u and v, and therefore u and v are related as ancestor and descendant. Then, $v < u$ implies that v is a proper ancestor of u. Thus, we have that v is the greatest proper ancestor of u such that $M(v) = M(B(u) \setminus \{e_{high1}(u), e_{high2}(u)\})$. Since the number of all those queries is $O(n)$, Algorithm 22 can answer all of them in linear time in total, according to Lemma 5.27. We conclude that Algorithm 25 runs in linear time. □

5.7 Computing Type-3α 4-cuts

Throughout this section, we assume that G is a 3-edge-connected graph with n vertices and m edges. All graph-related elements (e.g., vertices, edges, cuts, etc.) refer to G. Furthermore, we assume that we have computed a DFS-tree T of G rooted at a vertex r.

Lemma 5.41. *Let u, v, w be three vertices $\neq r$ such that w is a common ancestor of $\{u, v\}$, and u, v are not related as ancestor and descendant. Then there is a back-edge e such that $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ is a 4-cut if and only if either (i) $e \in B(u) \cup B(v)$ and $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$, or (ii) $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$.*

Proof. ($\Rightarrow$) Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$. Since w is a common ancestor of $\{u, v\}$, and u, v are not related as ancestor and descendant, we have that the subtrees $T(u)$ and $T(v)$, and the tree-paths $T[p(u), w]$ and $T[p(v), w]$, remain intact in $G \setminus C$. Let $e = (x, y)$. Since C is a 4-cut and e is a back-edge in C, by Lemma 3.14 we have that e is either in $B(u)$, or in $B(v)$, or in $B(w)$.

Let us assume first that $e \in B(u)$. Then x is a descendant of u, and therefore it cannot be a descendant of v (since u and v are not related as ancestor and descendant). Thus we have $e \notin B(v)$. Now let us assume, for the sake of contradiction, that $e \in B(w)$. Since C is a 4-cut, we have that $G' = G \setminus \{(u, p(u)), (v, p(v)), (w, p(w))\}$ is connected. In particular, u is connected with $p(u)$ in G'. Suppose first that there is a back-edge $(x', y') \in B(u)$ such that $y' \in T[p(u), w]$. Then u is still connected with $p(u)$ in $G' \setminus e$, contradicting the fact that C is a 4-cut of G. Thus, every back-edge $(x', y') \in B(u)$ must satisfy that y' is a proper ancestor of w. Similarly, we have that every back-edge $(x', y') \in B(v)$ must satisfy that y' is a proper ancestor of w. Thus, since u is connected with $p(u)$ in G', there must exist a back-edge $(x', y') \in B(w)$ such that x' is not a descendant of u or v. In particular, $(x', y') \neq e$. But now, by removing e from G', we can see that w remains connected with $p(w)$ through the path $T[w, x'], (x', y'), T[y', p(w)]$, in contradiction to the fact that C is a 4-cut of G. This shows that $e \notin B(w)$. Now it is not difficult to see that every back-edge in $(B(u) \setminus \{e\}) \cup B(v)$ must be in $B(w)$, for otherwise C is not a 4-cut of G. And conversely, every back-edge in $B(w)$ must be in $(B(u) \setminus \{e\}) \cup B(v)$, for otherwise C is not a 4-cut of G. Thus we have shown that $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$. Similarly, if we assume that $e \in B(v)$, we can use the analogous argument to show that $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$. In any case, we observe that we cannot have that e is both in $B(u) \cup B(v)$ and in $B(w)$ $(*)$.

Now let us assume that $e \in B(w)$. Then, by $(*)$ we have that $e \notin B(u) \cup B(v)$. Now let (x', y') be a back-edge in $B(u)$. Let us assume, for the sake of contradiction, that $y' \in T[p(u), w]$. Then notice that u is connected with $p(u)$ in the graph $G \setminus C$ through the path $T[u, x'], (x', y'), T[y', p(u)]$, in contradiction to the fact that C is a 4-cut of G. Thus we have that y' is a proper ancestor of w, and therefore $(x', y') \in B(w)$. This shows that $B(u) \subseteq B(w)$. Similarly, we have $B(v) \subseteq B(w)$ using the analogous argument. This shows that $B(u) \cup B(v) \subseteq B(w)$. Since e cannot be in $B(u) \cup B(v)$, this is strengthened to $B(u) \cup B(v) \subseteq B(w) \setminus \{e\}$. Conversely, let (x', y') be a back-edge in $B(w) \setminus \{e\}$. Let us assume that $(x', y') \notin B(u)$. This implies that x' is not a descendant of u (for otherwise we would have $(x', y') \in B(u)$, because y' is a proper ancestor of w, and therefore a proper ancestor of u). Let us suppose, for the sake of contradiction, that x' is not a descendant of v. But then we have that w is connected with $p(w)$ in $G \setminus C$ through the path $T[w, x'], (x', y'), T[y', p(w)]$, contradicting the fact that C is a 4-cut of G. Thus we have that x' is a descendant of v, and therefore $(x', y') \in B(v)$ (since y' is a proper ancestor of w, and therefore a proper ancestor of v). This means

that $B(w) \setminus \{e\} \subseteq B(u) \cup B(v)$, and therefore we have $B(w) \setminus \{e\} = B(u) \cup B(v)$. Since $e \in B(w) \setminus (B(u) \cup B(v))$, this implies that $B(w) = (B(u) \cup B(v)) \sqcup \{e\}$.

Finally, notice that the expression "$B(u) \cup B(v)$" can be strengthened to "$B(u) \sqcup B(v)$" everywhere, because u and v are not related as ancestor and descendant.

($\Leftarrow$) In the following, we let G' denote the graph $G \setminus \{(u, p(u)), (v, p(v)), (w, p(w))\}$. In every case, we will first show that all 3-set subsets of $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ are not 3-cuts of G. Then we will show that $G' \setminus e$ is disconnected.

Let us assume first that there is a back-edge $e \in B(u) \cup B(v)$ such that $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$. We may assume w.l.o.g. that $e \in B(u)$. Then we have that $e \notin B(v)$ and $e \notin B(w)$. Let $e = (x, y)$. Then this means that x is a descendant of u and y is a proper ancestor of u. Since u and v are not related as ancestor and descendant, we have that x is not a descendant of v. Therefore, the tree-path $T[x, u]$ remains intact in G'. Since $e \notin B(w)$, we have that y cannot be a proper ancestor of w (because otherwise we would have $(x, y) \in B(w)$, since x is a descendant of u, and therefore a descendant of w). Thus we have that u remains connected with $p(u)$ in G' through the path $T[u, x], (x, y), T[y, p(u)]$, and so $\{(u, p(u)), (v, p(v)), (w, p(w))\}$ is not a 3-cut of G. Since $e \notin B(v) \cup B(w)$, we have that the tree-path $T[x, y]$ remains intact in $G \setminus \{(v, p(v)), (w, p(w)), e\}$. Thus, the endpoints of e remain connected in $G \setminus \{(v, p(v)), (w, p(w)), e\}$, and so $\{(v, p(v)), (w, p(w)), e\}$ is not a 3-cut of G. Since the graph is 3-edge-connected, we have that there is a back-edge $(x', y') \in B(v)$. Since u and v are not related as ancestor and descendant, we have that the tree-paths $T[x', v]$ and $T[p(v), y']$ remain intact in $G \setminus \{(u, p(u)), (v, p(v)), e\}$. Thus, v remains connected with $p(v)$ in $G \setminus \{(u, p(u)), (v, p(v)), e\}$ through the path $T[v, x'], (x', y'), T[y', p(v)]$, and therefore $\{(u, p(u)), (v, p(v)), e\}$ is not a 3-cut of G. Since w is a proper ancestor of u, we have that w does not lie on the tree-path $T[x, u]$. And since y is not a proper ancestor of w, we have that y lies on the tree-path $T[p(u), w]$. Thus, u remains connected with $p(u)$ in $G \setminus \{(u, p(u)), (w, p(w)), e\}$ through the path $T[u, x], (x, y), T[y, p(u)]$. This shows that $\{(u, p(u)), (w, p(w)), e\}$ is not a 3-cut of G.

Now suppose that we remove e from G'. Consider the four parts $A = T(u)$, $B = T(v)$, $C = T(w) \setminus (T(u) \cup T(v))$ an $D = T(r) \setminus T(w)$. Observe that these parts are connected in $G' \setminus e$. Now, the only ways in which u can remain connected with $p(u)$ in $G' \setminus e$ are: (1) there is a back-edge from A to C, or (2) there is back-edge from A to D and a back-edge from D to C, or (3) there is a back-edge from A to D, a back-edge from D to B, and a back-edge from B to C. Possibility (1) is precluded from the

fact that $B(u) \setminus \{e\} \subseteq B(w)$. Possibility (2) is precluded from the fact that $B(w) \subset B(u) \sqcup B(v)$ (i.e., there is no back-edge from C to D). And possibility (3) is precluded from $B(v) \subset B(w)$ (i.e., there is no back-edge from B to C). Thus, we conclude that u is not connected with $p(u)$ in $G' \setminus e$, and so $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ is a 4-cut of G.

Now let us assume that there is a back-edge $e = (x,y)$ such that $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$. Then we have that $e \notin B(u) \cup B(v)$. This implies that x is not a descendant of u (because otherwise we would have $(x,y) \in B(u)$, since y is a proper ancestor of w, and therefore a proper ancestor of u). Similarly, we have that x is not a descendant of v. Thus, the tree-path $T[x,w]$ remains intact in G'. Furthermore, the tree-path $T[y,p(w)]$ remains intact in G'. This implies that w remains connected with $p(w)$ in G' through the path $T[w,x],(x,y),T[y,p(w)]$. Therefore, $\{(u,p(u)),(v,p(v)),(w,p(w))\}$ is not a 3-cut of G. Since $e \notin B(u) \cup B(v)$, we have that neither u nor v lies on the tree-path $T[x,y]$. Thus, the endpoints of e remain connected in $G \setminus \{(u,p(u)),(v,p(v)),e\}$, and so $\{(u,p(u)),(v,p(v)),e\}$ is not a 3-cut of G. Now consider the graph $G \setminus \{(u,p(u)),(w,p(w)),e\}$. Since G is 3-edge-connected, we have that there is a back-edge $(x',y') \in B(v)$. Then, $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that $(x',y') \in B(w)$ and $(x',y') \neq e$. Since u and v are not related as ancestor and descendant, we have that x' is not a descendant of u (because otherwise, x' would be a common descendant of u and v). Thus, u does not lie on the tree-path $T[x',w]$. Furthermore, we have that y' is a proper ancestor of w. Thus, w remains connected with $p(w)$ in $G \setminus \{(u,p(u)),(w,p(w)),e\}$, through the path $T[w,x'],(x',y'),T[y',p(w)]$. This shows that $\{(u,p(u)),(w,p(w)),e\}$ is not a 3-cut of G. Similarly, we can show that $\{(v,p(v)),(w,p(w)),e\}$ is not a 3-cut of G.

Now suppose that we remove e from G'. Consider the four parts $A = T(u)$, $B = T(v)$, $C = T(w) \setminus (T(u) \cup T(v))$ an $D = T(r) \setminus T(w)$. Observe that these parts are connected in $G' \setminus e$. Now, the only ways in which w can remain connected with $p(w)$ in $G' \setminus e$ are: (1) there is a back-edge from C to D, or (2) there is back-edge from C to A and a back-edge from A to D, or (3) there is a back-edge from C to B and a back-edge from B to D. Possibility (1) is precluded from the fact that $B(w) \setminus \{e\} = B(u) \sqcup B(v)$. Possibility (2) is precluded from the fact that $B(u) \subset B(w)$ (i.e., there is no back-edge from A to C). And possibility (3) is precluded from $B(v) \subset B(w)$ (i.e., there is no back-edge from B to C). Thus, we conclude that w is not connected with $p(w)$ in $G' \setminus e$, and so $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ is a 4-cut of G. □

According to Lemma 5.41, we distinguish two types of Type-3α 4-cuts: Type-$3\alpha i$ and Type-$3\alpha ii$. In both cases, the 4-cuts that we consider have the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where w is a common ancestor of $\{u,v\}$, and u,v are not related as ancestor and descendant. In the case of Type-$3\alpha i$ 4-cuts, we have that $e \in B(u) \sqcup B(v)$ and $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$. In Type-$3\alpha ii$ 4-cuts, we have $e \in B(w) \setminus (B(u) \sqcup B(v))$ and $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$.

5.7.1 Type-$3\alpha i$ 4-cuts

Lemma 5.42. *Let u,v,w be three vertices such that w is a common ancestor of $\{u,v\}$, and u,v are not related as ancestor and descendant. Suppose that there is a back-edge $e \in B(u) \sqcup B(v)$ such that $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ is a 4-cut. Then, either u is a descendant of the low1 child of $M(w)$ and v is a descendant of the low2 child of $M(w)$, or reversely.*

Proof. By the conditions of the lemma, we have that $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ is a Type-$3\alpha i$ 4-cut, and by Lemma 5.41 we have that $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$.

First we will show that both u and v are descendants of $M(w)$. Since u and v are not related as ancestor and descendant, we have that either none of them is an ancestor of $M(w)$, or one of them is an ancestor of $M(w)$, but the other is not related with $M(w)$ as ancestor or descendant. Suppose for the sake of contradiction that the second case is true, and assume w.l.o.g. that u is an ancestor of $M(w)$. This implies that v is not related as ancestor and descendant with $M(w)$. Since the graph is 3-edge-connected, we have that $|B(v)| \geq 2$. And since all back-edges in $B(v)$ are also in $B(w)$, except possibly e, we have that there is a back-edge $(x,y) \in B(v) \cap B(w)$. But then we have that x is a descendant of both v and $M(w)$, which implies that v and $M(w)$ are related as ancestor and descendant, a contradiction. Thus, we have that none of u and v is an ancestor of $M(w)$. The same argument also shows that it cannot be the case that one of u and v is not related as ancestor and descendant with $M(w)$. Thus we have that both u and v are proper descendants of $M(w)$.

Now let us assume, for the sake of contradiction, that $M(w)$ has less than two children. This implies that there is at least one back-edge of the form $(M(w),y)$ in $B(w)$, for a vertex y with $y < w$. Then $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$ implies that $e \notin B(w)$, and therefore $e \neq (M(w),y)$. But then, since both u and v are descendants of $M(w)$, we have that $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ cannot be a 4-cut, since w is connected

with $p(w)$ through the path $T[w, M(w)], (M(w), y), T[y, p(w)]$, a contradiction. This shows that $M(w)$ has at least two children. Furthermore, the same argument shows that there is no back-edge of the form $(M(w), y)$ in $B(w)$. Let c_1 and c_2 be the *low1* and *low2* children of $M(w)$, respectively.

Now let us assume, for the sake of contradiction, that u is neither a descendant of c_1 nor a descendant of c_2. We may assume w.l.o.g. that v is not a descendant of c_1 (because otherwise we have that v is not a descendant of c_2, and we can reverse the roles of c_1 and c_2 in the following). Since there is no back-edge of the form $(M(w), y)$ in $B(w)$, we have that there are at least two back-edges $(x_1, y_1), (x_2, y_2) \in B(w)$ such that x_1 is a descendant of c_1 and x_2 is a descendant of c_2. Since neither u nor v is a descendant of c_1, we have that $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ cannot be a 4-cut, since w is connected with $p(w)$ through the path $T[w, x_1], (x_1, y_1), T[y_1, p(w)]$, a contradiction. Thus we have shown that u is either a descendant of c_1 or a descendant of c_2. A similar argument shows that v is either a descendant of c_1 or a descendant of c_2. Furthermore, the same argument shows that it cannot be the case that both u and v are descendants of c_1, or that both of them are descendants of c_2. Thus the lemma follows. $\square$

In the following, we will assume w.l.o.g. that the back-edge of the Type-3αi 4-cuts that we consider leaps over u. Furthermore, we will consider the case that u is a descendant of the *low1* child of $M(w)$. The other case is treated similarly. Also, throughout this section we let $e(u)$ denote the back-edge $e_{high}(u)$.

Lemma 5.43. *Let u, v, w be three vertices such that w is a common ancestor of $\{u, v\}$, and u, v are not related as ancestor and descendant. Suppose that there is a back-edge $e \in B(u)$ such that $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ is a 4-cut. Then $e = (highD_1(u), high_1(u))$. Furthermore, suppose that u is a descendant of the low1 child c of $M(w)$. Then $M(B(u) \setminus \{e\}) = M(w, c)$.*

Proof. By Lemma 5.41 we have that $B(w) = (B(u) \setminus \{e\}) \sqcup B(v)$. Let $(x_1, y_1), \dots, (x_k, y_k)$ be the list of the back-edges that leap over u sorted in decreasing order w.r.t. their lower endpoint, so that $(x_1, y_1) = (highD_1(u), high_1(u))$. Let us assume, for the sake of contradiction, that $e = (x_i, y_i)$, for some $i \in \{2, \dots, k\}$. Since u is a descendant of w, we have that x_i is also a descendant of w. But since $e \notin B(w)$, we cannot have that y_i is a proper ancestor of w. But then we have that $(x_j, y_j) \notin B(w)$, for any $j \in \{1, \dots, i\}$, since y_j is a descendant of y_i. Since $i > 1$, this means that there are at

least two back-edges in $B(u) \setminus B(w)$, in contradiction to $B(w) = (B(u) \setminus \{e\}) \sqcup B(v)$. Thus we have that $e = (x_1, y_1)$.

Now let $(x, y) \in B(w)$ be a back-edge such that x is a descendant of c. By $B(w) = (B(u) \setminus \{e\}) \sqcup B(v)$ we have that either $(x, y) \in B(u) \setminus \{e\}$, or $(x, y) \in B(v)$. By Lemma 5.42 we have that v is a descendant of the *low2* child of $M(w)$, and so the second case is impossible (because otherwise we would have that x is a descendant of the *low2* child of $M(w)$). Thus we have that $(x, y) \in B(u) \setminus \{e\}$. Conversely, let (x, y) be a back-edge in $B(u) \setminus \{e\}$. Then by $B(w) = (B(u) \setminus \{e\}) \sqcup B(v)$ we have that $(x, y) \in B(w)$. And since u is a descendant of c, we have that x is also a descendant of c. Thus we have shown that $B(u) \setminus \{e\} = \{(x, y) \in B(w) \mid x \text{ is a descendant of } c\}$, and so we get $M(B(u) \setminus \{e\}) = M(w, c)$. □

Note 5.3. Notice that the argument in the proof of Lemma 5.43 works independently of the fact that c is the *low1* child of $M(w)$. In other words, we could have assumed that u is a descendant of the *low2* child c of $M(w)$. In this case, Lemma 5.42 would imply that v is a descendant of the *low1* child of $M(w)$, and we would still get $M(B(u) \setminus \{e\}) = M(w, c)$ with the same argument.

Lemma 5.44. *Let u, v, w be three vertices $\neq r$, such that w is a proper ancestor of $\{u, v\}$, and u, v are not related as ancestor and descendant. Let c_1 and c_2 be the low1 and low2 children of $M(w)$, respectively. Suppose that $\{(u, p(u)), (v, p(v)), (w, p(w)), e(u)\}$ is a 4-cut, and let us assume that u is a descendant of c_1. Then v is the lowest vertex with $M(v) = M(w, c_2)$ that is a proper descendant of w.*

Proof. Since $\{(u, p(u)), (v, p(v)), (w, p(w)), e(u)\}$ is a 4-cut, by Lemma 5.41 we have that $B(w) = (B(u) \setminus \{e(u)\}) \sqcup B(v)$. By Lemma 5.42 we have that v is a descendant of c_2. Let $S = \{(x, y) \in B(w) \mid x \text{ is a descendant of } c_2\}$. Then we have $M(w, c_2) = M(S)$. Now let (x, y) be a back-edge in $B(v)$. By $B(w) = (B(u) \setminus \{e(u)\}) \sqcup B(v)$ we have that $(x, y) \in B(w)$. And since v is a descendant of c_2, we have $(x, y) \in S$. Conversely, let (x, y) be a back-edge in S. Then by $B(w) = (B(u) \setminus \{e(u)\}) \sqcup B(v)$ we have that either $(x, y) \in B(u) \setminus \{(e(u))\}$ or $(x, y) \in B(v)$. But since x is a descendant of c_2, it cannot be the case that x is a descendant of u, because u is a descendant of c_1. Thus we have $(x, y) \in B(v)$. This shows that $S = B(v)$, and so we get $M(w, c_2) = M(v)$.

Now let us assume, for the sake of contradiction, that there is a proper ancestor v' of v with $M(v') = M(w, c_2)$, which is also a proper descendant of w. Then, since $M(v') = M(v)$, Lemma 3.2 implies that $B(v') \subseteq B(v)$. Now let (x, y) be a back-edge

in $B(v)$. Then, as previously, we have $(x, y) \in B(w)$. Thus, y is proper ancestor of w, and therefore a proper ancestor of v'. Furthermore, x is a descendant of v, and therefore a descendant of v'. This shows that $(x, y) \in B(v')$. Due to the generality of $(x, y) \in B(v)$, this implies that $B(v) \subseteq B(v')$. But then we get $B(v') = B(v)$, in contradiction to the fact that the graph is 3-edge-connected. Thus, v is the lowest proper descendant of w such that $M(v) = M(w, c_2)$. □

Here we distinguish two cases of Type-$3\alpha i$ 4-cuts, depending on whether $M(B(u)\setminus\{e(u)\}) = M(u)$ or $M(B(u)\setminus\{e(u)\}) \neq M(u)$. In the first case, we show how to compute all such 4-cuts in linear time. In the second case, the number of 4-cuts can be $\Omega(n^2)$. However, we show how to compute a collection of such 4-cuts in linear time, so that the rest of them are implied by this collection, plus that computed by Algorithm 24. A vertex u such that $M(B(u) \setminus \{e(u)\}) = M(u)$ is called a "special" vertex.

5.7.1.1 The case where $M(B(u) \setminus \{e_{high}(u)\}) = M(u)$

Lemma 5.45. *Let w, v be two vertices $\neq r$ such that w is a proper ancestor of v. Then there is at most one vertex u such that: u is a special vertex, it is a proper descendant of w, and $\{(u, p(u)), (v, p(v)), (w, p(w)), e(u)\}$ is a Type-3α 4-cut. If that is the case, assume w.l.o.g. that u is a descendant of the low1 child c of $M(w)$. Then u is either the lowest or the second-lowest proper descendant of w such that $M(u) = M(w, c)$.*

Proof. Let u be proper descendant of w such that u is a special vertex and $\{(u, p(u)), (v, p(v)), (w, p(w)), e(u)\}$ is a Type-3α 4-cut $(*)$. By Lemma 5.41 we have that $B(w) = (B(u) \setminus \{e(u)\}) \sqcup B(v)$. By Lemma 5.42 we may assume w.l.o.g. that u is a descendant of the *low1* child c of $M(w)$, and v is a descendant of the *low2* child of $M(w)$. By Lemma 5.43 we have that $M(B(u) \setminus \{e(u)\}) = M(w, c)$. Since u is a special vertex, we have $M(u) = M(B(u) \setminus \{e(u)\})$. Thus, $M(B(u) \setminus \{e(u)\}) = M(w, c)$ implies that $M(u) = M(w, c)$.

Now let us assume, for the sake of contradiction, that there is another vertex u' such that: u' is a special vertex, it is a proper descendant of w, and $\{(u', p(u')), (v, p(v)), (w, p(w)), e(u')\}$ is a Type-3α 4-cut. Then, since v is a descendant of the *low2* child of $M(w)$, by Lemma 5.42 we have that u' is a descendant of c. Thus, u' satisfies the same properties as u. This implies that $M(u') = M(w, c)$. Since u is an ancestor of $M(u) = M(w, c)$ and u' is an ancestor of $M(u') = M(w, c)$, we have that u and u' are related as ancestor and descendant (because they have a

common descendant). Let us assume w.l.o.g. that u' is a proper ancestor of u. Then, since $M(u) = M(u')$, Lemma 3.2 implies that $B(u') \subseteq B(u)$. Since the graph is 3-edge-connected, this can be strengthened to $B(u') \subset B(u)$. Thus, there is a back-edge (x, y) in $B(u) \setminus B(u')$. This implies that x is a descendant of u, and therefore a descendant of u'. Thus, y cannot be a proper ancestor of u'. Therefore, since y and u' are related as ancestor and descendant (since they have x as a common descendant), we have that y is a descendant of u'. Thus, since u' is a proper descendant of w, we have that y cannot be a proper ancestor of w, and so $(x, y) \notin B(w)$. Thus, since $(x, y) \in B(u)$, $B(w) = (B(u) \setminus \{e(u)\}) \sqcup B(v)$ implies that $(x, y) = e(u)$. Now, since $B(u') \subset B(u)$, we have $e(u') \in B(u)$. Since $(x, y) \notin B(u')$, we have $e(u') \neq (x, y)$. Also, we have $e(u') \notin B(w)$ (since $B(w) = (B(u') \setminus \{e(u')\}) \sqcup B(v)$, and $B(u') \cap B(v) = \emptyset$, because u' and v are descendants of different children of $M(w)$, and therefore they are not related as ancestor and descendant). Thus $B(u) \setminus B(w)$ contains at least two back-edges, in contradiction to $B(w) = (B(u) \setminus \{e(u)\}) \sqcup B(v)$. This shows that u is unique in satisfying property $(*)$.

Now let us suppose, for the sake of contradiction, that there are two distinct vertices u' and u'' that are lower than u, they are proper descendants of w, and satisfy $M(u') = M(u'') = M(u)$. Then we have that all u, u', u'' are related as ancestor and descendant. We may assume w.l.o.g. that $u'' < u'$. Thus, we have that u'' is a proper ancestor of u', and u' is a proper ancestor of u. Then, by Lemma 3.2 we have $B(u'') \subseteq B(u') \subseteq B(u)$. Since the graph is 3-edge-connected, this can be strengthened to $B(u'') \subset B(u') \subset B(u)$. Thus, there is a back-edge $(x, y) \in B(u) \setminus B(u')$, and a back-edge $(x', y') \in B(u') \setminus B(u'')$. Since $(x, y) \in B(u)$, we have that x is a descendant of u, and therefore a descendant of u'. Thus, since $(x, y) \notin B(u')$, it cannot be that y is a proper ancestor of u'. Similarly, since $(x', y') \in B(u') \setminus B(u'')$, it cannot be that y' is a proper ancestor of u''. Since both u' and u'' are proper descendants of w, this implies that neither y nor y' is a proper ancestor of w. Thus, $(x, y) \notin B(w)$ and $(x', y') \notin B(w)$. Since $(x, y) \notin B(u')$ and $(x', y') \in B(u')$, we have $(x, y) \neq (x', y')$. And since $(x', y') \in B(u')$ and $B(u') \subset B(u)$, we have $(x', y') \in B(u)$. But then (x, y) and (x', y') are two distinct back-edges in $B(u) \setminus B(w)$, in contradiction to $B(w) = ((B(u) \setminus \{e(u)\}) \sqcup B(v)$ (which implies that $B(u) \setminus B(w)$ consists of $e(u)$). This shows that u is either the lowest or the second-lowest proper descendant of w such that $M(u) = M(w, c)$. $\square$

Lemma 5.45 gives enough information to be able to compute efficiently all 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where w is common ancestor of $\{u,v\}$, u,v are not related as ancestor and descendant, $e \in B(u)$, and u is a special vertex.

This method is shown in Algorithm 26, for the case where u is a descendant of the *low1* child of $M(w)$. The case where u is a descendant of the *low2* child of $M(w)$ is treated similarly, by simply changing the roles of c_1 and c_2 in Lines 6 and 7, respectively. (I.e., we set "$c_1 \leftarrow$ *low2* child of $M(w)$" and "$c_2 \leftarrow$ *low1* child of $M(w)$".) The proof of correctness and linear complexity is given in Proposition 5.14.

Lemma 5.46. *Let u,v,w be three vertices such that w is a common ancestor of $\{u,v\}$, and u,v are not related as ancestor and descendant. Then, there is a back-edge $e \in B(u)$ such that $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ is a Type-3αi 4-cut if and only if: $high_2(u) < w$, $high_1(v) < w$, and $bcount(w) = bcount(u) + bcount(v) - 1$.*

Proof. ($\Rightarrow$) Since $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ is a Type-3αi 4-cut where $e \in B(u)$, we have that $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$. This implies that $bcount(w) + 1 = bcount(u) + bcount(v)$, from which we infer that $bcount(w) = bcount(u) + bcount(v) - 1$. Let (x,y) be a back-edge in $B(v)$. Then $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$ implies that $(x,y) \in B(w) \sqcup \{e\}$. Since $B(u) \cap B(v) = \emptyset$, we have that $(x,y) \neq e$. Thus, we have $(x,y) \in B(w)$. This implies that y is a proper ancestor of w, and therefore $y < w$. Due to the generality of $(x,y) \in B(v)$, this implies that $high_1(v) < w$. Now let us suppose, for the sake of contradiction, that $high_2(u) \geq w$. This implies that the *high1* and the *high2* edges of u are not in $B(w)$. But $e \in B(u)$ and $B(w) \sqcup \{e\} = B(u) \sqcup B(v)$ imply that precisely one back-edge from $B(u)$ is not in $B(w)$, a contradiction. Thus, we have $high_2(u) < w$.
($\Leftarrow$) Since v is a common descendant of $high_1(v)$ and w, we have that $high_1(v)$ and w are related as ancestor and descendant. Thus, $high_1(v) < w$ implies that $high_1(v)$ is a proper ancestor of w. Similarly, we have that $high_2(u)$ is a proper ancestor of w. Now, let (x,y) be a back-edge in $B(v)$. Then, x is a descendant of v, and therefore a descendant of w. Furthermore, y is an ancestor of $high_1(v)$, and therefore a proper ancestor of w. This shows that $(x,y) \in B(w)$. Due to the generality of $(x,y) \in B(v)$, this implies that $B(v) \subseteq B(w)$.

Now let $(x_1,y_1),\ldots,(x_k,y_k)$ be the list of the back-edges in $B(u)$ sorted in decreasing order w.r.t. their lower endpoint, so that we have $(x_1,y_1) = e(u)$. Let i be an index in $\{2,\ldots,k\}$. Then, we have that x_i is a descendant of u, and therefore a descendant of w. Furthermore, we have that $y_i \leq high_2(u)$, and therefore y_i is an ancestor of

high$_2(u)$, and therefore y_i is a proper ancestor of w. This shows that $(x_i, y_i) \in B(w)$. Thus, we have shown that $B(u) \setminus \{e(u)\} \subseteq B(w)$.

Since u and v are not related as ancestor and descendant, we have that $B(u) \cap B(v) = \emptyset$ (because otherwise, if there existed a back-edge in $B(u) \cap B(v)$, we would have that its higher endpoint would be a common descendant of both u and v). Thus, we have $(B(u) \setminus \{e(u)\}) \sqcup B(v) \subseteq B(w)$. Therefore, $\mathit{bcount}(w) = (\mathit{bcount}(u) - 1) + \mathit{bcount}(v)$ implies that $(B(u) \setminus \{e(u)\}) \sqcup B(v) = B(w)$. Since $B(u) \cap B(v) = \emptyset$, we have that $e(u) \notin B(v)$, and therefore $(B(u) \setminus \{e(u)\}) \sqcup B(v) = B(w)$ implies that $e(u) \notin B(w)$. Thus, $(B(u) \setminus \{e(u)\}) \sqcup B(v) = B(w)$ and $e(u) \notin B(v)$ imply that $B(u) \sqcup B(v) = B(w) \sqcup \{e(u)\}$. Thus, by Lemma 5.41 we have that $\{(u, p(u)), (v, p(v)), (w, p(w)), e(u)\}$ is a Type-3αi 4-cut. □

Proposition 5.14. *Algorithm 26 correctly computes all 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$, where w is a common ancestor of $\{u, v\}$, $e \in B(u)$, and u is a special vertex. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3αi 4-cut where w is a common ancestor of u and v, u is a special vertex, and $e \in B(u)$. Let c_1 and c_2 be the *low1* and the *low2* child of $M(w)$, respectively. Lemma 5.42 implies that either u is a descendant of c_1 and v is a descendant of c_2, or reversely. So let us assume w.l.o.g. that u is a descendant of c_1. Then Lemma 5.44 implies that v is the lowest proper descendant of w such that $M(v) = M(w, c_2)$. Lemma 5.43 implies that $e = e_{high}(u)$. Lemma 5.46 implies that $\mathit{bcount}(w) = \mathit{bcount}(u) + \mathit{bcount}(v) - 1$, $\mathit{high}_2(u) < w$ and $\mathit{high}_1(v) < w$. If we have that u is the lowest proper descendant of w such that $M(u) = M(w, c_1)$, then we can see that C will be marked in Line 12. Otherwise, by Lemma 5.45 we have that u is the second-lowest proper descendant of w such that $M(u) = M(w, c_1)$. This implies that $u = \mathit{prevM}(u')$, where u' is the lowest proper descendant of w such that $M(u') = M(w, c_1)$. Thus, C will be marked in Line 16.

Conversely, let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e(u)\}$ be a 4-element set that is marked in Line 12 or 16. Then, in either case we have $\mathit{bcount}(w) = \mathit{bcount}(u) + \mathit{bcount}(v) - 1$, $\mathit{high}_2(u) < w$ and $\mathit{high}_1(v) < w$. Furthermore, in either case we have $M(u) = M(w, c_1)$ and $M(v) = M(w, c_2)$. Therefore, Lemma 3.12 implies that u and v are not related as ancestor and descendant. Thus, Lemma 5.46 implies that C is indeed a Type-3αi 4-cut.

Algorithm 26: Compute all Type-3αi 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where w is a common ancestor of $\{u,v\}$, $e \in B(u)$, and u is a special vertex.

```
1  foreach vertex w ≠ r such that M(w) has at least two children do
2      compute M(w,c1) and M(w,c2), where c1 and c2 are the low1 and low2
         children of M(w), respectively
3  end
   // the case where u is a descendant of the low1 child of M(w); for the
      other case, simply reverse the roles of c1 and c2
4  foreach vertex w ≠ r do
5      if M(w) has less than two children then continue
6      let c1 ← low1 child of M(w)
7      let c2 ← low2 child of M(w)
8      if M(w,c1) = ⊥ or M(w,c2) = ⊥ then continue
9      let u be the lowest proper descendant of w such that M(u) = M(w,c1)
10     let v be the lowest proper descendant of w such that M(v) = M(w,c2)
11     if u is a special vertex and bcount(w) = (bcount(u) − 1) + bcount(v) and
         high_2(u) < w and high_1(v) < w then
12         mark {(u,p(u)),(v,p(v)),(w,p(w)),e(u)} as a 4-cut
13     end
14     u ← prevM(u)
15     if u is a special vertex and bcount(w) = (bcount(u) − 1) + bcount(v) and
         high_2(u) < w and high_1(v) < w then
16         mark {(u,p(u)),(v,p(v)),(w,p(w)),e(u)} as a 4-cut
17     end
18 end
```

Now we will argue about the complexity of Algorithm 26. By Proposition 3.5 we have that the values $M(w,c_1)$ and $M(w,c_2)$ can be computed in linear time in total, for all vertices $w \neq r$ such that $M(w)$ has at least two children, where c_1 and c_2 are the *low1* and *low2* children of $M(w)$ respectively. Thus, Line 1 can be performed in linear time. The vertices u and v in Lines 9 and 10 can be computed with Algorithm 22. Specifically, whenever we reach Line 9, we generate a query

$q(M^{-1}(M(w,c_1)),w)$. This will return the lowest vertex u with $M(u)=M(w,c_1)$ and $u>w$. Since $M(u)=M(w,c_1)$ implies that $M(u)$ is a common descendant of u and w, we have that u and w are related as ancestor and descendant. Thus, $u>w$ implies that u is a proper descendant of w. Thus, u is the lowest proper descendant of w such that $M(u)=M(w,c_1)$. We generate the analogous query to get v. Since the number of all those queries is $O(n)$, Algorithm 22 can answer all of them in $O(n)$ time, according to Lemma 5.27. We conclude that Algorithm 26 runs in linear time. $\square$

5.7.1.2 The case where $M(B(u)\setminus\{e_{high}(u)\})\neq M(u)$

Lemma 5.47. *Let u and u' be two distinct vertices $\neq r$ such that $M(u)\neq M(B(u)\setminus\{e(u)\})=M(B(u')\setminus\{e(u')\})\neq M(u')$. Then, $e(u)\notin B(u')$ and $e(u')\notin B(u)$. Furthermore, if $high_2(u)=high_2(u')$, then $B(u)\sqcup\{e(u')\}=B(u')\sqcup\{e(u)\}$.*

Proof. Let us assume w.l.o.g. that $u'<u$. Since the graph is 3-edge-connected, we have that $|B(u')|>1$. Thus, there is a back-edge $(x,y)\in B(u')\setminus\{e(u')\}$. Then, we have that x is a descendant of $M(B(u')\setminus\{e(u')\})=M(B(u)\setminus\{e(u)\})$, and therefore a descendant of $M(u)$, and therefore a descendant of u. Thus, x is a common descendant of u' and u, and therefore u and u' are related as ancestor and descendant. Thus, $u'<u$ implies that u' is a proper ancestor of u.

Let us suppose, for the sake of contradiction, that $e(u')\in B(u)$. Then, since $M(u')\neq M(B(u')\setminus\{e(u')\})$, we have that the higher endpoint of $e(u')$ is not a descendant of $M(B(u')\setminus\{e(u')\})$, and therefore it is not a descendant of $M(B(u)\setminus\{e(u)\})$. Furthermore, since $M(u)\neq M(B(u)\setminus\{e(u)\})$, we have that the higher endpoint of $e(u)$ is not a descendant of $M(B(u)\setminus\{e(u)\})$, and that this is the only back-edge in $B(u)$ with this property. Thus, since $e(u')\in B(u)$, we have that $e(u)=e(u')$. This implies that $high_1(u)=high_1(u')$. Thus, since u' is a proper ancestor of u, by Lemma 3.3 we have that $B(u)\subseteq B(u')$. Since the graph is 3-edge-connected, this can be strengthened to $B(u)\subset B(u')$. Thus, there is a back-edge $(x,y)\in B(u')\setminus B(u)$. Since $e(u')\in B(u)$ and $(x,y)\notin B(u)$, we have that $(x,y)\neq e(u')$. Thus, we have $(x,y)\in B(u')\setminus\{e(u')\}$, and therefore x is a descendant of $M(B(u')\setminus\{e(u')\})=M(B(u)\setminus\{e(u)\})$, and therefore a descendant of $M(u)$. Furthermore, y is a proper ancestor of u', and therefore a proper ancestor of u. This shows that $(x,y)\in B(u)$, a contradiction. Thus, we have shown that $e(u')\notin B(u)$. This implies that $e(u')\neq e(u)$.

Let us suppose, for the sake of contradiction, that $e(u)\in B(u')$. Then, since

$M(u') \neq M(B(u') \setminus \{e(u')\})$, we have that the higher endpoint of $e(u')$ is not a descendant of $M(B(u') \setminus \{e(u')\})$, and therefore it is not a descendant of $M(B(u) \setminus \{e(u)\})$. Furthermore, we have that $e(u')$ is the only back-edge in $B(u')$ with this property. Now, since $M(u) \neq M(B(u) \setminus \{e(u)\})$, we have that the higher endpoint of $e(u)$ is not a descendant of $M(B(u) \setminus \{e(u)\})$. Since $e(u) \in B(u')$, this implies that $e(u) = e(u')$, a contradiction. Thus, we have shown that $e(u) \notin B(u')$.

Now let (x, y) be a back-edge in $B(u) \setminus \{e(u)\}$. Then, x is a descendant of $M(B(u) \setminus \{e(u)\}) = M(B(u') \setminus \{e(u')\})$, and therefore a descendant of $M(u')$. Furthermore, y is an ancestor of $high_2(u) = high_2(u')$, and therefore a proper ancestor of u'. This shows that $(x, y) \in B(u')$. Due to the generality of $(x, y) \in B(u) \setminus \{e(u)\}$, this implies that $B(u) \setminus \{e(u)\} \subseteq B(u')$. And since $e(u') \notin B(u)$, this can be strengthened to $B(u) \setminus \{e(u)\} \subseteq B(u') \setminus \{e(u')\}$. Conversely, let (x, y) be a back-edge in $B(u') \setminus \{e(u')\}$. Then x is a descendant of $M(B(u') \setminus \{e(u')\}) = M(B(u) \setminus \{e(u)\})$, and therefore a descendant of $M(u)$. Furthermore, y is a proper ancestor of u', and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Due to the generality of $(x, y) \in B(u') \setminus \{e(u')\}$, this implies that $B(u') \setminus \{e(u')\} \subseteq B(u)$. And since $e(u) \notin B(u')$, this can be strengthened to $B(u') \setminus \{e(u')\} \subseteq B(u) \setminus \{e(u)\}$. Thus, we have $B(u) \setminus \{e(u)\} = B(u') \setminus \{e(u')\}$. Since $e(u') \notin B(u)$ and $e(u) \notin B(u')$, this implies that $B(u) \sqcup \{e(u')\} = B(u') \sqcup \{e(u)\}$. ☐

Lemma 5.48. *Let u, v, w be three vertices $\neq r$ such that w is a common ancestor of $\{u, v\}$, and u, v are not related as ancestor and descendant. Suppose that u is a non-special vertex such that $\{(u, p(u)), (v, p(v)), (w, p(w)), e(u)\}$ is a Type-3αi 4-cut. Let c be the child of $M(w)$ that is an ancestor of u, and let u' be the lowest non-special vertex such that $M(B(u') \setminus \{e(u')\}) = M(w, c)$ and u' is a proper descendant of w. Then, $\{(u', p(u')), (v, p(v)), (w, p(w)), e(u')\}$ is a Type-3αi 4-cut.*

Proof. By Lemma 5.42 we have that u is a descendant of a child c of $M(w)$. Then, by Lemma 5.43 we have that $M(B(u) \setminus \{e(u)\}) = M(w, c)$. Thus, since u is a non-special vertex that is a proper descendant of w, it makes sense to consider the lowest non-special vertex u' such that $M(B(u') \setminus \{e(u')\}) = M(w, c)$ and u' is a proper descendant of w. If $u' = u$, then by assumption we have that $\{(u', p(u')), (v, p(v)), (w, p(w)), e(u')\}$ is a Type-3αi 4-cut. So let us assume that $u' < u$. Notice that $M(w, c)$ is a common descendant of u' and u, and therefore u' and u are related as ancestor and descendant. Thus, $u' < u$ implies that u' is a proper ancestor of u.

Since $\{(u, p(u)), (v, p(v)), (w, p(w)), e(u)\}$ is a Type-3αi 4-cut, we have that $B(w) =$

$(B(u)\setminus\{e(u)\})\sqcup B(v)$. Let (x,y) be a back-edge in $B(u)\setminus\{e(u)\}$. Then, x is a descendant of $M(B(u)\setminus\{e(u)\})$, and therefore a descendant of $M(w,c)$, and therefore a descendant of $M(B(u')\setminus\{e(u')\})$, and therefore a descendant of $M(u')$. Furthermore, $B(w) = (B(u)\setminus\{e(u)\})\sqcup B(v)$ implies that $(x,y)\in B(w)$, and therefore y is a proper ancestor of w, and therefore a proper ancestor of u'. This shows that $B(u)\setminus\{e(u)\}\subseteq B(u')$. Since Lemma 5.47 implies that $e(u')\notin B(u)$, this can be strengthened to $B(u)\setminus\{e(u)\}\subseteq B(u')\setminus\{e(u')\}$. Conversely, let (x,y) be a back-edge in $B(u')\setminus\{e(u')\}$. Then x is a descendant of $M(B(u')\setminus\{e(u')\})$, and therefore a descendant of $M(w,c)$, and therefore a descendant of $M(B(u)\setminus\{e(u)\})$, and therefore a descendant of $M(u)$. Furthermore, y is a proper ancestor of u', and therefore a proper ancestor of u. This shows that $B(u')\setminus\{e(u')\}\subseteq B(u)$. Since Lemma 5.47 implies that $e(u)\notin B(u')$, this can be strengthened to $B(u')\setminus\{e(u')\}\subseteq B(u)\setminus\{e(u)\}$. Thus, we have $B(u)\setminus\{e(u)\} = B(u')\setminus\{e(u')\}$. Therefore, $B(w) = (B(u)\setminus\{e(u)\})\sqcup B(v)$ implies that $B(w) = (B(u')\setminus\{e(u')\})\sqcup B(v)$.

Let us suppose, for the sake of contradiction, that $e(u')\in B(v)$. Then, the higher endpoint of $e(u')$ is a common descendant of u' and v, and therefore u' and v are related as ancestor and descendant. Therefore, since u' is an ancestor of u, but u and v are not related as ancestor and descendant, we have that u' is an ancestor of both u and v. Since the graph is 3-edge-connected, we have that $|B(v)| > 1$. Thus, there is a back-edge $(x,y)\in B(v)\setminus\{e(u')\}$. Then, x is a descendant of v, and therefore a descendant of u'. Furthermore, $B(w) = (B(u)\setminus\{e(u)\})\sqcup B(v)$ implies that $(x,y)\in B(w)$, and therefore y is a proper ancestor of w, and therefore a proper ancestor of u'. This shows that $(x,y)\in B(u')$, in contradiction to (the disjointness of the sets in) $B(w) = (B(u')\setminus\{e(u')\})\sqcup B(v)$. Thus, we have $e(u')\notin B(v)$. Therefore, $B(w) = (B(u')\setminus\{e(u')\})\sqcup B(v)$ implies that $B(w)\sqcup\{e(u')\} = B(u)\sqcup B(v)$. Thus, by Lemma 5.41 we have that $\{(u',p(u')),(v,p(v)),(w,p(w)),e(u')\}$ is a Type-$3\alpha i$ 4-cut. $\square$

Lemma 5.49. *Let u,v,w be three vertices $\neq r$ such that w is a common ancestor of $\{u,v\}$, and u,v are not related as ancestor and descendant. Suppose that u is a non-special vertex and $C = \{(u,p(u)),(v,p(v)),(w,p(w)),e(u)\}$ is a Type-$3\alpha i$ 4-cut. Then, every other Type-$3\alpha i$ 4-cut C' of the form $\{(u',p(u')),(v,p(v)),(w,p(w)),e(u')\}$, where u' is a non-special vertex that is a proper descendant of w, is implied by C and some Type-$2ii$ 4-cuts that are computed by Algorithm 24.*

Proof. Since C is a Type-$3\alpha i$ 4-cut where w is a common ancestor of $\{u,v\}$, we have $B(w)\sqcup\{e(u)\} = B(u)\sqcup B(v)$. This implies that $B(u)\setminus\{e(u)\} = B(w)\setminus B(v)$.

Similarly, for the 4-cut C' we have $B(u') \setminus \{e(u')\} = B(w) \setminus B(v)$. Thus, we have $B(u) \setminus \{e(u)\} = B(u') \setminus \{e(u')\}$. Notice that we cannot have $e(u) \in B(u')$, because otherwise we would have $B(u) = B(u')$, in contradiction to the fact that the graph is 3-edge-connected. Similarly, we cannot have $e(u') \in B(u)$. Thus, $B(u) \setminus \{e(u)\} = B(u') \setminus \{e(u')\}$ implies that $B(u) \sqcup \{e(u')\} = B(u') \sqcup \{e(u)\}$. Then, Lemma 5.28 implies that $C'' = \{(u, p(u)), (u', p(u')), e(u), e(u')\}$ is a Type-2ii 4-cut. Notice that C' is implied by C and C'' through the pair of edges $\{(u', p(u')), e(u')\}$. Let $\mathcal{C}$ be the collection of 4-cuts computed by Algorithm 24. Then, by Proposition 5.12 we have that C'' is implied by $\mathcal{C}$ through the pair of edges $\{(u', p(u')), e(u')\}$. Thus, by Lemma 5.7 we have that C' is implied by $\mathcal{C} \cup \{C\}$ through the pair of edges $\{(u', p(u')), e(u')\}$. $\square$

Proposition 5.15. *Algorithm 27 computes a collection $\mathcal{C}$ of Type-3αi 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$, where w is a common ancestor of $\{u, v\}$, u is a non-special vertex, and $e \in B(u)$, and it runs in linear time. Furthermore, let $\mathcal{C}'$ be the collection of Type-2ii 4-cuts computed by Algorithm 24. Then, every Type-3αi 4-cut of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$, where w is a common ancestor of $\{u, v\}$, u is a non-special vertex, and $e \in B(u)$ is implied by $\mathcal{C} \cup \mathcal{C}'$.*

Proof. Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e(u)\}$ be a 4-element set that is marked in Line 25. Then we have that u and v are proper descendants of w such that $bcount(w) = (bcount(u) - 1) + bcount(v)$, $high_2(u) < w$ and $high_1(v) < w$. Furthermore, we have that u and v are not related as ancestor and descendant. Thus, Lemma 5.46 implies that there is a back-edge $e \in B(u)$ such that $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ is a Type-3αi 4-cut. By Lemma 5.43, this implies that $e = e(u)$. Thus, C is indeed a 4-cut. Let $\mathcal{C}$ be the collection of all 4-cuts marked in Line 25.

Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3αi 4-cut such that w is a common ancestor of $\{u, v\}$, u is a non-special vertex, and $e \in B(u)$. Let c_1 and c_2 be the *low1* and *low2* children of $M(w)$, respectively. Lemma 5.42 implies that either u is a descendant of c_1 and v is a descendant of c_2, or u is a descendant of c_2 and v is a descendant of c_1. Let us assume that u is a descendant of c_1. Then, Lemma 5.44 implies that v is the lowest proper descendant of v such that $M(v) = M(w, c_2)$. Lemma 5.43 implies that $M(B(u) \setminus \{e(u)\}) = M(w, c_1)$. Thus, we may consider the lowest proper descendant u' of w that is a non-special vertex such that $M(B(u') \setminus \{e(u')\}) = M(w, c_1)$. Then, Lemma 5.48 implies that $C' = \{(u', p(u')), (v, p(v)), (w, p(w)), e(u')\}$ is a Type-3αi 4-cut. Then, Lemma 5.46 implies that $bcount(w) = (bcount(u') - 1) + bcount(v)$,

Algorithm 27: Compute a collection of Type-3αi 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where w is a common ancestor of $\{u,v\}$, u is a non-special vertex, and $e \in B(u)$, so that all Type-3αi 4-cuts of this form are implied from this collection, plus that of the Type-$2ii$ 4-cuts computed by Algorithm 24

1 **foreach** *vertex* $u \neq r$ **do**
2 compute $M(B(u) \setminus \{e(u)\})$
3 **end**
4 **foreach** *vertex* x **do**
5 initialize a collection $\widetilde{U}(x) \leftarrow \emptyset$
6 **end**
7 **foreach** *vertex* $u \neq r$ **do**
8 let $x \leftarrow M(B(u) \setminus \{e(u)\})$
9 **if** $M(u) \neq x$ **then**
10 insert u into $\widetilde{U}(x)$
11 **end**
12 **end**
// $\widetilde{U}(x)$ contains all non-special vertices u with $M(B(u) \setminus \{e(u)\}) = x$
13 **foreach** *vertex* $w \neq r$ *such that* $M(w)$ *has at least two children* **do**
14 compute $M(w,c_1)$ and $M(w,c_2)$, where c_1 and c_2 are the *low1* and *low2* children of $M(w)$, respectively
15 **end**
// the case where u is a descendant of the *low1* child of $M(w)$; for the other case, simply reverse the roles of c_1 and c_2
16 **foreach** *vertex* $w \neq r$ **do**
17 **if** $M(w)$ *has less than two children* **then continue**
18 let $c_1 \leftarrow$ *low1* child of $M(w)$
19 let $c_2 \leftarrow$ *low2* child of $M(w)$
20 **if** $M(w,c_1) = \bot$ ***or*** $M(w,c_2) = \bot$ **then continue**
21 let u be the lowest proper descendant of w in $\widetilde{U}(M(w,c_1))$
22 let v be the lowest proper descendant of w such that $M(v) = M(w,c_2)$
23 **if** u *and* v *are related as ancestor and descendant* **then continue**
24 **if** $bcount(w) = (bcount(u) - 1) + bcount(v)$ ***and*** $high_2(u) < w$ ***and*** $high_1(v) < w$ **then**
25 mark $\{(u,p(u)),(v,p(v)),(w,p(w)),e(u)\}$ as a 4-cut
26 **end**
27 **end**

$high_2(u') < w$ and $high_1(v) < w$. Thus, notice that C' will be marked in Line 25, and therefore $C' \in \mathcal{C}$. Now, if $C' = C$, then it is trivially true that C is implied by $\mathcal{C}$. Otherwise, by Lemma 5.49 we have that C is implied by $\mathcal{C}' \cup \{C'\}$. Thus, we have that C is implied by $\mathcal{C} \cup \mathcal{C}'$.

Now we will argue about the complexity of Algorithm 27. By Proposition 3.6 we have that the values $M(B(u) \setminus \{e(u)\})$ can be computed in linear time in total, for all vertices $u \neq r$. Thus, the **for** loop in Line 1 can be performed in linear time. By Proposition 3.5, we have that the values $M(w, c_1)$ and $M(w, c_2)$ can be computed in linear time in total, for all vertices $w \neq r$ such that $M(w)$ has at least two children, where c_1 and c_2 are the *low1* and *low2* children of $M(w)$. Thus, the **for** loop in Line 13 can be performed in linear time. The vertices u and v in Lines 21 and 22 can be computed with Algorithm 22. Specifically, whenever we reach Line 21, we generate a query $q(\widetilde{U}(M(w, c_1)), w)$, which returns the lowest vertex u in $\widetilde{U}(M(w, c_1))$ such that $u > w$. $u \in \widetilde{U}(M(w, c_1))$ implies that $M(B(u) \setminus \{e(u)\}) = M(w, c_1)$, and therefore we have that $M(w, c_1)$ is a common descendant of u and w, and therefore u and w are related as ancestor and descendant. Thus, $u > w$ implies that u is a proper descendant of w. Thus, u is the lowest proper descendant of w that lies in $\widetilde{U}(M(w, c_1))$. Since the sets $\widetilde{U}$ are disjoint, and the total number of those queries is $O(n)$, Lemma 5.27 implies that Algorithm 22 can answer all those queries in $O(n)$ time in total. Similarly, the vertices v in Line 22 can be computed in $O(n)$ time in total. We conclude that Algorithm 27 runs in linear time. □

5.7.2 Type-3αii 4-cuts

We will distinguish the Type-3αii 4-cuts according to the following.

Lemma 5.50. *Let u, v, w be three vertices $\neq r$ such that w is a common ancestor of $\{u, v\}$, and u, v are not related as ancestor and descendant. Suppose that there is a back-edge $e = (x, y)$ such that $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$. Then we have the following cases (see also Figure 5.21).*

1. *x **is an ancestor of both** u **and** v. In this case, $x = M(w)$. Furthermore, u is a descendant of the low1 child of $\widetilde{M}(w)$ and v is a descendant of the low2 child of $\widetilde{M}(w)$, or reversely.*

2. *x **is an ancestor of** u**, but not an ancestor of** v. In this case, x is a descendant of the low1 child of $M(w)$ and v is a descendant of the low2 child of $M(w)$, or reversely.*

3. x ***is an ancestor of*** v***, but not an ancestor of*** u*. In this case,* x *is a descendant of the low1 child of* $M(w)$ *and* u *is a descendant of the low2 child of* $M(w)$*, or reversely.*

4. x ***is neither an ancestor of*** u ***nor an ancestor of*** v*. In this case, we have the following two subcases.*

   4.1 Two of $\{u, v, x\}$ *are descendants of the low1 child of* $M(w)$ *and the other is a descendant of the low2 child of* $M(w)$*, or reversely: two of* $\{u, v, x\}$ *are descendants of the low2 child of* $M(w)$ *and the other is a descendant of the low1 child of* $M(w)$*.*

   4.2 There is a permutation σ *of* $\{1, 2, 3\}$ *such that* u *is a descendant of the low*σ_1 *child of* $M(w)$*,* v *is a descendant of the low*σ_2 *child of* $M(w)$*, and* x *is a descendant of the low*σ_3 *child of* $M(w)$*.*

In cases $4.1, 4.2$ *we have* $l_2(x) \geq w$ *and* $low(c_1(x)) \geq w$ *(if* $c_1(x) \neq \perp$*).*

In all cases $1-4$*, we have* $y = l_1(x)$*.*

Proof. Before we consider the four cases in turn, we will show that $M(w)$ is the nearest common ancestor of $\{u, v, x\}$. The fact that $M(w)$ is an ancestor of x is an obvious consequence of $(x, y) \in B(w)$. Now, since the graph is 3-edge-connected, neither $B(u)$ nor $B(v)$ is empty. Thus there are back-edges $(x', y') \in B(u)$ and $(x'', y'') \in B(v)$. Then $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that $(x', y') \in B(w)$ and $(x'', y'') \in B(w)$, and therefore $M(w)$ is a common ancestor of $\{x', x''\}$. Since x' is a descendant of u and x'' is a descendant of v, we have that $M(w)$ is related as ancestor and descendant with both u and v. But since u, v are not related as ancestor and descendant, $M(w)$ must be an ancestor of both u and v. Thus far we have that $M(w)$ is a common ancestor of $\{u, v, x\}$. Now let us suppose, for the sake of contradiction, that $M(w)$ is not the nearest common ancestor of $\{u, v, x\}$. This means that there is a proper descendant c of $M(w)$ that is a common ancestor of $\{u, v, x\}$. Now let (x', y') be a back-edge in $B(w)$. Then $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $x' = x$, or x' is a descendant of u, or x' is a descendant of v. In any case, x' is a descendant of c. But due to the generality of $(x', y') \in B(w)$, this shows that $M(w)$ is a descendant of c, a contradiction. Thus we have that $M(w)$ is the nearest common ancestor of $\{u, v, x\}$.

Furthermore, we will show that u and v are proper descendants of $M(w)$. Otherwise, let us assume w.l.o.g. that u is not a proper descendant of $M(w)$. Since $M(w)$

is an ancestor of u, this means that $u = M(w)$. Now, since x is a descendant of $M(w)$, it is also a descendant of u. And since y is a proper ancestor of w, it is also a proper ancestor of u. But then we have $(x, y) \in B(u)$, contradicting (the disjointness of the union in) $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$.

(1) Suppose that x is an ancestor of both u and v. Then, since $M(w)$ is the nearest common ancestor of $\{u, v, x\}$, we have that $x = M(w)$. Furthermore, since u and v are proper descendants of $M(w)$, we have $u \neq x$ and $v \neq x$. Let $S = \{(x', y') \in B(w) \mid x' \neq M(w)\}$. Then we have $\widetilde{M}(w) = M(S)$. We will show that $\widetilde{M}(w)$ is the nearest common ancestor of $\{u, v\}$. Let (x', y') be a back-edge in S. Then $(x', y') \in B(w) \setminus \{e\}$, and so $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that x' is either a descendant of u or a descendant of v. This implies that x' is a descendant of $nca(u, v)$. Due to the generality of $(x', y') \in S$, we have that $M(S)$ is a descendant of $nca(u, v)$. Conversely, let (x', y') be a back-edge in $B(u)$ and let (x'', y'') be a back-edge in $B(v)$ (such back-edges exist, because the graph is 3-edge-connected). Then $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that x' and x'' are in S, and so $M(S)$ is a common ancestor of x' and x''. Then, since x' is a descendant of u and x'' is a descendant of v, we have that $M(S)$ is related to both u and v as ancestor and descendant. But since u and v are not related as ancestor and descendant, we have that $M(S)$ is an ancestor of both u and v. Thus, since $M(S)$ is a descendant of $nca(u, v)$, we have that $M(S)$ is the nearest common ancestor of $\{u, v\}$.

Since u and v are not related as ancestor and descendant, we have that there are children c_1 and c_2 of $M(S)$ such that u is a descendant of c_1 and v is a descendant of c_2. Then, since $B(u)$ and $B(v)$ are non-empty and $B(u) \cup B(v) \subset B(w)$, we have that $low(c_1) < w$ and $low(c_2) < w$. Now let us suppose, for the sake of contradiction, that there is also another child c of $M(w)$ that has $low(c) < w$ (i.e., $c \notin \{c_1, c_2\}$). Then neither u nor v is a descendant of c. Then $low(c) < w$ implies that there is a back-edge (x', y') such that x' is a descendant of c and y is a proper ancestor of w. Since c is a descendant of $M(S)$, which is a descendant of $M(w)$, we thus have that $(x', y') \in B(w)$. Then $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $(x', y') \in B(u) \cup B(v)$, or $(x', y') = e$. The case $(x', y') = e$ is rejected, because x' is a descendant of c, but c is not an ancestor of either u or v (whereas x is an ancestor of both u and v). Thus, we have $(x', y') \in B(u) \cup B(v)$, which implies that either x' is a descendant of u, or x' is a descendant of v. But this contradicts the fact that x' is a descendant of c (which is not related as ancestor and descendant with either u or v). Thus we have that c_1 and c_2 are the only children of $M(S)$ that have $low(c_i) < w$, for $i \in \{1, 2\}$, and so these

must coincide with the *low1* and the *low2* children of $M(S)$ (not necessarily in that order).

(2) Suppose that x is an ancestor of u, but not an ancestor of v. Then, since $M(w)$ is a common ancestor of $\{u, v, x\}$, we have that x is a proper descendant of $M(w)$ (otherwise x would be an ancestor of v). Since v is also a proper descendant of $M(w)$, we have that both x and v are descendants of children of $M(w)$. Furthermore, since u is a descendant of x, we have that u is a descendant of the same child of $M(w)$ as x.

Let us suppose, for the sake of contradiction, that x and v are descendants of the same child c of $M(w)$. Let (x', y') be a back-edge in $B(w)$. Then $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $x' = x$, or x' is a descendant of u, or x' is a descendant of v. In either case, we have that x' is a descendant of c. Due to the generality of (x', y'), this means that $M(w)$ is a descendant of c, a contradiction. Thus, x and v are descendants of different children of $M(w)$. Let c_1 be the child of $M(w)$ that is an ancestor of x, and let c_2 be the child of $M(w)$ that is an ancestor of v. Then the existence of the back-edge (x, y) implies that $low(c_1) < w$. And the fact that the graph is 3-edge-connected implies that $B(v) \neq \emptyset$, which further implies that $low(c_2) < w$, due to $B(v) \subset B(w)$.

Now let us suppose, for the sake of contradiction, that there is another child c of $M(w)$ (i.e., with $c \notin \{c_1, c_2\}$) that has $low(c) < w$. Then neither v nor x (and therefore neither u) is a descendant of c. Then $low(c) < w$ implies that there is a back-edge (x', y') such that x' is a descendant of c and y is a proper ancestor of w. Since c is a descendant of $M(w)$, we thus have that $(x', y') \in B(w)$. Then $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $x' = x$, or x' is a descendant of u, or x' is a descendant of v. But this contradicts the fact that x' is a descendant of c (which is not related as ancestor and descendant with either x, or u, or v). Thus we have that c_1 and c_2 are the only children of $M(w)$ that have $low(c_i) < w$, for $i \in \{1, 2\}$, and so these must coincide with the *low1* and the *low2* children of $M(w)$ (not necessarily in that order).

(3) The argument for this case is analogous to that for case (2).

(4) Suppose that x is neither an ancestor of u nor an ancestor of v. Then, since $M(w)$ is a common ancestor of $\{u, v, x\}$, we have that x is a proper descendant of $M(w)$ (otherwise x would be an ancestor of both u and v). Let us assume first that u and v are descendants of the same child c_1 of $M(w)$. Then we cannot have that x is also a descendant of c_1, because $M(w)$ is the nearest common ancestor of $\{u, v, x\}$

(and therefore we would have that $M(w)$ is a descendant of c_1). So let c_2 be the child of $M(w)$ that is an ancestor of x. Now we can argue as in (2), in order to demonstrate that c_1 and c_2 are the only children of $M(w)$ with $low(c_i) < w$, for $i \in \{1,2\}$, and so these must coincide with the *low1* and the *low2* children of $M(w)$ (not necessarily in that order).

Now let us assume that u and v are not descendants of the same child of $M(w)$. Let c_1 be the child of $M(w)$ that is an ancestor of u, and let c_2 be the child of $M(w)$ that is an ancestor of v. If we assume that x is a descendant of either c_1 or c_2, then we can argue as in (2), in order to demonstrate that c_1 and c_2 are the only children of $M(w)$ with $low(c_i) < w$, for $i \in \{1,2\}$, and so these must coincide with the *low1* and the *low2* children of $M(w)$ (not necessarily in that order). So let us assume that x is neither a descendant of c_1, nor a descendant of c_2, and let c_3 be the child of $M(w)$ that is an ancestor of x. Then we can argue as in (2) in order to demonstrate that c_1, c_2 and c_3 are the only children of $M(w)$ with $low(c_i) < w$, for $i \in \{1,2,3\}$, and so these must coincide with the *low1*, the *low2*, and the *low3* children of $M(w)$ (not necessarily in that order).

Since $(x,y) \in B(w)$, we have that e is a back-edge in $B(x)$ whose lower endpoint is lower than w. Now let us suppose, for the sake of contradiction, that there is one more back-edge $e' = (x',y') \in B(x)$ such that $y' < w$. Then $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $e' \in B(u)$, or $e' \in B(v)$, or $e' = e$. The last case is rejected by assumption. If $e' \in B(u)$, then x' is a descendant of u. Therefore, x and u are related as ancestor and descendant, since they have x' as a common descendant. Then, since x is not an ancestor of u, we have that x is a descendant of u. But since y is a proper ancestor of w, it is also a proper ancestor of u, and therefore $(x,y) \in B(u)$, which is impossible. Thus, the case $e' \in B(u)$ is rejected. Similarly, the case $e' \in B(v)$ is also rejected. But then there are no viable options left, and so we are led to a contradiction. This shows that $e = (x,y)$ is the unique back-edge in $B(x) \cap B(w)$. Thus, we have $e = (x, l_1(x))$, and we have $l_2(x) \geq w$ and $low(c_1(x)) \geq w$ (if $c_1(x) \neq \bot$).

Finally, let us suppose, for the sake of contradiction, that there is a back-edge of the form $(x,y') \in B(w)$ such that $(x,y') \neq (x,y)$. Then, $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $(x,y') \in B(u)$ or $(x,y') \in B(v)$. This implies that x is a descendant of u or v, respectively. Thus, since y is a proper ancestor of w, we have that $(x,y) \in B(u)$ or $(x,y) \in B(v)$, respectively, in contradiction to (the disjointness in) $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$. This shows that (x,y) is the only back-edge with higher endpoint x such

that y is a proper ancestor of w. Let us suppose, for the sake of contradiction, that $y \neq l_1(x)$. Then there is a back-edge $(x, y') \neq (x, y)$ such that $y' \leq y$. Since x is a common descendant of y and y', we have that y and y' are related as ancestor and descendant. Thus, $y' \leq y$ implies that y' is an ancestor of y, and therefore y' is a proper ancestor of w, a contradiction. Thus, we have $y = l_1(x)$. □

Lemma 5.51. *Let u, v, w be three vertices $\neq r$ such that w is a common ancestor of $\{u, v\}$ and u, v are not related as ancestor and descendant. If there is a back-edge e such that $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$, then u is the lowest proper descendant of w in $M^{-1}(M(u))$. Similarly, v is the lowest proper descendant of w in $M^{-1}(M(v))$.*

Proof. We will provide the argument for u, since that for v is similar. Let us suppose, for the sake of contradiction, that there is a proper descendant u' of w with $M(u') = M(u)$ such that u' is lower than u. Then we have that u' is a proper ancestor of u, and Lemma 3.2 implies that $B(u') \subseteq B(u)$. This can be strengthened to $B(u') \subset B(u)$, since the graph is 3-edge-connected. Thus, there is a back-edge $(x, y) \in B(u) \setminus B(u')$. Then x is a descendant of $M(u)$, and therefore a descendant of $M(u')$. Thus, it cannot be that y is a proper ancestor of u', for otherwise we would have $(x, y) \in B(u')$. This implies that y cannot be a proper ancestor of w, for otherwise it would be a proper ancestor of u'. Thus we have that $(x, y) \notin B(w)$. But this contradicts $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$, which implies that $B(u) \subset B(w)$. Thus we have that u is the lowest proper descendant of w in $M^{-1}(M(u))$. □

Lemma 5.52. *Let u, v, w be three vertices $\neq r$ such that w is a common ancestor of $\{u, v\}$ and u, v are not related as ancestor and descendant. Then there is a back-edge e such that $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ if and only if $bcount(w) = bcount(u) + bcount(v) + 1$ and $high_1(u) < w$ and $high_1(v) < w$.*

Proof. ($\Rightarrow$) $bcount(w) = bcount(u) + bcount(v) + 1$ is an immediate consequence of $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$. Now let (x, y) be a back-edge in $B(u)$. Then $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that (x, y) is in $B(w)$, and therefore y is a proper ancestor of w, and therefore $y < w$. Due to the generality of $(x, y) \in B(u)$, this shows that $high_1(u) < w$. Similarly, we get $high_1(v) < w$.

($\Leftarrow$) Let (x, y) be a back-edge in $B(u)$. Then, x is a descendant of u, and therefore a descendant of w. Furthermore, y is a proper ancestor of u. Thus, since y and w have u as a common descendant, we have that y and w are related as ancestor

and descendant. Since $(x, y) \in B(u)$, we have that y is an ancestor of $high_1(u)$, and therefore $y \leq high_1(u)$. Thus, $high_1(u) < w$ implies that y is a proper ancestor of w. This shows that $(x, y) \in B(w)$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(w)$. Similarly, we have $B(v) \subseteq B(w)$. Since u and v are not related as ancestor and descendant, we have $B(u) \cap B(v) = \emptyset$. Thus, $B(u) \sqcup B(v) \subseteq B(w)$. Now $bcount(w) = bcount(u) + bcount(v) + 1$ implies that $|B(w) \setminus (B(u) \sqcup B(v))| = 1$, and so there is a back-edge e such that $B(w) \setminus (B(u) \sqcup B(v)) = \{e\}$. This means that $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$. □

First, we consider case (1) of Lemma 5.50.

Lemma 5.53. *Let case* (1) *of Lemma 5.50 be true. Then* $l_1(M(w)) < w$, $l_2(M(w)) \geq w$, *and* $e = (M(w), l_1(M(w)))$. *Let* c_1 *and* c_2 *be the low1 and low2 children of* $\widetilde{M}(w)$, *respectively. Assume w.l.o.g. that* u *is a descendant of* c_1 *and* v *is a descendant of* c_2. *Then* $M(u) = M(w, c_1)$ *and* $M(v) = M(w, c_2)$.

Proof. Since $x = M(w)$ and (x, y) is a back-edge in $B(w)$, we have that $l_1(M(w)) < w$. Let us suppose, for the sake of contradiction, that $l_2(M(w)) < w$. Then there is a back-edge $(x, y') \neq (x, y)$ such that $(x, y') \in B(w)$. Since u is a descendant of c_1 and v is a descendant of c_2, we have that x is not a descendant of either u or v. Thus, (x, y) and (x, y') are two distinct back-edges that leap over w and none of them is in $B(u)$ or $B(v)$. This contradicts the fact that $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$, which implies that exactly one back-edge in $B(w)$ is not in $B(u) \cup B(v)$. Thus, we have that $l_2(M(w)) \geq w$. Since $e = (M(w), y)$ satisfies $y < w$, we have that $y = l_1(M(w))$.

Now we will provide the arguments for u, since those for v are similar. Let $S = \{(x', y') \in B(w) \mid x' \text{ is a descendant of } c_1\}$. Then $M(S) = M(w, c_1)$. Let (x', y') be a back-edge in $B(u)$. Then $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that $(x', y') \in B(w)$. Since x' is a descendant of u and u is a descendant of c_1, we have that x' is a descendant of c_1. Thus we have $(x', y') \in S$. Due to the generality of $(x', y') \in B(u)$, this shows that $M(u)$ is a descendant of $M(S)$. Conversely, let (x', y') be a back-edge in S. Then we have that x' is a descendant of c_1 and y' is a proper ancestor of w. Furthermore, $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $x' = x$, or x' is a descendant of u, or x' is a descendant of v. The case $x' = x$ is rejected, because $x = M(w)$. Also, x' cannot be a descendant of v, for otherwise x' would be a descendant of c_2. Thus x' is a descendant of u. Then, since y' is a proper ancestor of w, we infer that $(x', y') \in B(u)$.

Due to the generality of $(x', y') \in S$, this shows that $M(S)$ is a descendant of $M(u)$. This concludes the proof that $M(u) = M(w, c_1)$. $\square$

According to Lemma 5.53, we can compute all 4-cuts in case (1) of Lemma 5.50 as follows. First, we only have to consider those $w \neq r$ such that $l_1(M(w)) < w$ and $\widetilde{M}(w)$ has at least two children. In this case, let c_1 and c_2 be the *low1* and *low2* children of $\widetilde{M}(w)$. Then we compute $M(w, c_1)$ and $M(w, c_2)$. If none of $M(w, c_1)$ and $M(w, c_2)$ is $\bot$, then, according to Lemma 5.51, we have that u is the lowest proper descendant of w that has $M(u) = M(w, c_1)$, and v is the lowest proper descendant of w that has $M(v) = M(w, c_2)$. Then, according to Lemma 5.52, we have that $\{(u, p(u)), (v, p(v)), (w, p(w)), (M(w), l_1(M(w)))\}$ is a 4-cut if and only if $high(u) < w$, $high(v) < w$, and $bcount(w) = bcount(u) + bcount(v) + 1$. The procedure for computing those 4-cuts is given in Algorithm 28. The proof of correctness and linear complexity is given in Proposition 5.16.

Proposition 5.16. *Algorithm 28 correctly computes all Type-3αii 4-cuts of the form* $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$*, where* w *is a common ancestor of* $\{u, v\}$*, and* e *satisfies* (1) *of Lemma 5.50. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3αii 4-cut, where w is a common ancestor of $\{u, v\}$, and e satisfies (1) of Lemma 5.50. Then, Lemma 5.53 implies that $e = (M(w), l_1(M(w)))$. Furthermore, let c_1 and c_2 be the *low1* and *low2* children of $\widetilde{M}(w)$, respectively. Then w.l.o.g. we have that $M(u) = M(w, c_1)$ and $M(v) = M(w, c_2)$. Then, Lemma 5.51 implies that u is the lowest proper descendant of w with $M(u) = M(w, c_1)$, and v is the lowest proper descendant of w with $M(v) = M(w, c_2)$. Lemma 5.52 implies that $bcount(w) = bcount(u) + bcount(v) + 1$, $high_1(u) < w$ and $high_1(v) < w$. Thus, all conditions are satisfied for C to be marked in Line 22.

Conversely, suppose that a 4-element set $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e(w)\}$ is marked in Line 22. Then we have that u and v are descendants of w such that $bcount(w) = bcount(u) + bcount(v) + 1$, $high_1(u) < w$ and $high_1(v) < w$. Since $M(u) = M(w, c_1)$ and $M(v) = M(w, c_2)$, where c_1 and c_2 are different children of $M(w)$, by Lemma 3.12 we have that u and v are not related as ancestor and descendant. Therefore, Lemma 5.52 implies that there is a back-edge e such that $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$. Since $l_1(M(w)) < w$, we have that $(M(w), l_1(M(w)))$ is a back-edge in $B(w)$. Since u and v are proper descendants of $M(w)$, we have that this back-edge

Algorithm 28: Compute all Type-3αii 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where w is a common ancestor of $\{u,v\}$, and e satisfies (1) of Lemma 5.50.

```
1  foreach vertex w ≠ r do
2      if l_1(M(w)) < w then
3          let e(w) = (M(w), l_1(M(w)))
4          compute M̃(w)
5      end
6  end
7  foreach vertex w ≠ r do
8      if M̃(w) has at least two children then
9          let c_1 and c_2 be the low1 and low2 children of M̃(w)
10         compute M(w, c_1) and M(w, c_2)
11     end
12 end
13 foreach vertex w ≠ r do
14     if l_1(M(w)) ≥ w then continue
15     if M̃(w) has less than two children then continue
16     let c_1 ← low1 child of M̃(w)
17     let c_2 ← low2 child of M̃(w)
18     if low(c_1) ≥ w or low(c_2) ≥ w then continue
19     let u be the lowest proper descendant of w such that M(u) = M(w, c_1)
20     let v be the lowest proper descendant of w such that M(v) = M(w, c_2)
       // u and v are not related as ancestor and descendant
21     if bcount(w) = bcount(u) + bcount(v) + 1 and high_1(u) < w and high_1(v) < w
        then
22         mark {(u,p(u)),(v,p(v)),(w,p(w)),e(w)} as a 4-cut
23     end
24 end
```

does not belong to $B(u)\cup B(v)$. Thus, $e=(M(w),l_1(M(w)))$, and therefore C is indeed a Type-3αii 4-cut.

Now we will argue about the complexity of Algorithm 28. By Proposition 3.5, we

have that the values $\widetilde{M}(w)$ can be computed in linear time in total, for all vertices $w \neq r$ (see the first paragraph in Section 3.6). Thus, the **for** loop in Line 1 can be performed in linear time. By Proposition 3.5 we have that the values $M(w,c_1)$ and $M(w,c_2)$ can be computed in linear time in total, for all vertices $w \neq r$ such that $\widetilde{M}(w)$ has at least two children, where c_1 and c_2 are the *low1* and *low2* children of $\widetilde{M}(w)$, respectively. Thus, the **for** loop in Line 7 can be performed in linear time. In order to compute the vertices u and v in Lines 19 and 20, respectively, we use Algorithm 22. Specifically, whenever we reach Line 19, we generate a query $q(M^{-1}(M(w,c_1)),w)$. This will return the lowest vertex u with $M(u) = M(w,c_1)$ such that $u > w$. Since $M(u) = M(w,c_1)$ is a common descendant of u and w, we have that u and w are related as ancestor and descendant. Thus, $u > w$ implies that u is a proper descendant of w. Thus, u is the lowest vertex with $M(u) = M(w,c_1)$ such that u is a proper descendant of w. Since the number of all those queries is $O(n)$, Algorithm 22 can compute them in linear time in total, according to Lemma 5.27. The same is true for the queries for v in Line 20. We conclude that Algorithm 28 has a linear-time implementation. □

Now we consider case (2) of Lemma 5.50. Notice that due to the symmetry between cases (2) and (3) of Lemma 5.50, case (3) essentially coincides with case (2) (after switching the labels of u and v), and thus we do not have to provide a different algorithm for case (3).

Lemma 5.54. *Let case* (2) *of Lemma 5.50 be true. Let* c_1 *and* c_2 *be the low1 and low2 children of* $M(w)$*, respectively. Assume w.l.o.g. that* x *is a descendant of* c_1 *and* v *is a descendant of* c_2*. Then* $e = (M(w,c_1), l_1(M(w,c_1)))$*. Let* $S = \{(x',y') \in B(w) \mid x' \text{ is a descendant of the low1 child of } x\}$*. Then* $M(u) = M(S)$ *and* $M(v) = M(w,c_2)$*.*

Proof. First we will show that $e = (M(w,c_1), l_1(M(w,c_1)))$. Let (x',y') be a back-edge in $B(w)$ such that x' is a descendant of c_1. Then $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $(x',y') = e$, or $(x',y') \in B(u)$, or $(x',y') \in B(v)$. Only the last case is rejected, since v is a descendant of c_2, and thus it cannot be an ancestor of x'. Thus we have that the nearest common ancestor of x and $M(u)$ is an ancestor of x'. Due to the generality of $(x',y') \in B(w)$ with x' a descendant of c_1, this shows that the nearest common ancestor of x and $M(u)$ is an ancestor of $M(w,c_1)$. Conversely, if (x',y') is a back-edge such that either $(x',y') = e$ or $(x',y') \in B(u)$, then x' is a descendant of c_1, and $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that $(x',y') \in B(w)$. Thus, x' is a descendant of $M(w,c_1)$, and so the nearest common ancestor of x and $M(u)$ is a descendant of

$M(w, c_1)$. This shows that $M(w, c_1) = nca\{x, M(u)\}$. Since x is an ancestor of u, it is also an ancestor of $M(u)$, and so $nca\{x, M(u)\} = x$. This shows that $x = M(w, c_1)$. Since $(x, y) \in B(w)$, we have that $l_1(M(w, c_1)) < w$. Now let us suppose, for the sake of contradiction, that $l_2(M(w, c_1)) < w$. Then there is a back-edge $(x, y') \neq (x, y)$ such that $y' < w$, and thus we have $(x, y') \in B(w)$. Notice that since $(x, y) \notin B(u)$, it cannot be the case that x is a descendant of u. Furthermore, x cannot be a descendant of v, because v is a descendant of c_2 whereas x is a descendant of c_1. Thus, none of (x, y) and (x, y') is in $B(u)$ or $B(v)$. But this contradicts $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$, which implies that there is only one back-edge in $B(w)$ that is not in $B(u)$ or $B(v)$. This shows that $l_2(M(w, c_1)) \geq w$. Thus, since $y < w$, we have that $y = l_1(M(w, c_1))$.

Now we will provide the arguments for u, since those for v are basically given in the proof of Lemma 5.53. Since $(x, y) \in B(w)$, it cannot be the case that $x = u$, for otherwise we would have that $(x, y) \in B(u)$, contradicting (the disjointness of the union in) $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$. Thus, u is a proper descendant of x. Let c be the child of x that is an ancestor of u. Then, since $B(u)$ is non-empty and $B(u) \subset B(w)$, we have that $low(c) < w$. Now let us suppose, for the sake of contradiction, that there is also another child c' of x that has $low(c') < w$ (i.e., $c' \neq c$). This means that there is a back-edge (x', y') such that x' is a descendant of c' and y' is a proper ancestor of w. Then $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $x' = x$, or x' is a descendant of u, or x' is a descendant of v. $x' = x$ is rejected, since x' is a descendant of c'. Furthermore, x' cannot be a descendant of v, because this would imply that x and v are related as ancestor and descendant, contradicting the fact that x and v are descendants of different children of $M(w)$. Thus we have that x' is a descendant of u, and therefore a descendant of c, a contradiction. Thus c is the unique child of x that satisfies $low(c) < w$, and so it must be the *low1* child of x.

Now let (x', y') be a back-edge in $B(u)$. Then we have that x' is a descendant of u, and therefore a descendant of the *low1* child of x. Furthermore, $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that $(x', y') \in B(w)$. This shows that $(x', y') \in S$. Due to the generality of $(x', y') \in B(u)$, this implies that $B(u) \subseteq S$. Conversely, let (x', y') be a back-edge in S. Then (x', y') is in $B(w)$, and so $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $x' = x$, or x' is a descendant of u, or x' is a descendant of v. Since x' is a descendant of the *low1* child of x, the only viable option is that x' is a descendant of u. Since y' is a proper ancestor of w, it is also a proper ancestor of u. This shows that $(x', y') \in B(u)$. Due to the generality of $(x', y') \in S$, this implies that $S \subseteq B(v)$. Thus

we have shown that $B(u) = S$, and so $M(u) = M(S)$ is derived. $\square$

According to Lemma 5.54, we can compute all 4-cuts in case (2) of Lemma 5.50 as follows. First, we only have to consider those $w \neq r$ such that $M(w)$ has at least two children. In this case, let c_1 and c_2 be the *low1* and *low2* children of $M(w)$, respectively. Then we compute $M(w, c_1)$ and $M(w, c_2)$. If none of $M(w, c_1)$ and $M(w, c_2)$ is $\perp$, then we keep considering w only if $l_1(M(w, c_1)) < w$ and $M(w, c_1)$ has at least one child. In this case, let c_1' be the *low1* child of $M(w, c_1)$. Then, we have that $e = (M(w, c_1), l_1(M(w, c_1)))$, and, according to Lemma 5.51, we have that u is the lowest proper descendant of w that has $M(u) = M(w, c_1')$, and v is the lowest proper descendant of w that has $M(v) = M(w, c_2)$. Then, according to Lemma 5.52, we have that $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ is a 4-cut if and only if $high(u) < w$, $high(v) < w$, and $bcount(w) = bcount(u) + bcount(v) + 1$. The procedure for computing those 4-cuts is given in Algorithm 29. The proof of correctness and linear complexity is given in Proposition 5.17.

Proposition 5.17. *Algorithm 29 correctly computes all Type-3αii 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$, where w is a common ancestor of $\{u, v\}$, and e satisfies (2) of Lemma 5.50. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3αii 4-cut where w is a common ancestor of u and v, and e satisfies (2) of Lemma 5.50. Let us also assume that the higher endpoint x of e is a descendant of the *low1* child of $M(w)$ (the other case, where x is a descendant of the *low2* child of $M(w)$, is treated similarly). Let c_1 and c_2 be the *low1* and *low2* children of $M(w)$, respectively. Furthermore, let c_1' be the *low1* child of $M(w, c_1)$. Then Lemma 5.54 implies that $e = (M(w, c_1), l_1(M(w, c_1)))$, $M(u) = M(w, c_1')$ and $M(v) = M(w, c_2)$. Then, Lemma 5.51 implies that u is the lowest proper descendant of w with $M(u) = M(w, c_1')$, and v is the lowest proper descendant of w with $M(v) = M(w, c_2)$. Finally, Lemma 5.52 implies that $bcount(w) = bcount(u) + bcount(v) + 1$, $high_1(u) < w$ and $high_1(v) < w$. Thus, we have that C will be marked in Line 22.

Conversely, let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a 4-element set that is marked in Line 22. Then we have that u and v are descendants of w such that $bcount(w) = bcount(u) + bcount(v) + 1$, $high_1(u) < w$ and $high_1(v) < w$. We will show that the comment in Line 20 is true: i.e., u and v are not related as ancestor and descendant. This is a consequence of the fact that $M(u) = M(w, c_1')$, $M(v) = M(w, c_2)$,

Algorithm 29: Compute all Type-3αii 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where w is a common ancestor of $\{u,v\}$, and e satisfies (2) of Lemma 5.50.

// We assume that the higher endpoint of e is a descendant of the *low1* child of $M(w)$; the other case is treated similarly, by reversing the roles of c_1 and c_2

1 **foreach** *vertex* $w \neq r$ **do**
2   **if** $M(w)$ *has less than two children* **then continue**
3   let $c_1 \leftarrow$ *low1* child of $M(w)$
4   let $c_2 \leftarrow$ *low2* child of $M(w)$
5   compute $M(w,c_1)$ and $M(w,c_2)$
6   **if** $M(w,c_1)=\perp$ ***or*** $l_1(M(w,c_1)) \geq w$ ***or*** $M(w,c_1)$ *has no children* **then continue**
7   let $e(w) \leftarrow (M(w,c_1), l_1(M(w,c_1)))$
8   let $c_1' \leftarrow$ *low1* child of $M(w,c_1)$
9   compute $M(w,c_1')$
10 **end**
11 **foreach** *vertex* $w \neq r$ **do**
12   **if** $M(w)$ *has less than two children* **then continue**
13   let $c_1 \leftarrow$ *low1* child of $M(w)$
14   let $c_2 \leftarrow$ *low2* child of $M(w)$
15   **if** *low*$(c_1) \geq w$ ***or*** *low*$(c_2) \geq w$ **then continue**
16   **if** $l_1(M(w,c_1)) \geq w$ ***or*** $M(w,c_1)$ *has no children* **then continue**
17   let $c_1' \leftarrow$ *low1* child of $M(w,c_1)$
18   **if** *low*$(c_1') \geq w$ **then continue**
19   let u be the lowest proper descendant of w such that $M(u)=M(w,c_1')$
20   let v be the lowest proper descendant of w such that $M(v)=M(w,c_2)$
  // u and v are not related as ancestor and descendant
21   **if** *bcount*$(w)=$ *bcount*$(u)+$ *bcount*$(v)+1$ ***and*** *high*$_1(u)<w$ ***and*** *high*$_1(v)<w$ **then**
22   mark $\{(u,p(u)),(v,p(v)),(w,p(w)),e(w)\}$ as a 4-cut
23   **end**
24 **end**

and c'_1 is a descendant of the *low1* child c_1 of $M(w)$, whereas c_2 is the *low2* child of $M(w)$. Thus, Lemma 3.12 implies that u and v are not related as ancestor and descendant. Therefore, Lemma 5.52 implies that there is a back-edge e such that $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$. Since we have that $l_1(M(w,c_1)) < w$, we have that the back-edge $e(w) = (M(w,c_1), l_1(M(w,c_1)))$ is in $B(w)$. Since $M(u)$ is a proper descendant of $M(w,c_1)$, we have that $e(w) \notin B(u)$. And since $M(v) = M(w,c_2)$, we have that $e(w) \notin B(v)$. Thus, $e = e(w)$, and therefore C is indeed a Type-3αii 4-cut.

Now we will argue about the complexity of Algorithm 29. For every $w \neq r$ such that $M(w)$ has at least two children, we have to compute $M(w,c_1)$ and $M(w,c_2)$, where c_1 and c_2 are the *low1* and *low1* children of $M(w)$. By Proposition 3.5 this can be done in linear time in total, for all such vertices w. Then, for every such w, if $M(w,c_1) \neq \bot$ and $M(w,c_1)$ has at least one child, we have to compute $M(w,c'_1)$, where c'_1 is the *low1* child of $M(w,c_1)$. Again, by Proposition 3.5, all these calculations take linear time in total. Thus, the **for** loop in Line 1 can be performed in linear time. In order to compute the vertices u and v in Lines 19 and 20, we can use Algorithm 22, as explained e.g. in the proof of Proposition 5.16. According to Lemma 5.27, all these computations take $O(n)$ time in total. We conclude that Algorithm 29 has a linear-time implementation. □

Now we consider case (4.1) of Lemma 5.50.

Lemma 5.55. *Let case* (4.1) *of Lemma 5.50 be true. Let* c_1 *be the low1 child of* $M(w)$*, and let* c_2 *be the low2 child of* $M(w)$*. Let us assume that two of* $\{u,v,x\}$ *are descendants of* c_1*. Let* c'_1 *and* c'_2 *be the low1 and the low2 child of* $M(w,c_1)$*, respectively. Then we have the following three cases.*

(1) u ***and*** v ***are descendants of*** c_1***, and*** x ***is a descendant of*** c_2*. Then we have* $M(u) = M(w,c'_1)$ *and* $M(v) = M(w,c'_2)$ *(or reversely). Furthermore, we have* $(x,y) = (M(w,c_2), l_1(M(w,c_2)))$*.*

(2) u ***and*** x ***are descendants of*** c_1***, and*** v ***is a descendant of*** c_2*. Then we have* $M(u) = M(w,c'_1)$ *and* $(x,y) = (M(w,c'_2), l_1(M(w,c'_2)))$ *(or* $M(u) = M(w,c'_2)$ *and* $(x,y) = (M(w,c'_1), l_1(M(w,c'_1)))$*). Furthermore, we have* $M(v) = M(w,c_2)$*.*

(3) v ***and*** x ***are descendants of*** c_1***, and*** u ***is a descendant of*** c_2*. Then we have* $M(v) = M(w,c'_1)$ *and* $(x,y) = (M(w,c'_2), l_1(M(w,c'_2)))$ *(or* $M(v) = M(w,c'_2)$ *and* $(x,y) = (M(w,c'_1), l_1(M(w,c'_1)))$*). Furthermore, we have* $M(u) = M(w,c_2)$*.*

Proof. Let us consider case (1) first. Let $S = \{(x',y') \in B(w) \mid x' \text{ is a descendant of } c_1\}$. Then we have $M(S) = M(w,c_1)$. Let (x',y') be a back-edge in $B(w)$. Then, $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $(x',y') \in B(u) \sqcup B(v)$ or $(x',y') = (x,y)$. The case $(x',y') = (x,y)$ is rejected, because x' is a descendant of c_1, whereas x is a descendant of c_2 (and c_1, c_2 are not related as ancestor and descendant). Thus, we have $(x',y') \in B(u) \sqcup B(v)$. Due to the generality of $(x',y') \in S$, this implies that $S \subseteq B(u) \sqcup B(v)$. Conversely, let $(x',y') \in B(u)$. Then x' is a descendant of u, and therefore a descendant of c_1. Furthermore, $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that $(x',y') \in B(w)$. Thus, we have $(x',y') \in S$. Due to the generality of $(x',y') \in B(u)$, this implies that $B(u) \subseteq S$. Similarly, we can show that $B(v) \subseteq S$. Thus, we have $S \subseteq B(u) \sqcup B(v)$. Since $B(u) \sqcup B(v) \subseteq S$, this can be strengthened to $S = B(u) \sqcup B(v)$. Therefore, $M(S)$ is an ancestor of both $M(u)$ and $M(v)$. Let us suppose, for the sake of contradiction, that $M(u) = M(S)$. Then, $M(u)$ is an ancestor of $M(v)$, and therefore u is an ancestor of $M(v)$. Thus, $M(v)$ is a common descendant of v and u, in contradiction to the fact that u and v are not related as ancestor and descendant. Thus, we have that $M(u)$ is a proper descendant of $M(S)$. Similarly, we can show that $M(v)$ is a proper descendant of $M(S)$.

Let us suppose, for the sake of contradiction, that there is a back-edge of the form $(M(S),z)$ in S. Then, since $S = B(u) \sqcup B(v)$, we have that either $(M(S),z) \in B(u)$, or $(M(S),z) \in B(v)$. The first case implies that $M(S)$ is a descendant of $M(u)$, and therefore $M(S) = M(u)$ (since $M(S)$ is an ancestor of $M(u)$), which is impossible. Thus, the case $(M(S),z) \in B(u)$ is rejected. Similarly, we can reject $(M(S),z) \in B(v)$. Therefore, there are no viable options left, and so we have arrived at a contradiction. This shows that there is no back-edge of the form $(M(S),z)$ in S. Thus, there are at least two back-edges (x_1,y_1) and (x_2,y_2) in S such that x_1 is a descendant of the *low1* child c_1' of $M(S)$, and x_2 is a descendant of the *low2* child c_2' of $M(S)$. Since $S = B(u) \sqcup B(v)$, we have that $(x_1,y_1) \in B(u) \sqcup B(v)$ and $(x_2,y_2) \in B(u) \sqcup B(v)$. Notice that we cannot have that both $(x_1,y_1) \in B(u)$ and $(x_2,y_2) \in B(u)$, because this would imply that $M(u)$ is an ancestor of $nca\{x_1,x_2\} = M(S)$, and so $M(S) = M(u)$ (since $M(S)$ is an ancestor of $M(u)$), which is impossible. Similarly, it cannot be that both (x_1,y_1) and (x_2,y_2) are in $B(v)$. Thus, we have that one of (x_1,y_1) and (x_2,y_2) is in $B(u)$, and the other is in $B(v)$. Let us assume w.l.o.g. that $(x_1,y_1) \in B(u)$ and $(x_2,y_2) \in B(v)$. This implies that $M(u)$ is an ancestor of x_1, and $M(v)$ is an ancestor of x_2. Since x_1 is a common descendant of c_1' and $M(u)$, we have that c_1' and $M(u)$

are related as ancestor and descendant. Similarly, since x_2 is a common descendant of c'_2 and $M(v)$, we have that c'_2 and $M(v)$ are related as ancestor and descendant.

Now let $S_1 = \{(x', y') \in B(w) \mid x' \text{ is a descendant of } c'_1\}$. Then we have $M(S_1) = M(w, c'_1)$. Let (x', y') be a back-edge in S_1. Then, x' is a descendant of c'_1, and therefore a descendant of $M(S)$. Thus, since $(x', y') \in B(w)$, we have that $(x', y') \in S$. Since $S = B(u) \sqcup B(v)$, this implies that either $(x', y') \in B(u)$ or $(x', y') \in B(v)$. Let us suppose, for the sake of contradiction, that $(x', y') \in B(v)$. Then, x' is a descendant of $M(v)$. Thus, we have that x' is a common descendant of c'_1 and $M(v)$, and therefore c'_1 and $M(v)$ are related as ancestor and descendant. Since $M(v)$ is related as ancestor and descendant with c'_2, but c'_1 and c'_2 are not related as ancestor and descendant (because they have the same parent), we have that $M(v)$ is an ancestor of both c'_1 and c'_2. This implies that $M(v)$ is an ancestor of $nca\{c'_1, c'_2\} = M(S)$, and therefore $M(S) = M(v)$ (since $M(S)$ is an ancestor of $M(v)$), which is impossible. Thus, the case $(x', y') \in B(v)$ is rejected, and so we have $(x', y') \in B(u)$. Due to the generality of $(x', y') \in S_1$, this implies that $S_1 \subseteq B(u)$. Conversely, let (x', y') be a back-edge in $B(u)$. Then $S = B(u) \sqcup B(v)$ implies that $(x', y') \in S$. Thus, since there is no back-edge of the form $(M(S), z)$ in S, we have that x' is a descendant of a child c of $M(S)$. Let us suppose, for the sake of contradiction, that $c \neq c'_1$. Since $(x', y') \in B(u)$, we have that x' is a descendant of $M(u)$. Thus, since x' is a common descendant of c and $M(u)$, we have that c and $M(u)$ are related as ancestor and descendant. Since $M(u)$ is related as ancestor and descendant with c'_1, but c and c'_1 are not related as ancestor and descendant (since they have the same parent), we have that $M(u)$ is a common ancestor of c'_1 and c, and thus $M(u)$ is an ancestor of $nca\{c, c'_1\} = M(S)$. Since $M(S)$ is an ancestor of $M(u)$, this implies that $M(S) = M(u)$, which is impossible. Thus, we have that $c = c'_1$, and therefore x' is a descendant of c'_1. Furthermore, since $(x', y') \in B(w)$, we have that $(x', y') \in S_1$. Due to the generality of $(x', y') \in B(u)$, this implies that $B(u) \subseteq S_1$. Thus, since $S_1 \subseteq B(u)$, we have that $S_1 = B(u)$. This implies that $M(S_1) = M(u)$, and therefore $M(w, c'_1) = M(u)$. Similarly, we can show that $M(w, c'_2) = M(v)$.

Now let $S' = \{(x', y') \in B(w) \mid x' \text{ is a descendant of } c_2\}$. Notice that $M(w, c_2) = M(S')$, and $(x, y) \in S'$. Let us suppose, for the sake of contradiction, that there is a back-edge (x', y') in S' such that $(x', y') \neq (x, y)$. Then, $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that $(x', y') \in B(u) \sqcup B(v)$, and therefore $(x', y') \in S$. This implies that x' is a descendant of c_1, which is impossible, since x' is a descendant of c_2 (and c_1, c_2 are not

related as ancestor and descendant). Thus, we have that $S' = \{(x,y)\}$, and therefore $M(S') = x$, and therefore $M(w,c_2) = x$. Lemma 5.50 implies that $(x,y) = (x, l_1(x))$, and so $e = (M(w,c_2), l_1(M(w,c_2)))$.

The arguments for cases (2) and (3) are similar to those we have used for case (1). □

Based on Lemma 5.55, we can compute all Type-3αii 4-cuts that satisfy (4.1) of Lemma 5.50 as follows. First, it is sufficient to consider only those $w \neq r$ such that $M(w)$ has at least two children. Let c_1 and c_2 be the *low1* and *low2* children of $M(w)$. Let $\{(u,p(u)),(v,p(v)),(w,p(w)),(x,y)\}$ be a 4-cut that satisfies (4.1) of Lemma 5.50, where u and v are both descendants of w, and (x,y) is the back-edge in $B(w)\setminus(B(u)\sqcup B(v))$. Then there are six different possibilities:

(1) u and v are descendants of c_1, and x is a descendant of c_2.

(2) u and x are descendants of c_1, and v is a descendant of c_2.

(3) v and x are descendants of c_1, and u is a descendant of c_2.

(4) u and v are descendants of c_2, and x is a descendant of c_1.

(5) u and x are descendants of c_2, and v is a descendant of c_1.

(6) v and x are descendants of c_2, and u is a descendant of c_1.

Notice that cases (2)-(3) and (5)-(6) are equivalent from an algorithmic perspective, because the names of the variables do not matter (i.e., the names of u and v can be exchanged). Thus, the possible cases that we have to consider are reduced to four.

First, we may have that two of the vertices from $\{u,v,x\}$ are descendants of c_1, and the other is a descendant of c_2. In this case, we need to have computed $M(w,c_1')$ and $M(w,c_2')$, where c_1' and c_2' are the *low1* and *low2* children of $M(w,c_1)$. Furthermore, we need to have computed $M(w,c_2)$. Now, we may have that both u and v are descendants of c_1, and x is a descendant of c_2. In this case, we have that $M(u) = M(w,c_1')$ and $M(v) = M(w,c_2')$ (or reversely), and $(x,y) = (M(w,c_2), l_1(M(w,c_2)))$. Otherwise, we have that both u and x are descendants of c_1, and v is a descendant of c_2. Then, we have that either $M(u) = M(w,c_1')$ and $(x,y) = (M(w,c_2'), l_1(M(w,c_2')))$, or $M(u) = M(w,c_2')$ and $(x,y) = (M(w,c_1'), l_1(M(w,c_1')))$. In either case, we have $M(v) = M(w,c_2)$. On the other hand, we may have that two of the vertices from

$\{u, v, x\}$ are descendants of c_2, and the other is a descendant of c_1. This case is treated similarly (we just reverse the roles of c_1 and c_2). The process that we follow in order to compute all those 4-cuts is shown in Algorithm 30. The proof of correctness and linear complexity is given in Proposition 5.18.

Proposition 5.18. *Algorithm 30 correctly computes all Type-3αii 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$, where w is a common ancestor of $\{u, v\}$, and e satisfies (4.1) of Lemma 5.50. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3αii 4-cut, where w is a common ancestor of $\{u, v\}$, and e satisfies (4.1) of Lemma 5.50. Let the higher endpoint of e be x, and let us assume that two of $\{u, v, x\}$ are descendants of the *low1* child c_1 of $M(w)$, and the other is a descendant of the *low2* child c_2 of $M(w)$. (The other case is treated similarly, by reversing the roles of c_1 and c_2.) Let c'_1 be the *low1* child of $M(w, c_1)$, and let c'_2 be the *low2* child of $M(w, c_1)$. The possible cases here are: (1) u and v are descendants of c_1, or (2) u and x are descendants of c_1, or (3) v and x are descendants of c_1. We note that case (3) can be subsumed by case (2) (by exchanging the names of u and v), and thus we may ignore it. In case (1), Lemma 5.55 implies (w.l.o.g.) that $M(u) = M(w, c'_1)$, $M(v) = M(w, c'_2)$ and $e = (M(w, c_2), l_1(M(w, c_2)))$. Then, by Lemma 5.51 we have that u is the lowest proper descendant of w that has $M(u) = M(w, c'_1)$, and v is the lowest proper descendant of w that has $M(v) = M(w, c'_2)$. Furthermore, according to Lemma 5.52, we have that $\mathit{bcount}(w) = \mathit{bcount}(u) + \mathit{bcount}(v) + 1$, $\mathit{high}(u) < w$ and $\mathit{high}(v) < w$. Thus, C will be marked in Line 11. In case (2), Lemma 5.55 implies that either $M(u) = M(w, c'_1)$, $e = (M(w, c'_2), l_1(M(w, c'_2)))$ and $M(v) = M(w, c_2)$, or $M(u) = M(w, c'_2)$, $e = (M(w, c'_1), l_1(M(w, c'_1)))$ and $M(v) = M(w, c_2)$. Thus, the same argument as before implies that C will be marked in Line 16 or 21, respectively.

Conversely, let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), (x, y)\}$ be a 4-element set that is marked in Line 11, or 16, or 21. Suppose first that C is marked in Line 11. Then we have $(x, y) = (M(w, c_2), l_1(M(w, c_2)))$, $\mathit{bcount}(w) = \mathit{bcount}(u) + \mathit{bcount}(v) + 1$, $\mathit{high}(u) < w$ and $\mathit{high}(v) < w$. Furthermore, we have $M(u) = M(w, c'_1)$ and $M(v) = M(w, c'_2)$, and therefore we can use a similar argument as in the proof of Lemma 3.12 in order to show that u and v are not related as ancestor and descendant. (The argument hinges on the fact that c'_1 and c'_2 are different children of the same vertex, and w is an ancestor of both u and v.) Thus, Lemma 5.52 implies that there is a back-edge e such

Algorithm 30: Compute all Type-3αii 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where w is a common ancestor of $\{u,v\}$, and e satisfies (4.1) of Lemma 5.50.

```
// We consider the case in which two vertices from {u, v, x} are
   descendants of the low1 child of M(w) and the other is a descendant
   of the low2 child of M(w), where x is the higher endpoint of e; the
   other case is treated similarly, by reversing the roles of c1 and c2
   below
1  foreach vertex w ≠ r do
2      if M(w) has less than two children then continue
3      let c1 and c2 be the low1 and low2 children of M(w)
4      if M(w, c1) = ⊥ or M(w, c2) = ⊥ then continue
5      if M(w, c1) has less than two children then continue
6      let c'1 and c'2 be the low1 and low2 children of M(w, c1)
7      if M(w, c'1) = ⊥ or M(w, c'2) = ⊥ then continue
8      let u be the lowest proper descendant of w that has M(u) = M(w, c'1)
9      let v be the lowest proper descendant of w that has M(v) = M(w, c'2)
10     if high(u) < w and high(v) < w and bcount(w) = bcount(u) + bcount(v) + 1
        then
11         mark {(u, p(u)), (v, p(v)), (w, p(w)), (M(w, c2), l1(M(w, c2)))} as a
            Type-3αii 4-cut
12     end
13     let u be the lowest proper descendant of w that has M(u) = M(w, c'1)
14     let v be the lowest proper descendant of w that has M(v) = M(w, c2)
15     if high(u) < w and high(v) < w and bcount(w) = bcount(u) + bcount(v) + 1
        then
16         mark {(u, p(u)), (v, p(v)), (w, p(w)), (M(w, c'2), l1(M(w, c'2)))} as a
            Type-3αii 4-cut
17     end
18     let u be the lowest proper descendant of w that has M(u) = M(w, c'2)
19     let v be the lowest proper descendant of w that has M(v) = M(w, c2)
20     if high(u) < w and high(v) < w and bcount(w) = bcount(u) + bcount(v) + 1
        then
21         mark {(u, p(u)), (v, p(v)), (w, p(w)), (M(w, c'1), l1(M(w, c'1)))} as a
            Type-3αii 4-cut
22     end
23 end
```

that $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$. Since $M(w, c_2) \neq \bot$, we have that there is a back-edge $(x', y') \in B(w)$ such that x' is a descendant of c_2. Thus, we have $(x', y') \notin B(u) \cup B(v)$ (because otherwise, we would have that x' is a descendant of either $M(u)$ or $M(v)$, and therefore a descendant of either $M(w, c_1')$ or $M(w, c_2')$, and therefore a descendant of c_1). Thus, we have that $(x', y') = e$, and that this is the only back-edge in $B(w)$ that stems from $T(c_2)$. Thus, we have $e = (M(w, c_2), l_1(M(w, c_2)))$, and therefore $e = (x, y)$. This shows that C is indeed a Type-3αii 4-cut. With similar arguments we can show that, if C is marked in Lines 16 or 21, then C is a Type-3αii 4-cut.

Now we will argue about the complexity of Algorithm 30. Notice that for every $w \neq r$ such that $M(w)$ has at least two children, we have to compute the values $M(w, c_1)$ and $M(w, c_2)$, where c_1 and c_2 are the *low1* and *low2* children of $M(w)$. According to Proposition 3.5, these computations take linear time in total, for all such vertices w. Then, if $M(w, c_1) \neq \bot$ and $M(w, c_1)$ has at least two children, we have to compute the values $M(w, c_1')$ and $M(w, c_2')$, where c_1' and c_2' are the *low1* and *low2* children of $M(w, c_1)$. According to Proposition 3.5, these computation take linear time in total, for all such vertices w. Finally, the vertices u and v in Lines 8, 9, 13, 14, 18 and 19, can be computed with Algorithm 22, as explained e.g. in the proof of Proposition 5.16. According to Lemma 5.27, all these computations take $O(n)$ time in total. We conclude that Algorithm 30 has a linear-time implementation. $\square$

Now we consider case (4.2) of Lemma 5.50.

Lemma 5.56. *Let case* (4.2) *of Lemma 5.50 be true. Let* c_1, c_2, *and* c_3, *be the low1, low2, and low3 children of* $M(w)$ *(not necessarily in that order). Let us assume that* u *is a descendant of* c_1, v *is a descendant of* c_2, *and* x *is a descendant of* c_3. *Then* $M(u) = M(w, c_1)$, $M(v) = M(w, c_2)$, *and* $e = (M(w, c_3), l_1(M(w, c_3)))$.

Proof. Let $S_1 = \{(x', y') \in B(w) \mid x' \text{ is a descendant of } c_1\}$. Then, $M(w, c_1) = M(S_1)$. Let (x', y') be a back-edge in $B(u)$. Then x' is a descendant of u, and therefore a descendant of c_1. Furthermore, $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that $(x', y') \in B(w)$. Thus, we have $(x', y') \in S_1$. This implies that x' is a descendant of $M(S_1)$. Due to the generality of $(x', y') \in B(u)$, this implies that $M(u)$ is a descendant of $M(S_1)$. Conversely, let (x', y') be a back-edge in S_1. Then $(x', y') \in B(w)$, and so $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $(x', y') \in B(u)$, or $(x', y') \in B(v)$, or $(x', y') = (x, y)$. Since x' is a descendant of c_1, we have that x' cannot be a descendant of either c_2 or c_3. Thus, the cases $(x', y') \in B(v)$ and $(x', y') = (x, y)$ are rejected.

(Because $(x',y') \in B(v)$ would imply that x' is a descendant of v, and therefore a descendant of c_2; and $x' = x$ would imply that x' is a descendant of c_3.) Thus, we are left with the case $(x',y') \in B(u)$. This implies that x' is a descendant of $M(u)$. Due to the generality of $(x',y') \in S_1$, this implies that $M(S_1)$ is a descendant of $M(u)$. Since $M(u)$ is a descendant of $M(S_1)$, this shows that $M(u) = M(S_1)$, and therefore $M(u) = M(w,c_1)$. Similarly, we can show that $M(v) = M(w,c_2)$.

Now let $S_3 = \{(x',y') \in B(w) \mid x' \text{ is a descendant of } c_3\}$. Then, $M(w,c_3) = M(S_3)$. We obviously have $(x,y) \in S_3$, and so x is a descendant of $M(S_3)$. Let us suppose, for the sake of contradiction, that there is a back-edge $(x',y') \in S_3$ such that $(x',y') \neq (x,y)$. Then we have that $(x',y') \in B(w)$, and so $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that either $(x',y') \in B(u)$, or $(x',y') \in B(v)$, or $(x',y') = (x,y)$. The last case is rejected by assumption. If $(x',y') \in B(u)$, then we have that x' is a descendant of u, and so u and c_3 are related as ancestor and descendant (since they have x' as a common descendant). Since u is not a descendant of c_3, we have that u is a proper ancestor of c_3, and therefore an ancestor of $M(w)$. But this is impossible, since u is a descendant of c_1. Thus, the case $(x',y') \in B(u)$ is rejected. Similarly, the case $(x',y') \in B(v)$ is also rejected. But then there are no viable options left, and so we have a contradiction. Thus, we have that (x,y) is the only back-edge in S_3, and so $M(S_3) = x$. By Lemma 5.50, we have that $e = (x, l_1(x))$. ☐

In order to compute all Type-3αii 4-cuts that satisfy (4.2) of Lemma 5.50, we can apply the information provided by Lemma 5.56 as follows. First, we notice that we need to process only those vertices $w \neq r$ such that $M(w)$ has at least three children. Let c_1, c_2 and c_3 be the *low1*, the *low2* and the *low3* child of $M(w)$. We assume that we have $M(w,c_1) \neq \bot$, $M(w,c_2) \neq \bot$ and $M(w,c_3) \neq \bot$. First, we find the lowest proper descendant u of w that has $M(u) = M(w,c_1)$, and the lowest proper descendant v of w that has $M(v) = M(w,c_2)$ (this is according to Lemma 5.51). Then we check whether $\mathit{high}(u) < w$, $\mathit{high}(v) < w$, and $\mathit{bcount}(w) = \mathit{bcount}(u) + \mathit{bcount}(v) + 1$, in order to establish with the use of Lemma 5.52 that we indeed have a Type-3αii 4-cut. If that is the case, then we know that the back-edge of this 4-cut is $(M(w,c_3), l_1(M(w,c_3)))$. Otherwise, we find the lowest proper descendant u of w that has $M(u) = M(w,c_1)$, and the lowest proper descendant v of w that has $M(v) = M(w,c_3)$, and we perform the same checks. The back-edge in this case is $(M(w,c_2), l_1(M(w,c_2)))$. Finally, we find the lowest proper descendant u of w that has $M(u) = M(w,c_2)$, and the lowest proper

descendant v of w that has $M(v) = M(w, c_3)$. Again, we perform the same checks; the back-edge in this case is $(M(w, c_1), l_1(M(w, c_1)))$. The procedure for finding all those 4-cuts is shown in Algorithm 31. The proof of correctness and linear complexity is given in Proposition 5.19.

Proposition 5.19. *Algorithm 31 correctly computes all Type-3αii 4-cuts of the form* $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$*, where* w *is a common ancestor of* $\{u, v\}$*, and* e *satisfies* (4.2) *of Lemma 5.50. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3αii 4-cut, where w is a common ancestor of $\{u, v\}$, and e satisfies (4.2) of Lemma 5.50. Let x be the higher endpoint of e, and let c_1, c_2 and c_3 be the *low1*, *low2* and *low3* children of $M(w)$, respectively. Let us suppose first that u is a descendant of c_1, v is a descendant of c_2, and x is a descendant of c_3. Then Lemma 5.56 implies that $M(u) = M(w, c_1)$, $M(v) = M(w, c_2)$ and $e = (M(w, c_3), l_1(M(w, c_3)))$. Then, Lemma 5.51 implies that u is the lowest proper descendant of w such that $M(u) = M(w, c_1)$, and v is the lowest proper descendant of w such that $M(v) = M(w, c_2)$. Lemma 5.52 implies that $bcount(w) = bcount(u) + bcount(v) + 1$, $high(u) < w$ and $high(v) < w$. Thus, C will be marked in Line 8. Similarly, if we assume that u is a descendant of c_1, v is a descendant of c_3, and x is a descendant of c_2, or that u is a descendant of c_2, v is a descendant of c_3, and x is a descendant of c_1, then we have that C will be marked in Line 13, or 18, respectively. (The other cases that we have tacitly ignored, e.g., the case where u is a descendant of c_2 and v is a descendant of c_1, are basically subsumed in the cases that we considered; to see this, just exchange the names of the variables u and v.)

Conversely, let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), (x, y)\}$ be a 4-element set that is marked in Line 8, or 13, or 18. Let us suppose first that C is marked in Line 8. Then we have $(x, y) = (M(w, c_3), l_1(M(w, c_3)))$, $bcount(w) = bcount(u) + bcount(v) + 1$, $high(u) < w$ and $high(v) < w$. Furthermore, since $M(u) = M(w, c_1)$ and $M(v) = M(w, c_2)$, and c_1, c_2 are different children of $M(w)$, Lemma 3.12 implies that u and v are not related as ancestor and descendant. Thus, Lemma 5.52 implies that there is a back-edge e such that $B(w) = (B(u) \sqcup B(v)) \sqcup \{e\}$. Then, since $M(w, c_3) \neq \bot$, we have that there is a back-edge $(x', y') \in B(w)$ such that x' is a descendant of c_3. Then, we have $(x', y') \notin B(u) \cup B(v)$ (because otherwise, we would have that x' is a descendant of either $M(u)$ or $M(v)$, and therefore a descendant of either $M(w, c_1)$

Algorithm 31: Compute all Type-3αii 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where w is a common ancestor of $\{u,v\}$, and e satisfies (4.2) of Lemma 5.50.

```
1  foreach vertex w ≠ r do
2      if M(w) has less than three children then continue
3      let c1, c2 and c3 be the low1, low2 and low3 children of M(w)
4      if either of M(w,c1), M(w,c2) or M(w,c3) is ⊥ then continue
5      let u be the lowest proper descendant of w such that M(u) = M(w,c1)
6      let v be the lowest proper descendant of w such that M(v) = M(w,c2)
7      if high(u) < w and high(v) < w and bcount(w) = bcount(u) + bcount(v) + 1
        then
8          mark {(u,p(u)),(v,p(v)),(w,p(w)),(M(w,c3),l1(M(w,c3)))} as a
            Type-3αii 4-cut
9      end
10     let u be the lowest proper descendant of w such that M(u) = M(w,c1)
11     let v be the lowest proper descendant of w such that M(v) = M(w,c3)
12     if high(u) < w and high(v) < w and bcount(w) = bcount(u) + bcount(v) + 1
        then
13         mark {(u,p(u)),(v,p(v)),(w,p(w)),(M(w,c2),l1(M(w,c2)))} as a
            Type-3αii 4-cut
14     end
15     let u be the lowest proper descendant of w such that M(u) = M(w,c2)
16     let v be the lowest proper descendant of w such that M(v) = M(w,c3)
17     if high(u) < w and high(v) < w and bcount(w) = bcount(u) + bcount(v) + 1
        then
18         mark {(u,p(u)),(v,p(v)),(w,p(w)),(M(w,c1),l1(M(w,c1)))} as a
            Type-3αii 4-cut
19     end
20 end
```

or $M(w, c_2)$, and therefore a descendant of either c_1 or c_2, which is impossible). This implies that $(x', y') = e$, and that (x', y') is the only back-edge in $B(w)$ that stems from $T(c_3)$. Thus, we have that $e = (M(w, c_3), l_1(M(w, c_3)))$, and therefore $e = (x, y)$. This shows that C is indeed a Type-3αii 4-cut. Similarly, if we have that C is marked in Line 13 or 18, then we can use a similar argument in order to show that C is a Type-3αii 4-cut.

Now we will argue about the complexity of Algorithm 31. According to Proposition 3.5, we can compute the values $M(w, c_1)$, $M(w, c_2)$ and $M(w, c_3)$ in linear time in total, for every vertex $w \neq r$ such that $M(w)$ has at least three children, where c_1, c_2 and c_3 are the *low1*, *low2* and *low3* children of $M(w)$, respectively. The values u and v in Lines 5, 6, 10, 11, 15 and 16, can be computed with Algorithm 22, as explained e.g. in the proof of Proposition 5.16. According to Lemma 5.27, all these computations take $O(n)$ time in total. We conclude that Algorithm 31 runs in linear time. □

5.8 Computing Type-3β 4-cuts

Throughout this section, we assume that G is a 3-edge-connected graph with n vertices and m edges. All graph-related elements (e.g., vertices, edges, cuts, etc.) refer to G. Furthermore, we assume that we have computed a DFS-tree T of G rooted at a vertex r.

Lemma 5.57. *Let u, v, w be three vertices $\neq r$ such that w is proper ancestor of v and v is a proper ancestor of u, and let e be a back-edge. Then $C' = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ is a 4-cut if and only if one of the following is true. (See Figure 5.23.)*

(1) $e \in B(u) \cap B(v) \cap B(w)$ *and* $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$

(2) $e \in B(w)$, $e \notin B(v) \cup B(u)$, *and* $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$

(3) $e \in B(u)$, $e \notin B(v) \cup B(w)$, *and* $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$

(4) $e \in B(v)$ *and* $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$

Proof. ($\Rightarrow$) Consider the parts $A = T(u)$, $B = T(v) \setminus T(u)$, $C = T(w) \setminus T(v)$, and $D = T(r) \setminus T(w)$. Observe that every one of those parts remains connected in $G \setminus$

$\{(u,p(u)),(v,p(v)),(w,p(w))\}$. Since C' is a 4-cut, by Lemma 3.14 we have that $e \in B(u) \cup B(v) \cup B(w)$.

Let us suppose, first, that $e \in B(u) \cap B(v) \cap B(w)$. This means that e connects A and D. Thus, since C' is a 4-cut, we have that A and D must be disconnected in $G \setminus C'$, and so e is the unique back-edge that connects A and D. Furthermore, since e connects A and D, notice that there is no back-edge from A to B, or from B to C, or from C to D, for otherwise u would be connected with $p(u)$, or v would be connected with $p(v)$, or w would be connected with $p(w)$, respectively, in $G \setminus C'$, in contradiction to the fact that C' is a 4-cut of G. Let e' be a back-edge in $B(v) \setminus \{e\}$. Then this can be a back-edge from A to C, or from B to D. The first case implies that e' is in $B(u)$, and the second case implies that e' is in $B(w)$. This shows that $B(v) \setminus \{e\} \subseteq B(u) \cup B(w)$, which implies that $B(v) \setminus \{e\} \subseteq (B(u) \setminus \{e\}) \cup (B(w) \setminus \{e\})$. Conversely, let e' be a back-edge in $(B(u) \cup B(w)) \setminus \{e\}$. Then this is either a back-edge from A to C (if $e' \in B(u) \setminus \{e\}$), or from B to D (if $e' \in B(w) \setminus \{e\}$). Thus we have that $(B(u) \cup B(w)) \setminus \{e\} \subseteq B(v)$, which implies that $(B(u) \setminus \{e\}) \cup (B(w) \setminus \{e\}) \subseteq B(v) \setminus \{e\}$. We infer that $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \cup (B(w) \setminus \{e\})$. Now let e' be a back-edge in $B(u) \setminus \{e\}$. Then this can only be a back-edge from A to C. Thus, e' cannot be in $B(w) \setminus \{e\}$, because all the back-edges in $B(w)$ have their lower endpoint in D. Thus we have $(B(u) \setminus \{e\}) \cap (B(w) \setminus \{e\}) = \emptyset$, and so it is proper to write $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$ (case (1)).

From now on, let us suppose that $e \notin B(u) \cap B(v) \cap B(w)$. We will show that e belongs to exactly one of $B(u)$, $B(v)$, or $B(w)$. So let us suppose, for the sake of contradiction, that this is not true. Then there are three possible cases to consider: either (I) $e \in B(u) \cap B(v)$ and $e \notin B(w)$, or (II) $e \in B(u) \cap B(w)$ and $e \notin B(v)$, or (III) $e \in B(v) \cap B(w)$ and $e \notin B(u)$. Suppose that (I) is true. Then e is either from A to C, or from A to D. The second case is rejected since $e \notin B(w)$. Thus, e connects A and C. Since C' is a 4-cut of G, we have that e is the only back-edge from A to C. Furthermore, we have that there are no back-edges from B to C, or from C to D. Thus, C becomes disconnected from the rest of the graph in $G \setminus \{(v,p(v)),(w,p(w)),e\}$, in contradiction to the fact that C' is a 4-cut of G. This shows that (I) cannot be true. Now suppose that (II) is true. Since $e \in B(u) \cap B(w)$, we have that e must be a back-edge from A to D. But this contradicts $e \notin B(v)$. Thus, (II) cannot be true. Finally, suppose that (III) is true. Then $e \in B(v) \cap B(w)$ implies that e either connects A and D, or B and D. The first case is rejected, since $e \notin B(u)$. Thus, e connects B

and D. Since C' is a 4-cut of G, we have that e is the only back-edge from B to D. Furthermore, we have that there are no back-edges from A to B, or from B to C. Thus, B becomes disconnected from the rest of the graph in $G \setminus \{(u,p(u)),(v,p(v)),e\}$, in contradiction to the fact that C' is a 4-cut of G. This shows that (III) cannot be true. Since all cases (I)-(III) lead to a contradiction, we have that e belongs to exactly one of $B(u)$, $B(v)$, or $B(w)$.

Now suppose that $e \in B(w)$. Since $e \notin B(u) \cup B(v)$, we have that e connects C and D. Since C' is a 4-cut, we have that e is the only back-edge that connects C and D. Furthermore, there are no back-edges from A to B, or from B to C. Now let e' be a back-edge in $B(v)$. Then e' is either a back-edge in $B(u)$, or it connects B and D. Thus we have that $e' \in B(u) \cup (B(w) \setminus \{e\})$. This shows that $B(v) \subseteq B(u) \cup (B(w) \setminus \{e\})$. Conversely, let e' be a back-edge in $B(u) \cup (B(w) \setminus \{e\})$. If $e' \in B(u)$, then e' is also in $B(v)$, because there is no back-edge from A to B. And if $e' \in B(w) \setminus \{e\}$, we have that $e' \in B(v)$ because e is the only back-edge from C to D. This shows that $B(u) \cup (B(w) \setminus \{e\}) \subseteq B(v)$, and so we have $B(v) = B(u) \cup (B(w) \setminus \{e\})$. Let us suppose, for the sake of contradiction, that there is a back-edge in $B(u) \cap B(w)$. Then this is a back-edge from A to D. Since there is no back-edge from A to B, we have that $B(u) \subseteq B(v)$. Since the graph is 3-edge-connected, we have that $B(u) \neq B(v)$. Thus, there is a back-edge in $B(v) \setminus B(u)$. Since there are no back-edges from B to C, this must be a back-edge from B to D. Since e is the unique back-edge from C to D, we have that there must be at least one back-edge from A to C, because otherwise C becomes disconnected from the rest of the graph in $G \setminus \{(v,p(v)),(w,p(w)),e\}$. But the existence of back-edges from A to D, from B to D, and from A to C, (and the fact that e is a back-edge from C to D), implies that all parts A-D remain connected in $G \setminus C'$, a contradiction. This shows that $B(u) \cap B(w) = \emptyset$, and therefore it is correct to write $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ (case (2)).

Now suppose that $e \in B(u)$. Since $e \notin B(v) \cup B(w)$, we have that e connects A and B. Since C' is a 4-cut, we have that e is the only back-edge that connects A and B. Furthermore, there are no back-edges from B to C, or from C to D. Now let e' be a back-edge in $B(v)$. Then e' is either a back-edge in $B(u)$, or a back-edge from B to C, or a back-edge from B to D. The case that e' connects B and C is forbidden, and so e' is either in $B(u)$ or in $B(w)$. This shows that $B(v) \subseteq B(u) \cup B(w)$, which can be strengthened to $B(v) \subseteq (B(u) \setminus \{e\}) \cup B(w)$, since $e \notin B(v)$. Conversely, let e' be a back-edge in $(B(u) \setminus \{e\}) \cup B(w)$. If $e' \in B(u) \setminus \{e\}$, then e' is in $B(v)$, because e is the

only back-edge from A to B. And if $e' \in B(w)$, then we have $e' \in B(v)$, because there are no back-edges from C to D. This shows that $(B(u) \setminus \{e\}) \cup B(w) \subseteq B(v)$, and thus we have $B(v) = (B(u) \setminus \{e\}) \cup B(w)$. Now let us suppose, for the sake of contradiction, that there is a back-edge in $B(u) \cap B(w)$. Then this is a back-edge from A to D. Since e the only back-edge from A to B and there is no back-edge from B to C, we have that there must exist a back-edge from B to D, because otherwise B becomes disconnected from the rest of the graph in $G \setminus \{(u, p(u)), (v, p(v)), e\}$, contradicting the fact that C' is a 4-cut of G. Since the graph is 3-edge-connected, we have that $B(w) \neq B(v)$. Thus, since there are no back-edges from C to D or from B to C, we have that there must exist a back-edge from A to C. But now, since there is a back-edge from A to D, a back-edge from B to D, and a back-edge from A to C, (and e is a back-edge from A to B), we have that all parts A-D remain connected in $G \setminus C'$, a contradiction. Thus we have that $B(u) \cap B(w) = \emptyset$, and so it correct to write $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ (case (3)).

Finally, let us suppose that $e \in B(v)$. Since $e \notin B(u) \cup B(w)$, we have that e connects B and C. Since C' is a 4-cut, we have that e is the only back-edge that connects B and C. Furthermore, there are no back-edges from A to B, or from C to D. Now let e' be a back-edge in $B(v) \setminus \{e\}$. Then e' is either a back-edge in $B(u)$, or a back-edge from B to C, or a back-edge from B to D. The case that e' is a back-edge from B to C is rejected, since e is the only back-edge with this property. Thus we have that e' is either in $B(u)$ or in $B(w)$. This shows that $e' \in B(u) \cup B(w)$, and so we have $B(v) \setminus \{e\} \subseteq B(u) \cup B(w)$. This is equivalent to $B(v) \subseteq (B(u) \cup B(w)) \sqcup \{e\}$, since $e \notin B(u) \cup B(w)$. Conversely, let e' be a back-edge in $B(u) \cup B(w)$. If $e' \in B(u)$, then, since there is no back-edge from A to B, we have that $e' \in B(v)$. And if $e' \in B(w)$, then, since there is no back-edge from C to D, we have that $e' \in B(v)$. This shows that $B(u) \cup B(w) \subseteq B(v)$, which can be strengthened to $(B(u) \cup B(w)) \sqcup \{e\} \subseteq B(v)$, since $e \in B(v)$ and $e \notin B(u) \cup B(w)$. Thus we have that $B(v) = (B(u) \cup B(w)) \sqcup \{e\}$. Now let us suppose, for the sake of contradiction, that $B(u) \cap B(w) \neq \emptyset$. Then there exists at least one back-edge from A to D. Since there is no back-edge from A to B, and e is the only back-edge from B to C, we have that there must exist a back-edge from B to D, because otherwise B would become disconnected from the rest of the graph in $G \setminus \{(u, p(u)), (v, p(v)), e\}$, in contradiction to the fact that C' is a 4-cut of G. Furthermore, since there is no back-edge from C to D, and e is the only back-edge from B to C, we have that there must exist a back-edge from A to C, because otherwise

C would become disconnected from the rest of the graph in $G\setminus\{(v,p(v)),(w,p(w)),e\}$, in contradiction to the fact that C' is a 4-cut of G. But now, since there is a back-edge from A to D, and a back-edge from B to D, and a back-edge from A to C, (and e is a back-edge from B to C), we have that all parts A-D remain connected in $G\setminus C'$, a contradiction. Thus we have that $B(u)\cap B(w)=\emptyset$, and so it is correct to write $B(v)=(B(u)\sqcup B(w))\sqcup\{e\}$ (case (4)).

($\Leftarrow$) We have to show that $G\setminus C'$ is disconnected in every one of cases (1)-(4), but $G\setminus C''$ is connected for every proper subset C'' of C'. Since the graph is 3-edge-connected, it is sufficient to prove that no 3-element subset of C' is a 3-cut of G. Furthermore, since the graph is 3-edge-connected, we have that $|B(u)|>1$, $|B(v)|>1$ and $|B(w)|>1$. Consider the parts $A=T(u)$, $B=T(v)\setminus T(u)$, $C=T(w)\setminus T(v)$, and $D=T(r)\setminus T(w)$. Observe that every one of those parts remains connected in $G\setminus\{(u,p(u)),(v,p(v)),(w,p(w))\}$. Notice that there are six different types of back-edges that connect the parts A-D: back-edges from A to B, from A to C, from A to D, from B to C, from B to D, and from C to D. Notice that such a back-edge is contained either in $B(u)$, or in $B(v)$, or in $B(w)$ (or in intersections or unions between those sets).

Suppose first that (1) is true. Then, since $e\in B(u)\cap B(w)$, we have that e connects A and D. The disjointness of the union in $B(v)\setminus\{e\}=(B(u)\setminus\{e\})\sqcup(B(w)\setminus\{e\})$ implies that e is the only back-edge in $B(u)\cap B(w)$, and so it is the only back-edge from A to D. Let e' be a back-edge in $B(u)\setminus\{e\}$. Then, since $e'\in B(u)$, we have that e' is either from A to B, or from A to C, or from A to D. The latter case is rejected, since we have that e is the only back-edge with this property. The case that e' is from A to B is rejected, because $B(u)\setminus\{e\}\subseteq B(v)$. Thus, e' is a back-edge from A to C. Let e'' be a back-edge in $B(w)\setminus\{e\}$. Then, since $e''\in B(w)$, we have that e'' is either a back-edge from A to D, or from B to D, or from C to D. The case that e'' is from A to D is rejected, because e is the only back-edge with this property. And the case that e'' is from C to D is rejected, since we have $B(w)\setminus\{e\}\subseteq B(v)$. Thus, e'' is a back-edge from B to D. Since $B(v)\setminus\{e\}=(B(u)\setminus\{e\})\sqcup(B(w)\setminus\{e\})$ and $e\in B(u)\cap B(v)\cap B(w)$, we have exhausted all possibilities for the back-edges that connect the parts A-D.

Now we have collected enough information to see why C' is a 4-cut of G. The only different types of back-edges that connect the parts A-D are: at least one back-edge e' from A to C, at least one back-edge e'' from B to D, and e is the unique back-edge from A to D. In this situation, observe that, if we remove C' from G, then G

becomes disconnected into the connected components $A \cup C$ and $B \cup D$. If we remove $\{(u,p(u)),(v,p(v)),e\}$ from G, then there remains a path $A \xrightarrow{e'} C \xrightarrow{(w,p(w))} D$, and so this is not a 3-cut of G (because the endpoints of e remain connected). If we remove $\{(u,p(u)),(w,p(w)),e\}$ from G, then there remains a path $A \xrightarrow{e'} C \xrightarrow{(p(v),v)} B \xrightarrow{e''} D$, and so this is not a 3-cut of G (because the endpoints of e remain connected). If we remove $\{(v,p(v)),(w,p(w)),e\}$ from G, then there remains a path $A \xrightarrow{(u,p(u))} B \xrightarrow{e''} D$, and so this is not a 3-cut of G (because the endpoints of e remain connected). Finally, if we remove $\{(u,p(u)),(v,p(v)),(w,p(w))\}$ from G, then there remains a path $C \xrightarrow{e'} A \xrightarrow{e} D \xrightarrow{e''} B$, and so this is not a 3-cut of G (because all parts A-D remain connected). Thus, we have that C' is a 4-cut of G.

Now suppose that (2) is true. Then, since $e \in B(w)$ and $e \notin B(u) \cup B(v)$, we have that e connects C and D. $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $B(w) \setminus \{e\} \subseteq B(v)$, and therefore e is the unique back-edge from C to D (because all other back-edges in $B(w)$ lie in $B(v)$). Notice that $B(u) \cap (B(w) \setminus \{e\}) = \emptyset$ and $e \notin B(u)$, implies that there is no back-edge in $B(u) \cap B(w)$. Let e' be a back-edge in $B(u)$. Then e' is either from A to B, or from A to C, or from A to D. The latter case is rejected, because there is no back-edge in $B(u) \cap B(w)$. The case that e' is from A to B is also rejected, because $e' \in B(v)$. Thus, e' is a back-edge from A to C. Let e'' be a back-edge in $B(w) \setminus \{e\}$. Then e'' is either a back-edge from A to D, or from B to D, or from C to D. The latter case is rejected, because e is the only back-edge with this property. The case that e'' is from A to D is rejected, since there is no back-edge in $B(u) \cap B(w)$. Thus, e'' is a back-edge from B to D. Since $B(v) = B(u) \sqcup (B(w) \setminus \{e\}$, we have exhausted all different combinations for back-edges that connect the parts A-D.

Now we have collected enough information to see why C' is a 4-cut of G. The only different types of back-edges that connect the parts A-D are: at least one back-edge e' from A to C, at least one back-edge e'' from B to D, and e is the unique back-edge from C to D. In this situation, observe that, if we remove C' from G, then G becomes disconnected into the connected components $A \cup C$ and $B \cup D$. Then we can argue as before, in order to show that no 3-element subset of C' is a 3-cut of G. Thus, we have that C' is a 4-cut of G.

Now suppose that (3) is true. Then, since $e \in B(u)$ and $e \notin B(v) \cup B(w)$, we have that e connects A and B. $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that $B(u) \setminus \{e\} \subseteq B(v)$, and therefore e is the unique back-edge from A to B (because all other back-edges in $B(u)$ lie in $B(v)$). Notice that $(B(u) \setminus \{e\}) \cap B(w) = \emptyset$ and $e \notin B(w)$, implies that

there is no back-edge in $B(u) \cap B(w)$. Let e' be a back-edge in $B(u) \setminus \{e\}$. Then e' is either from A to B, or from A to C, or from A to D. The latter case is rejected, because there is no back-edge in $B(u) \cap B(w)$. The case that e' is from A to B is also rejected, because e is the unique back-edge with this property. Thus, e' is a back-edge from A to C. Let e'' be a back-edge in $B(w)$. Then e'' is either a back-edge from A to D, or from B to D, or from C to D. The latter case is rejected, because we have $B(w) \subseteq B(v)$, and therefore all back-edges in $B(w)$ lie in $B(v)$. The case that e'' is from A to D is rejected, since there is no back-edge in $B(u) \cap B(w)$. Thus, e'' is a back-edge from B to D. Since $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$, we have exhausted all different combinations for back-edges that connect the parts A-D.

Now we have collected enough information to see why C' is a 4-cut of G. The only different types of back-edges that connect the parts A-D are: at least one back-edge e' from A to C, at least one back-edge e'' from B to D, and e is the unique back-edge from A to B. In this situation, observe that, if we remove C' from G, then G becomes disconnected into the connected components $A \cup C$ and $B \cup D$. Then we can argue as before, in order to show that no 3-element subset of C' is a 3-cut of G. Thus, we have that C' is a 4-cut of G.

Finally, suppose that (4) is true. Then, since $e \in B(v)$ and $e \notin B(u) \cup B(w)$, we have that e connects B and C. Since $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, we have that all back-edges in $B(v)$, except e, are either in $B(u)$ or in $B(w)$. Therefore, e is the unique back-edge from B to C. Since $B(u) \cap B(w) = \emptyset$, we have that there is no back-edge from A to D. Let e' be a back-edge in $B(u)$. Then e' is either from A to B, or from A to C, or from A to D. The latter case is rejected, because there is no back-edge in $B(u) \cap B(w)$. The case that e' is from A to B is also rejected, because $e \in B(v)$. Thus, e' is a back-edge from A to C. Let e'' be a back-edge in $B(w)$. Then e'' is either a back-edge from A to D, or from B to D, or from C to D. The latter case is rejected, because we have $B(w) \subseteq B(v)$. The case that e'' is from A to D is rejected, since there is no back-edge in $B(u) \cap B(w)$. Thus, e'' is a back-edge from B to D. Since $B(v) = (B(v) \sqcup B(w)) \sqcup \{e\}$, we have exhausted all different combinations for back-edges that connect the parts A-D.

Now we have collected enough information to see why C' is a 4-cut of G. The only different types of back-edges that connect the parts A-D are: at least one back-edge e' from A to C, at least one back-edge e'' from B to D, and e is the unique back-edge from B to C. In this situation, observe that, if we remove C' from G, then G becomes

disconnected into the connected components $A \cup C$ and $B \cup D$. Then we can argue as before, in order to show that no 3-element subset of C' is a 3-cut of G. Thus, we have that C' is a 4-cut of G. □

We distinguish two types of Type-3β 4-cuts – Type-3βi and Type-3βii – depending on whether the M points of v and w (after the removal of e) coincide. The Type-3βi 4-cuts have the following classification, corresponding to the cases in Lemma 5.57.

1. $e \in B(u) \cap B(v) \cap B(w)$, $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$ and $M(B(v) \setminus \{e\}) \neq M(B(w) \setminus \{e\})$
2. $e \in B(w)$, $e \notin B(u) \cup B(v)$, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ and $M(v) \neq M(B(w) \setminus \{e\})$
3. $e \in B(u)$, $e \notin B(v) \cup B(w)$, $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ and $M(v) \neq M(w)$
4. $e \in B(v)$, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ and $M(B(v) \setminus \{e\}) \neq M(w)$

The Type-3βii 4-cuts are classified as the Type-3βi 4-cuts above, the only difference being that the inequalities are replaced with equalities in each case.

The Type-3βi 4-cuts are easier to handle. This is because, given v, we have enough information to find u and w relatively easily. On the other hand, for Type-3βii 4-cuts we have to apply more sophisticated methods, because it is not straightforward what are the possible u and w that may induce with v a 4-cut of this type. The general idea to compute those 4-cuts is basically to calculate a set of candidates u, for each v, that may induce with v and a w a 4-cut of the desired kind. The search space for w, given v, is known, but not computed (in most cases). Then, given v and u, we can relatively easily determine a w that may induce a desired 4-cut with u and v. Then we apply a criterion in order to check that we indeed get a 4-cut. The remaining 4-cuts (if it is impossible to compute all of them in linear time), are implied from the collection we have computed, plus that returned by Algorithm 24.

5.8.1 Type-3βi 4-cuts

5.8.1.1 Case (1) of Lemma 5.57

Lemma 5.58. *Let u, v, w be three vertices $\neq r$ such that w is proper ancestor of v, v is a proper ancestor of u, and there is a back-edge e such that $e \in B(u) \cap B(v) \cap B(w)$, $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$, and $M(B(v) \setminus \{e\}) \neq M(B(w) \setminus \{e\})$. Then $e = (lowD(u), low(u))$, $high(u) < v$, $low_2(u) \geq w$, and $M(w) = M(v)$. Furthermore,*

$M(u) = M(v,c)$, where c is either the low1 or the low2 child of $M(w)$. Finally, u is the lowest proper descendant of v that has $M(u) = M(v,c)$.

Proof. Since $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$ and $e \in B(v) \cap B(u) \cap B(w)$, we have that $B(u) \subseteq B(v)$ and $B(w) \subseteq B(v)$. Let (x,y) be a back-edge in $B(u)$. Then $B(u) \subseteq B(v)$ implies that $(x,y) \in B(v)$, and therefore y is a proper ancestor of v, and therefore $y < v$. Due to the generality of $(x,y) \in B(u)$, this implies that $high(u) < v$.

The disjoint union in $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$, and the fact that $e \in B(u) \cap B(w)$, implies that $B(u)$ and $B(w)$ intersect only at e. Let us suppose, for the sake of contradiction, that $low_2(u) < w$. Let (x,y) be the *low2* back-edge of u. Then x is a descendant of u, and therefore a descendant of v, and therefore a descendant of w. Since (x,y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of y and w, and therefore y and w are related as ancestor and descendant. Since $y = low_2(u)$ and $low_2(u) < w$, we have that $y < w$, and so y is a proper ancestor of w. This shows that $(x,y) \in B(w)$. Now let (x',y') be the *low1* back-edge of u. Then we have $low_1(u) \leq low_2(u)$, and therefore $y' \leq y$. Since $y < w$, this implies that $y' < w$. But then we can show as previously that $(x',y') \in B(w)$, and so $B(u) \cap B(w)$ contains at least two back-edges (i.e., (x,y) and (x',y')), a contradiction. Thus, we have that $low_2(u) \geq w$. This implies that only the *low1* back-edge of u may be in $B(w)$, and so we have that e is the *low1* back-edge of u.

Now we will show that $M(w) = M(v)$. Since $e \in B(u) \cap B(w)$, we have that the higher endpoint of e is a descendant of both u and $M(w)$. Thus, u and $M(w)$ are related as ancestor and descendant. Let us suppose, for the sake of contradiction, that $M(w)$ is a descendant of u. Since the graph is 3-edge-connected, there is a back-edge $(x,y) \in B(w) \setminus \{e\}$. Then, we have that x is a descendant of $M(w)$, and therefore a descendant of u. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x,y) \in B(u)$. But this contradicts the disjointness of the union in $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$, which implies that there is at most one back-edge in $B(u) \cap B(w)$. This shows that $M(w)$ is a proper ancestor of u. Now, since $B(w) \subseteq B(v)$, we have that $M(w)$ is a descendant of $M(v)$. Conversely, let (x,y) be a back-edge in $B(v)$. If $(x,y) = e$, then we have that $(x,y) \in B(w)$, and therefore x is a descendant of $M(w)$. Otherwise, $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$ implies that either $(x,y) \in B(u)$ or $(x,y) \in B(w)$. If $(x,y) \in B(u)$, then x is a descendant of u, and therefore a descendant of $M(w)$. And if $(x,y) \in B(w)$, then x is a descendant of $M(w)$. Thus, in either case we have

that x is a descendant of $M(w)$. Due to the generality of $(x, y) \in B(v)$, this shows that $M(v)$ is a descendant of $M(w)$. Thus, we have $M(w) = M(v)$.

Since $B(u) \subseteq B(v)$, we have that $M(u)$ is a descendant of $M(v)$. Let us suppose, for the sake of contradiction, that $M(u) = M(v)$. Then, since u is a proper descendant of v, Lemma 3.2 implies that $B(v) \subseteq B(u)$. Therefore, $B(u) \subseteq B(v)$ implies that $B(u) = B(v)$, in contradiction to the fact that the graph is 3-edge-connected. Thus, we have that $M(u) \neq M(v)$, and therefore $M(u)$ is a proper descendant of $M(v)$. Let c be the child of $M(v)$ that is an ancestor of $M(u)$.

Let us suppose, for the sake of contradiction, that c is neither the *low1* nor the *low2* child of $M(v)$. Let $e = (x, y)$. Since $e \in B(u)$, we have that x is a descendant of $M(u)$, and therefore a descendant of c. Furthermore, we have $e \in B(v)$. Thus, y is a proper ancestor of v, and therefore a proper ancestor of $M(v)$, and therefore a proper ancestor of c. This shows that $e \in B(c)$. Since $e \in B(w)$, we have that y is a proper ancestor of w, and therefore $y < w$. Thus, since $e \in B(c)$, we have that $low(c) \leq y < w$. Let c_1 and c_2 be the *low1* and the *low2* child of $M(v)$, respectively. Since c is neither c_1 nor c_2, we have that $low(c_1) \leq low(c_2) \leq low(c) < w$. Now let (x', y') be a back-edge in $B(c_1)$ such that $y' = low(c_1)$. Then, x' is a descendant of c_1, and therefore a descendant of $M(v)$, and therefore a descendant of $M(w)$, and therefore a descendant of w. Furthermore, since (x', y') is a back-edge, y' is an ancestor of x'. Thus, since x' is a common descendant of y' and w, we have that y' and w are related as ancestor and descendant. Then, $y' = low(c_1) < w$ implies that y' is a proper ancestor of w. This shows that $(x', y') \in B(w)$. Since x is a descendant of c, and x' is a descendant of c_1, and $c \neq c_1$, we have that $x \neq x'$ (because otherwise x would be a common descendant of c and c_1, and so c and c_1 would be related as ancestor and descendant, which is absurd). Thus, $(x', y') \in B(w)$ can be strengthend to $(x', y') \in B(w) \setminus \{e\}$. This implies that x' is a descendant of $M(B(w) \setminus \{e\})$. Thus, $M(B(w) \setminus \{e\})$ is an ancestor of a descendant of c_1. Similarly, we can show that $M(B(w) \setminus \{e\})$ is an ancestor of a descendant of c_2. Since c_1 and c_2 are not related as ancestor and descendant, this implies that $M(B(w) \setminus \{e\})$ is an ancestor of $nca\{c_1, c_2\} = M(v) = M(w)$. Since $M(B(w) \setminus \{e\})$ is a descendant of $M(w)$, this implies that $M(B(w) \setminus \{e\}) = M(w) = M(v)$. Since $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$, we have $B(w) \setminus \{e\} \subseteq B(v) \setminus \{e\}$, and therefore $M(B(w) \setminus \{e\})$ is a descendant of $M(B(v) \setminus \{e\})$, which is a descendant of $M(v)$. Thus, since $M(B(w) \setminus \{e\}) = M(v)$, we have $M(B(w) \setminus \{e\}) = M(B(v) \setminus \{e\})$, in contradiction to the assumption of the lemma. Thus, we have that c is either the *low1* or the *low2*

child of $M(v)$.

Let us suppose, for the sake of contradiction, that there is a proper descendant u' of v, that is lower than u and has $M(u') = M(v,c)$. Then, since $M(u') = M(u)$ and $u' < u$, we have that u' is a proper ancestor of u, and Lemma 3.2 implies that $B(u') \subseteq B(u)$. Since the graph is 3-edge-connected, this can be strengthened to $B(u') \subset B(u)$. Thus, there is a back-edge $(x,y) \in B(u) \setminus B(u')$. Then, x is a descendant of $M(u) = M(u')$. Furthermore, $B(u) \subseteq B(v)$ implies that $(x,y) \in B(v)$, and therefore y is a proper ancestor of v, and therefore a proper ancestor of u'. This shows that $(x,y) \in B(u')$, a contradiction. Thus, we have that u is the lowest proper descendant of v that has $M(u) = M(v,c)$. □

Lemma 5.58 provides enough information to guide us into the search for all Type-$3\beta i$ 4-cuts that satisfy (1) of Lemma 5.57. According to Lemma 5.58, we have to find, for every vertex $v \neq r$, the lowest proper descendant u of v that has $M(u) = M(v,c)$, where c is either the *low1* or the *low2* child of $M(v)$. Then, w should satisfy that $M(w) = M(v)$ and $bcount(w) = bcount(v) - bcount(u) + 1$. We note that this w, if it exists, is unique, because it has the same M point as v. Then, once we collect the triple u, v, w we apply the criterion provided by Lemma 5.59, in order to check that we indeed get a 4-cut. This procedure is shown in Algorithm 32. The proof of correctness and linear complexity is given in Proposition 5.20.

Lemma 5.59. *Let $v \neq r$ be a vertex such that $M(v)$ has a child c. Let u be a proper descendant of u such that $M(u)$ is a descendant of c, and let w be a proper ancestor of v such that $M(w) = M(v)$. Then there is a back-edge $e \in B(u) \cap B(v) \cap B(w)$ such that $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$ if and only if: $high(u) < v$, $w \leq low_2(u)$, and $bcount(v) = bount(u) + bcount(w) - 1$.*

Proof. ($\Rightarrow$) $bcount(v) = bount(u) + bcount(w) - 1$ is an immediate consequence of $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$ and $e \in B(u) \cap B(v) \cap B(w)$. Since $B(u) \setminus \{e\} \subseteq B(v) \setminus \{e\}$ and $e \in B(u) \cap B(v)$, we have that $B(u) \subseteq B(v)$. This implies that $high(u) < v$ (because every back-edge $(x,y) \in B(u)$ must have $y < v$). Since $B(u) \cap B(w) \neq \emptyset$ and w is a proper ancestor of u, we have that the *low*-edge of u is in $B(w)$. And since $(B(u) \setminus \{e\}) \cap (B(w) \setminus \{e\}) = \emptyset$ and $e \in B(u) \cap B(w)$, we have that the *low*-edge of u is precisely e. Then, since $(B(u) \setminus \{e\}) \cap (B(w) \setminus \{e\}) = \emptyset$ and u is a proper descendant of w, we have that no back-edge e' in $B(u) \setminus \{e\}$ has lower point that is lower than w (because this would imply that $e' \in B(w)$). This implies that $low_2(u) \geq w$.

($\Leftarrow$) Let (x,y) be a back-edge in $B(u)$. Then, x is a descendant of $M(u)$, and therefore a descendant of c, and therefore a descendant of $M(v)$, and therefore a descendant of v. Furthermore, since (x,y) is a back-edge, y is an ancestor of x. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Since $(x,y) \in B(u)$, we have that y is an ancestor of $high(u)$, and therefore $y \leq high(u)$. Thus, $high(u) < v$ implies that $y < v$, and therefore y is a proper ancestor of v. This shows that $(x,y) \in B(v)$. Due to the generality of $(x,y) \in B(u)$, this implies that $B(u) \subseteq B(v)$. Since w is a proper ancestor of v with $M(w) = M(v)$, by Lemma 3.2 we have that $B(w) \subseteq B(v)$.

Let us suppose, for the sake of contradiction, that $B(w) \cap B(v) = \emptyset$. Then, $B(u) \subseteq B(v)$ and $B(w) \subseteq B(v)$ imply that $B(u) \sqcup B(w) \subseteq B(v)$, and therefore $bcount(u) + bcount(w) \leq bcount(v)$, in contradiction to $bcount(v) = bcount(u) + bcount(w) - 1$. Thus, we have that $B(w) \cap B(v) \neq \emptyset$.

Let $e = (x,y)$ be a back-edge in $B(u)$ such that $y = low_1(u)$. Let us suppose, for the sake of contradiction, that $e \notin B(w)$. Consider a back-edge $(x',y') \in B(u) \setminus \{(x,y)\}$. Then, we have that $y' \geq low_2(u)$. Thus, $low_2(u) \geq w$ implies that $y' \geq w$. This implies that y' cannot be a proper ancestor of w, and therefore $(x',y') \notin B(w)$. Due to the generality of $(x',y') \in B(u) \setminus \{e\}$, this implies that $B(w) \cap (B(u) \setminus \{e\}) = \emptyset$. Therefore, since $e \notin B(w)$, we have that $B(w) \cap B(u) = \emptyset$, a contradiction. Thus, we have that $e \in B(w)$. Furthermore, since $low_2(u) \geq w$, this is the only back-edge in $B(u)$ that is also in $B(w)$.

Thus, since $B(u) \subseteq B(v)$, $B(w) \subseteq B(v)$, $B(u) \cap B(w) = \{e\}$ and $bcount(v) = bcount(u) + bcount(w) - 1$, we have that $B(v) = ((B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})) \sqcup \{e\}$, and therefore $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$. $\square$

Proposition 5.20. *Algorithm 32 correctly computes all Type-3βi 4-cuts that satisfy (1) of Lemma 5.57. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ be a Type-3βi 4-cut such that w is a proper ancestor of v, and v is a proper ancestor of u, where the back-edge e satisfies (1) of Lemma 5.57. Then, by Lemma 5.58 we have that $M(u) = M(v,c)$, where c is either the *low1* or the *low2* child of $M(v)$. Let us assume that c is the *low1* child of $M(v)$ (the other case is treated similarly). Then, Lemma 5.58 implies that u is the lowest proper descendant of v such that $M(u) = M(v,c)$, $e = (lowD(u), low(u))$,

Algorithm 32: Compute all Type-3βi 4-cuts that satisfy (1) of Lemma 5.57

1. **foreach** *vertex* $v \neq r$ **do**
2. let c_1 be the *low1* child of $M(v)$
3. let c_2 be the *low2* child of $M(v)$
4. compute $M(v, c_1)$ and $M(v, c_2)$
5. **end**
6. initialize an array A of size m
7. **foreach** *vertex* $x \neq r$ **do**
8. let c_1 be the *low1* child of x
9. let c_2 be the *low2* child of x
10. **foreach** $z \in M^{-1}(x)$ **do**
11. set $A[\mathit{bcount}(z)] \leftarrow z$
12. **end**
13. **foreach** $v \in M^{-1}(x)$ **do**
    // consider the case where u is a descendant of the *low1* child of x; the other case is treated similarly, by substituting c_1 with c_2
14. let u be the lowest proper descendant of v with $M(u) = M(v, c_1)$
15. let $w \leftarrow A[\mathit{bcount}(v) - \mathit{bcount}(u) + 1]$
16. **if** $w \neq \bot$ ***and*** $w < v$ ***and*** $\mathit{high}(u) < v$ ***and*** $w \leq \mathit{low}_2(u)$ **then**
17. mark $\{(u, p(u)), (v, p(v)), (w, p(w)), (\mathit{lowD}(u), \mathit{low}(u))\}$ as a 4-cut
18. **end**
19. **end**
20. **foreach** $z \in M^{-1}(x)$ **do**
21. set $A[\mathit{bcount}(z)] \leftarrow \bot$
22. **end**
23. **end**

and $M(w) = M(v)$. Lemma 5.59 implies that $bcount(w) = bcount(v) - bcount(u) + 1$, $high(u) < v$ and $w \leq low_2(u)$.

Now, when the **for** loop in Line 7 processes $x = M(v)$, we will eventually reach the **for** loop in Line 13, because $v \in M^{-1}(x)$. Notice that when we reach Line 14 when the **for** loop in Line 13 processes v, we have that the variable "u" is assigned precisely u. Now consider the variable "w" in Line 15. We claim that the $bcount(v) - bcount(u) + 1$ entry of the array A is precisely w. To see this, observe that the **for** loop in Line 10 has processed w (because $M(w) = M(v)$), and at some point inserted into the $bcount(v) - bcount(u) + 1$ entry of A the value w. But then, this entry was not altered afterwards, because Lemma 3.7 implies that all vertices with the same M point have different *bcount* values (since our graph is 3-edge-connected). Thus, the variable "w" in Line 15 holds the value w, and therefore C will be marked in Line 17, since all the conditions to reach this line are satisfied.

Conversely, suppose that a 4-element set $C = \{(u, p(u)), (v, p(v)), (w, p(w)), (x, y)\}$ is marked in Line 17. Then, we have that $(x, y) = (lowD(u), low(u))$, u is a proper descendant of v, $M(u)$ is a descendant of the *low1* child of $M(v)$, w is a proper ancestor of v with $M(w) = M(v)$ (due to $w < v$ in Line 16), $bcount(w) = bcount(v) - bcount(u) + 1$, $high(u) < v$ and $w < low_2(u)$. Thus, Lemma 5.59 implies that there is a back-edge $e \in B(u) \cap B(v) \cap B(w)$ such that $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$. Since $B(u) \cap B(w) \neq \emptyset$ and w is a proper ancestor of u, we have that the *low*-edge of u is in $B(w)$. And since $(B(u) \setminus \{e\}) \cap (B(w) \setminus \{e\}) = \emptyset$, we have that the *low*-edge of u is precisely e. Thus, C is indeed a 4-cut (that satisfies (1) of Lemma 5.57).

Now we will argue about the complexity of Algorithm 32. By Proposition 3.5, we have that the values $M(v, c_1)$ and $M(v, c_2)$ can be computed in linear time in total, for all vertices $v \neq r$, where c_1 and c_2 are the *low1* and *low2* children of $M(v)$. Thus, the **for** loop in Line 1 can be performed in linear time. In order to compute u in Line 14, we use Algorithm 22. Specifically, whenever we reach this line, we generate a query $q(M^{-1}(M(v, c_1)), v)$. This returns the lowest vertex u with $M(u) = M(v, c_1)$ such that $u > v$. Since $M(u) = M(v, c_1)$ implies that $M(u)$ is a common descendant of v and u, we have that v and u are related as ancestor and descendant. Thus, $u > v$ implies that u is a proper descendant of v. Therefore, we have that u is the lowest proper descendant of v such that $M(u) = M(v, c_1)$. Since the number of all those queries is $O(n)$, by Lemma 5.27 we have that Algorithm 22 answers all of them in linear time in total. It is easy to see that the remaining operations of Algorithm 32 take $O(n)$

time in total. We conclude that Algorithm 32 runs in linear time. □

5.8.1.2 Case (2) of Lemma 5.57

Lemma 5.60. *Let u, v, w be three vertices $\neq r$ such that w is proper ancestor of v, v is a proper ancestor of u, and there is a back-edge $e \in B(w)$ such that $e \notin B(v) \cup B(u)$, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$, and $M(v) \neq M(B(w) \setminus \{e\})$. Then w is an ancestor of $low(u)$, $M(w) \neq M(v)$, $M(B(w) \setminus \{e\}) \neq M(w)$, and e is either $(L_1(w), l(L_1(w)))$ or $(R_1(w), l(R_1(w)))$. Furthermore, let c_1 be the low1 child of $M(v)$, and let c_2 be the low2 child of $M(v)$. Then $M(B(w) \setminus \{e\}) = M(v, c_1)$, and u is the lowest proper descendant of v such that $M(u) = M(v, c_2)$.*

Proof. Let us suppose, for the sake of contradiction, that w is not an ancestor of $low(u)$. $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $B(u) \subseteq B(v)$. Thus, the *low*-edge of u is in $B(v)$. This implies that $low(u)$ is a proper ancestor of v. Then, since v is a common descendant of $low(u)$ and w, we have that $low(u)$ and w are related as ancestor and descendant. Since w is not an ancestor of $low(u)$, this implies that w is a proper descendant of $low(u)$. Now let (x', y') be the *low*-edge of u. Then we have that x' is a descendant of u, and therefore a descendant of w. Furthermore, $y' = low(u)$ is a proper ancestor of w. This shows that $(x', y') \in B(w)$. But since $B(u) \cap (B(w) \setminus \{e\}) = \emptyset$, this implies that $(x', y') = e$, contradicting the fact that $e \notin B(u)$. This shows that w is an ancestor of $low(u)$.

Let $e = (x, y)$. Since $e \in B(w)$, we have that y is a proper ancestor of w, and therefore a proper ancestor of v. Thus, since $e \notin B(v)$, we have that x cannot be a descendant of $M(v)$. Since $M(w)$ is an ancestor of x, this implies that $M(w)$ cannot be a descendant of $M(v)$, and therefore $M(w) \neq M(v)$. Since $B(w) \setminus \{e\} \subseteq B(v)$, we have that $M(B(w) \setminus \{e\})$ is a descendant of $M(v)$. Thus, since $M(w)$ cannot be a descendant of $M(v)$, we have $M(w) \neq M(B(w) \setminus \{e\})$. By Lemma 3.9, this implies that either $e = e_L(w)$ or $e = e_R(w)$.

$B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $B(u) \subseteq B(v)$, and therefore $M(u)$ is a descendant of $M(v)$. Thus, since $M(u)$ is a common descendant of u and $M(v)$, we have that u and $M(v)$ are related as ancestor and descendant. Let us suppose, for the sake of contradiction, that u is not proper descendant of $M(v)$. Then, u is an ancestor of $M(v)$. Let (x', y') be a back-edge in $B(v)$. Then x' is a descendant of $M(v)$, and therefore a descendant of u. Furthermore, y' is a proper ancestor of v, and therefore

a proper ancestor of u. This shows that $(x', y') \in B(u)$. Due to the generality of $(x', y') \in B(v)$, this implies that $B(v) \subseteq B(u)$. But $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $B(u) \subseteq B(v)$, and therefore we have $B(u) = B(v)$, in contradiction to the fact that the graph is 3-edge-connected. This shows that u is a proper descendant of $M(v)$.

Let us suppose, for the sake of contradiction, that there is a back-edge of the form $(M(v), z)$ in $B(v)$. Then, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that either $(M(v), z) \in B(u)$, or $(M(v), z) \in B(w) \setminus \{e\}$. The first case is rejected, because it implies that $M(v)$ is a descendant of u. Thus, we have $(M(v), z) \in B(w) \setminus \{e\}$. This implies that $M(v)$ is a descendant of $M(B(w) \setminus \{e\})$. But $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $B(w) \setminus \{e\} \subseteq B(v)$, and therefore $M(B(w) \setminus \{e\})$ is a descendant of $M(v)$, and therefore we have $M(v) = M(B(w) \setminus \{e\})$, contradicting one of the assumptions in the statement of the lemma. Thus, we have that there is no back-edge of the form $(M(v), z)$ in $B(v)$. This implies that $low(c_1) < v$ and $low(c_2) < v$, where c_1 and c_2 is the *low1* and the *low2* child of $M(v)$, respectively.

Let us suppose, for the sake of contradiction, that there is a back-edge $(x', y') \in B(w) \setminus \{e\}$ such that x' is not a descendant of c_1. $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $(x', y') \in B(v)$. Thus, since there is no back-edge of the form $(M(v), z)$ in $B(v)$, we have that x' is a descendant of a child c of $M(v)$. We have that y' is a proper ancestor of v, and therefore a proper ancestor of $M(v)$, and therefore a proper ancestor of c. This shows that $(x', y') \in B(c)$. Thus, we have $low(c) < w$. Since $c \neq c_1$, we have that $low(c_1) \leq low(c)$. Thus, $low(c) < w$ implies that $low(c_1) < w$. This means that there is a back-edge $(x'', y'') \in B(c_1)$ such that $y'' < w$. Then x'' is a descendant of $M(v)$, and therefore a descendant of v, and therefore a descendant of w. Furthermore, since x'' is a common descendant of y'' and w, we have that y'' and w are related as ancestor and descendant. Thus, $y'' < w$ implies that y'' is a proper ancestor of w. This shows that $(x'', y'') \in B(w)$. Since x'' is a descendant of $M(v)$ and y'' is a proper ancestor of v, we also have that $(x'', y'') \in B(v)$. Thus, $e \notin B(v)$ implies that $(x'', y'') \neq e$. Therefore, $(x'', y'') \in B(w)$ can be strengthened to $(x'', y'') \in B(w) \setminus \{e\}$. This implies that x'' is a descendant of $M(B(w) \setminus \{e\})$. Since $(x', y') \in B(w) \setminus \{e\}$, we also have that x' is a descendant of $M(B(w) \setminus \{e\})$. This implies that $M(B(w) \setminus \{e\})$ is an ancestor of $nca\{x', x''\}$. Since x' and x'' are descendants of different children of $M(v)$, we have that $nca\{x', x''\} = M(v)$. But then we have that $M(B(w) \setminus \{e\})$ is an ancestor of $M(v)$, and therefore we have $M(B(w) \setminus \{e\}) = M(v)$, since $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $M(B(w) \setminus \{e\})$ is a descendant of $M(v)$. This contradicts the assumption

$M(B(w) \setminus \{e\}) \neq M(v)$ in the statement of the lemma. Thus, we have that all back-edges in $B(w) \setminus \{e\}$ have their higher endpoint in $T(c_1)$.

Since $low(c_2) < v$, we have that there is a back-edge $(x', y') \in B(c_2)$ such that $y' < v$. Then, x' is a descendant of c_2, and therefore a descendant of $M(v)$, and therefore a descendant of v. Thus, since x' is a common descendant of y' and v, we have that y' and v are related as ancestor and descendant, and therefore $y' < v$ implies that y' is a proper ancestor of v. This shows that $(x', y') \in B(v)$. Thus, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that either $(x', y') \in B(u)$, or $(x', y') \in B(w) \setminus \{e\}$. The second case is rejected, since all back-edges in $B(w) \setminus \{e\}$ have their higher endpoint in $T(c_1)$. Thus, $(x', y') \in B(u)$. This implies that x' is a descendant of u. Since u is a proper descendant of $M(v)$, we have that u is a descendant of a child c of $M(v)$. Then, we have that x' is a descendant of c_2 (by assumption) and also a descendant of c (since it is a descendant of u). Thus, we have that c and c_2 are related as ancestor and descendant. Therefore, since they have the same parent, they must coincide. In other words, we have $c = c_2$.

Let $S_1 = \{(x', y') \in B(v) \mid x' \text{ is a descendant of } c_1\}$. Then, $M(S_1) = M(v, c_1)$. Let (x', y') be a back-edge in S_1. Then $(x', y') \in B(v)$, and therefore $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that either $(x', y') \in B(u)$, or $(x', y') \in B(w) \setminus \{e\}$. Let us suppose, for the sake of contradiction, that $(x', y') \in B(u)$. Then, x' is a descendant of u, and therefore a descendant of c_2. Thus, x' is a common descendant of c_1 and c_2, and therefore c_1 and c_2 are related as ancestor and descendant, which is impossible. Thus, the case $(x', y') \in B(u)$ is rejected, and so we have $(x', y') \in B(w) \setminus \{e\}$. Due to the generality of $(x', y') \in S_1$, this implies that $S_1 \subseteq B(w) \setminus \{e\}$. Conversely, let (x', y') be a back-edge in $B(w) \setminus \{e\}$. Then, we have shown that x' is a descendant of c_1. Furthermore, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $(x', y') \in B(v)$. This shows that $(x', y') \in S_1$. Due to the generality of $(x', y') \in B(w) \setminus \{e\}$, this implies that $B(w) \setminus \{e\} \subseteq S_1$. Thus, since $S_1 \subseteq B(w) \setminus \{e\}$, we have $S_1 = B(w) \setminus \{e\}$. This implies that $M(S_1) = M(B(w) \setminus \{e\})$, and therefore $M(v, c_1) = M(B(w) \setminus \{e\})$.

Let $S_2 = \{(x', y') \in B(v) \mid x' \text{ is a descendant of } c_2\}$. Then, $M(S_2) = M(v, c_2)$. Let (x', y') be a back-edge in S_2. Then $(x', y') \in B(v)$, and therefore $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that either $(x', y') \in B(u)$, or $(x', y') \in B(w) \setminus \{e\}$. The case $(x', y') \in B(w) \setminus \{e\}$ is rejected, because it implies that $(x', y') \in S_1$ (and obviously we have $S_1 \cap S_2 = \emptyset$). Thus, we have $(x', y') \in B(u)$. Due to the generality of $(x', y') \in S_2$, this implies that $S_2 \subseteq B(u)$. Conversely, let (x', y') be a back-edge in $B(u)$. Then, x' is a descendant of u, and therefore a descendant of c_2. Furthermore, $B(v) =$

$B(u) \sqcup (B(w) \setminus \{e\})$ implies that $(x', y') \in B(v)$. This shows that $(x', y') \in S_2$. Due to the generality of $(x', y') \in B(u)$, this implies that $B(u) \subseteq S_2$. Thus, since $S_2 \subseteq B(u)$, we have $S_2 = B(u)$. This implies that $M(S_2) = M(u)$, and therefore $M(v, c_2) = M(u)$.

Let us suppose, for the sake of contradiction, that there is a proper descendant u' of v that is lower than u and such that $M(u') = M(v, c_2)$. Then, since $M(u') = M(u)$ and $u' < u$, we have that u' is a proper ancestor of u, and Lemma 3.2 implies that $B(u') \subseteq B(u)$. Since the graph is 3-edge-connected, this can be strengthened to $B(u') \subset B(u)$. Thus, there is a back-edge $(x', y') \in B(u) \setminus B(u')$. Then, x' is a descendant of u, and therefore a descendant of u'. Furthermore, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $(x', y') \in B(v)$, and therefore y' is a proper ancestor of v, and therefore a proper ancestor of u'. This shows that $(x', y') \in B(u')$, a contradiction. Thus, we have that u is the lowest proper descendant of v such that $M(u) = M(v, c_2)$. $\square$

Lemma 5.61. *Let u, v, w be three vertices $\neq r$ such that w is proper ancestor of v, v is a proper ancestor of u, and there is a back-edge $e \in B(w)$ such that $e \notin B(v) \cup B(u)$, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$, and $M(v) \neq M(B(w) \setminus \{e\})$. Let c_1 be the low1 child of $M(v)$, and let w' be the greatest ancestor of low(u) with the property that there is a back-edge $e' \in B(w')$ such that $M(w') \neq M(B(w') \setminus \{e'\}) = M(v, c_1)$. Then, we have that $e' \notin B(v) \cup B(u)$, $B(v) = B(u) \sqcup (B(w') \setminus \{e'\})$ and $M(v) \neq M(B(w') \setminus \{e'\})$. Furthermore, if $w' \neq w$, then $B(w) \sqcup \{e'\} = B(w') \sqcup \{e\}$.*

Proof. By Lemma 5.60 we have that w is an ancestor of *low*(u), $M(B(w) \setminus \{e\}) \neq M(w)$, and $M(B(w) \setminus \{e\}) = M(v, c_1)$. Thus, we may consider the greatest ancestor w' of *low*(u) with the property that there is a back-edge $e' \in B(w')$ such that $M(w') \neq M(B(w') \setminus \{e'\}) = M(v, c_1)$. We may assume that $w' \neq w$, because otherwise there is nothing to show. Thus we have $w' > w$. Notice that $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $B(u) \subseteq B(v)$, and therefore the *low*-edge of u is in $B(v)$. This implies that *low*(u) is a proper ancestor of v, and therefore w' is also a proper ancestor of v.

Let us suppose, for the sake of contradiction, that the higher endpoint of e' is a descendant of $M(B(w') \setminus \{e'\})$. Let (x, y) be a back-edge in $B(w')$. If $(x, y) = e'$, then x is a descendant of $M(B(w') \setminus \{e'\})$. Otherwise, if $(x, y) \neq e'$, then we have that $(x, y) \in B(w') \setminus \{e'\}$, and therefore x is a descendant of $M(B(w') \setminus \{e'\})$. Thus, in any case we have that x is a descendant of $M(B(w') \setminus \{e'\})$. Due to the generality of $(x, y) \in B(w')$, this implies that $M(w')$ is a descendant of $M(B(w') \setminus \{e'\})$. But we have $B(w') \setminus \{e'\} \subseteq B(w')$, and therefore $M(B(w') \setminus \{e'\})$ is a descendant of $M(w')$,

and therefore $M(B(w') \setminus \{e'\}) = M(w')$, a contradiction. This shows that the higher endpoint of e' is not a descendant of $M(B(w') \setminus \{e'\})$. Similarly, we can see that the higher endpoint of e is not a descendant of $M(B(w) \setminus \{e\})$.

Since v is a common descendant of w and w', we have that w and w' are related as ancestor and descendant. Thus, $w' > w$ implies that w' is a proper descendant of w. Let (x, y) be a back-edge in $B(w) \setminus \{e\}$. Then x is a descendant of $M(B(w) \setminus \{e\})$, and therefore a descendant of $M(v, c_1)$, and therefore a descendant of v, and therefore a descendant of w'. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of w'. This shows that $(x, y) \in B(w')$. Since x is a descendant of $M(B(w) \setminus \{e\}) = M(B(w') \setminus \{e'\})$, we have that x is not the higher endpoint of e', and therefore $(x, y) \neq e'$. Thus, $(x, y) \in B(w')$ can be strengthened to $(x, y) \in B(w') \setminus \{e'\}$. Due to the generality of $(x, y) \in B(w) \setminus \{e\}$, this implies that $B(w) \setminus \{e\} \subseteq B(w') \setminus \{e'\}$. Conversely, let (x, y) be a back-edge in $B(w') \setminus \{e'\}$. Then x is a descendant of $M(B(w') \setminus \{e'\})$, and therefore a descendant of $M(v, c_1)$, and therefore a descendant of v. Furthermore, y is a proper ancestor of w', and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Then, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w) \setminus \{e\}$. The case $(x, y) \in B(u)$ is rejected, because w' is an ancestor of $low(u)$ (and therefore there is no back-edge in $B(u)$ whose lower endpoint is low enough to be a proper ancestor of w). Thus, we have $(x, y) \in B(w) \setminus \{e\}$. Due to the generality of $(x, y) \in B(w') \setminus \{e'\}$, this implies that $B(w') \setminus \{e'\} \subseteq B(w) \setminus \{e\}$. Thus, since we have showed the reverse inclusion too, we have that $B(w') \setminus \{e'\} = B(w) \setminus \{e\}$. Then, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $B(v) = B(u) \sqcup (B(w') \setminus \{e'\})$.

Let us suppose, for the sake of contradiction, that $e \in B(w')$. Then, since $e \in B(w)$ and $B(w') \setminus \{e'\} = B(w) \setminus \{e\}$, we have that $e = e'$ and $B(w') = B(w)$, which contradicts the fact that the graph is 3-edge-connected. Thus, $e \notin B(w')$. Similarly, we have $e' \notin B(w)$. Thus, $B(w') \setminus \{e'\} = B(w) \setminus \{e\}$ implies that $B(w') \sqcup \{e\} = B(w) \sqcup \{e'\}$. Since w' is an ancestor of $low(u)$, we have $B(u) \cap B(w') = \emptyset$. In particular, this implies that $e' \notin B(u)$. Therefore, $B(v) = B(u) \sqcup (B(w') \setminus \{e'\})$ implies that $e' \notin B(u) \cup B(v)$. $M(B(w') \setminus \{e'\}) = M(v, c_1)$ implies that $M(B(w') \setminus \{e'\})$ is a descendant of c_1, and therefore a proper descendant of $M(v)$. Thus, $M(v) \neq M(B(w') \setminus \{e'\})$. ☐

Now we have collected enough information in order to show how to compute a collection of Type-$3\beta i$ 4-cuts that satisfy (2) of Lemma 5.57, so that all 4-cuts of this type are implied by this collection, plus that returned by Algorithm 24.

Based on Lemma 5.61, it is sufficient to compute all Type-3βi 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$, such that e satisfies (2) of Lemma 5.57, where w is the greatest ancestor of *low*(u) such that $M(w) \neq M(B(w) \setminus \{e\}) = M(v, c_1)$, where c_1 is the *low1* child of $M(v)$. By Lemma 5.60 we have that either $e = e_L(w)$, or $e = e_R(w)$. Thus, we distinguish two cases, depending on whether $e = e_L(w)$ or $e = e_R(w)$. By Lemma 5.60, we have that u is the lowest proper descendant of v such that $M(u) = M(v, c_2)$, where c_2 is the *low2* child of $M(v)$. Thus, given $v \neq r$, we know exactly what are the u and w thay may possibly induce a 4-cut with v. We use the criterion provided by Lemma 5.62 in order to check whether we get a 4-cut from v, u and w. The whole procedure is shown in Algorithm 33. The proof of correctness and linear complexity is given in Proposition 5.21.

Lemma 5.62. *Let $v \neq r$ be a vertex such that $M(v)$ has at least two distinct children c_1 and c_2. Let w be a proper ancestor of v with the property that $M(w) \neq M(v)$ and there is a back-edge e such that $M(B(w) \setminus \{e\})$ is a descendant of c_1, and let u be a proper descendant of v such that $M(u)$ is a descendant of c_2. Suppose that high$(u) < v$, and bcount$(v) =$ bcount$(u) +$ bcount$(w) - 1$. Then, $e \notin B(u) \cup B(v)$ and $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$.*

Proof. Let (x, y) be a back-edge in $B(u)$. Then x is a descendant of u, and therefore a descendant of v. Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Since $(x, y) \in B(u)$, we have that y is an ancestor of *high*(u), and therefore $y \leq$ *high*(u). Thus, *high*$(u) < v$ implies that $y < v$, and therefore y is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(v)$.

Let (x, y) be a back-edge in $B(w) \setminus \{e\}$. Then x is a descendant of $M(B(w) \setminus \{e\})$, and therefore a descendant of c_1, and therefore a descendant of $M(v)$. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(w) \setminus \{e\}$, this implies that $B(w) \setminus \{e\} \subseteq B(v)$.

Let us suppose, for the sake of contradiction, that there is a back-edge $(x, y) \in B(u) \cap (B(w) \setminus \{e\})$. Then x is a descendant of both $M(u)$ and $M(B(w) \setminus \{e\})$, and therefore a descendant of both c_2 and c_1. This implies that c_1 and c_2 are related as ancestor and descendant, which is absurd. Thus, we have that $B(u) \cap (B(w) \setminus \{e\}) = \emptyset$.

Then, since $B(u) \subseteq B(v)$ and $B(w) \setminus \{e\} \subseteq B(v)$ and $B(u) \cap (B(w) \setminus \{e\}) = \emptyset$ and

bcount$(v) = $ *bcount*$(u) + $ *bcount*$(w) - 1$, we have that $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$.

Let us suppose, for the sake of contradiction, that $e \in B(u)$. Let x be the higher endpoint of e. Notice that $M(w) = nca\{x, M(B(w)\setminus\{e\})\}$. Since $e \in B(u)$, we have that x is a descendant of $M(u)$, and therefore a descendant of c_2. Since $M(B(w) \setminus \{e\}) = M(v, c_1)$, we have that $M(B(w) \setminus \{e\})$ is a descendant of c_1. Thus, we have that $nca\{x, M(B(w)\setminus\{e\})\} = M(v)$, in contradiction to the assumption $M(w) \neq M(v)$. This shows that $e \notin B(u)$. Thus, $B(v) = B(u) \sqcup (B(w)\setminus\{e\})$ implies that $e \notin B(u) \cup B(v)$. □

Proposition 5.21. *Algorithm 33 computes a collection $\mathcal{C}$ of Type-3βi 4-cuts that satisfy (2) of Lemma 5.57, and it runs in linear time. Furthermore, every Type-3βi 4-cut that satisfies (2) of Lemma 5.57 is implied by $\mathcal{C} \cup \mathcal{C}'$, where $\mathcal{C}'$ is the collection of Type-2ii 4-cuts returned by Algorithm 24.*

Proof. Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e_L(w)\}$ be a 4-element set that is marked in Line 22. Then we have that *bcount*$(v) = $ *bcount*$(u) + $ *bcount*$(w) - 1$, $M(w) \neq M(v)$ and *high*$(u) < v$. Since $M(u) = M(v, c_2)$ we have that $M(u)$ is a descendant of c_2. Since $w \in W_L(M(v, c_1))$ we have that $M(B(w) \setminus \{e_L(w)\}) = M(v, c_1)$, and therefore $M(B(w) \setminus \{e_L(w)\})$ is a descendant of c_1. We have that u is a proper descendant of v, and therefore *high*$(u) < v$ implies that *high*(u) is a proper ancestor of v. This implies that *low*(u) is also a proper ancestor of v, and therefore w being an ancestor of *low*(u) implies that w is a proper ancestor of v. Thus, all the conditions of Lemma 5.62 are satisfied, and therefore we have that $e_L(w) \notin B(u) \cup B(v)$ and $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$. Since $M(B(w) \setminus \{e_L(w)\})$ is a descendant of c_1, we have that $M(B(w) \setminus \{e_L(w)\}) \neq M(v)$. Thus, we have that C is a Type-3βi 4-cut, that satisfies (2) of Lemma 5.57. So let $\mathcal{C}$ be the collection of all 4-cuts marked in Line 22.

Now let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3βi 4-cut such that w is an ancestor of v, and v is an ancestor of u, and e satisfies (2) of Lemma 5.57. Let c_1 and c_2 be the *low1* and *low2* children of $M(v)$, respectively. Then, Lemma 5.60 implies that w is an ancestor of *low*(u), $M(w) \neq M(B(w)\setminus\{e\}) = M(v, c_1)$, and u is the lowest proper descendant of v such that $M(u) = M(v, c_2)$. Thus, we may consider the greatest ancestor w' of *low*(u) with the property that there is a back-edge $e' \in B(w')$ such that $M(w') \neq M(B(w') \setminus \{e'\}) = M(v, c_1)$. Since $M(w') \neq M(B(w') \setminus \{e'\})$, Lemma 3.9 implies that either $e' = e_L(w')$ or $e' = e_R(w')$. Let us assume that $e' = e_L(w')$. (The other case is treated similarly.) Then, Lemma 5.61 implies that $e_L(w') \notin B(u) \cup B(v)$ and $B(v) = B(u) \sqcup (B(w') \setminus \{e_L(w')\})$. Thus, $C' = \{(u, p(u)), (v, p(v)), (w', p(w')), e_L(w')\}$ is a

Algorithm 33: Compute a collection of Type-3βi 4-cuts that satisfy (2) of Lemma 5.57, so that all of them are implied by this collection, plus that returned by Algorithm 24

```
   // We deal with the case that the back-edge of the 4-cut is e_L(w); the
      other case is treated similarly
1  foreach vertex w ≠ r do
2  |  compute M(B(w) \ {e_L(w)})
3  end
4  foreach vertex x do
5  |  let W_L(x) be the collection of all w ≠ r such that
   |    M(w) ≠ M(B(w) \ {e_L(w)}) = x
6  end
7  foreach vertex v ≠ r do
8  |  if M(v) has at least two children then
9  |  |  let c_1 be the low1 child of M(v)
10 |  |  let c_2 be the low2 child of M(v)
11 |  |  compute M(v, c_1) and M(v, c_2)
12 |  end
13 end
14 foreach vertex v ≠ r do
15 |  if M(v) has less than two children then continue
16 |  let c_1 be the low1 child of M(v)
17 |  let c_2 be the low2 child of M(v)
18 |  if low(c_1) ≥ v or low(c_2) ≥ v then continue
19 |  let u be the lowest proper descendant of v such that M(u) = M(v, c_2)
20 |  let w be the greatest ancestor of low(u) such that w ∈ W_L(M(v, c_1))
21 |  if M(w) ≠ M(v) and high(u) < v and bcount(v) = bcount(u) + bcount(w) − 1
   |    then
22 |  |  mark {(u, p(u)), (v, p(v)), (w, p(w)), e_L(w)} as a 4-cut
23 |  end
24 end
```

4-cut that satisfies (2) of Lemma 5.57. Furthermore, since $M(B(w') \setminus \{e'\}) = M(v, c_1)$, we have that $M(B(w') \setminus \{e'\})$ is a descendant of c_1, and therefore $M(B(w') \setminus \{e'\}) \neq M(v)$. Thus, C' is a Type-$3\beta i$ 4-cut. Then, Lemma 5.60 implies that $M(w') \neq M(v)$. Since, $e_L(w') \notin B(u) \cup B(v)$ and $B(v) = B(u) \sqcup (B(w') \setminus \{e_L(w')\})$, we have $\mathit{bcount}(v) = \mathit{bcount}(u) + \mathit{bcount}(w) - 1$. Furthermore, we have $B(u) \subseteq B(v)$, and therefore $\mathit{high}(u) < v$ (because the lower endpoints of all back-edges in $B(u)$ are proper ancestors of v). Thus, all the conditions are satisfied for C' to be marked in Line 22, and therefore we have $C \in \mathcal{C}$. Now, if $C' = C$, then there is nothing to show. Otherwise, we have $w' \neq w$, and therefore Lemma 5.61 implies that $B(w) \sqcup \{e_L(w')\} = B(w') \sqcup \{e\}$. Thus, Lemma 5.28 implies that $C'' = \{(w, p(w)), (w', p(w')), e, e_L(w')\}$ is a Type-$2ii$ 4-cut. Notice that C is implied by C' and C'' through the pair of edges $\{(w, p(w)), e\}$. By Proposition 5.12 we have that C'' is implied by $\mathcal{C}'$ through the pair of edges $\{(w, p(w)), e\}$, where $\mathcal{C}'$ is the collection of Type-$2ii$ 4-cuts computed by Algorithm 24. Therefore, by Lemma 5.7 we have that C is implied by $\mathcal{C}' \cup \{C'\}$. This shows that $\mathcal{C} \cup \mathcal{C}'$ implies all Type-$3\beta i$ 4-cuts that satisfy (2) of Lemma 5.57.

Now we will argue about the complexity of Algorithm 33. By Proposition 3.6 we have that the values $M(B(w) \setminus \{e_L(w)\})$ can be computed in linear time in total, for all vertices $w \neq r$. Thus, the **for** loop in Line 1 can be performed in linear time. By Proposition 3.5 we have that the values $M(v, c_1)$ and $M(v, c_2)$ can be computed in linear time in total, for all vertices $v \neq r$ such that $M(v)$ has at last two children, where c_1 and c_2 are the *low1* and *low2* children of $M(v)$. Thus, the **for** loop in Line 7 can be performed in linear time. In order to compute the vertex u in Line 19 we use Algorithm 22. Specifically, whenever we reach Line 19, we generate a query $q(M^{-1}(M(v, c_2)), v)$. This will return the lowest vertex u with $M(u) = M(v, c_2)$ such that $u > v$. Since $M(u) = M(v, c_2)$, we have that $M(u)$ is a common descendant of v and u, and therefore v and u are related as ancestor and descendant. Thus, $u > v$ implies that u is a proper descendant of v, and therefore we have that u is the greatest proper descendant of v such that $M(u) = M(v, c_2)$. Similarly, in order to compute the vertex v in Line 20 we use Algorithm 22. Specifically, whenever we reach Line 20, we generate a query $q'(W_L(M(v, c_1)), \mathit{low}(u))$. This is to return the greatest w in $W_L(M(v, c_1))$ such that $w \leq \mathit{low}(u)$. Since $w \in W_L(M(v, c_1))$, we have that $M(B(w) \setminus \{e_L(w)\}) = M(v, c_1)$. This implies that $M(v, c_1)$ is a common descendant of v and w, and therefore v and w are related as ancestor and descendant. Notice that, since we have reached Line 20, we have that u in Line 19 satisfies $M(u) = M(v, c_2)$.

This implies that $low(u) < v$. To see this, let (x, y) be a back-edge in $B(v)$ such that x is a descendant of c_2. (Such a back-edge exists, because $low(c_2) < v$, since we have passed the check in Line 18.) Then x is a descendant of $M(v, c_2)$, and therefore a descendant of $M(u)$. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Therefore, since $low(u) \leq y$ and y is a proper ancestor of v, we have $low(u) < v$. Furthermore, we have that $low(u)$ is an ancestor of y, and therefore an ancestor of v. Then, since v is a common descendant of $low(u)$ and w, we have that $low(u)$ and w are related as ancestor and descendant, and then $w \leq low(u)$ implies that w is an ancestor of $low(u)$. Thus, w is the greatest ancestor of $low(u)$ such that $w \in W_L(M(v, c_1))$. Now, since the total number of all q' queries is $O(n)$, and since the sets W_L are pairwise disjoint, by Lemma 5.27 we have that Algorithm 22 can answer all those queries in linear time in total. We conclude that Algorithm 33 runs in linear time. □

5.8.1.3 Case (3) of Lemma 5.57

Lemma 5.63. *Let u, v, w be three vertices $\neq r$ such that w is proper ancestor of v, v is a proper ancestor of u, and there is a back-edge $e \in B(u)$ such that $e \notin B(v) \cup B(w)$, $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$, and $M(v) \neq M(w)$. Then $e = (highD(u), high(u))$ and $high_2(u) < v$. Furthermore, let c_1 and c_2 be the low1 and low2 child of $M(v)$, respectively. Then, $M(B(u) \setminus \{e_{high}(u)\}) = M(v, c_2)$, and w is the greatest proper ancestor of v such that $M(w) = M(v, c_1)$.*

Proof. $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ and $e \notin B(v)$ imply that all back-edges in $B(u)$, except e, are in $B(v)$. Let us suppose, for the sake of contradiction, that $e_{high}(u) \in B(v)$. This implies that $high(u)$ is a proper ancestor of v. Let (x, y) be a back-edge in $B(u)$. Then x is a descendant of u, and therefore a descendant of v. Furthermore, y is an ancestor of $high(u)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(v)$, in contradiction to the fact that $e \in B(u) \setminus B(v)$. This shows that $e_{high}(u) \notin B(v)$, and therefore $e = e_{high}(u)$. Since $B(u) \setminus \{e\} \subseteq B(v)$, we have that the *high2* back-edge of $B(u)$ is in $B(v)$, and therefore $high_2(u)$ is a proper ancestor of v, and therefore $high_2(u) < v$.

$B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that $B(w) \subseteq B(v)$, and therefore $M(w)$ is a descendant of $M(v)$. Thus, since $M(v) \neq M(w)$, we have that $M(w)$ is a proper descendant of $M(v)$. Let c be the child of $M(v)$ that is an ancestor of $M(w)$. Let (x, y)

be a back-edge in $B(w)$. Then x is a descendant of $M(w)$, and therefore a descendant of c, and therefore a descendant of $M(v)$. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of $M(v)$, and therefore a proper ancestor of c. This shows that $(x,y) \in B(c)$. Since y is a proper ancestor of w, we have $y < w$. Thus, since $(x,y) \in B(c)$, we have $low(c) \leq y < w$.

Let us suppose, for the sake of contradiction, that $c \neq c_1$. Since $low(c_1) \leq low(c)$, $low(c) < w$ implies that $low(c_1) < w$. Thus, there is a back-edge $(x,y) \in B(c_1)$ such that $y < w$. Then, x is a descendant of c_1, and therefore a descendant of $M(v)$, and therefore a descendant of v, and therefore a descendant of w. Since (x,y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of w and y, and therefore w and y are related as ancestor and descendant. Then, since $y < w$, we have that y is a proper ancestor of w. This shows that $(x,y) \in B(w)$. This implies that x is a descendant of $M(w)$, and therefore x is a descendant of c. Thus, x is a common descendant of c and c_1, and therefore c and c_1 are related as ancestor and descendant, which is absurd. Thus, we have that $c = c_1$.

$B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that $B(u) \setminus \{e\} \subseteq B(v)$, and therefore $M(B(u) \setminus \{e\})$ is a descendant of $M(v)$. Let us suppose, for the sake of contradiction, that $M(B(u) \setminus \{e\}) = M(v)$. Let (x,y) be a back-edge in $B(w)$. Then $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that $(x,y) \in B(v)$. Therefore, x is a descendant of $M(v)$, and therefore a descendant of $M(B(u) \setminus \{e\})$, and therefore a descendant of u. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x,y) \in B(u)$. Since $(x,y) \in B(w)$ and $e \notin B(w)$, we have that $(x,y) \neq e$. Therefore, $(x,y) \in B(u)$ can be strengthened to $(x,y) \in B(u) \setminus \{e\}$. But then we have a contradiction to (the disjointness of the union in) $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$. This shows that $M(B(u) \setminus \{e\}) \neq M(v)$, and therefore $M(B(u) \setminus \{e\})$ is a proper descendant of $M(v)$. So let c' be the child of $M(v)$ that is an ancestor of $M(B(u) \setminus \{e\})$.

Let (x,y) be a back-edge in $B(u) \setminus \{e\}$. Then x is a descendant of $M(B(u) \setminus \{e\})$, and therefore a descendant of c'. Furthermore, $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that $(x,y) \in B(v)$, and therefore y is a proper ancestor of v, and therefore a proper ancestor of $M(v)$, and therefore a proper ancestor of c'. This shows that $(x,y) \in B(c')$. Since y is a proper ancestor of v, we have $y < v$. Thus, since $(x,y) \in B(c')$, we have $low(c') \leq y < v$.

Let us suppose, for the sake of contradiction, that there is a back-edge of the form $(M(v), z)$ in $B(v)$. Then $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that either $(M(v), z) \in$

$B(u) \setminus \{e\}$, or $(M(v), z) \in B(w)$. The first case implies that $M(v)$ is a descendant of $M(B(u) \setminus \{e\})$, and therefore a descendant of c', which is a absurd. The second case implies that $M(v)$ is a descendant of $M(w)$, and therefore a descendant of c_1, which is also absurd. This shows that there is no back-edge of the form $(M(v), z)$ in $B(v)$. Thus, Lemma 3.13 implies that $low(c_2) < v$.

Let us suppose, for the sake of contradiction, that $c' \neq c_2$. Since $low(c_2) < v$, there is a back-edge $(x, y) \in B(c_2)$ such that $y < v$. Then, x is a descendant of c_2, and therefore a descendant of $M(v)$, and therefore a descendant of v. Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, since x is a common descendant of both v and y, we have that v and y are related as ancestor and descendant. Thus, $y < v$ implies that y is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Then, $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that either $(x, y) \in B(u) \setminus \{e\}$, or $(x, y) \in B(w)$. The first case implies that x is a descendant of $M(B(u) \setminus \{e\})$, and therefore a descendant of c', which is absurd, since x is a descendant of c_2 (and c', c_2 are not related as ancestor and descendant, since they have the same parent and $c' \neq c_2$). The second case implies that x is a descendant of $M(w)$, and therefore a descendant of c_1, which is absurd, since x is a descendant of c_2 (and c_1, c_2 are not related as ancestor and descendant, since they have the same parent). There are no viable options left, and so we have arrived at a contradiction. This shows that $c' = c_2$.

Let $S_1 = \{(x, y) \in B(v) \mid x \text{ is a descendant of } c_1\}$. Then, $M(S_1) = M(v, c_1)$. Let (x, y) be a back-edge in S_1. Then, $(x, y) \in B(v)$, and therefore $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that either $(x, y) \in B(u) \setminus \{e\}$, or $(x, y) \in B(w)$. Let us suppose, for the sake of contradiction, that $(x, y) \in B(u) \setminus \{e\}$. Then x is a descendant of $M(B(u) \setminus \{e\})$, and therefore a descendant of c_2. Thus, x is a common descendant of c_1 and c_2, and so c_1 and c_2 are related as ancestor and descendant, which is absurd. Thus, the case $(x, y) \in B(u) \setminus \{e\}$ is rejected, and so we have $(x, y) \in B(w)$. Due to the generality of $(x, y) \in S_1$, this implies that $S_1 \subseteq B(w)$. Conversely, let (x, y) be a back-edge in $B(w)$. Then, x is a descendant of $M(w)$, and therefore a descendant of c_1, and therefore a descendant of $M(v)$. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$, and so we have $(x, y) \in S_1$. Due to the generality of $(x, y) \in B(w)$, this implies that $B(w) \subseteq S_1$. Thus, since $S_1 \subseteq B(w)$, we have $S_1 = B(w)$, and therefore $M(S_1) = M(w)$, and therefore $M(v, c_1) = M(w)$.

Let $S_2 = \{(x, y) \in B(v) \mid x \text{ is a descendant of } c_2\}$. Then, $M(S_2) = M(v, c_2)$. Let (x, y) be a back-edge in S_2. Then, $(x, y) \in B(v)$, and therefore $B(v) = (B(u) \setminus \{e\}) \sqcup$

$B(w)$ implies that either $(x, y) \in B(u) \setminus \{e\}$, or $(x, y) \in B(w)$. Let us suppose, for the sake of contradiction, that $(x, y) \in B(w)$. Then x is a descendant of $M(w)$, and therefore a descendant of c_1. Thus, x is a common descendant of c_1 and c_2, and so c_1 and c_2 are related as ancestor and descendant, which is absurd. Thus, the case $(x, y) \in B(w)$ is rejected, and so we have $(x, y) \in B(u) \setminus \{e\}$. Due to the generality of $(x, y) \in S_2$, this implies that $S_2 \subseteq B(u) \setminus \{e\}$. Conversely, let (x, y) be a back-edge in $B(u) \setminus \{e\}$. Then, x is a descendant of $M(B(u) \setminus \{e\})$, and therefore a descendant of c_2. Furthermore, since $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$, we have that $(x, y) \in B(v)$. This shows that $(x, y) \in S_2$. Due to the generality of $(x, y) \in B(u) \setminus \{e\}$, this implies that $B(u) \setminus \{e\} \subseteq S_2$. Thus, since $S_2 \subseteq B(u) \setminus \{e\}$, we have $S_2 = B(u) \setminus \{e\}$, and therefore $M(S_2) = M(B(u) \setminus \{e\})$, and therefore $M(v, c_2) = M(B(u) \setminus \{e\})$.

Let us suppose, for the sake of contradiction, that there is a proper ancestor w' of v that is greater than w and has $M(w') = M(v, c_1)$. Since $M(w') = M(w)$ and $w' > w$, we have that w' is a proper descendant of w, and Lemma 3.2 implies that $B(w) \subseteq B(w')$. Since the graph is 3-edge-connected, this can be strengthend to $B(w) \subset B(w')$. This implies that there is a back-edge $(x, y) \in B(w') \setminus B(w)$. Then, x is a descendant of $M(w')$, and therefore a descendant of $M(v, c_1)$, and therefore a descendant of v. Furthermore, y is a proper ancestor of w', and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Then, $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that either $(x, y) \in B(u) \setminus \{e\}$, or $(x, y) \in B(w)$. The case $(x, y) \in B(w)$ is rejected (since $(x, y) \in B(w') \setminus B(w)$), and therefore $(x, y) \in B(u) \setminus \{e\}$. This implies that x is a descendant of $M(B(u) \setminus \{e\}) = M(v, c_2)$, and therefore a descendant of c_2. But then we have that x is a descendant of both c_2 and c_1, and therefore c_1 and c_2 are related as ancestor and descendant, which is absurd. This shows that w is the greatest proper ancestor of v that has $M(w) = M(v, c_1)$. $\square$

We distinguish two cases for the Type-$3\beta i$ 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$, where u is a descendant of v, v is a descendant of w, and e satisfies (3) of Lemma 5.57: either $M(u) = M(B(u) \setminus \{e\})$, or $M(u) \neq M(B(u) \setminus \{e\})$. In the first case, we can compute all such 4-cuts explicitly. In the second case, we compute only a subcollection $\mathcal{C}$ of them, so that the rest are implied by $\mathcal{C} \cup \mathcal{C}'$, where $\mathcal{C}'$ is the collection of Type-$2ii$ 4-cuts that are computed by Algorithm 24. Our result is summarized and proved in Proposition 5.22.

The reason that the 4-cuts in the first case can be computed explicitly is the

following.

Lemma 5.64. *Let u, v, w be three vertices $\neq r$ such that u is a proper descendant of v, v is a proper descendant of w, and $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$, where e is a back-edge in $B(u)$ such that $e \notin B(v)$. Suppose that $M(u) = M(B(u) \setminus \{e\})$. Then u is either the lowest or the second-lowest proper descendant of v in $M^{-1}(M(u))$.*

Proof. Let us suppose, for the sake of contradiction, that u is neither the lowest nor the second-lowest proper descendant of v in $M^{-1}(M(u))$. This means that there are two proper descendants u' and u'' of v, such that $u > u' > u''$ and $M(u') = M(u'') = M(u)$. Since $M(u) = M(u') = M(u'')$, we have that u, u' and u'' are related as ancestor and descendant, and therefore $u > u' > u''$ implies that u is a proper descendant of u', and u' is a proper descendant of u''. Then, Lemma 3.2 implies that $B(u'') \subseteq B(u') \subseteq B(u)$. Since the graph is 3-edge-connected, this can be strengthened to $B(u'') \subset B(u') \subset B(u)$. Thus, there is a back-edge $(x, y) \in B(u) \setminus B(u')$, and a back-edge $(x', y') \in B(u') \setminus B(u'')$. Since $(x, y) \in B(u)$, we have that x is a descendant of $M(u)$, and therefore a descendant of u'. Thus, since $(x, y) \notin B(u')$, we have that y is not a proper ancestor of u'. This implies that y is not a proper ancestor of v either, and therefore $(x, y) \notin B(v)$. Similarly, since $(x', y') \in B(u') \setminus B(u'')$, we have that y' is not a proper ancestor of u'', and therefore not a proper ancestor of v, and therefore $(x', y') \notin B(v)$. Notice that both (x, y) and (x', y') are in $B(u)$ (since $B(u'') \subset B(u') \subset B(u)$). Furthermore, since $(x', y') \in B(u')$ and $(x, y) \notin B(u')$, we have that $(x, y) \neq (x', y')$. Thus, (x, y) and (x', y') are two distinct back-edges in $B(u)$ that are not in $B(v)$. But this contradicts the fact $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$, which implies that e is the only back-edge from $B(u)$ that is not in $B(v)$. This shows that u is either the lowest or the second-lowest proper descendant of v in $M^{-1}(M(u))$. $\square$

In the case where $M(u) \neq M(B(u) \setminus \{e\})$, we consider the lowest proper descendant u' of v that satisfies $M(u') \neq M(B(u') \setminus \{e_{high}(u')\}) = M(B(u) \setminus \{e\})$. Then, the following shows that we still get a Type-$3\beta i$ 4-cut with v and w.

Lemma 5.65. *Let u, v, w be three vertices $\neq r$ such that w is a proper ancestor of v, v is a proper ancestor of u, and there is a back-edge $e \in B(u)$ such that $e \notin B(v) \cup B(w)$ and $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$. Suppose that $M(u) \neq M(B(u) \setminus \{e\})$. Let u' be the lowest proper descendant of v such that $M(u') \neq M(B(u') \setminus \{e_{high}(u')\}) = M(B(u) \setminus \{e\})$. Then we have $e_{high}(u') \notin B(v) \cup B(w)$ and $B(v) = (B(u') \setminus \{e_{high}(u')\}) \sqcup B(w)$. Furthermore, if $u' \neq u$, then $B(u) \sqcup \{e_{high}(u')\} = B(u') \sqcup \{e\}$.*

Proof. By the proof of Lemma 5.63, we have $e = e_{high}(u)$. (This result does not rely on the supposition $M(w) \neq M(v)$, which is included in the statement of Lemma 5.63.) Thus, it makes sense to consider the lowest proper descendant u' of v that satisfies $M(u') \neq M(B(u') \setminus \{e_{high}(u')\}) = M(B(u) \setminus \{e\})$. If $u' = u$, then there is nothing to show. So let us assume that $u' \neq u$. Then, due to the minimality of u', we have $u' < u$. Since $M(B(u') \setminus \{e_{high}(u')\}) = M(B(u) \setminus \{e\})$, we have that $M(B(u) \setminus \{e\})$ is a common descendant of u and u'. Therefore, u and u' are related as ancestor and descendant, and then $u' < u$ implies that u' is a proper ancestor of u.

Since $M(u') \neq M(B(u') \setminus \{e_{high}(u')\})$, we have that $e_{high}(u')$ is the only back-edge in $B(u')$ whose higher endpoint is not a descendant of $M(B(u') \setminus \{e_{high}(u')\})$. Similarly, since $M(u) \neq M(B(u) \setminus \{e\})$, we have that e is the only back-edge in $B(u)$ whose higher endpoint is not a descendant of $M(B(u) \setminus \{e\})$.

Let (x, y) be a back-edge in $B(u') \setminus \{e_{high}(u')\}$. Then x is a descendant of $M(B(u') \setminus \{e_{high}(u')\})$, and therefore a descendant of $M(B(u) \setminus \{e\})$, and therefore a descendant of $M(u)$. Furthermore, y is a proper ancestor of u', and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Due to the generality of $(x, y) \in B(u') \setminus \{e_{high}(u')\}$, this implies that $B(u') \setminus \{e_{high}(u')\} \subseteq B(u)$. Since the higher endpoint of e is not a descendant of $M(B(u) \setminus \{e\}) = M(B(u') \setminus \{e_{high}(u')\})$, we have that $e \notin B(u') \setminus \{e_{high}(u')\}$. Thus, $B(u') \setminus \{e_{high}(u')\} \subseteq B(u)$ can be strengthened to $B(u') \setminus \{e_{high}(u')\} \subseteq B(u) \setminus \{e\}$. Conversely, let (x, y) be a back-edge in $B(u) \setminus \{e\}$. Then x is a descendant of $M(B(u) \setminus \{e\})$, and therefore a descendant of $M(B(u') \setminus \{e_{high}(u')\})$, and therefore a descendant of $M(u')$. Furthermore, $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that $(x, y) \in B(v)$, and therefore y is a proper ancestor of v, and therefore a proper ancestor of u'. This shows that $(x, y) \in B(u')$. Due to the generality of $(x, y) \in B(u) \setminus \{e\}$, this implies that $B(u) \setminus \{e\} \subseteq B(u')$. Since the higher endpoint of $e_{high}(u')$ is not a descendant of $M(B(u') \setminus \{e_{high}(u')\}) = M(B(u) \setminus \{e\})$, we have that $e_{high}(u') \notin B(u) \setminus \{e\}$. Thus, $B(u) \setminus \{e\} \subseteq B(u')$ can be strengthened to $B(u) \setminus \{e\} \subseteq B(u') \setminus \{e_{high}(u')\}$.

Thus, we have shown that $B(u) \setminus \{e\} = B(u') \setminus \{e_{high}(u')\}$. Then, notice that we cannot have $e \in B(u')$ or $e_{high}(u') \in B(u)$, because otherwise we get $B(u) = B(u')$, in contradiction to the fact that the graph is 3-edge-connected. Thus, $B(u) \setminus \{e\} = B(u') \setminus \{e_{high}(u')\}$ implies that $B(u) \sqcup \{e_{high}(u')\} = B(u') \sqcup \{e\}$. Furthermore, since $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ and $B(u) \setminus \{e\} = B(u') \setminus \{e_{high}(u')\}$, we infer that $B(v) = (B(u') \setminus \{e_{high}(u')\}) \sqcup B(w)$.

Since the graph is 3-edge-connected, we have $|(B(u')| > 1$. Thus, there is a back-

edge $(x, y) \in B(u') \setminus \{e_{high}(u')\}$. Then we have that x is a descendant of u', and therefore a descendant of v, and therefore a descendant of w. Thus, since $(B(u') \setminus \{e_{high}(u')\}) \cap B(w) = \emptyset$, we have that y is not a proper ancestor of w. Since (x, y) is a back-edge, x is a descendant of y. Thus, x is a common descendant of y and w, and therefore y and w are related as ancestor and descendant. Thus, since y is not a proper ancestor of w, we have that y is a descendant of w, and therefore $y \geq w$. Since $(x, y) \in B(u')$, we have that $high_1(u') \geq y$, and therefore $high_1(u') \geq w$. This implies that $e_{high}(u') \notin B(w)$ (because the lower endpoint of $e_{high}(u')$ is not low enough to be a proper ancestor of w). Then, $B(v) = (B(u') \setminus \{e_{high}(u')\}) \sqcup B(w)$ implies that $e_{high}(u') \notin B(v) \cup B(w)$. $\square$

Lemma 5.66. *Let $v \neq r$ be a vertex such that $M(v)$ has at least two children. Let u be a proper descendant of v such that $M(B(u) \setminus \{e_{high}(u)\})$ is a proper descendant of $M(v)$, and let w be a proper ancestor of v such that $M(w)$ is a proper descendant of $M(v)$. Suppose that $M(u)$ and $M(w)$ are not related as ancestor and descendant, $high_2(u) < v$, and $bcount(u) = bcount(v) - bcount(w) + 1$. Then, $e_{high}(u) \notin B(v) \cup B(w)$ and $B(v) = (B(u) \setminus \{e_{high}(u)\}) \sqcup B(w)$.*

Proof. Let (x, y) be a back-edge in $B(u) \setminus \{e_{high}(u)\}$. Then, x is a descendant of $M(B(u) \setminus \{e_{high}(u)\})$, and therefore a descendant of $M(v)$, and therefore a descendant of v. Since (x, y) is a back-edge, x is a descendant of y. Thus, x is a common descendant of y and v, and therefore y and v are related as ancestor and descendant. Since $(x, y) \in B(u) \setminus \{e_{high}(u)\}$, we have that y is an ancestor of $high_2(u)$, and therefore $y \leq high_2(u)$. Thus, since $high_2(u) < v$, we have that y is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u) \setminus \{e_{high}(u)\}$, this implies that $B(u) \setminus \{e_{high}(u)\} \subseteq B(v)$.

Let (x, y) be a back-edge in $B(w)$. Then, x is a descendant of $M(w)$, and therefore a descendant of $M(v)$. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(w)$, this implies that $B(w) \subseteq B(v)$.

Let us suppose, for the sake of contradiction, that $B(u) \cap B(w) \neq \emptyset$. Then there is a back-edge $(x, y) \in B(u) \cap B(w)$. This implies that x is a descendant of both $M(u)$ and $M(w)$, and therefore $M(u)$ and $M(w)$ are related as ancestor and descendant. A contradiction. Thus, we have that $B(u) \cap B(w) = \emptyset$.

Now, since $B(u) \setminus \{e_{high}(u)\} \subseteq B(v)$, $B(w) \subseteq B(v)$ and $B(u) \cap B(w) = \emptyset$, we have that $(B(u) \setminus \{e_{high}(u)\}) \sqcup B(w) \subseteq B(v)$. Then, $bcount(u) = bcount(v) - bcount(w) + 1$

implies that $(B(u) \setminus \{e_{high}(u)\}) \sqcup B(w) = B(v)$. Since $B(u) \cap B(w) = \emptyset$, we have that $e_{high}(u) \notin B(w)$. Thus, $(B(u) \setminus \{e_{high}(u)\}) \sqcup B(w) = B(v)$ implies that $e_{high}(u) \notin B(v)$, and so we have $e_{high}(u) \notin B(v) \cup B(w)$. □

Proposition 5.22. *Algorithm 34 computes a collection $\mathcal{C}$ of Type-3βi 4-cuts that satisfy* (3) *of Lemma 5.57, and it runs in linear time. Furthermore, every Type-3βi 4-cut that satisfies* (3) *of Lemma 5.57 is implied by $\mathcal{C} \cup \mathcal{C}'$, where $\mathcal{C}'$ is the collection of Type-2ii 4-cuts returned by Algorithm 24.*

Proof. Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e_{high}(u)\}$ be a 4-element set that is marked in Lines 18, or 22, or 26. Then we have that $bcount(v) = bcount(u) + bcount(w) - 1$ and $high_2(u) < v$. Furthermore, w is a proper ancestor of v such that $M(w)$ is a proper descendant of c_1 (since $M(w) = M(v, c_1)$). In Lines 15 and 20, it is clear that u is a proper descendant of v such that $M(B(u) \setminus \{e_{high}(u)\})$ is a proper descendant of c_2. Then, in Line 24, we still have that $M(B(u) \setminus \{e_{high}(u)\})$ is a proper descendant of $M(v)$, and now u is even greater than previously. Furthermore, since $M(u) = M(B(u) \setminus \{e_{high}(u)\})$ and $M(B(u) \setminus \{e_{high}(u)\}) = M(v, c_2)$, we have that u is an ancestor of $M(v, c_2)$, and therefore u is still a proper descendant of v. Thus, all the conditions of Lemma 5.66 are satisfied, regardless of whether C is marked in Line 18, or 22, or 26, and therefore we have that $e_{high}(u) \notin B(v) \cup B(w)$ an $B(v) = (B(u) \setminus \{e_{high}(u)\}) \sqcup B(w)$. Thus, C is indeed a 4-cut that satisfies (3) of Lemma 5.57. Furthermore, since $M(w) = M(v, c_1)$, we have that $M(w)$ is a proper descendant of $M(v)$, and therefore $M(w) \neq M(v)$. This implies that C is a Type-3αi 4-cut. Thus, the collection $\mathcal{C}$ of all 4-element sets that are marked in Lines 18 or 22 is a collection of Type-3βi 4-cuts that satisfy (3) of Lemma 5.57.

Now let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3βi 4-cut, where w is a proper ancestor of v, and v is a proper ancestor of u, and e satisfies (3) of Lemma 5.57. Let c_1 and c_2 be the *low1* and *low2* children of $M(v)$, respectively. Then Lemma 5.63 implies that $e = e_{high}(u)$, $high_2(u) < v$, $M(B(u) \setminus \{e\}) = M(v, c_2)$, and w is the greatest proper ancestor of v that has $M(w) = M(v, c_1)$. Furthermore, since C satisfies (3) of Lemma 5.57, we have that $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$, which implies that $bcount(v) = bcount(u) + bcount(w) - 1$.

First, suppose that $M(u) = M(B(u) \setminus \{e_{high}(u)\})$. Then, by Lemma 5.64 we have that u is either the lowest or the second-lowest proper descendant of v such that $M(u) =$

Algorithm 34: Compute a collection of Type-3βi 4-cuts that satisfy (3) of Lemma 5.57, so that all of them are implied by this collection, plus that returned by Algorithm 24

```
1  foreach vertex u ≠ r do
2      compute M(B(u) \ {e_high(u)})
3  end
4  foreach vertex v ≠ r do
5      if M(v) has at least two children then
6          let c_1 be the low1 child of M(v)
7          let c_2 be the low2 child of M(v)
8          compute M(v, c_1) and M(v, c_2)
9      end
10 end
11 foreach vertex v ≠ r do
12     if M(v) has less than two children then continue
13     let c_1 be the low1 child of M(v)
14     let c_2 be the low2 child of M(v)
15     let u be the lowest proper descendant of v that has
         M(u) ≠ M(B(u) \ {e_high(u)}) = M(v, c_2)
16     let w be the greatest proper ancestor of v that has M(w) = M(v, c_1)
17     if high_2(u) < v and bcount(v) = bcount(u) + bcount(w) − 1 then
18         mark {(u, p(u)), (v, p(v)), (w, p(w)), e_high(u)} as a 4-cut
19     end
20     let u be the lowest proper descendant of v that has
         M(u) = M(B(u) \ {e_high(u)}) = M(v, c_2)
21     if high_2(u) < v and bcount(v) = bcount(u) + bcount(w) − 1 then
22         mark {(u, p(u)), (v, p(v)), (w, p(w)), e_high(u)} as a 4-cut
23     end
24     u ← prevM(u)
25     if high_2(u) < v and bcount(v) = bcount(u) + bcount(w) − 1 then
26         mark {(u, p(u)), (v, p(v)), (w, p(w)), e_high(u)} as a 4-cut
27     end
28 end
```

$M(B(u) \setminus \{e_{high}(u)\}) = M(v, c_2)$. Thus, notice that C will be marked in Line 22 or 26, respectively, and therefore $C \in \mathcal{C}$. So let us assume that $M(u) \neq M(B(u) \setminus \{e_{high}(u)\})$.

Now, it makes sense to consider the lowest proper descendant u' of v that has $M(u') \neq M(B(u') \setminus \{e_{high}(u')\}) = M(v, c_2)$. If $u' = u$, then notice that C satisfies enough conditions to be marked in Line 18, and therefore $C \in \mathcal{C}$. Otherwise, Lemma 5.65 implies that $e_{high}(u') \notin B(v) \cup B(w)$ and $B(v) = (B(u') \setminus \{e_{high}(u')\}) \sqcup B(w)$. Therefore, $C' = \{(u', p(u')), (v, p(v)), (w, p(w)), e_{high}(u')\}$ is a 4-cut that satisfies (3) of Lemma 5.57. Furthermore, since $M(w) = M(v, c_1)$, we have that $M(w) \neq M(v)$, and therefore C' is a Type-$3\beta i$ 4-cut. Then, since u' is the lowest proper descendant of v that has $M(u') \neq M(B(u') \setminus \{e_{high}(u')\}) = M(v, c_2)$, we have that C' will be marked in Line 18, and therefore $C' \in \mathcal{C}$. Furthermore, Lemma 5.65 implies that $B(u) \sqcup \{e_{high}(u')\} = B(u') \sqcup \{e_{high}(u)\}$, and therefore Lemma 5.28 implies that $C'' = \{(u, p(u)), (u', p(u')), e_{high}(u), e_{high}(u')\}$ is a Type-$2ii$ 4-cut. Observe that C is implied by C' and C'' through the pair of edges $\{(u, p(u)), e_{high}(u)\}$. According to Proposition 5.12, we have that C'' is implied by $\mathcal{C}'$ through the pair of edges $\{(u, p(u)), e_{high}(u)\}$, where $\mathcal{C}'$ is the collection of Type-$2ii$ 4-cuts computed by Algorithm 24. Then, by Lemma 5.7 we have that C is implied by $\mathcal{C}' \cup \{C'\}$. Thus, we have shown that all Type-$3\beta i$ 4-cuts that satisfy (3) of Lemma 5.57 are implied by $\mathcal{C} \cup \mathcal{C}'$.

Now we will argue about the complexity of Algorithm 34. By Proposition 3.6 we have that the values $M(B(u) \setminus \{e_{high}(u)\})$ can be computed in linear time in total, for all vertices $u \neq r$. Thus, the **for** loop in Line 1 can be performed in linear time. By Proposition 3.5 we have that the values $M(v, c_1)$ and $M(v, c_2)$ can be computed in linear time in total, for all vertices $v \neq r$ such that $M(v)$ has at least two children, where c_1 and c_2 are the *low1* and *low2* children of $M(v)$. Thus, the **for** loop in Line 4 can be performed in linear time. The vertices u and w in Lines 15, 20 and 16 can be computed with the use of Algorithm 22. Specifically, for every vertex x, we compute the collection $U(x)$ of all vertices $u \neq r$ such that $M(u) \neq M(B(u) \setminus \{e_{high}(u)\}) = x$. Then, the U sets are pairwise disjoint, and we can compute them easily in $O(n)$ time, once we have computed all $M(B(u) \setminus \{e_{high}(u)\})$ values. Now, when we reach Line 15, we generate a query $q(U(M(v, c_2)), v)$. This is to return the lowest vertex u in $U(M(v, c_2))$ that has $u > v$. Since $u \in U(M(v, c_2))$, we have that $M(B(u) \setminus \{e_{high}(u)\}) = M(v, c_2)$. This implies that $M(v, c_2)$ is a common descendant of u and v, and therefore u and v are related as ancestor and descendant. Thus, $u > v$ implies that u is a proper descendant of v, and therefore we have that u is the lowest proper descendant of v such that

$M(u) \neq M(B(u) \setminus \{e_{high}(u)\}) = M(v, c_2)$. Since the number of all those queries is $O(n)$, Lemma 5.27 implies that Algorithm 22 can compute all of them in linear time in total. Similarly, we can have the answers for the w vertices in Line 16 and the u vertices in Line 20 in $O(n)$ time in total. We conclude that Algorithm 34 has a linear-time implementation. $\square$

5.8.1.4 Case (4) of Lemma 5.57

For the Type-3βi 4-cuts that satisfy (4) of Lemma 5.57, we distinguish two cases, depending on whether $M(v) \neq M(B(v) \setminus \{e\})$ or $M(v) = M(B(v) \setminus \{e\})$. In the first case, by Lemma 3.9 we have that e is either $e_L(v)$ or $e_R(v)$. By Lemma 5.68 below, we know precisely how to locate u and w: u is the lowest proper descendant of v such that $M(u) = M(v, c_2)$, and w is the greatest proper ancestor of v such that $M(w) = M(w, c_1)$, where c_1 and c_2 are the *low1* and *low2* children of $M(B(v) \setminus \{e\})$, respectively. Thus, the procedure for computing all Type-3βi 4-cuts that satisfy (4) of Lemma 5.57 and $M(v) \neq M(B(v) \setminus \{e\})$ is shown in Algorithm 35. The proof of correctness and linear complexity is given in Proposition 5.23.

In the second case, we first determine u and w according to Lemma 5.69 and Lemma 5.70 (by considering all the different cases), and then e is uniquely determined by the relation $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. We use the criterion provided by Lemma 5.71 in order to check whether we indeed have a 4-cut. Thus, the procedure for computing all Type-3βi 4-cuts that satisfy (4) of Lemma 5.57 and $M(v) = M(B(v) \setminus \{e\})$ is shown in Algorithm 36. The proof of correctness is given in Proposition 5.24.

Let u, v, w be three vertices $\neq r$. Then we let $e(u, v, w)$ denote the pair $(\mathit{XorDesc}(u) \oplus \mathit{XorDesc}(v) \oplus \mathit{XorDesc}(w), \mathit{XorAnc}(u) \oplus \mathit{XorAnc}(v) \oplus \mathit{XorAnc}(w))$. (We note that $e(u, v, w)$ is not necessarily an edge of the graph.)

Lemma 5.67. *Let u, v, w be three vertices $\neq r$ such that there is a back-edge $e \in B(v)$ with $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. Then $e = e(u, v, w)$.*

Proof. Let $e = (x, y)$. Then $B(v) = (B(u) \sqcup B(v)) \sqcup \{e\}$ implies that $\mathit{XorDesc}(v) = \mathit{XorDesc}(u) \oplus \mathit{XorDesc}(w) \oplus x$ and $\mathit{XorAnc}(v) = \mathit{XorAnc}(u) \oplus \mathit{XorAnc}(w) \oplus y$. This implies that $x = \mathit{XorDesc}(u) \oplus \mathit{XorDesc}(v) \oplus \mathit{XorDesc}(w)$ and $y = \mathit{XorAnc}(u) \oplus \mathit{XorAnc}(v) \oplus \mathit{XorAnc}(w)$. $\square$

Lemma 5.68. *Let u, v, w be three vertices $\neq r$ such that w is proper ancestor of v, v is a proper ancestor of u, and there is a back-edge $e \in B(v)$ such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ and $M(B(v) \setminus \{e\}) \neq M(w)$. Suppose that $M(v) \neq M(B(v) \setminus \{e\})$. Let c_1 and c_2 be the low1 and low2 children of $M(B(v) \setminus \{e\})$, respectively. Then, u is the lowest proper descendant of v such that $M(u) = M(v, c_2)$, and w is the greatest proper ancestor of v such that $M(w) = M(v, c_1)$.*

Proof. Let $z = M(B(v) \setminus \{e\})$. Notice that, since $M(v) \neq M(B(v) \setminus \{e\})$, we have that the higher endpoint of e is not a descendant of z.

Since $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, we have $B(w) \subseteq B(v) \setminus \{e\}$, and therefore $M(w)$ is a descendant of $M(B(v) \setminus \{e\}) = z$. Since by assumption we have that $M(w) \neq z$, this implies that $M(w)$ is a proper descendant of z. Let c be the child of z that is an ancestor of $M(w)$. Let us suppose, for the sake of contradiction, that $c \neq c_1$. Let (x, y) be a back-edge in $B(w)$. Then x is a descendant of $M(w)$, and therefore a descendant of c. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v, and therefore a proper ancestor of $M(v)$, and therefore a proper ancestor of z, and therefore a proper ancestor of c. This shows that $(x, y) \in B(c)$, and therefore we have $low(c) \leq y$. Since y is a proper ancestor of w, we have $y < w$. Thus, $low(c) \leq y$ implies that $low(c) < w$. Now, since c_1 is the *low1* child of z, we have $low(c_1) \leq low(c)$, and therefore $low(c_1) < w$. This implies that there is a back-edge $(x', y') \in B(c_1)$ such that $y' < w$. Then we have that x' is a descendant of c_1, and therefore a descendant of z, and therefore a descendant of v, and therefore a descendant of w. Since (x', y') is a back-edge, we have that x' is a descendant of y'. Thus, x' is a common descendant of y' and w, and therefore y' and w are related as ancestor and descendant. Thus, $y' < w$ implies that y' is a proper ancestor of w. This shows that $(x', y') \in B(w)$, and therefore we have that x' is a descendant of $M(w)$, and therefore a descendant of c. Thus, x' is a common descendant of c and c_1, and therefore c and c_1 are related as ancestor and descendant. But this is impossible, since c and c_1 are supposed to be distinct children of z. Thus, we have $c = c_1$.

Since $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, we have $B(u) \subseteq B(v) \setminus \{e\}$, and therefore $M(u)$ is a descendant of $M(B(v) \setminus \{e\}) = z$. Let us suppose, for the sake of contradiction, that $M(u) = z$. Let (x, y) be a back-edge in $B(w)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x, y) \in B(v) \setminus \{e\}$, and therefore x is a descendant of $M(B(v) \setminus \{e\}) = z$, and therefore a descendant of $M(u)$. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v, and therefore a proper ancestor of u. This shows

that $(x,y) \in B(u)$, in contradiction to the fact that $B(u) \cap B(w) = \emptyset$. Thus, we have that $M(u)$ is a proper descendant of z. Let c' be the child of z that is an ancestor of $M(u)$.

Let us suppose, for the sake of contradiction, that c' is neither c_1 nor c_2. Let (x,y) be a back-edge in $B(u)$. This implies that x is a descendant of $M(u)$, and therefore a descendant of c'. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x,y) \in B(v)$. Then, y is a proper ancestor of v, and therefore a proper ancestor of $M(v)$, and therefore a proper ancestor of z, and therefore a proper ancestor of c'. This shows that $(x,y) \in B(c')$, and therefore $low(c') \leq y$. Since y is a proper ancestor of v, we have $y < v$. Thus, $low(c') \leq y$ implies that $low(c') < v$. Since c' is neither c_1 nor c_2, we have that $low(c_2) \leq low(c')$, and therefore $low(c_2) < v$. This implies that there is a back-edge $(x,y) \in B(c_2)$ such that $y < v$. Then x is a descendant of c_2, and therefore a descendant of z, and therefore a descendant of v. Since (x,y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Thus, $y < v$ implies that y is a proper ancestor of v. This shows that $(x,y) \in B(v)$. Then, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x,y) \in B(u)$, or $(x,y) \in B(w)$, or $(x,y) = e$. Let us suppose first that $(x,y) \in B(u)$. Then x is a descendant of $M(u)$, and therefore a descendant of c'. But then x is a common descendant of c_2 and c', which is impossible (since c_2 and c' are two distinct children of z). Now let us suppose that $(x,y) \in B(w)$. Then x is a descendant of $M(w)$, and therefore a descendant of c_1. But then x is a common descendant of c_1 and c_2, which is impossible (since c_1 and c_2 are two distinct children of z). The case $(x,y) = e$ is also rejected, because the higher endpoint of e is not a descendant of z. Thus, there are no viable options left, and so we have arrived at a contradiction. This shows that c' is either c_1 or c_2.

Let us suppose, for the sake of contradiction, that $c' = c_1$. Now let (x,y) be a back-edge in $B(v) \setminus \{e\}$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x,y) \in B(u)$ or $(x,y) \in B(w)$. If $(x,y) \in B(u)$, then x is a descendant of $M(u)$, and therefore a descendant of c_1. If $(x,y) \in B(w)$, then x is a descendant of $M(w)$, and therefore a descendant of c_1. In either case, then, we have that x is a descendant of c_1. Due to the generality of $(x,y) \in B(v) \setminus \{e\}$, this implies that $M(B(v) \setminus \{e\})$ is a descendant of c_1. But this is impossible, because c_1 is a child of $z = M(B(v) \setminus \{e\})$. Thus, we have $c' \neq c_1$, and therefore we infer that $c' = c_2$.

Let $S_1 = \{(x,y) \in B(v) \mid x \text{ is a descendant of } c_1\}$. Then $M(S_1) = M(v, c_1)$. Let

(x, y) be a back-edge in S_1. Then x is a descendant of c_1 and $(x, y) \in B(v)$. $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. Since the higher endpoint of e is not a descendant of z, the case $(x, y) = e$ is rejected. Let us suppose, for the sake of contradiction, that $(x, y) \in B(u)$. Then x is a descendant of $M(u)$, and therefore a descendant of c_2. Thus, x is a common descendant of c_1 and c_2, and therefore c_1 and c_2 are related as ancestor and descendant. But this is absurd, and therefore the case $(x, y) \in B(u)$ is rejected. Thus, we are left with the case $(x, y) \in B(w)$. Due to the generality of $(x, y) \in S_1$, this implies that $S_1 \subseteq B(w)$. Conversely, let (x, y) be a back-edge in $B(w)$. Then x is a descendant of $M(w)$, and therefore a descendant of c_1. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x, y) \in B(v)$. This shows that $(x, y) \in S_1$. Due to the generality of $(x, y) \in B(w)$, this implies that $B(w) \subseteq S_1$. Since we have shown the reverse inclusion too, we infer that $B(w) = S_1$. This implies that $M(w) = M(S_1)$, and therefore $M(w) = M(v, c_1)$.

Let $S_2 = \{(x, y) \in B(v) \mid x \text{ is a descendant of } c_2\}$. Then $M(S_2) = M(v, c_2)$. Let (x, y) be a back-edge in S_2. Then x is a descendant of c_2 and $(x, y) \in B(v)$. $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. Since the higher endpoint of e is not a descendant of z, the case $(x, y) = e$ is rejected. The case $(x, y) \in B(w)$ is also rejected, because this implies that $(x, y) \in S_1$ (whereas the sets S_1 and S_2 are disjoint, since c_1 and c_2 are not related as ancestor and descendant). Thus, we are left with the case $(x, y) \in B(u)$. Due to the generality of $(x, y) \in S_2$, this implies that $S_2 \subseteq B(u)$. Conversely, let (x, y) be a back-edge in $B(u)$. Then x is a descendant of $M(u)$, and therefore a descendant of c_2. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x, y) \in B(v)$. This shows that $(x, y) \in S_2$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq S_2$. Since we have shown the reverse inclusion too, we infer that $B(u) = S_2$. This implies that $M(u) = M(S_2)$, and therefore $M(u) = M(v, c_2)$.

Let us suppose, for the sake of contradiction, that u is not the lowest proper descendant of v such that $M(u) = M(v, c_2)$. This means that there is a proper descendant u' of v such that $u' < u$ and $M(u') = M(u)$. This implies that u' is a proper ancestor of u, and therefore Lemma 3.2 implies that $B(u') \subseteq B(u)$. Since the graph is 3-edge-connected, this can be strengthened to $B(u') \subset B(u)$. Thus, there is a back-edge $(x, y) \in B(u) \setminus B(u')$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x, y) \in B(v)$. Since $(x, y) \in B(u)$, we have that x is a descendant of $M(u) = M(u')$. Since $(x, y) \in B(v)$, we have that y is a proper ancestor of v, and therefore a proper ancestor of u'. This

shows that $(x,y) \in B(u')$, a contradiction. Thus, we have shown that u is the lowest proper descendant of v in $M^{-1}(M(u))$.

Let us suppose, for the sake of contradiction, that w is not the greatest proper ancestor of v such that $M(w) = M(v,c_1)$. This means that there is a proper ancestor w' of v such that $w' > w$ and $M(w') = M(w)$. This implies that w' is a proper descendant of w, and therefore Lemma 3.2 implies that $B(w) \subseteq B(w')$. Since the graph is 3-edge-connected, this can be strengthened to $B(w) \subset B(w')$. Thus, there is a back-edge $(x,y) \in B(w') \setminus B(w)$. Since $(x,y) \in B(w')$, we have that x is a descendant of $M(w') = M(w)$, and therefore a descendant of c_1, and therefore a descendant of z, and therefore a descendant of $M(v)$. Furthermore, we have that y is a proper ancestor of w', and therefore a proper ancestor of v. This shows that $(x,y) \in B(v)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x,y) \in B(u)$, or $(x,y) \in B(w)$, or $(x,y) = e$. The case $(x,y) \in B(w)$ is rejected, since $(x,y) \in B(w') \setminus B(w)$. The case $(x,y) \in B(u)$ implies that x is a descendant of $M(u)$, and therefore a descendant of c_2. But this is absurd, since x is a descendant of c_1 (and c_1, c_2 cannot have a common descendant). The case $(x,y) = e$ is also rejected, because the higher endpoint of e is not a descendant of z. Thus, there are no viable options left, and so we have arrived at a contradiction. This shows that w is the greatest proper ancestor of v such that $M(w) = M(v,c_1)$. $\square$

Lemma 5.69. *Let u, v, w be three vertices $\neq r$ such that w is proper ancestor of v, v is a proper ancestor of u, and there is a back-edge $e \in B(v)$ such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ and $M(B(v) \setminus \{e\}) \neq M(w)$. Suppose that $M(v) = M(B(v) \setminus \{e\})$. Let c_1, c_2 and c_3 be the low1, the low2 and the low3 child of $M(v)$, respectively. Then we have that either **(w.1)** $M(w) = M(v,c_1)$, or **(w.2)** $M(w) = M(v,c_1')$, where c_1' is the low1 child of $M(v,c_1)$. Furthermore, we have that either **(u.1)** $M(u) = M(v,c_2)$, or **(u.2)** $M(u) = M(v,c_1'')$, where c_1'' is the low1 child of $M(v,c_2)$, or **(u.3)** $M(u) = M(v,c_2'')$, where c_2'' is the low2 child of $M(v,c_2)$, or **(u.4)** $M(u) = M(v,c_3)$.*

Proof. Since $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, we have $B(w) \subseteq B(v)$, and therefore $M(w)$ is a descendant of $M(v)$. Since $M(v) = M(B(v) \setminus \{e\})$ and $M(B(v) \setminus \{e\}) \neq M(w)$, we have $M(v) \neq M(w)$. Therefore, $M(w)$ is a proper descendant of $M(v)$. Let c be the child of $M(v)$ that is an ancestor of $M(w)$. Let us suppose, for the sake of contradiction, that $c \neq c_1$. Let (x,y) be a back-edge in $B(w)$. Then x is a descendant of $M(w)$, and therefore a descendant of c. Furthermore, y is a proper ancestor of w,

and therefore a proper ancestor of v, and therefore a proper ancestor of $M(v)$, and therefore a proper ancestor of c. This shows that $(x, y) \in B(c)$, and therefore we have $low(c) \leq y$. Since y is a proper ancestor of w, we have $y < w$. Thus, $low(c) \leq y$ implies that $low(c) < w$. Now, since c_1 is the *low1* child of $M(v)$, we have $low(c_1) \leq low(c)$, and therefore $low(c_1) < w$. This implies that there is a back-edge $(x', y') \in B(c_1)$ such that $y' < w$. Then we have that x' is a descendant of c_1, and therefore a descendant of $M(v)$, and therefore a descendant of v, and therefore a descendant of w. Since (x', y') is a back-edge, we have that x' is a descendant of y'. Thus, x' is a common descendant of y' and w, and therefore y' and w are related as ancestor and descendant. Thus, $y' < w$ implies that y' is a proper ancestor of w. This shows that $(x', y') \in B(w)$, and therefore we have that x' is a descendant of $M(w)$, and therefore a descendant of c. Thus, x' is a common descendant of c and c_1, and therefore c and c_1 are related as ancestor and descendant. But this is impossible, since c and c_1 are supposed to be distinct children of $M(v)$. Thus, we have $c = c_1$.

Since $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, we have $B(u) \subseteq B(v)$, and therefore $M(u)$ is a descendant of $M(v)$. Let us suppose, for the sake of contradiction, that $M(u) = M(v)$. Let (x, y) be a back-edge in $B(w)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x, y) \in B(v)$, and therefore x is a descendant of $M(v)$, and therefore a descendant of $M(u)$. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$, in contradiction to the fact that $B(u) \cap B(w) = \emptyset$. Thus, we have that $M(u)$ is a proper descendant of $M(v)$. Let c' be the child of $M(v)$ that is an ancestor of $M(u)$.

Let us suppose, for the sake of contradiction, that c' is neither c_1, nor c_2, nor c_3. Let (x, y) be a back-edge in $B(u)$. Then x is a descencant of $M(u)$, and therefore a descendant of c'. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x, y) \in B(v)$, and therefore y is a proper ancestor of $M(v)$, and therefore a proper ancestor of c'. This shows that $(x, y) \in B(c')$, and therefore $low(c') \leq y$. Since $(x, y) \in B(v)$, we have that y is a proper ancestor of v, and therefore $y < v$. Thus, $low(c') \leq y$ implies that $low(c') < v$. Now, since c' is neither c_1, nor c_2, nor c_3, we have $low(c_1) \leq low(c_2) \leq low(c_3) \leq low(c') < v$. Since $low(c_2) < v$, there is a back-edge $(x, y) \in B(c_2)$ such that $y < v$. Then x is a descendant of c_2, and therefore a descendant of $M(v)$, and therefore a descendant of v. Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Then, $y < v$ implies that y is a proper ancestor of y. This shows

that $(x, y) \in B(v)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. The case $(x, y) \in B(u)$ is rejected, because it implies that x is a descendant of $M(u)$, and therefore a descendant of c', and therefore c' and c_2 have x as a common descendant, which is absurd. The case $(x, y) \in B(w)$ is also rejected, because it implies that x is a descendant of $M(w)$, and therefore a descendant of c_1, and therefore c_1 and c_2 have x as a common descendant, which is absurd. Thus, the only viable option is $(x, y) = e$. Now, since *low*$(c_3) < v$, there is a back-edge $(x', y') \in B(c_3)$ such that $y' < v$. We can follow the same reasoning as for (x, y), in order to infer that $(x', y') = e$. But then we have $x' = x$, and therefore x is a common descendant of c_2 and c_3, which is impossible. This shows that c' is either c_1, or c_2, or c_3.

Let us suppose, for the sake of contradiction, that $c' = c_1$. Let (x, y) be a back-edge in $B(v) \setminus \{e\}$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$. If $(x, y) \in B(u)$, then x is a descendant of $M(u)$, and therefore a descendant of c_1. If $(x, y) \in B(w)$, then x is a descendant of $M(w)$, and therefore a descendant of c_1. In any case, then, we have that x is a descendant of c_1. Due to the generality of $(x, y) \in B(v) \setminus \{e\}$, this implies that $M(B(v) \setminus \{e\})$ is a descendant of c_1. But this is impossible, because by assumption we have $M(B(v) \setminus \{e\}) = M(v)$, and c_1 is a child of $M(v)$. This shows that $c' \neq c_1$. Thus, we have that either $c' = c_2$, or $c' = c_3$.

Let (x, y) be a back-edge in $B(w)$. Then x is a descendant of $M(w)$, and therefore a descendant of c_1. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x, y) \in B(v)$. This shows that x is a descendant of $M(v, c_1)$. Due to the generality of $(x, y) \in B(w)$, this implies that $M(w)$ is a descendant of $M(v, c_1)$. If $M(w) = M(v, c_1)$, then we get **(w.1)**. So let us assume that $M(w)$ is a proper descendant of $M(v, c_1)$. Then there is a back-edge $(x_1, y_1) \in B(v)$, such that x_1 is a descendant of c_1, but not a descendant of $M(w)$. Since x_1 is a descendant of c_1, it cannot be a descendant of c_2 or c_3, and therefore it cannot be a descendant of $M(u)$. Thus, we have $(x_1, y_1) \notin B(u)$. And since x_1 is not a descendant of $M(w)$, we have $(x_1, y_1) \notin B(w)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x_1, y_1) = e$.

Now, since $M(w)$ is a proper descendant of $M(v, c_1)$, we have that $M(w)$ is a descendant of a child $\tilde{c}$ of $M(v, c_1)$. Let c_1' be the *low1* child of $M(v, c_1)$. Let us suppose, for the sake of contradiction, that $\tilde{c} \neq c_1'$. Let (x, y) be a back-edge in $B(w)$. Then x is a descendant of $M(w)$, and therefore a descendant of $\tilde{c}$. Furthermore,

$B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x, y) \in B(v)$, and therefore y is a proper ancestor of v, and therefore a proper ancestor of $M(v)$, and therefore a proper ancestor of c_1, and therefore a proper ancestor of $M(v, c_1)$, and therefore a proper ancestor of $\tilde{c}$. This shows that $(x, y) \in B(\tilde{c})$, and therefore $low(\tilde{c}) \leq y$. Since y is a proper ancestor of w, we have $y < w$. Therefore, $low(\tilde{c}) \leq y$ implies that $low(\tilde{c}) < w$. Since $\tilde{c} \neq c'_1$ and c'_1 is the *low1* child of $M(v, c_1)$, we have $low(c'_1) \leq low(\tilde{c}) < w$. This implies that there is a back-edge $(x, y) \in B(c'_1)$ such that $y < w$. Then x is a descendant of c'_1, and therefore a descendant of $M(v, c_1)$, and therefore a descendant of v. Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Since w is a proper ancestor of v, we have $w < v$. Then, $y < w$ implies that $y < v$, and therefore we have that y is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. The case $(x, y) \in B(u)$ is rejected, because it implies that x is a descendant of $M(u)$, and therefore a descendant of either c_2 or c_3, whereas x is a descendant of c_1. The case $(x, y) \in B(w)$ is also rejected, because it implies that x is a descendant of $M(w)$, and therefore a descendant of $\tilde{c}$, whereas x is a descendant of c'_1. Thus, we are left with the case $(x, y) = e$. Since $e \in B(v)$, we have that x is a descendant of v, and therefore a descendant of w. Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of w and y, and therefore w and y are related as ancestor and descendant. Then, $y < w$ implies that y is a proper ancestor of w. But this shows that $(x, y) \in B(w)$, in contradiction to the fact that $e \notin B(w)$. This shows that $\tilde{c} = c'_1$.

Let us suppose, for the sake of contradiction, that the higher endpoint of e is a descendant of c'_1. Let (x, y) be a back-edge in $B(v)$ such that x is a descendant of c_1. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. The case $(x, y) \in B(u)$ is rejected, because it implies that x is a descendant of $M(u)$, and therefore a descendant of either c_2 or c_3, whereas x is a descendant of c_1. Now, if $(x, y) \in B(w)$, then x is a descendant of $M(w)$, and therefore a descendant of c'_1. And if $(x, y) = e$, then by supposition we have that x is a descendant of c'_1. In either case, then, we have that x is a descendant of c'_1. Due to the generality of $(x, y) \in B(v)$ such that x is a descendant of c_1, this implies that $M(v, c_1)$ is a descendant of c'_1. But this is impossible, since c'_1 is a child of $M(v, c_1)$. This shows that the higher endpoint of e is not a descendant of c'_1.

Now let $S = \{(x,y) \in B(v) \mid x \text{ is a descendant of } c'_1\}$. Then we have $M(S) = M(v,c'_1)$. Let (x,y) be a back-edge in S. Then we have that x is a descendant of c'_1 and $(x,y) \in B(v)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x,y) \in B(u)$, or $(x,y) \in B(w)$, or $(x,y) = e$. The case $(x,y) \in B(u)$ is rejected, because it implies that x is a descendant of $M(u)$, and therefore a descendant of either c_2 or c_3, whereas x is a descendant of c'_1, and therefore a descendant of $M(v,c_1)$, and therefore a descendant of c_1. The case $(x,y) = e$ is also rejected, because we have shown that the higher endpoint of e is not a descendant of c'_1. Thus, we are left with $(x,y) \in B(w)$. Due to the generality of $(x,y) \in S$, this shows that $S \subseteq B(w)$. Conversely, let (x,y) be a back-edge in $B(w)$. Then x is a descendant of $M(w)$, and therefore a descendant of c'_1. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x,y) \in B(v)$. This shows that $(x,y) \in S$. Due to the generality of $(x,y) \in B(w)$, this implies that $B(w) \subseteq S$. Since we have showed the reverse inclusion too, we infer that $B(w) = S$. This implies that $M(w) = M(S)$, and therefore $M(w) = M(v,c'_1)$. Thus, we have established **(w.2)**.

Now, let us first consider the case that $c' = c_3$ (this is the shortest one). Let (x,y) be a back-edge in $B(u)$. Then x is a descendant of $M(u)$, and therefore a descendant of c_3. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x,y) \in B(v)$, and therefore y is a proper ancestor of v, and therefore a proper ancestor of $M(v)$, and therefore a proper ancestor of c_3. This shows that $(x,y) \in B(c_3)$, and therefore $low(c_3) \leq y$. Since y is a proper ancestor of v, we have $y < v$. Therefore, $low(c_3) \leq y$ implies that $low(c_3) < v$. Then, we have $low(c_2) \leq low(c_3) < v$. This implies that there is a back-edge $(x,y) \in B(c_2)$ such that $y < v$. Then x is a descendant of c_2, and therefore a descendant of $M(v)$, and therefore a descendant of v. Since (x,y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Then, $y < v$ implies that y is a proper ancestor of v. This shows that $(x,y) \in B(v)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x,y) \in B(u)$, or $(x,y) \in B(w)$, or $(x,y) = e$. The case $(x,y) \in B(u)$ is rejected, because it implies that x is a descendant of $M(u)$, and therefore a descendant of c_3, whereas x is a descendant of c_2. The case $(x,y) \in B(w)$ is also rejected, because it implies that x is a descendant of $M(w)$, and therefore a descendant of c_1, whereas x is a descendant of c_2. Thus, $(x,y) = e$ is the only viable option. This implies that the higher endpoint of e is a descendant of c_2.

Now let $S_3 = \{(x,y) \in B(v) \mid x \text{ is a descendant of } c_3\}$. Then we have $M(S_3) = M(v,c_3)$. Let (x,y) be a back-edge in S_3. Then x is a descendant of c_3 and $(x,y) \in B(v)$.

Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x,y) \in B(u)$, or $(x,y) \in B(w)$, or $(x,y) = e$. The case $(x,y) \in B(w)$ is rejected, because it implies that x is a descendant of $M(w)$, and therefore a descendant of c_1, whereas x is a descendant of c_3. The case $(x,y) = e$ is also rejected, because we have shown that the higher endpoint of e is a descendant of c_2. Thus, $(x,y) \in B(u)$ is the only acceptable option. Due to the generality of $(x,y) \in S_3$, this implies that $S_3 \subseteq B(u)$. Conversely, let (x,y) be a back-edge in $B(u)$. Then x is a descendant of $M(u)$, and therefore a descendant of c_3. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x,y) \in B(v)$. This shows that $(x,y) \in S_3$. Due to the generality of $(x,y) \in B(u)$, this implies that $B(u) \subseteq S_3$. Since we have shown the reverse inclusion too, we infer that $B(u) = S_3$. This implies that $M(u) = M(S_3)$, and therefore $M(u) = M(v, c_3)$. Thus, we get **(u.4)**.

So let us assume that $c' = c_2$ (which is the only case left). Let (x,y) be a back-edge in $B(u)$. Then x is a descendant of $M(u)$, and therefore a descendant of c_2. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x,y) \in B(v)$. This shows that x is a descendant of $M(v, c_2)$. Due to the generality of $(x,y) \in B(u)$, this implies that $M(u)$ is a descendant of $M(v, c_2)$. Now, if $M(u) = M(v, c_2)$, then we get **(u.1)**. So let us assume that $M(u)$ is a proper descendant of $M(v, c_2)$. Then there is a child c'' of $M(v, c_2)$ that is an ancestor of $M(u)$.

Let c''_1 be the *low1* child of $M(v, c_2)$, and let c''_2 be the *low2* child of $M(v, c_2)$. Let us suppose, for the sake of contradiction, that c'' is neither the *low1* nor the *low2* child of $M(v, c_2)$. Let (x,y) be a back-edge in $B(u)$. Then x is a descendant of $M(u)$, and therefore a descendant of c''. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x,y) \in B(v)$, and therefore y is a proper ancestor of v, and therefore a proper ancestor of $M(v)$, and therefore a proper ancestor of $M(v, c_2)$, and therefore a proper ancestor of c''. This shows that $(x,y) \in B(c'')$, and therefore $low(c'') \leq y$. Since y is a proper ancestor of v, we have $y < v$. Therefore, $low(c'') \leq y$ implies that $low(c'') < v$. Therefore, we have $low(c''_1) \leq low(c'') < v$. This implies that there is a back-edge $(x,y) \in B(c''_1)$ such that $y < v$. Then x is a descendant of c''_1, and therefore a descendant of $M(v, c_2)$, and therefore a descendant of v. Since (x,y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Then, $y < v$ implies that y is a proper ancestor of v. This shows that $(x,y) \in B(v)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x,y) \in B(u)$, or $(x,y) \in B(w)$, or $(x,y) = e$. The case $(x,y) \in B(u)$ is rejected, because it implies that x is a descendant

of $M(u)$, and therefore a descendant of c'', whereas x is a descendant of c''_1. The case $(x,y) \in B(w)$ is also rejected, because it implies that x is a descendant of $M(w)$, and therefore a descendant of c_1, whereas x is a descendant of c''_1, and therefore a descendant of $M(v,c_2)$, and therefore a descendant of c_2. Thus, $(x,y) = e$ is the only acceptable option, and thus the higher endpoint of e is a descendant of c''_1. Similarly, since c'' is neither c''_1 nor c''_2, we have that $low(c''_2) \leq low(c'') < v$, and therefore we can show with the same argument that the higher endpoint of e is a descendant of c''_2, which is absurd. Thus, our initial supposition cannot be true, and therefore we have that c'' is either c''_1 or c''_2.

Now let us suppose, for the sake of contradiction, that the higher endpoint of e is a descendant of c''. Let (x,y) be a back-edge in $B(v)$ such that x is a descendant of c_2. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x,y) \in B(u)$, or $(x,y) \in B(w)$, or $(x,y) = e$. The case $(x,y) \in B(w)$ is rejected, because it implies that x is a descendant of $M(w)$, and therefore a descendant of c_1, whereas x is a descendant of c_2. Thus, we have that either $(x,y) \in B(u)$, or $(x,y) = e$. If $(x,y) \in B(u)$, then x is a descendant of $M(u)$, and therefore a descendant of c''. If $(x,y) = e$, then by supposition we have that x is a descendant of c''. In either case, then, we have that x is a descendant of c''. Due to the generality of $(x,y) \in B(v)$ such that x is a descendant of c_2, this implies that $M(v,c_2)$ is a descendant of c''. But this is impossible, because c'' is a child of $M(v,c_2)$. Thus, we have that the higher endpoint of e is not a descendant of c''.

Now let $S' = \{(x,y) \in B(v) \mid x \text{ is a descendant of } c''\}$. Then we have $M(S') = M(v,c'')$. Let (x,y) be a back-edge in S'. Then x is a descendant of c'' and $(x,y) \in B(v)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x,y) \in B(u)$, or $(x,y) \in B(w)$, or $(x,y) = e$. The case $(x,y) \in B(w)$ is rejected, because it implies that x is a descendant of $M(w)$, and therefore a descendant of c_1, whereas x is a descendant of c'', and therefore a descendant of $M(v,c_2)$, and therefore a descendant of c_2. The case $(x,y) = e$ is also rejected, because we have shown that the higher endpoint of e is not a descendant of c''. Thus, $(x,y) \in B(u)$ is the only acceptable option. Due to the generality of $(x,y) \in S'$, this implies that $S' \subseteq B(u)$. Conversely, let (x,y) be a back-edge in $B(u)$. Then x is a descendant of $M(u)$, and therefore a descendant of c''. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x,y) \in B(v)$. This shows that $(x,y) \in S'$. Due to the generality of $(x,y) \in B(u)$, this implies that $B(u) \subseteq S'$. Since we have shown the reverse inclusion too, we infer that $B(u) = S'$. This implies that $M(u) = M(S')$, and therefore $M(u) = M(v,c'')$. Thus, if $c'' = c''_1$, then we get **(u.2)**. And if $c'' = c''_2$,

then we get **(u.3)**. □

Lemma 5.70. *Let u, v, w be three vertices $\neq r$ such that w is proper ancestor of v, v is a proper ancestor of u, and there is a back-edge $e \in B(v)$ such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ and $M(B(v) \setminus \{e\}) \neq M(w)$. Suppose that $M(v) = M(B(v) \setminus \{e\})$. Then u is the lowest proper descendant of v in $M^{-1}(M(u))$. In case* ***(w.1)*** *of Lemma 5.69 we have that w is either the greatest or the second-greatest proper ancestor of v in $M^{-1}(M(w))$. In case* ***(w.2)*** *of Lemma 5.69 we have that w is the greatest proper ancestor of v in $M^{-1}(M(w))$.*

Proof. Let us suppose, for the sake of contradiction, that u is not the lowest proper descendant of v in $M^{-1}(M(u))$. This means that there is a proper descendant u' of v such that $u' < u$ and $M(u') = M(u)$. This implies that u' is a proper ancestor of u, and then Lemma 3.2 implies that $B(u') \subseteq B(u)$. Since the graph is 3-edge-connected, this can be strengthened to $B(u') \subset B(u)$. Thus, there is a back-edge $(x, y) \in B(u) \setminus B(u')$. Then x is a descendant of $M(u) = M(u')$. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x, y) \in B(v)$, and therefore y is a proper ancestor of v, and therefore a proper ancestor of u'. But this shows that $(x, y) \in B(u')$, a contradiction. Thus, u is the lowest proper descendant of v in $M^{-1}(M(u))$.

Let c_1, c_2 and c_3 be the *low1*, *low2* and *low3* children of $M(v)$, respectively. Then Lemma 5.69 implies that $M(u)$ is either a descendant of c_2 or a descendant of c_3. In either case, then, we have that no descendant of $M(u)$ can be a descendant of c_1. Furthermore, Lemma 5.69 implies that either **(w.1)** $M(w) = M(v, c_1)$, or **(w.2)** $M(w) = M(v, c_1')$, where c_1' is the *low1* child of $M(v, c_1)$.

Let us assume first that **(w.1)** is true. That is, we have $M(w) = M(v, c_1)$. Let us suppose, for the sake of contradiction, that w is neither the greatest nor the second-greatest proper ancestor of v in $M^{-1}(M(w))$. This means that there are proper ancestors w' and w'' of v such that $w'' > w' > w$ and $M(w'') = M(w') = M(w)$. This implies that w'' is a proper descendant of w', and w' is a proper descendant of w. Then, Lemma 3.2 implies that $B(w) \subseteq B(w') \subseteq B(w'')$. Since the graph is 3-edge-connected, this can be strengthened to $B(w) \subset B(w') \subset B(w'')$. Thus, there are back-edges $(x, y) \in B(w') \setminus B(w)$ and $(x', y') \in B(w'') \setminus B(w')$. Then x is a descendant of $M(w') = M(w)$, and therefore a descendant of $M(v, c_1)$, and therefore a descendant of v. Furthermore, y is a proper ancestor of w', and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. The case $(x, y) \in B(u)$ is

rejected, because it implies that x is a descendant of $M(u)$, and therefore it is not a descendant of c_1, whereas x is a descendant of $M(v, c_1)$, and therefore a descendant of c_1. The case $(x, y) \in B(w)$ is rejected because $(x, y) \in B(w') \setminus B(w)$. Thus, the only viable option is $(x, y) = e$. Then, with the same argument we can show that $(x', y') = e$, and therefore we have $(x, y) = (x', y')$. But this contradicts the fact that $(x, y) \in B(w')$ and $(x', y') \notin B(w')$. Thus, we have shown that w is either the greatest or the second-greatest proper ancestor of v in $M^{-1}(M(w))$.

Now let us assume that **(w.2)** is true. That is, we have $M(w) = M(v, c_1')$, where c_1' is the *low1* child of $M(v, c_1)$. Let us suppose, for the sake of contradiction, that the higher endpoint of e is a descendant of c_1'. Let (x, y) be a back-edge in $B(v)$ such that x is a descendant of c_1. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. The case $(x, y) \in B(u)$ is rejected, because it implies that x is a descendant of $M(u)$, and therefore it is not a descendant of c_1, whereas we have that x is a descendant of c_1. Thus, we have that either $(x, y) \in B(w)$ or $(x, y) = e$. If $(x, y) \in B(w)$, then x is a descendant of $M(w)$, and therefore a descendant of $M(v, c_1')$, and therefore a descendant of c_1'. If $(x, y) = e$, then by supposition we have that x is a descendant of c_1'. In either case, then, we have that x is a descendant of c_1'. Due to the generality of $(x, y) \in B(v)$ such that x is a descendant of c_1, this implies that $M(v, c_1)$ is a descendant of c_1'. But this is impossible, because c_1' is a child of $M(v, c_1)$. Thus, we have that the higher endpoint of e is not a descendant of c_1'.

Now let us suppose, for the sake of contradiction, that w is not the greatest proper ancestor of v in $M^{-1}(M(w))$. This means that there is a proper ancestor w' of v such that $w' > w$ and $M(w') = M(w) = M(v, c_1')$. This implies that w' is a proper descendant of w, and therefore Lemma 3.2 implies that $B(w) \subseteq B(w')$. Since the graph is 3-edge-connected, this can be strengthened to $B(w) \subset B(w')$. Thus, there is a back-edge $(x, y) \in B(w') \setminus B(w)$. Then x is a descendant of $M(w') = M(w) = M(v, c_1')$, and therefore a descendant of v. Furthermore, y is a proper ancestor of w', and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. The case $(x, y) \in B(u)$ is rejected, because it implies that x is a descendant of $M(u)$, and therefore it is not a descendant of c_1, whereas x is a descendant of $M(v, c_1')$, and therefore a descendant of c_1', and therefore a descendant of $M(v, c_1)$, and therefore a descendant of c_1. The case $(x, y) = e$ is also rejected, because we have shown that the higher endpoint of e is not a descendant of c_1', whereas x is a descendant of $M(v, c_1')$, and therefore a descendant

of c_1'. Thus, $(x, y) \in B(w)$ is the only viable option left. But this contradicts the fact that $(x, y) \in B(w') \setminus B(w)$. Thus, we conclude that w is the greatest proper ancestor of v in $M^{-1}(M(w))$. □

Lemma 5.71. *Let u, v, w be three vertices $\neq r$ such that u is a proper descendant of v, and v is a proper descendant of w. Suppose that (1) $M(u)$ and $M(w)$ are descendants of $M(v)$ that are not related as ancestor and descendant, (2) $high(u) < v$, and (3) $bcount(v) = bcount(u) + bcount(w) + 1$. Then there is a back-edge e such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$.*

Proof. Let (x, y) be a back-edge in $B(u)$. Then x is a descendant of $M(u)$, and therefore a descendant of $M(v)$, and therefore a descendant of v. Furthermore, y is an ancestor of $high(u)$, and therefore $y \leq high(u)$. Then, $high(u) < v$ implies that $y < v$. Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Then, $y < v$ implies that y is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(v)$. Let (x, y) be a back-edge in $B(w)$. Then x is a descendant of $M(w)$, and therefore a descendant of $M(v)$. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(w)$, this implies that $B(w) \subseteq B(v)$. Let us suppose, for the sake of contradiction, that $B(u) \cap B(w) \neq \emptyset$. Then there is a back-edge $(x, y) \in B(u) \cap B(w)$. This implies that x is a descendant of both $M(u)$ and $M(w)$, contradicting the fact that $M(u)$ and $M(w)$ are not related as ancestor and descendant. Thus, we have $B(u) \cap B(w) = \emptyset$. Now, since $B(u) \subseteq B(v)$, $B(w) \subseteq B(v)$, $B(u) \cap B(w) = \emptyset$, and $bcount(v) = bcount(u) + bcount(w) + 1$, we infer that there is a back-edge e such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. □

Proposition 5.23. *Algorithm 35 correctly computes all Type-3βi 4-cuts that satisfy (4) of Lemma 5.57 and $M(v) \neq M(B(v) \setminus \{e\})$. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3βi 4-cut, where w is a proper ancestor of v, v is a proper ancestor of u, e satisfies (4) of Lemma 5.57 and $M(v) \neq M(B(v) \setminus \{e\})$. Since $M(v) \neq M(B(v) \setminus \{e\})$, Lemma 3.9 implies that either $e = e_L(v)$ or $e = e_R(v)$. Let us assume that $e = e_L(v)$ (the other case is treated similarly). Let c_1 be the *low1* child of $M(B(v) \setminus \{e_L(v)\})$, and let c_2 be the *low2* child of $M(B(v) \setminus \{e_L(v)\})$. Then Lemma 5.68 implies that u is the lowest proper descendant of v such that $M(u) = M(v, c_2)$, and w is the greatest proper ancestor of v such that

Algorithm 35: Compute all Type-3βi 4-cuts that satisfy (4) of Lemma 5.57 and $M(v) \neq M(B(v) \setminus \{e\})$

```
1  foreach vertex v ≠ r do
2      compute M(B(v) \ {e_L(v)}) and M(B(v) \ {e_R(v)})
3  end
4  foreach vertex v ≠ r do
5      if M(B(v) \ {e_L(v)}) has at least two children then
6          let c_1 be the low1 child of M(B(v) \ {e_L(v)})
7          let c_2 be the low2 child of M(B(v) \ {e_L(v)})
8          compute M(v, c_1) and M(v, c_2)
9      end
10     if M(B(v) \ {e_R(v)}) has at least two children then
11         let c_1 be the low1 child of M(B(v) \ {e_R(v)})
12         let c_2 be the low2 child of M(B(v) \ {e_R(v)})
13         compute M(v, c_1) and M(v, c_2)
14     end
15 end
16 foreach vertex v ≠ r do
17     if M(B(v) \ {e_L(v)}) has at least two children then
18         let c_1 be the low1 child of M(B(v) \ {e_L(v)})
19         let c_2 be the low2 child of M(B(v) \ {e_L(v)})
20         let u be the lowest proper descendant of v such that M(u) = M(v, c_2)
21         let w be the greatest proper ancestor of v such that M(w) = M(v, c_1)
22         if high(u) < v and bcount(v) = bcount(u) + bcount(w) + 1 then
23             mark {(u, p(u)), (v, p(v)), (w, p(w)), e(u, v, w)} as a 4-cut
24         end
25     end
26     if M(B(v) \ {e_R(v)}) has at least two children then
27         let c_1 be the low1 child of M(B(v) \ {e_R(v)})
28         let c_2 be the low2 child of M(B(v) \ {e_R(v)})
29         let u be the lowest proper descendant of v such that M(u) = M(v, c_2)
30         let w be the greatest proper ancestor of v such that M(w) = M(v, c_1)
31         if high(u) < v and bcount(v) = bcount(u) + bcount(w) + 1 then
32             mark {(u, p(u)), (v, p(v)), (w, p(w)), e(u, v, w)} as a 4-cut
33         end
34     end
35 end
```

$M(w) = M(v, c_1)$. Since (4) of Lemma 5.57 is satisfied, we have $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. This implies that $bcount(v) = bcount(u) + bcount(w) + 1$. Furthermore, this implies that $B(u) \subseteq B(v)$, and therefore $high(u) < v$ (because the lower endpoint of every back-edge in $B(u)$ is a proper ancestor of v). Lemma 5.67 implies that $e = e(u, v, w)$. Thus, C will be marked in Line 23.

Conversely, let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e(u, v, w)\}$ be a 4-element set that is marked in Line 23 or 32. Let us assume that C is marked in Line 23 (the other case is treated similarly). Then we have that u is a proper descendant of v such that $M(u) = M(v, c_2)$ and w is a proper ancestor of v such that $M(w) = M(v, c_1)$, where c_1 and c_2 are the *low1* and *low2* children of $M(B(v) \setminus \{e_L(v)\})$, respectively. Then we have that both $M(u)$ and $M(w)$ are descendants of $M(v)$. Furthermore, $M(u)$ is a descendant of c_1 and $M(w)$ is a descendant of c_2. Thus, $M(u)$ and $M(w)$ are not related as ancestor and descendant. Then, we also have that $high(u) < v$ and $bcount(v) = bcount(u) + bcount(w) + 1$. Thus, all the conditions of Lemma 5.71 are satisfied, and therefore we have $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. Then, Lemma 5.67 implies that $e = e(u, v, w)$. Thus, C is a 4-cut that satisfies (4) of Lemma 5.57. Therefore, C is correctly marked in Line 23 as a 4-cut.

Now will argue about the complexity of Algorithm 35. By Proposition 3.6, we have that the values $M(B(v) \setminus \{e_L(v)\})$ and $M(B(v) \setminus \{e_R(v)\})$ can be computed in linear time in total, for every vertex $v \neq r$. Thus, the **for** loop in Line 1 can be performed in linear time. Proposition 3.5 implies that all values $M(v, c_1)$ and $M(v, c_2)$ can be computed in linear time in total, for every vertex $v \neq r$ such that $M(B(v) \setminus \{e_L(v)\})$ (resp., $M(B(v) \setminus \{e_R(v)\})$) has at least two children, where c_1 and c_2 are the *low1* and *low2* children of $M(B(v) \setminus \{e_L(v)\})$ (resp., $M(B(v) \setminus \{e_R(v)\})$). Thus, the **for** loop in Line 4 can be performed in linear time. The vertices u and w in Lines 20, 21, 29 and 30 can be computed in linear time in total with Algorithm 22 (see e.g. the proof of Proposition 5.21, on how we generate the queries that provide the vertices u and w). We conclude that Algorithm 35 runs in linear time. $\square$

Proposition 5.24. *Algorithm 36 correctly computes all Type-3βi 4-cuts that satisfy (4) of Lemma 5.57 and $M(v) = M(B(v) \setminus \{e\})$. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3βi 4-cut, where w is a proper ancestor of v, v is a proper ancestor of u, e satisfies (4) of Lemma 5.57 and $M(v) = M(B(v) \setminus \{e\})$. We will use the notation that is introduced in Lines 2, 3 and

Algorithm 36: Compute all Type-3βi 4-cuts that satisfy (4) of Lemma 5.57 and $M(v) = M(B(v) \setminus \{e\})$

1. **foreach** *vertex* $v \neq r$ **do**
2.   let $c_1(v)$, $c_2(v)$ and $c_3(v)$ denote the *low1*, *low2* and *low3* children of $M(v)$, respectively
3.   let $c'_1(v)$ denote the *low1* child of $M(v, c_1)$
4.   let $c''_1(v)$ and $c''_2(v)$ denote the *low1* and *low2* children of $M(v, c_2)$, respectively
5. **end**
6. **foreach** *vertex* $v \neq r$ **do**
   // case **(w.1)** of Lemma 5.69
7.   let w be the greatest proper ancestor of v such that $M(w) = M(v, c_1(v))$
8.   **if** $w \neq \perp$ **then**
     // case **(u.1)** of Lemma 5.69
9.   let u be the lowest proper descendant of v such that $M(u) = M(v, c_2(v))$
10.   **if** $high(u) < v$ ***and*** $bcount(v) = bcount(u) + bcount(w) + 1$ **then**
11.   mark $\{(u, p(u)), (v, p(v)), (w, p(w)), e(u, v, w)\}$ as a 4-cut
12.   **end**
     // case **(u.2)** of Lemma 5.69
13.   let u be the lowest proper descendant of v such that $M(u) = M(v, c''_1(v))$
14.   **if** $high(u) < v$ ***and*** $bcount(v) = bcount(u) + bcount(w) + 1$ **then**
15.   mark $\{(u, p(u)), (v, p(v)), (w, p(w)), e(u, v, w)\}$ as a 4-cut
16.   **end**
     // case **(u.3)** of Lemma 5.69
17.   let u be the lowest proper descendant of v such that $M(u) = M(v, c''_2(v))$
18.   **if** $high(u) < v$ ***and*** $bcount(v) = bcount(u) + bcount(w) + 1$ **then**
19.   mark $\{(u, p(u)), (v, p(v)), (w, p(w)), e(u, v, w)\}$ as a 4-cut
20.   **end**
     // case **(u.4)** of Lemma 5.69
21.   let u be the lowest proper descendant of v such that $M(u) = M(v, c_3(v))$
22.   **if** $high(u) < v$ ***and*** $bcount(v) = bcount(u) + bcount(w) + 1$ **then**
23.   mark $\{(u, p(u)), (v, p(v)), (w, p(w)), e(u, v, w)\}$ as a 4-cut
24.   **end**
25.   **end**
26. $w \leftarrow prevM(w)$
27.   **if** $w \neq \perp$ ***and*** w *is a proper ancestor of* v **then**
28.   perform the same steps as in Lines 9 to 24
29.   **end**
   // case **(w.2)** of Lemma 5.69
30.   let w be the greatest proper ancestor of v such that $M(w) = M(v, c'_1(v))$
31.   **if** $w \neq \perp$ **then**
32.   perform the same steps as in Lines 9 to 24
33.   **end**
34. **end**

4: i.e., $c_1(v)$ is the *low1* child of $M(v)$, $c_2(v)$ is the *low2* child of $M(v)$, $c_3(v)$ is the *low3* child of $M(v)$, $c'_1(v)$ is the *low1* child of $M(v, c_1)$, $c''_1(v)$ is the *low1* child of $M(v, c_2)$, and $c''_2(v)$ is the *low2* child of $M(v, c_2)$. (We note that some of those values may be *null*.)

Then, according to Lemma 5.69, we have two cases for w: either **(w.1)** $M(w) = M(v, c_1(v))$, or **(w.2)** $M(w) = M(v, c'_1(v))$. Furthermore, we have four cases for u: either **(u.1)** $M(u) = M(v, c_2(v))$, or **(u.2)** $M(u) = M(v, c''_1(v))$, or **(u.3)** $M(u) = M(v, c''_2(v))$, or **(u.4)** $M(u) = M(v, c_3(v))$. By Lemma 5.70, we have that u, in any case, is the lowest proper descendant of v with the respective property. Furthermore, w in case **(w.2)** is the greatest proper ancestor of v with this property, whereas in case **(w.1)** it is either the greatest or the second-greatest proper ancestor of v. Thus, there are twelve distinct cases in total.

Since C satisfies (4) of Lemma 5.57, we have that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. This implies that $bcount(v) = bcount(u) + bcount(w) + 1$. Furthermore, we have $B(u) \subseteq B(v)$, and therefore $high(u) < v$ (because the lower endpoint of every back-edge in $B(u)$ is a proper ancestor of v). By Lemma 5.67 we have $e = e(u, v, w)$. Thus, it is clear that, if w satisfies **(w.1)** and is the greatest proper ancestor of v with this property, or it satisfies **(w.2)**, then the condition in Line 8 or Line 31 will be satisfied, and therefore C will be marked at some point between Lines 9 to 24. Otherwise, we have that w satisfies **(w.1)** and is the second-greatest proper ancestor of v with this property. So let w' be the greatest proper ancestor of v that satisfies $M(w') = M(v, c_1(v))$. Then we have $w = prevM(w')$, because $prevM(w')$ is the lowest vertex in $M^{-1}(M(w'))$ that is greater than w'. Thus, Line 27 will be satisfied, and therefore C will be marked at some point between Lines 9 to 24.

Conversely, let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e(u, v, w)\}$ be a 4-element set that is marked at some point between Lines 9 to 24, where we have entered the condition in Line 8, or 27, or 31. Then, in either case, we have the following facts. First, u is a proper descendant of v, and w is a proper ancestor of v. Second, $M(w)$ is a descendant of $c_1(v)$, and $M(u)$ is a descendant of either $c_2(v)$ or $c_3(v)$. Thus, both $M(w)$ and $M(u)$ are descendants of $M(v)$, but they are not related as ancestor and descendant. And third, we have $high(u) < v$ and $bcount(v) = bcount(u) + bcount(w) + 1$. Thus, all the conditions of Lemma 5.71 are satisfied, and therefore we have that there is a back-edge e such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. Then Lemma 5.67 implies that $e = e(u, v, w)$. Thus, C is a 4-cut that satisfies (4) of Lemma 5.57, and therefore

it is correctly marked as a 4-cut.

Now we will argue about the complexity of Algorithm 36. For every vertex $v \neq r$, we generate queries for computing $M(v, c_1(v))$, $M(v, c_2(v))$ and $M(v, c_3(v))$ (for those of $c_1(v)$, $c_2(v)$ and $c_3(v)$ that exist). By Proposition 3.5, we can have the answer to all those queries in linear time in total. Then, for every $v \neq r$ such that $M(v, c_1(v)) \neq \bot$ and $c_1'(v) \neq \bot$, we generate a query for computing $M(v, c_1'(v))$. Similarly, if $M(v, c_2(v)) \neq \bot$, then we generate queries for computing $M(v, c_1''(v))$ and $M(v, c_2''(v))$ (for those of $c_1''(v)$ and $c_2''(v)$ that exist). According to Proposition 3.5, all those queries can be answered in linear time in total. We keep pointers from every vertex v to all its respective values that were computed (i.e., a pointer to $M(v, c_1(v))$, a pointer to $M(v, c_1''(v))$, etc.). In order to get the vertices u and w throughout (e.g., in Line 7 and Line 9), we first collect all the queries for those vertices, and we make appropriate use of Algorithm 22. The way to do this (and the guarantee of correctness) has been already explained in previous algorithms (e.g., in the proof of Proposition 5.21). By Lemma 5.27, we can have the answer to all those queries in linear time in total. It is easy to see that all other operations in Algorithm 36 take $O(n)$ time in total. We conclude that Algorithm 36 runs in linear time. $\square$

5.8.2 Type-3βii 4-cuts

5.8.2.1 Type-3βii-1 4-cuts

Now we consider case (1) of Lemma 5.57.

Let u, v, w be three vertices $\neq r$ such that w is proper ancestor of v, v is a proper ancestor of u, and there is a back-edge $e \in B(u) \cap B(v) \cap B(w)$ such that $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$ and $M(B(v) \setminus \{e\}) = M(B(w) \setminus \{e\})$. By Lemma 5.57, we have that $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ is a 4-cut; we call this a Type-3βii-1 4-cut.

The following lemma provides some useful information concerning this type of 4-cuts.

Lemma 5.72. *Let u, v, w be three vertices such that (u, v, w) induces a Type-3βii-1 4-cut, and let e be the back-edge of the 4-cut induced by (u, v, w). Then $e = (lowD(u), low(u))$. Furthermore, $low(u) < w$, $low_2(u) \geq w$, $high(u) = high(v)$, w is an ancestor of $high(v)$ and $M(w) = M(v)$. Finally, if u' is a vertex such that $u \geq u' \geq v$ and $high(u') = high(v)$, then u' is an ancestor of u.*

Proof. Since (u, v, w) induces a Type-3βii-1 4-cut, we have that $e \in B(u) \cap B(v) \cap B(w)$. Since $e \in B(w)$, we have that the lower endpoint of e is strictly lower than w. And since $e \in B(u)$, we have that $low(u)$ is at least as low as the lower endpoint of e. This shows that $low(u) < w$.

Let us suppose, for the sake of contradiction, that $low_2(u) < w$. Let (x, y) be a back-edge in $B(u)$ such that $y = low_2(u)$. Then x is a descendant of u, and therefore a descendant of v, and therefore a descendant of w. Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of w and y, and therefore w and y are related as ancestor and descendant. Since $y = low_2(u)$ and $low_2(u) < w$, we have $y < w$. This implies that y is a proper ancestor of w. This shows that $(x, y) \in B(w)$. With the same argument, we have that the *low*-edge of u is also in $B(w)$. Since (u, v, w) induces a Type-3βii-1 4-cut, we have $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$. Since $e \in B(u) \cap B(w)$, this implies that there is only one back-edge in $B(u) \cap B(w)$, a contradiction. Thus, we have $w \leq low_2(u)$. Furthermore, since $e \in B(u) \cap B(w)$, we have that e is the only back-edge in $B(u)$ whose lower endpoint is low enough to be lower than w, and therefore e is the *low*-edge of u.

Since $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$ and $e \in B(u) \cap B(v)$, we have $B(u) \subseteq B(v)$. This implies that $high(v) \geq high(u)$. Let us suppose, for the sake of contradiction, that $high(v) > high(u)$. Let (x, y) be a back-edge in $B(v)$ such that $y = high(v)$. Then $y > high(u)$, and therefore $(x, y) \notin B(u)$. Thus, $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$ implies that $(x, y) \in B(w)$. Then we have that y is a proper ancestor of w, and therefore $y < w$. But we have $w \leq low(u) \leq high(u) < high(v) = y$, a contradiction. This shows that $high(u) = high(v)$.

Since both w and $high(v)$ are ancestors of v, we have that w and $high(v)$ are related as ancestor and descendant. Now let us suppose, for the sake of contradiction, that w is a proper descendant of $high(v)$. Let (x, y) be a back-edge in $B(u)$ such that $y = high(u)$. Then, since $high(u) = high(v)$, we have that $y = high(v)$, and therefore y is a proper ancestor of w. But since $y = high(u)$, this implies that $low_2(u) < w$, a contradiction. This shows that w is an ancestor of $high(v)$.

Since $B(w) \setminus \{e\} \subseteq B(v) \setminus \{e\}$ and $e \in B(v) \cap B(w)$, we have that $B(w) \subseteq B(v)$. This implies that $M(w)$ is a descendant of $M(v)$. Since $e \in B(u)$, we have that $M(u)$ is an ancestor of the higher endpoint of e. Furthermore, since $e \in B(w)$, we have that $M(w)$ is an ancestor of the higher endpoint of e. Thus, since $M(u)$ and $M(w)$ have a common descendant, they are related as ancestor and descendant. Since u

is an ancestor of $M(u)$, this implies that $M(w)$ and u are also related as ancestor and descendant. Let us suppose, for the sake of contradtion, that $M(w)$ is not an ancestor of u. Then we have that $M(w)$ is a proper descendant of u. Since the graph is 3-edge-connected, we have that there is a back-edge $(x, y) \in B(w) \setminus \{e\}$. Then x is a descendant of $M(w)$, and therefore a descendant of u. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. But then we have $(x, y) \in (B(u) \setminus \{e\}) \cap (B(w) \setminus \{e\})$, a contradiction. Thus, we have that $M(w)$ is an ancestor of u, and therefore an ancestor of $M(u)$. Now let (x, y) be a back-edge in $B(v)$. If $(x, y) = e$, then $e \in B(w)$, and therefore x is a descendant of $M(w)$. Otherwise, since $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$, we have that either $(x, y) \in B(u) \setminus \{e\}$, or $(x, y) \in B(w) \setminus \{e\}$. If $(x, y) \in B(u) \setminus \{e\}$, then x is a descendant of u, and therefore x is a descendant of $M(w)$. If $(x, y) \in B(w) \setminus \{e\}$, then $M(w)$ is an ancestor of x. Thus, in every case we have that $M(w)$ is an ancestor of x. Due to the generality of $(x, y) \in B(v)$, this shows that $M(w)$ is an ancestor of $M(v)$. Thus, since $M(w)$ is a descendant of $M(v)$, we infer that $M(w) = M(v)$.

Now let u' be a vertex such that $u \geq u' \geq v$ and $\mathit{high}(u') = \mathit{high}(v)$. Since u is a descendant of v and $u \geq u' \geq v$, we have that u' is also a descendant of v. Furthermore, since $u \geq u'$, we have that either u' is an ancestor of u, or it is not related as ancestor and descendant with u. Now let us suppose, for the sake of contradiction, that u' is not an ancestor of u. Then u' is not related as ancestor and descendant with u. Let (x, y) be a back-edge in $B(u')$ with $y = \mathit{high}(u')$. Then x is a descendant of u', and therefore a descendant of v. Thus, $y = \mathit{high}(u') = \mathit{high}(v)$ implies that (x, y) is in $B(v)$. Since u' is not related as ancestor and descendant with u, we have that x is not a descendant of u. Thus, $(x, y) \notin B(u)$. This implies that $e \neq (x, y)$, and thus $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$ implies that $(x, y) \in B(w)$, and therefore y is a proper ancestor of w. But $y = \mathit{high}(v)$ and $\mathit{high}(v)$ is a descendant of w, a contradiction. This shows that u' is an ancestor of u. $\square$

We will provide a method to compute all Type-3βii-1 4-cuts in linear time. The idea is to compute, for every vertex v, a set $U_1(v)$ of proper descendants u of v that have the potential to participate in a triple (u, v, w) that induces a Type-3βii-1 4-cut. (These sets have the property that their total size is $O(n)$.) Then, for every $u \in U_1(v)$, we search for all w with $M(w) = M(v)$ that may participate in a triple (u, v, w) that induces a Type-3βii-1 4-cut. (In fact, we can show that such a w, if it exists, is unique.)

It is sufficient to restrict our search to w with $M(w) = M(v)$, according to Lemma 5.72.

Now let $v \neq r$ be a vertex with $\mathit{nextM}(v) \neq \bot$. Let S be the segment of $H(\mathit{high}(v))$ that contains v and is maximal w.r.t. the property that its elements are related as ancestor and descendant (i.e., we have $S = S(v)$). Then we let $U_1(v)$ denote the collection of all vertices $u \in S$ such that either (1) u is a proper descendant of v with $\mathit{nextM}(v) > \mathit{low}_2(u) \geq \mathit{lastM}(v)$, or (2) u is the lowest proper descendant of v in S such that $\mathit{low}_2(u) \geq \mathit{nextM}(v)$.

Lemma 5.73. *Let (u, v, w) be a triple of vertices that induces a Type-3βii-1 4-cut. Then $u \in U_1(v)$.*

Proof. Since (u, v, w) induces a Type-3βii-1 4-cut, we have that u is a proper descendant of v, and by Lemma 5.72 we have $\mathit{high}(u) = \mathit{high}(v)$. Let u' be a vertex such that $u \geq u' \geq v$ and $\mathit{high}(u') = \mathit{high}(v)$. Then Lemma 5.72 implies that u' is an ancestor of u. This shows that u and v belong to a segment of $H(\mathit{high}(v))$ with the property that its elements are related as ancestors and descendants. Thus, we have $u \in S(v)$.

By Lemma 5.72 we have that w is a proper ancestor of v with $M(w) = M(v)$. Thus, $\mathit{nextM}(v) \neq \bot$ and $w \leq \mathit{nextM}(v)$. Now, if u satisfies $\mathit{nextM}(v) > \mathit{low}_2(u) \geq \mathit{lastM}(v)$, then u satisfies enough conditions to be in $U_1(v)$. Otherwise, if $\mathit{nextM}(v) > \mathit{low}_2(u) \geq \mathit{lastM}(v)$ is not true, then either $\mathit{low}_2(u) \geq \mathit{nextM}(v)$ or $\mathit{low}_2(u) < \mathit{lastM}(v)$. Since $\mathit{lastM}(v) \leq w$, the case $\mathit{low}_2(u) < \mathit{lastM}(v)$ is rejected by Lemma 5.72 (because this ensures that $\mathit{low}_2(u) \geq w$, and we have that $w \geq \mathit{lastM}(v)$). Thus we have $\mathit{low}_2(u) \geq \mathit{nextM}(v)$.

Now let us suppose, for the sake of contradiction, that there is a vertex $u' \in S(v)$ that is a proper descendant of v, it is lower than u, and satisfies $\mathit{low}_2(u') \geq \mathit{nextM}(v)$. This implies that u' is a proper ancestor of u (because all vertices in $S(v)$ are related as ancestor and descendant). Now let (x, y) be a back-edge in $B(u)$. Then x is a descendant of u, and therefore a descendant of u'. Furthermore, y is an ancestor of $\mathit{high}(u) = \mathit{high}(u')$, and therefore it is a proper ancestor of u'. This shows that $(x, y) \in B(u')$, and therefore we have $B(u) \subseteq B(u')$. This can be strengthened to $B(u) \subset B(u')$, since the graph is 3-edge-connected. Thus, there is a back-edge $(x, y) \in B(u') \setminus B(u)$. Then, x is a descendant of u', and therefore a descendant of v. Furthermore, y is an ancestor of $\mathit{high}(u')$, and therefore it is a proper ancestor of v (since $\mathit{high}(u') = \mathit{high}(v)$). This shows that $(x, y) \in B(v)$. Since (u, v, w) induces a Type-3βii-1 4-cut, we have $B(v) \setminus \{e\} \subseteq B(u) \cup B(w)$ and $e \in B(u) \cap B(v) \cap B(w)$, where e is the back-edge of

the 4-cut induced by (u, v, w). Since $(x, y) \in B(v)$ and $(x, y) \notin B(u)$, this implies that $(x, y) \neq e$ and $(x, y) \in B(w)$. Thus, since $low_2(u') \geq nextM(v)$ and $w \leq nextM(v)$, we have that $(x, y) = (lowD_1(u'), low_1(u'))$, and (x, y) is the only back-edge in $B(u')$ whose lower endpoint is lower than w. Now, since $e \in B(u)$, we have that the higher endpoint of e is a descendant of u, and therefore a descendant of u'. Then, since $e \in B(w)$, we have that the lower endpoint of e is a proper ancestor of w, and therefore a proper ancestor of v, and therefore a proper ancestor of u'. This shows that $e \in B(u')$. But we have that (x, y) is the only back-edge in $B(u')$ whose lower endpoint is lower than w, and therefore $(x, y) = e$, a contradiction. We conclude that u is the lowest vertex in $S(v)$ that is a proper descendant of v with $low_2(u) \geq nextM(v)$. Thus, u satisfies enough conditions to be in $U_1(v)$. □

Lemma 5.74. *Let v and v' be two vertices $\neq r$ such that $nextM(v) \neq \bot$ and $nextM(v') \neq \bot$. Suppose that v' is a proper descendant of v with $high(v') = high(v)$. Then $nextM(v') < lastM(v)$.*

Proof. Since $high(v') = high(v)$ and v' is a proper descendant of v, by Lemma 3.3 we have that $B(v') \subseteq B(v)$. Since the graph is 3-edge-connected, this can be strengthened to $B(v') \subset B(v)$. This implies that $M(v')$ is a descendant of $M(v)$. Since $high(v') = high(v)$, Lemma 3.7 implies that $M(v') \neq M(v)$ (for otherwise we would have $B(v') = B(v)$). Thus, $M(v')$ is a proper descendant of $M(v)$.

Now let w be a proper ancestor of v with $M(w) = M(v)$, and let w' be a proper ancestor of v' with $M(w') = M(v')$. Then $nextM(v) \geq w \geq lastM(v)$ and $nextM(v') \geq w' \geq lastM(v')$. Let us suppose, for the sake of contradiction, that $w \leq w'$. Since w is an ancestor of v, it is also an ancestor of v'. Thus, w and w' have v' as a common descendant, and therefore they are related as ancestor and descendant. Since $w \leq w'$, this implies that w is an ancestor of w'. Let us suppose, for the sake of contradiction, that w' is a descendant of v. This implies that w' is a proper descendant of $high(v) = high(v')$. Now let (x, y) be a back-edge in $B(v')$. Then x is a descendant of $M(v') = M(w')$. Furthermore, y is an ancestor of $high(v')$, and therefore a proper ancestor of w'. This shows that $(x, y) \in B(w')$, and thus we have $B(v') \subseteq B(w')$. Conversely, since $M(v') = M(w')$ and w' is a proper ancestor of v', Lemma 3.2 implies that $B(w') \subseteq B(v')$. Thus we have $B(v') = B(w')$, a contradiction. This shows that w' is not a descendant of v. Since w' and v have v' as a common descendant, we have that w' and v are related as ancestor and descendant. Thus, w' is a proper ancestor of

v. Therefore, $M(v) = M(w)$ is a descendant of w'. Thus, since w is an ancestor of w' and $M(w)$ is a descendant of w', by Lemma 3.1 we have that $M(w)$ is a descendant of $M(w')$. But $M(w) = M(v)$ and $M(w') = M(v')$, and so we have a contradiction to the fact that $M(v')$ is a proper descendant of $M(v)$. This shows that $w > w'$. Due to the generality of w and w', we conclude that $\mathit{lastM}(v) > \mathit{nextM}(v')$. $\square$

Lemma 5.75. *Let v and v' be two vertices with $\mathit{nextM}(v) \neq \bot$ and $\mathit{nextM}(v') \neq \bot$, such that $S(v') = S(v)$ and v' is a proper descendant of v. If $U_1(v') = \emptyset$, then $U_1(v) = \emptyset$. If $U_1(v') \neq \emptyset$, then the lowest vertex in $U_1(v)$ (if it exists) is greater than, or equal to, the greatest vertex in $U_1(v')$.*

Proof. First, let us suppose that there is a vertex u in $U_1(v)$. We will show that u is a proper descendant of v'. So let us suppose, for the sake of contradiction, that u is not a proper descendant of v'. Since $u \in U_1(v)$, we have $u \in S(v)$. Thus, since $v' \in S(v') = S(v)$, we have that u and v' are related as ancestor and descendant. Then, since u is not a proper descendant of v', we have that u is an ancestor of v'. Since u and v' are in $S(v)$, we have $\mathit{high}(u) = \mathit{high}(v) = \mathit{high}(v')$. Thus, Lemma 3.3 implies that $B(v') \subseteq B(u)$. Since $\mathit{nextM}(v) \neq \bot$ and $\mathit{nextM}(v') \neq \bot$ and v' is a proper descendant of v, Lemma 5.74 implies that $\mathit{nextM}(v') < \mathit{lastM}(v)$. Let $w = \mathit{nextM}(v')$. Since $M(w) = M(v')$ and w is a proper ancestor of v', Lemma 3.2 implies that $B(w) \subseteq B(v')$. Thus, we have $B(w) \subseteq B(u)$. Since the graph is 3-edge-connected, we have $|B(w)| \geq 2$. Notice that the lower endpoint of every back-edge in $B(w)$ is lower than w, and therefore lower than $\mathit{lastM}(v)$. Thus, $B(w) \subseteq B(u)$ implies that $\mathit{low}_2(u) < \mathit{lastM}(v)$, contradicting the fact that $u \in U_1(v)$. This shows that u is a proper descendant of v'.

Now let us suppose, for the sake of contradiction, that $U_1(v') = \emptyset$ and $U_1(v) \neq \emptyset$. Let u be a vertex in $U_1(v)$. Then we have shown that u is a proper descendant of v'. Since $u \in U_1(v)$, we have $u \in S(v)$, and therefore $u \in S(v')$. Furthermore, we have $\mathit{low}_2(u) \geq \mathit{lastM}(v)$. By Lemma 5.74 we have $\mathit{nextM}(v') < \mathit{lastM}(v)$. This implies that $\mathit{low}_2(u) \geq \mathit{nextM}(v')$. Thus, we can consider the lowest proper descendant u' of v' in $S(v')$ that satisfies $\mathit{low}_2(u') \geq \mathit{nextM}(v')$. But then we have $u' \in U_1(v')$, a contradiction. This shows that, if $U_1(v') = \emptyset$, then $U_1(v) = \emptyset$.

Now let us assume that $U_1(v) \neq \emptyset$. This implies that $U_1(v') \neq \emptyset$. Let u be a vertex in $U_1(v)$, and let u' be a vertex in $U_1(v')$. We have shown that u is a proper descendant of v'. Since $u \in U_1(v)$, we have $u \in S(v) = S(v')$. Furthermore, we have

$low_2(u) \geq lastM(v)$. By Lemma 5.74 we have $nextM(v') < lastM(v)$. This implies that $low_2(u) \geq nextM(v')$. Since $u' \in U_1(v')$, we have that either (1) $low_2(u') < nextM(v')$, or (2) u' is the lowest vertex in $S(v')$ that is a proper descendant of v' such that $low_2(u') \geq nextM(v')$. Case (2) implies that $u' \leq u$ (due to the minimality of u'). So let us assume that case (1) is true. Let us suppose, for the sake of contradiction, that $u \leq u'$. Since $u \in S(v)$ and $u' \in S(v')$ and $S(v) = S(v')$, we have that u and u' are related as ancestor and descendant. Thus, $u \leq u'$ implies that u is an ancestor of u'. Furthermore, we have that u and u' have the same *high* point. Thus, Lemma 3.3 implies that $B(u') \subseteq B(u)$. This implies that $low_2(u) \leq low_2(u')$. But we have $low_2(u) \geq nextM(v')$ and $low_2(u') < nextM(v')$, a contradiction. This shows that case (1) implies too that $u' \leq u$. Due to the generality of $u \in U_1(v)$ and $u' \in U_1(v')$, this implies that the lowest vertex in $U_1(v)$ (if it exists) is greater than, or equal to, the greatest vertex in $U_1(v')$. □

Based on Lemma 5.75, we can provide an efficient algorithm for computing the sets $U_1(v)$, for all vertices $v \neq r$ such that $nextM(v) \neq \bot$. The computation takes place on segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant. Specifically, let $v \neq r$ be a vertex such that $nextM(v) \neq \bot$. Then we have that $U_1(v) \subset S(v)$. In other words, $U_1(v)$ is a subset of the segment of $H(high(v))$ that contains v and is maximal w.r.t. the property that its elements are related as ancestor and descendant. So let $z_1, \ldots, z_k$ be the vertices of $S(v)$, sorted in decreasing order. Then, we have that $v = z_i$, for an $i \in \{1, \ldots, k\}$. By definition, $U_1(v)$ contains every vertex u in $\{z_1, \ldots, z_{i-1}\}$ such that either $nextM(v) > low_2(u) \geq lastM(v)$, or u is the lowest vertex in this set such that $low_2(u) \geq nextM(v)$.

As an implication of Lemma 3.3, we have that the vertices in $\{z_1, \ldots, z_{i-1}\}$ are sorted in increasing order w.r.t. their B set, and therefore they are sorted in decreasing order w.r.t. their low_2 point. In other words, we have $B(z_1) \subseteq \cdots \subseteq B(z_{i-1})$, and therefore $low_2(z_1) \geq \cdots \geq low_2(z_{i-1})$. Thus, it is sufficient to process the vertices from $\{z_1, \ldots, z_{i-1}\}$ in reverse order, in order to find the first vertex u that has $low_2(u) \geq lastM(v)$. Then, we keep traversing this set in reverse order, and, as long as the low_2 point of every vertex u that we meet is lower than $nextM(v)$, we insert u into $U_1(v)$. Then, once we reach a vertex with low_2 point no lower than $nextM(v)$, we also insert it into $U_1(v)$, and we are done.

Now, if there is a proper ancestor v' of v in $S(v)$ such that $high(v') = high(v)$, then we have that $S(v) = S(v')$. If $nextM(v') \neq \bot$, then we have that $U_1(v')$ is defined. Then

we can follow the same process as above in order to compute $U_1(v')$. Furthermore, according to Lemma 5.75, it is sufficient to start from the greatest element of $U_1(v)$ (i.e., the one that was inserted last into $U_1(v)$). In particular, if $U_1(v) = \emptyset$, then it is certain that $U_1(v') = \emptyset$, and therefore we are done. Otherwise, we just pick up the computation from the greatest vertex in $U_1(v)$. In order to perform efficiently those computations, we first compute, for every vertex x, the collection $\mathcal{S}(x)$ of the segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant. For every vertex x, this computation takes $O(|H(x)|)$ time using Algorithm 10, according to Lemma 3.22. Since every vertex $\neq r$ participates in exactly one set of the form $H(x)$, we have that the total size of all $\mathcal{S}(x)$, for all vertices x, is $O(n)$. Then it is sufficient to process separately all segments of $\mathcal{S}(x)$, for every vertex x, as described above, by starting the computation each time from the first vertex v of the segment that satisfies $\mathit{nextM}(v) \neq \bot$. The whole procedure is shown in Algorithm 37. The result is formally stated in Lemma 5.76.

Lemma 5.76. *Algorithm 37 correctly computes the collections of vertices $U_1(v)$, for all $v \neq r$ with $\mathit{nextM}(v) \neq \bot$. Furthermore, it has a linear-time implementation.*

Proof. This was discussed in the main text, in the three paragraphs above Algorithm 37. It is easy to see that Algorithm 37 implements precisely the idea that we described in those paragraphs. □

Now we will show how to use the sets U_1 in order to compute all Type-3βii-1 4-cuts.

Corollary 5.12. *Let (u, v, w) and (u, v, w') be two triples of vertices that induce a Type-3βii-1 4-cut. Then $w = w'$.*

Proof. By Lemma 5.72 we have that the 4-cuts induced by (u, v, w) and (u, v, w') have the same back-edge (that is, $(\mathit{lowD}(u), \mathit{low}(u))$). Thus, Lemma 5.2 implies that $w = w'$. □

According to Corollary 5.12, for every $u \in U_1(v)$, where $v \neq r$ is a vertex with $\mathit{nextM}(v) \neq \bot$, there is at most one w such that (u, v, w) induces a Type-3βii-1 4-cut. Thus, the idea is to process all $u \in U_1(v)$, in order to find the w in $M^{-1}(M(v))$ (if it exists) such that (u, v, w) induces a 4-cut.

Given a w such that $M(w) = M(v)$ and $w < v$, the following lemma provides a criterion in order to check whether (u, v, w) induces a 4-cut.

Algorithm 37: Compute the sets $U_1(v)$, for all vertices v such that $nextM(v) \neq \bot$

```
1  foreach vertex x do
2      compute the collection S(x) of the segments of H(x) that are maximal
         w.r.t. the property that their elements are related as ancestor and
         descendant
3  end
4  foreach v ≠ r such that nextM(v) ≠ ⊥ do
5      set U_1(v) ← ∅
6  end
7  foreach vertex x do
8      foreach segment S ∈ S(x) do
9          let v be the first vertex in S
10         while v ≠ ⊥ and nextM(v) = ⊥ do
11             v ← next_S(v)
12         end
13         if v = ⊥ then continue
14         let u ← prev_S(v)
15         while v ≠ ⊥ do
16             while u ≠ ⊥ and low_2(u) < lastM(v) do
17                 u ← prev_S(u)
18             end
19             while u ≠ ⊥ and low_2(u) < nextM(v) do
20                 insert u into U_1(v)
21                 u ← prev_S(u)
22             end
23             if u ≠ ⊥ then
24                 insert u into U_1(v)
25             end
26             v ← next_S(v)
27             while v ≠ ⊥ and nextM(v) = ⊥ do
28                 v ← next_S(v)
29             end
30         end
31     end
32 end
```

Lemma 5.77. *Let u, v, w be three vertices $\neq r$ such that u is a proper descendant of v, v is a proper descendant of w, $high(u) = high(v)$, $M(w) = M(v)$, $w \leq low_2(u)$, $low(u) < w$, and $bcount(v) = bcount(u) + bcount(w) - 1$. Then there is a back-edge $e \in B(u) \cap B(v) \cap B(w)$ such that $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$.*

Proof. Since u is a descendant of v such that $high(u) = high(v)$, Lemma 3.3 implies that $B(u) \subseteq B(v)$. Since w is an ancestor of v such that $M(w) = M(v)$, Lemma 3.2 implies that $B(w) \subseteq B(v)$. Since $w \leq low_2(u)$, we have that there is at most one back-edge in $B(u)$ that may also be in $B(w)$ (i.e., the *low*-edge of u). Let (x, y) be a back-edge in $B(u)$ such that $y = low(u)$. Then x is a descendant of u, and therefore a descendant of v, and therefore a descendant of w. Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of w and y, and therefore w and y are related as ancestor and descendant. Then, $y = low(u) < w$ implies that y is a proper ancestor of w. This shows that $(x, y) \in B(w)$. Therefore, we have that the *low*-edge e of u is in $B(w)$. Furthermore, since $low(u) < w < v$, the same argument shows that $e \in B(v)$. Now, since $B(u) \subseteq B(v)$, $B(w) \subseteq B(v)$, $B(u) \cap B(w) = \{e\}$, $e \in B(v)$ and $bcount(v) = bcount(u) + bcount(w) - 1$, we have that $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$. □

Lemma 5.78. *Let (u, v, w) be a triple of vertices that induces a Type-3βii-1 4-cut. Then w is the greatest proper ancestor of v with $M(w) = M(v)$ and $w \leq low_2(u)$.*

Proof. Let us suppose, for the sake of contradiction, that there is a proper ancestor w' of v such that $M(w') = M(v)$, $w' \leq low_2(u)$, and $w' > w$. Then we have that $M(w') = M(w)$, and so w' is related as ancestor and descendant with w. Since $w' > w$, we have that w' is a proper descendant of w. Thus, Lemma 3.2 implies that $B(w) \subseteq B(w')$. Since the graph is 3-edge-connected, this can be strengthened to $B(w) \subset B(w')$. Similarly, since w' is a proper ancestor of v with $M(w') = M(v)$, we get $B(w') \subset B(v)$. Now, since $B(w) \subset B(w')$, there is a back-edge $(x, y) \in B(w') \setminus B(w)$. Since $B(w') \subset B(v)$, we have that $(x, y) \in B(v)$. Since (u, v, w) induces a Type-3βii-1 4-cut, we have that $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$ and $e \in B(u) \cap B(v) \cap B(w)$, where e is the back-edge of the 4-cut induced by (u, v, w). Thus, $(x, y) \in B(v)$ implies that either (1) $(x, y) = e$, or (2) $(x, y) \in B(u)$, or (3) $(x, y) \in B(w)$. Case (3) is immediately rejected, because $(x, y) \in B(w') \setminus B(w)$. Thus, (1) is also rejected (since $e \in B(w)$). Therefore, only (2) can be true. Then, since $low_2(u) \geq w'$ and $(x, y) \in B(w')$, we have that $(x, y) = (lowD(u), low(u))$. But then Lemma 5.72 implies that $e = (lowD(u), low(u))$, a

contradiction. This shows that w is the greatest proper ancestor of v with $M(w) = M(v)$ and $w \leq low_2(u)$. □

Lemma 5.78 suggests the following algorithm in order to find all Type-3βii-1 4-cuts: for every vertex $v \neq r$ such that $nextM(v) \neq \bot$, and every $u \in U_1(v)$, find the greatest proper ancestor w of v such that $M(w) = M(v)$ and $low_2(u) \geq w$, and then check whether (u, v, w) induces a Type-3βii-1 4-cut using Lemma 5.77. This procedure is shown in Algorithm 38. The proof of correctness and linear complexity is given in Proposition 5.25.

Algorithm 38: Compute all Type-3βii-1 4-cuts

```
1  foreach vertex v ≠ r such that nextM(v) ≠ ⊥ do
2      compute the set U_1(v)
3  end
4  foreach vertex v ≠ r such that nextM(v) ≠ ⊥ do
5      foreach u ∈ U_1(v) do
6          let w be the greatest proper ancestor of v such that w ≤ low_2(u) and
            M(w) = M(v)
7          if low(u) < w and bcount(v) = bcount(u) + bcount(w) − 1 then
8              mark {(u, p(u)), (v, p(v)), (w, p(w)), (lowD(u), low(u))} as a 4-cut
9          end
10     end
11 end
```

Proposition 5.25. *Algorithm 38 computes all Type-3βii-1 4-cuts. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ be a Type-3βii-1 4-cut, where w is a proper ancestor of v, and v is a proper ancestor of u. Then we have $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$, and therefore $bcount(v) = bcount(u) + bcount(w) - 1$. Lemma 5.72 implies that $e = (lowD(u), low(u))$ and $low(u) < w$. Lemma 5.73 implies that $u \in U_1(v)$. Lemma 5.78 implies that w be the greatest proper ancestor of v such that $w \leq low_2(u)$ and $M(w) = M(v)$. Thus, C satisfies all the conditions to be marked in Line 8.

Conversely, let $C = \{(u, p(u)), (v, p(v)), (w, p(w)), (lowD(u), low(u))\}$ be a 4-element set that is marked in Line 8. Since $u \in U_1(v)$, we have that u is a proper descendant

of v with $high(u) = high(v)$. Since w is derived in Line 6, we have that w is a proper ancestor of v with $M(w) = M(v)$ and $w \leq low_2(u)$. Since the condition in Line 7 is satisfied, we have $low(u) < w$ and $bcount(v) = bcount(u) + bcount(w) - 1$. Thus, all the conditions of Lemma 5.77 are satisfied, and therefore we have that there is a back-edge e such that $e \in B(u) \cap B(v) \cap B(w)$ and $B(v) \setminus \{e\} = (B(u) \setminus \{e\}) \sqcup (B(w) \setminus \{e\})$. By the proof of Lemma 5.72, we have $e = (lowD(u), low(u))$ (this result is independent of the condition $M(B(w) \setminus \{e\}) = M(B(v) \setminus \{e\})$ that is implicit in the statement of this lemma). Thus, we have that C is a 4-cut that satisfies (1) of Lemma 5.57.

Now we will argue about the complexity of Algorithm 38. By Lemma 5.76 we have that the sets $U_1(v)$ can be computed in linear time in total, for all vertices $v \neq r$ such that $nextM(v) \neq \bot$. Thus, the **for** loop in Line 1 can be performed in linear time. In order to compute the vertex w in Line 6 we use Algorithm 22. Specifically, whenever we reach Line 6, we generate a query of the form $q(M^{-1}(M(v)), min\{p(v), low_2(u)\})$. This is to return the greatest w in $M^{-1}(M(v))$ such that $w \leq p(v)$ and $w \leq low_2(u)$. Since $M(w) = M(v)$, $w \leq p(v)$ implies that w is a proper ancestor of v. Thus, w is the greatest proper ancestor of v with $M(w) = M(v)$ such that $w \leq low_2(u)$. Since the number of all those queries is $O(n)$, Lemma 5.27 implies that they can be answered in linear time in total, using Algorithm 22. We conclude that Algorithm 38 runs in linear time. $\square$

5.8.2.2 Type-3βii-2 4-cuts

Now we consider case (2) of Lemma 5.57.

Let u, v, w be three vertices $\neq r$ such that w is proper ancestor of v, v is a proper ancestor of u, and there is a back-edge $e \in B(w)$ such that $e \notin B(v) \cup B(u)$, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ and $M(v) = M(B(w) \setminus \{e\})$. By Lemma 5.57, we have that $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ is a 4-cut, and we call this a Type-3βii-2 4-cut.

The following lemma provides some useful information concerning this type of 4-cuts.

Lemma 5.79. *Let u, v, w be three vertices such that (u, v, w) induces a Type-3βii-2 4-cut, and let e be the back-edge of the 4-cut induced by (u, v, w). Then e is either $(L_1(w), l(L_1(w)))$ or $(R_1(w), l(R_1(w)))$. Furthermore, $high(u) = high(v)$, $M(w) \neq M(B(w) \setminus \{e\})$, w is an ancestor of $high(v)$, $low(v) < w$ and $w \leq low(u)$. Finally, if u' is a vertex such that $u \geq u' \geq v$ and $high(u') = high(v)$, then u' is an ancestor of u.*

Proof. Since (u, v, w) induces a Type-3βii-2 4-cut, we have that $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$, $M(v) = M(B(w) \setminus \{e\})$, $e \in B(w)$ and $e \notin B(v) \cup B(u)$. Let us suppose, for the sake of contradiction, that both $(L_1(w), l(L_1(w)))$ and $(R_1(w), l(R_1(w)))$ are back-edges in $B(v)$. Then $nca(L_1(w), R_1(w))$ is a descendant of $M(v)$, which means that $M(w)$ is a descendant of $M(v)$. Since $M(w)$ is an ancestor of $M(B(w) \setminus \{e\})$ and $M(B(w) \setminus \{e\}) = M(v)$, we have that $M(w)$ is an ancestor of $M(v)$. Thus we have that $M(w) = M(v)$. Since w is an ancestor of v, this implies that $B(w) \subseteq B(v)$, which implies that $e \in B(v)$, a contradiction. Thus we have that at least one of $(L_1(w), l(L_1(w)))$ and $(R_1(w), l(R_1(w)))$ is not in $B(v)$. Due to $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$, we have that e is the only back-edge in $B(w)$ that cannot be in $B(v)$, and therefore this coincides with either $(L_1(w), l(L_1(w)))$ or $(R_1(w), l(R_1(w)))$. Observe that in the argument that we used we arrived at a contradiction from $M(w) = M(v)$. Thus, $M(w) \neq M(v)$, and therefore $M(w) \neq M(B(w) \setminus \{e\})$.

Now let (x, y) be a back-edge in $B(u)$. Then $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $(x, y) \in B(v)$, and therefore $y \leq high(v)$. Due to the generality of $(x, y) \in B(u)$, this shows that $high(u) \leq high(v)$. Now let us suppose, for the sake of contradiction, that there is a back-edge (x, y) in $B(v)$ such that $y > high(u)$. This implies that $(x, y) \notin B(u)$. Thus, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $(x, y) \in B(w)$, which implies that $y < w$. Then $y > high(u)$ implies that $high(u) < w$. Now let (x', y') be a back-edge in $B(u)$. Then x' is a descendant of u, and therefore a descendant of v, and therefore a descendant of w. Furthermore, we have $y' \leq high(u) < w$. Since (x', y') is a back-edge, we have that x' is a descendant of y'. Thus, x' is a common descendant of y' and w, and therefore y' and w are related as ancestor and descendant, and therefore $y' < w$ implies that y' is a proper ancestor of w. This shows that $(x', y') \in B(w)$, which is impossible, since $B(u) \cap B(w) = \emptyset$. Thus we have that every back-edge $(x, y) \in B(v)$ has $y \leq high(u)$, and therefore $high(v) \leq high(u)$. This shows that $high(u) = high(v)$.

Since both w and $high(v)$ are ancestors of v, we have that w and $high(v)$ are related as ancestor and descendant. Now let us suppose, for the sake of contradiction, that w is a proper descendant of $high(v)$. Let (x, y) be a back-edge in $B(u)$ such that $y = high(u)$. Then, since $high(u) = high(v)$, we have that $y = high(v)$, and therefore y is a proper ancestor of w. Furthermore, x is a descendant of u, and therefore a descendant of w. Thus we have $(x, y) \in B(w)$, contradicting $B(u) \cap B(w) = \emptyset$. This shows that w is an ancestor of $high(v)$.

Since u is a descendant of w such that $B(u) \cap B(w) = \emptyset$, we have that $low(u) \geq w$.

Since $B(w) \setminus \{e\}$ is not empty, $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that there is a back-edge $(x, y) \in B(v) \cap B(w)$, and therefore $y < w$, and therefore $low(v) < w$.

Now let u' be a vertex such that $u \geq u' \geq v$ and $high(u') = high(v)$. Since u is a descendant of v and $u \geq u' \geq v$, we have that u' is also a descendant of v. Furthermore, since $u \geq u'$, we have that either u' is an ancestor of u, or it is not related as ancestor and descendant with u. Now let us suppose, for the sake of contradiction, that u' is not an ancestor of u. Then u' is not related as ancestor and descendant with u. Let (x, y) be a back-edge in $B(u')$ with $y = high(u')$. Then x is a descendant of u', and therefore a descendant of v. Thus, $y = high(u') = high(v)$ implies that (x, y) is in $B(v)$. Since u' is not related as ancestor and descendant with u, we have that x is not a descendant of u. Thus, $(x, y) \notin B(u)$. Now $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $(x, y) \in B(w) \setminus \{e\}$, and therefore y is a proper ancestor of w. But $y = high(v)$ and $high(v)$ is a descendant of w, a contradiction. This shows that u' is an ancestor of u. ☐

According to Lemma 5.79, if a triple of vertices (u, v, w) induces a Type-3βii-2 4-cut, then the back-edge e of this 4-cut is either $(L_1(w), l(L_1(w)))$ or $(R_1(w), l(R_1(w)))$. In the following we will show how to handle the case where $e = (L_1(w), l(L_1(w)))$. To be specific, we will provide an algorithm that computes a collection of Type-3βii-2 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$, where $e = (L_1(w), l(L_1(w)))$, so that all 4-cuts of this form are implied from this collection, plus that returned by Algorithm 24. The algorithms, the propositions and the arguments for the case $e = (R_1(w), l(R_1(w)))$ are similar. Thus, in this section, for every triple (u, v, w) that we consider that induces a Type-3βii-2 4-cut, we assume that $e = e_L(w)$.

For convenience, we distinguish two cases of Type-3βii-2 4-cuts. First, we have the case where $L_1(w)$ is not a descendant of $high(v)$. In this case, we compute only a sub-collection of the 4-cuts, which, together with the collection of Type-2ii 4-cuts returned by Algorithm 24, implies all the Type-3βii-2 4-cuts of this type (see Proposition 5.26). Then we have the case where $L_1(w)$ is a descendant of $high(v)$. In this case we can compute all those Type-3βii-2 4-cuts in linear time explicitly (see Proposition 5.27).

The case where $L_1(w)$ is not a descendant of $high(v)$

Let (u, v, w) be a triple of vertices that induces a Type-3βii-2 4-cut, and let e be

the back-edge of this 4-cut. Then we have $e \in B(w)$ and $M(B(w) \setminus \{e\}) = M(v)$. Furthermore, by Lemma 5.79 we have that $M(w) \neq M(B(w) \setminus \{e\})$. Now, for every vertex $v \neq r$, let $W(v)$ be the collection of all vertices $w \neq r$ such that: (1) $M(B(w) \setminus \{e_L(w)\}) \neq M(w)$, (2) $M(B(w) \setminus \{e_L(w)\}) = M(v)$, and (3) $L_1(w)$ is not a descendant of *high*(v). In particular, if $W(v) \neq \emptyset$, then we define *firstW*(v) = *max*$(W(v))$ and *lastW*(v) = *min*$(W(v))$.

Lemma 5.80. *Let v and w be two vertices such that $w \in W(v)$. Then w is a proper ancestor of high(v).*

Proof. Since $w \in W(v)$, we have that $M(B(w) \setminus \{e_L(w)\}) = M(v)$. Since the graph is 3-edge-connected, there is a back-edge $(x, y) \in B(w) \setminus \{e_L(w)\}$. Thus, x is a descendant of $M(B(w) \setminus \{e_L(w)\}) = M(v)$, and therefore a descendant of v, and therefore a descendant of *high*(v). Since $(x, y) \in B(w)$, we have that x is a descendant of w. Thus, x is a common descendant of *high*(v) and w, and therefore *high*(v) and w are related as ancestor and descendant.

Now let us suppose, for the sake of contradiction, that w is not a proper ancestor of *high*(v). Thus, we have that w is a descendant of *high*(v). Then, since $L_1(w)$ is a descendant of w, we have that $L_1(w)$ is a descendant of *high*(v), in contradiction to the fact that $w \in W(v)$. This shows that w is a proper ancestor of *high*(v). □

We will show how to compute the values *firstW*(v) and *lastW*(v), for all vertices v. With those values we can determine in constant time whether $W(v) \neq \emptyset$, for a vertex v, by simply checking whether *firstW*$(v) \neq \bot$. First, for every vertex x, we let $W_0(x)$ denote the list of all vertices $w \neq r$ with $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = x$, sorted in decreasing order. Notice that, for every vertex $v \neq r$, we have $W(v) \subseteq W_0(M(v))$. Now we have the following.

Lemma 5.81. *Let v be a vertex such that $W(v) \neq \emptyset$. Then lastW(v) is the last entry in $W_0(M(v))$.*

Proof. Let w = *lastW*(v), and let w' be the last entry in $W_0(M(v))$. Thus, we have $w' \leq w$. Let us suppose, for the sake of contradiction, that $w' \neq w$. Since w = *lastW*(v), we have that w is the lowest vertex in $W_0(M(v))$ such that $L_1(w)$ is not a descendant of *high*(v). Thus, since $w' \neq w$, we have that $L_1(w')$ is a descendant of *high*(v). Since $w \in W_0(M(v))$, we have that $M(v) = M(B(w) \setminus \{e_L(w)\}) \neq M(w)$. Thus, $e_L(w)$ is the only back-edge in $B(w)$ whose higher endpoint (i.e., $L_1(w)$) is not a descendant

of $M(v)$. Similarly, since $w' \in W_0(M(v))$, we have that $e_L(w')$ is the only back-edge in $B(w')$ whose higher endpoint (i.e., $L_1(w')$) is not a descendant of $M(v)$. Notice that $M(B(w) \setminus \{e_L(w)\}) = M(B(w') \setminus \{e_L(w')\}) = M(v)$. This implies that $M(v)$ is a common descendant of w and w', and therefore w and w' are related as ancestor and descendant. Thus, $w' \leq w$ implies that w' is an ancestor of w.

Since $w = \mathit{lastW}(v)$, by Lemma 5.80 we have that w is a proper ancestor of $\mathit{high}(v)$. Now let $(x, y) = e_L(w')$. This implies that $x = L_1(w')$. Then we have that x is a descendant of $\mathit{high}(v)$, and therefore a descendant of w. Furthermore, y is a proper ancestor of w', and therefore a proper ancestor of w. This shows that $(x, y) \in B(w)$. Since $(x, y) = e_L(w')$, we have that x is not a descendant of $M(v)$. Thus, since (x, y) is a back-edge in $B(w)$ such that x is not a descendant of $M(v)$, we have that $(x, y) = e_L(w)$. But this contradicts the fact that $L_1(w)$ is not a descendant of $\mathit{high}(v)$. We conclude that $w' = w$. □

Lemma 5.82. *Let v and v' be two vertices with $M(v') = M(v)$ such that v' is a proper ancestor of v. Suppose that $\mathit{firstW}(v') \neq \perp$. Then $\mathit{firstW}(v) \neq \perp$, and $\mathit{firstW}(v') \leq \mathit{firstW}(v)$.*

Proof. Let $w = \mathit{firstW}(v')$. Then we have that $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = M(v')$ and $L_1(w)$ is not a descendant of $\mathit{high}(v')$. Since $v' \neq v$ and the graph is 3-edge-connected, we have that $B(v) \neq B(v')$. Thus, since v is a proper descendant of v' with $M(v) = M(v')$, Lemma 3.6 implies that v' is an ancestor of $\mathit{high}(v)$. Since $\mathit{high}(v')$ is a proper ancestor of v', this implies that $\mathit{high}(v')$ is a proper ancestor of $\mathit{high}(v)$. Thus, since $L_1(w)$ is not a descendant of $\mathit{high}(v')$, we have that $L_1(w)$ is not a descendant of $\mathit{high}(v)$. This shows that $w \in W(v)$, and therefore $\mathit{firstW}(v) \neq \perp$ and $\mathit{firstW}(v) \geq w$. □

Using the information provided by Lemmata 5.81 and 5.82, we can provide an efficient algorithm for computing the values $\mathit{firstW}(v)$ and $\mathit{lastW}(v)$, for all vertices v. First, for every vertex x, we collect all vertices $w \neq r$ such that $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = x$ into a list $W_0(x)$, and we have $W_0(x)$ sorted in decreasing order. The computation of the values $M(B(w) \setminus \{e_L(w)\})$, for all $w \neq r$, takes linear time in total, according to Proposition 3.6. Then, the construction of the lists $W_0(x)$ takes $O(n)$ time in total, using bucket-sort. Now, for every vertex $v \neq r$, it is sufficient to check the last entry w in $W_0(M(v))$ in order to see if $w = \mathit{lastW}(v)$, according to Lemma 5.81. Since $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = M(v)$, we have that $w \in W(v)$ if and only if $L_1(w)$ is not a descendant of $\mathit{high}(v)$. So this can be easily checked in constant time.

In order to compute the values *firstW*, we traverse the lists $M^{-1}(x)$ and $W_0(x)$ simultaneously, for every vertex x. To be specific, let $v \neq r$ be a vertex. Then, if $w = \mathit{firstW}(v)$ exists, we have that $w \in W_0(M(v))$. (Notice that v itself lies in $M^{-1}(M(v))$.) Then, by Lemma 5.80 we have that w is a proper ancestor of $\mathit{high}(v)$, and therefore a proper ancestor of v. Thus, it is sufficient to reach the greatest w' in $W_0(M(v))$ that is a proper ancestor of v. Then, as long as w' does not satisfy the property that $L_1(w')$ is not a descendant of $\mathit{high}(v)$, we keep traversing the list $W_0(M(v))$. Eventually we will reach w. Now, if there is a proper ancestor v' of v with $M(v') = M(v)$ such that $\mathit{firstW}(v') \neq \bot$, then by Lemma 5.82 we have that $\mathit{firstW}(v') \leq \mathit{firstW}(v)$. Thus, it is sufficient to pick up the search for $\mathit{firstW}(v')$ in $W_0(M(v'))$ from the last entry in $W_0(M(v'))$ that we accessed. This procedure for computing the values $\mathit{firstW}(v)$ and $\mathit{lastW}(v)$, for all vertices v, is shown in Algorithm 39. Our result is summarized in Lemma 5.83.

Lemma 5.83. *Algorithm 39 correctly computes the values* $\mathit{firstW}(v)$ *and* $\mathit{lastW}(v)$*, for all vertices* v*, in total linear time. If for a vertex* v *we have* $W(v) = \emptyset$*, then* $\mathit{firstW}(v) = \bot$*.*

Recall that, for every vertex x, we let $H(x)$ denote the list of all vertices $v \neq r$ such that $\mathit{high}(v) = x$, sorted in decreasing order. Then, for every vertex $v \neq r$, we let $S(v)$ denote the segment of $H(\mathit{high}(v))$ that contains v and is maximal w.r.t. the property that its elements are related as ancestor and descendant. Furthermore, we let $U(v)$ denote the subsegment of $S(v)$ that contains all the proper descendants of v. Now, for every vertex v such that $W(v) \neq \emptyset$, we let $U_2(v)$ be the collection of all $u \in U(v)$ such that: either (1) $\mathit{firstW}(v) > \mathit{low}(u) \geq \mathit{lastW}(v)$, or (2) u is the lowest vertex in $U(v)$ such that $\mathit{low}(u) \geq \mathit{firstW}(v)$.

Lemma 5.84. *Let* (u, v, w) *be a triple of vertices that induces a Type-3βii-2 4-cut, where* $L_1(w)$ *is not a descendant of* $\mathit{high}(v)$*. Then* $u \in U_2(v)$ *and* $w \in W(v)$*.*

Proof. Let e be the back-edge in the 4-cut induced by (u, v, w). Due to our assumption in this subsection, we have that $e = e_L(w)$. Furthermore, we have $M(B(w) \setminus \{e\}) = M(v)$, and Lemma 5.79 implies that $M(w) \neq M(B(w) \setminus \{e\})$. Thus, since $L_1(w)$ is not a descendant of $\mathit{high}(v)$, w satisfies all the conditions to be in $W(v)$.

Since (u, v, w) induces a Type-3βii-2 4-cut, Lemma 5.79 implies that $\mathit{high}(u) = \mathit{high}(v)$. In other words, we have $u \in H(\mathit{high}(v))$. Now let u' be a vertex such that $u \geq u' \geq v$ and $u' \in H(\mathit{high}(v))$. This means that we have $\mathit{high}(u') = \mathit{high}(v)$, and

Algorithm 39: Compute the values *firstW*(v) and *lastW*(v), for all vertices $v \neq r$

```
1  foreach vertex w ≠ r do
2  |  compute the value M(B(w) \ {e_L(w)})
3  end
4  foreach vertex x do
5  |  let W_0(x) be the list of all vertices w ≠ r with
   |    M(w) ≠ M(B(w) \ {e_L(w)}) = x, sorted in decreasing order
6  |  let M^{-1}(x) be the list of all vertices v ≠ r with M(v) = x, sorted in
   |    decreasing order
7  end
8  foreach vertex v do
9  |  let firstW(v) ← ⊥ and lastW(v) ← ⊥
10 end
11 foreach vertex v ≠ r do
12 |  let w be the last entry in W_0(M(v))
13 |  if L_1(w) is not a descendant of high(v) then
14 |  |  set lastW(v) ← w
15 |  end
16 end
17 foreach vertex x do
18 |  let v be the first entry in M^{-1}(x)
19 |  let w be the first entry in W_0(x)
20 |  while v ≠ ⊥ do
21 |  |  while w ≠ ⊥ and w ≥ v do
22 |  |  |  w ← next_{W_0(x)}(w)
23 |  |  end
24 |  |  while w ≠ ⊥ and L_1(w) is a descendant of high(v) do
25 |  |  |  w ← next_{W_0(x)}(w)
26 |  |  end
27 |  |  if w ≠ ⊥ then
28 |  |  |  set firstW(v) ← w
29 |  |  end
30 |  |  v ← nextM(v)
31 |  end
32 end
```

therefore Lemma 5.79 implies that u' is an ancestor of u. This shows that $u \in S(v)$. Since u is a proper descendant of v, this implies that $u \in U(v)$.

Since $w \in W(v)$, we have that $w \geq \mathit{lastW}(v)$. By Lemma 5.79 we have $w \leq \mathit{low}(u)$. Thus, $w \geq \mathit{lastW}(v)$ implies that $\mathit{low}(u) \geq \mathit{lastW}(v)$. If $\mathit{low}(u) < \mathit{firstW}(v)$, then by definition we have $u \in U_2(v)$, and the proof is complete. Otherwise, we have $\mathit{low}(u) \geq \mathit{firstW}(v)$. Let us suppose, for the sake of contradiction, that there is a vertex $u' \in U(v)$ that is lower than u and satisfies $\mathit{low}(u') \geq \mathit{firstW}(v)$. Since $u' \in U(v)$, we have that u' is a proper descendant of v, and so we have $u' > v$. Furthermore, since $u' \in U(v)$, we have that $\mathit{high}(u') = \mathit{high}(v)$. Then, since $\mathit{high}(u') = \mathit{high}(v)$ and $u > u' > v$, by Lemma 5.79 we have that u' is an ancestor of u. Thus, since $\mathit{high}(u') = \mathit{high}(v) = \mathit{high}(u)$, by Lemma 3.3 we have that $B(u) \subseteq B(u')$. Since the graph is 3-edge-connected, this can be strengthened to $B(u) \subset B(u')$. Thus, there is a back-edge $(x, y) \in B(u') \setminus B(u)$. Then x is a descendant of u', and therefore a descendant of v. Furthermore, y is an ancestor of $\mathit{high}(u') = \mathit{high}(v)$, and therefore it is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Then, since (u, v, w) induces a Type-3βii-2 4-cut, we have that $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$. Since $(x, y) \notin B(u)$, this implies that $(x, y) \in B(w) \setminus \{e\}$. Since $w \in W(v)$, we have that $w \leq \mathit{firstW}(v)$. But we have supposed that $\mathit{low}(u') \geq \mathit{firstW}(v)$. This implies that $\mathit{low}(u') \geq w$, which implies that $y \geq w$ (since $(x, y) \in B(u')$, and therefore $\mathit{low}(u') \leq y$). This means that y cannot be a proper ancestor of w, in contradiction to the fact that $(x, y) \in B(w)$. This shows that u is the lowest vertex in $U(v)$ that has $\mathit{low}(u) \geq \mathit{firstW}(v)$. Thus, by definition, we have $u \in U_2(v)$. □

Lemma 5.85. *Let (u, v, w) be a triple of vertices that induces a Type-3βii-2 4-cut, where $L_1(w)$ is not a descendant of $\mathit{high}(v)$. Let w' be the greatest vertex in $W(v)$ that has $w' \leq \mathit{low}(u)$. Then (u, v, w') also induces a Type-3βii-2 4-cut. Furthermore, if $w' \neq w$, then $B(w) \sqcup \{e_L(w')\} = B(w') \sqcup \{e_L(w)\}$.*

Proof. By the assumption throughout this subsection, we have that the back-edge in the 4-cut induced by (u, v, w) is $e = e_L(w)$. By Lemma 5.84 we have that $w \in W(v)$, and by Lemma 5.79 we have that $w \leq \mathit{low}(u)$. Thus, we may consider the greatest vertex w' in $W(v)$ that has $w' \leq \mathit{low}(u)$. We will assume that $w' \neq w$, because otherwise there is nothing to show.

Let (x, y) be a back-edge in $B(w') \setminus \{e_L(w')\}$. Since $w' \in W(v)$, we have $M(B(w') \setminus \{e_L(w')\}) = M(v)$. This implies that x is a descendant of $M(v)$. Furthermore, since

$w' \in W(v)$, by Lemma 5.80 we have that w' is a proper ancestor of $high(v)$, and therefore a proper ancestor of v. This implies that y is a proper ancestor of v. Thus we have that $(x, y) \in B(v)$. Since $w' \leq low(u)$, we have that $B(u) \cap B(w') = \emptyset$. Thus, we have that $(x, y) \notin B(u)$, and therefore $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ implies that $(x, y) \in B(w) \setminus \{e\}$. Due to the generality of $(x, y) \in B(w') \setminus \{e_L(w')\}$, this shows that $B(w') \setminus \{e_L(w')\} \subseteq B(w) \setminus \{e_L(w)\}$. Conversely, let (x, y) be a back-edge in $B(w) \setminus \{e\}$. Since $w \in W(v)$, we have that $M(B(w) \setminus \{e\}) = M(v)$. This implies that x is a descendant of $M(v)$, and therefore a descendant of v, and therefore a descendant of $high(v)$. Since $w' \in W(v)$, by Lemma 5.80 we have that w' is a proper ancestor of $high(v)$. Thus, we have that x is a descendant of w'. Since w and w' are both in $W(v)$, Lemma 5.80 implies that they are both proper ancestors of $high(v)$. Thus, w and w' are related as ancestor and descendant. Due to the maximality of w', we have that $w' > w$, and therefore w' is a proper descendant of w. Then, since $(x, y) \in B(w)$, we have that y is a proper ancestor of w, and therefore a proper ancestor of w'. Since x is a descendant of w', this shows that $(x, y) \in B(w')$. Since $w' \in W(v)$, we have that the higher endpoint of $e_L(w')$ is not a descendant of $high(v)$. Thus, since x is a descendant of $high(v)$, we have that $(x, y) \neq e_L(w')$, and therefore $(x, y) \in B(w') \setminus \{e_L(w')\}$. Thus we have shown that $B(w') \setminus \{e_L(w')\} = B(w) \setminus \{e\}$. This implies that $B(v) = B(u) \sqcup (B(w) \setminus \{e\})$ is equivalent to $B(v) = B(u) \sqcup (B(w') \setminus \{e_L(w')\})$, and therefore, by Lemma 5.57, (u, v, w') induces a 4-cut. By definition, this is a Type-3βii-2 4-cut.

Let us suppose, for the sake of contradiction, that $e_L(w') = e_L(w)$. Then, since $B(w') \setminus \{e_L(w')\} = B(w) \setminus \{e\}$, we have that $B(w') = B(w)$, in contradiction to the fact that the graph is 3-edge-connected. Thus, we have $e_L(w') \neq e_L(w)$. Therefore, $B(w') \setminus \{e_L(w')\} = B(w) \setminus \{e\}$ implies that $B(w) \sqcup \{e_L(w')\} = B(w') \sqcup \{e_L(w)\}$. $\square$

Lemma 5.86. *Let v and v' be two vertices with $W(v) \neq \emptyset$ and $W(v') \neq \emptyset$ such that v is a proper ancestor of v' and $high(v) = high(v')$. Then $lastW(v) > firstW(v')$.*

Proof. Let w and w' be two vertices such that $w \in W(v)$ and $w' \in W(v')$. Then it is sufficient to show that $w > w'$ (because $lastW(v) = min(W(v))$ and $firstW(v') = max(W(v'))$). So let us suppose, for the sake of contradiction, that $w \leq w'$. Since v is a proper ancestor of v' with $high(v) = high(v')$, Lemma 3.3 implies that $B(v') \subseteq B(v)$. This implies that $M(v')$ is a descendant of $M(v)$. But we cannot have that $M(v') = M(v)$, because the graph is 3-edge-connected (and otherwise, $high(v) = high(v')$ would imply $B(v) = B(v')$, by Lemma 3.7). Thus, $M(v')$ is a proper descendant of $M(v)$.

Notice that we cannot have $w = w'$, because $w \in W(v)$ and $w' \in W(v')$ imply that $M(B(w) \setminus \{e_L(w)\}) = M(v)$ and $M(B(w') \setminus \{e_L(w')\}) = M(v')$ (and we showed that we cannot have $M(v) = M(v')$). Thus, we have $w < w'$. Since $w \in W(v)$, by Lemma 5.80 we have that w is an ancestor of $high(v)$, and therefore a proper ancestor of v, and therefore a proper ancestor of v'. And since $w' \in W(v')$, by Lemma 5.80 we have that w' is an ancestor of $high(v')$, and therefore a proper ancestor of v'. Thus, v' is a common descendant of w and w', and therefore w and w' are related as ancestor and descendant. Thus, $w < w'$ implies that w is a proper ancestor of w'.

Since $w \in W(v)$, we have that $M(B(w) \setminus \{e_L(w)\}) = M(v)$. Let us suppose, for the sake of contradiction, that all back-edges in $B(w) \setminus \{e_L(w)\}$ have their higher endpoint in $T(M(v'))$. (We note that $B(w) \setminus \{e_L(w)\}$ is not empty, since the graph is 3-edge-connected.) Then we have that $M(v')$ is an ancestor of $M(B(w) \setminus \{e_L(w)\}) = M(v)$, contradicting the fact that $M(v')$ is a proper descendant of $M(v)$. This shows that there is at least one back-edge $(x, y) \in B(w) \setminus \{e_L(w)\}$ such that x is not a descendant of $M(v')$. Since $M(B(w) \setminus \{e_L(w)\}) = M(v)$, we have that x is a descendant of $M(v)$. Therefore, x is a descendant of v, and therefore a descendant of $high(v) = high(v')$. Thus, it cannot be the case that $(x, y) = e_L(w')$ (because $w' \in W(v')$ implies that $L_1(w')$ is not a descendant of $high(v')$). Now, since x is a descendant of $high(v')$, and $high(v')$ is a descendant of w' (by Lemma 5.80), we have that x is a descendant of w'. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of w'. This shows that $(x, y) \in B(w')$. But since $(x, y) \neq e_L(w')$, we have that $(x, y) \in B(w') \setminus \{e_L(w')\}$. Thus, since $w' \in W(v')$, we have that $M(B(w') \setminus \{e_L(w')\}) = M(v')$, and therefore x is a descendant of $M(v')$, contradicting the fact that x is not a descendant of $M(v')$. We conclude that $w > w'$. Due to the generality of $w \in W(v)$ and $w' \in W(v')$, this implies that $lastW(v) > firstW(v')$. ☐

Lemma 5.87. *Let v and v' be two vertices with $W(v) \neq \emptyset$ and $W(v') \neq \emptyset$ such that v' is a proper descendant of v. Suppose that v and v' belong to the same segment S of $H(high(v))$ that is maximal w.r.t. the property that its elements are related as ancestor and descendant. If $U_2(v') = \emptyset$, then $U_2(v) = \emptyset$. If $U_2(v) \neq \emptyset$, then the lowest vertex in $U_2(v)$ is at least as great as the greatest vertex in $U_2(v')$.*

Proof. Let us suppose, for the sake of contradiction, that there is a vertex $u \in U_2(v)$, but $U_2(v')$ is empty. Since $u \in U_2(v)$ we have that $u \in S$, and therefore u is related as ancestor and descendant with v'. Let us suppose, for the sake of contradiction,

that u is an ancestor of v'. Since $W(v') \neq \emptyset$, there is a vertex $w \in W(v')$. This implies that $M(B(w) \setminus \{e_L(w)\}) = M(v')$. Since the graph is 3-edge-connected, we have that $|B(w)| > 1$. Thus, there is a back-edge $(x, y) \in B(w) \setminus \{e_L(w)\}$. Then we have that x is a descendant of $M(B(w) \setminus \{e_L(w)\})$, and therefore a descendant of $M(v')$, and therefore a descendant of v', and therefore a descendant of u. Furthermore, we have that y is a proper ancestor of w. Since $w \in W(v')$, by Lemma 5.80 we have that w is an ancestor of $\mathit{high}(v') = \mathit{high}(v)$. Since $u \in U_2(v)$, we have that $\mathit{high}(u) = \mathit{high}(v)$. Thus, we have that y is a proper ancestor of w, which is an ancestor of $\mathit{high}(u)$, which is a proper ancestor of u. Since x is a descendant of u, this shows that $(x, y) \in B(u)$. Therefore, $\mathit{low}(u) \leq y$. Since $(x, y) \in B(w)$, we have that y is a proper ancestor of w, and therefore $y < w$. Thus, $\mathit{low}(u) \leq y$ implies that $\mathit{low}(u) < w$. Now, Lemma 5.86 implies that $\mathit{firstW}(v') < \mathit{lastW}(v)$. Thus, since $w \leq \mathit{firstW}(v')$, we have that $w < \mathit{lastW}(v)$. But then $\mathit{low}(u) < w$ implies that $\mathit{low}(u) < \mathit{lastW}(v)$, in contradiction to $u \in U_2(v)$. Thus, our last supposition cannot be true, and therefore we have that u is a proper descendant of v'.

Now let us gather the information we have concerning u. We know that u is a proper descendant of v', and it belongs to S. Furthermore, since $u \in U_2(v)$ we have that $\mathit{low}(u) \geq \mathit{lastW}(v)$, and by Lemma 5.86 we have that $\mathit{lastW}(v) > \mathit{firstW}(v')$. This implies that $\mathit{low}(u) \geq \mathit{firstW}(v')$. Thus, we can consider the lowest proper descendant u' of v' in S that has $\mathit{low}(u') \geq \mathit{firstW}(v')$, and so $U_2(v')$ cannot be empty. A contradiction. Thus, we have shown that $U_2(v') = \emptyset$ implies that $U_2(v) = \emptyset$.

Now let us assume that $U_2(v) \neq \emptyset$. This implies that $U_2(v') \neq \emptyset$. Let u be any vertex in $U_2(v)$, and let u' be any vertex in $U_2(v')$. Let us suppose, for the sake of contradiction, that $u < u'$. Since $u \in U_2(v)$, we have that $u \in S$. And since $u' \in U_2(v')$, we have that $u' \in S$. Thus, u and u' are related as ancestor and descendant. Therefore, $u < u'$ implies that u is a proper ancestor of u'. Since $u' \in U_2(v')$, we have that either $\mathit{low}(u') < \mathit{firstW}(v')$, or u' is the lowest proper descendant of v' in S such that $\mathit{low}(u') \geq \mathit{firstW}(v')$.

Let us suppose, first, that $\mathit{low}(u') < \mathit{firstW}(v')$. Since u is an ancestor of u' with $\mathit{high}(u) = \mathit{high}(u')$ (since u and u' are in S), by Lemma 3.3 we have that $B(u') \subseteq B(u)$. This implies that $\mathit{low}(u) \leq \mathit{low}(u')$. Therefore, $\mathit{low}(u') < \mathit{firstW}(v')$ implies that $\mathit{low}(u) < \mathit{firstW}(v')$. Lemma 5.86 implies that $\mathit{firstW}(v') < \mathit{lastW}(v)$. Therefore, we have $\mathit{low}(u) < \mathit{lastW}(v)$, in contradiction to the fact that $u \in U_2(v)$. Thus, our last supposition cannot be true, and therefore we have that u' is the lowest proper de-

scendant of v' in S such that $low(u') \geq firstW(v')$. Now, since $u \in U_2(v)$, we can argue as above in order to establish that u is a proper descendant of v' (the argument above did not make use of the assumption $U_2(v') = \emptyset$, and so it can be applied here too). But then, since u is a proper descendant of v' in S such that $u < u'$, the minimality of u' implies that $low(u) < firstW(v')$, and so we can arrive again at $low(u) < lastW(v)$, in contradiction to $u \in U_2(v)$. This shows that $u \geq u'$. Due to the generality of $u \in U_2(v)$, this implies that the lowest vertex in $U_2(v)$ is at least as great as u'. And due to the generality of $u' \in U_2(v')$, this implies that the lowest vertex in $U_2(v)$ is at least as great as the greatest vertex in $U_2(v')$. □

Based on Lemma 5.87, we can provide an efficient algorithm for computing the sets $U_2(v)$, for all vertices $v \neq r$ such that $W(v) \neq \emptyset$. The computation takes place on segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant. Specifically, let $v \neq r$ be a vertex such that $W(v) \neq \emptyset$. Then we have that $U_2(v) \subset S(v)$. In other words, $U_2(v)$ is a subset of the segment of $H(high(v))$ that contains v and is maximal w.r.t. the property that its elements are related as ancestor and descendant. So let $z_1, \dots, z_k$ be the vertices of $S(v)$, sorted in decreasing order. Then, we have that $v = z_i$, for an $i \in \{1, \dots, k\}$. By definition, $U_2(v)$ contains every vertex u in $\{z_1, \dots, z_{i-1}\}$ such that either $firstW(v) > low(u) \geq lastW(v)$, or u is the lowest vertex in this set such that $low(u) \geq firstW(v)$. As an implication of Lemma 3.5, we have that the vertices in $\{z_1, \dots, z_{i-1}\}$ are sorted in decreasing order w.r.t. their *low* point. Thus, it is sufficient to process the vertices from $\{z_1, \dots, z_{i-1}\}$ in reverse order, in order to find the first vertex u that has $low(u) \geq lastW(v)$. Then, we keep traversing this set in reverse order, and, as long as the *low* point of every vertex u that we meet is lower than $firstW(v)$, we insert u into $U_2(v)$. Then, once we reach a vertex with *low* point no lower than $firstW(v)$, we also insert it into $U_2(v)$, and we are done.

Now, if there is a proper ancestor v' of v in $S(v)$ such that $high(v') = high(v)$, then we have that $S(v) = S(v')$. If $W(v') \neq \emptyset$, then we have that $U_2(v')$ is defined. Then we can follow the same process as above in order to compute $U_2(v')$. Furthermore, according to Lemma 5.87, it is sufficient to start from the greatest element of $U_2(v)$ (i.e., the one that was inserted last into $U_2(v)$). In particular, if $U_2(v) = \emptyset$, then it is certain that $U_2(v') = \emptyset$, and therefore we are done. Otherwise, we just pick up the computation from the greatest vertex in $U_2(v)$. In order to perform efficiently

those computations, first we compute, for every vertex x, the collection $\mathcal{S}(x)$ of the segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant. For every vertex x, this computation takes $O(|H(x)|)$ time using Algorithm 10, according to Lemma 3.22. Since every vertex $\neq r$ participates in exactly one set of the form $H(x)$, we have that the total size of all $\mathcal{S}(x)$, for all vertices x, is $O(n)$. Then it is sufficient to process separately all segments of $\mathcal{S}(x)$, for every vertex x, as described above, by starting the computation each time from the first vertex v of the segment that satisfies $W(v) \neq \emptyset$. The whole procedure is shown in Algorithm 40. The result is formally stated in Lemma 5.88.

Lemma 5.88. *Algorithm 40 correctly computes the sets $U_2(v)$, for all vertices $v \neq r$ such that $W(v) \neq \emptyset$. Furthermore, it runs in linear time.*

Proof. The idea behind Algorithm 40 and its correctness has been discussed in the main text above. It is easy to see that Algorithm 40 runs in $O(n)$ time, provided that we have computed the vertices $\mathit{firstW}(v)$ and $\mathit{lastW}(v)$ for every vertex v (if they exist). This can be achieved in linear time according to Lemma 5.83. Thus, Algorithm 40 has a linear-time implementation. □

Lemma 5.89. *Let u, v, w be three vertices such that $u \in U_2(v)$, $w \in W(v)$, $w \leq \mathit{low}(u)$, and $\mathit{bcount}(v) = \mathit{bcount}(u) + \mathit{bcount}(w) - 1$. Then, $e_L(w) \notin B(u) \cup B(v)$, and $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$.*

Proof. Since $u \in U_2(v)$ we have that u is a proper descendant of v with $\mathit{high}(u) = \mathit{high}(v)$. Thus, Lemma 3.3 implies that $B(u) \subseteq B(v)$. Since $w \in W(v)$, we have that $M(B(w) \setminus \{e_L(w)\}) = M(v)$. Now let (x, y) be a back-edge in $B(w) \setminus \{e_L(w)\}$. Then, x is a descendant of $M(B(w) \setminus \{e_L(w)\}) = M(v)$. Furthermore, y is a proper ancestor of w. Since $w \in W(v)$, by Lemma 5.80 we have that w is an ancestor of $\mathit{high}(v)$. This implies that y is a proper ancestor of $\mathit{high}(v)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(w) \setminus \{e_L(w)\}$, this implies that $B(w) \setminus \{e_L(w)\} \subseteq B(v)$. Let (x, y) be a back-edge in $B(u)$. Then $\mathit{low}(u) \leq y$, and therefore $w \leq \mathit{low}(u)$ implies that $w \leq y$. Thus, y cannot be a proper ancestor of w, and therefore $(x, y) \notin B(w)$. Due to the generality of $(x, y) \in B(u)$, this shows that $B(u) \cap B(w) = \emptyset$. Now, since $B(u) \subseteq B(v)$ and $B(w) \setminus \{e_L(w)\} \subseteq B(v)$ and $B(u) \cap B(w) = \emptyset$ and $\mathit{bcount}(v) = \mathit{bcount}(u) + \mathit{bcount}(w) - 1$, we have that $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$. Furthermore, since $B(u) \cap B(w) = \emptyset$, we have that

Algorithm 40: Compute the sets $U_2(v)$, for all vertices v such that $W(v) \neq \emptyset$

```
1  let 𝒱 be the collection of all vertices v such that W(v) ≠ ∅
2  foreach vertex x do
3      compute the collection 𝒮(x) of the segments of H(x) that are maximal
         w.r.t. the property that their elements are related as ancestor and
         descendant
4  end
5  foreach v ∈ 𝒱 do
6      set U_2(v) ← ∅
7  end
8  foreach vertex x do
9      foreach segment S ∈ 𝒮(x) do
10         let v be the first vertex in S
11         while v ≠ ⊥ and v ∉ 𝒱 do
12             v ← next_S(v)
13         end
14         if v = ⊥ then continue
15         let u ← prev_S(v)
16         while v ≠ ⊥ do
17             while u ≠ ⊥ and low(u) < lastW(v) do
18                 u ← prev_S(u)
19             end
20             while u ≠ ⊥ and low(u) < firstW(v) do
21                 insert u into U_2(v)
22                 u ← prev_S(u)
23             end
24             if u ≠ ⊥ then
25                 insert u into U_2(v)
26             end
27             v ← next_S(v)
28             while v ≠ ⊥ and v ∉ 𝒱 do
29                 v ← next_S(v)
30             end
31         end
32     end
33 end
```

$e_L(w) \notin B(u)$, and therefore $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$ implies that $e_L(w) \notin B(v)$. Thus, we have $e_L(w) \notin B(u) \cup B(v)$. $\square$

Let $\mathcal{C}_{3\beta ii2}$ denote the collection of all Type-$3\beta ii$-2 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e_L(w)\}$, such that u is a descendant of v, v is a descendant of w, and $L_1(w)$ is not a descendant of $\mathit{high}(v)$. Now we are ready to describe the algorithm for computing a collection $\mathcal{C}$ of enough 4-cuts in $\mathcal{C}_{3\beta ii2}$, so that all 4-cuts in $\mathcal{C}_{3\beta ii2}$ are implied from this collection, plus that computed by Algorithm 24. So let (u, v, w) be a triple of vertices that induces a 4-cut $C \in \mathcal{C}_{3\beta ii2}$. Then, by Lemma 5.84 we have that $u \in U_2(v)$ and $w \in W(v)$, and by Lemma 5.79 we have that $w \leq \mathit{low}(u)$. Now let w' be the greatest vertex in $W(v)$ that satisfies $w' \leq \mathit{low}(u)$. Then, by Lemma 5.85 we have that (u, v, w') also induces a 4-cut $C' \in \mathcal{C}_{3\beta ii2}$. Furthermore, if $w' \neq w$, then Lemma 5.85 implies that $B(w) \sqcup \{e_L(w')\} = B(w') \sqcup \{e_L(w)\}$. Thus, we have that C is implied by C', plus some Type-$2ii$ 4-cuts that are computed by Algorithm 24 (see Proposition 5.26). Thus, it is sufficient to have computed, for every vertex v such that $W(v) \neq \emptyset$, and every $u \in U_2(v)$, the greatest proper ancestor w of v that satisfies $w \leq \mathit{low}(u)$, and then check if the triple (u, v, w) induces a Type-$3\beta ii$-2 4-cut. This procedure is shown in Algorithm 41. Our result is summarized in Proposition 5.26.

Proposition 5.26. *Algorithm 41 computes a collection of 4-cuts $\mathcal{C} \subseteq \mathcal{C}_{3\beta ii2}$, and it runs in linear time. Furthermore, let $\mathcal{C}'$ be the collection of Type-$2ii$ 4-cuts computed by Algorithm 24. Then, every 4-cut in $\mathcal{C}_{3\beta ii2}$ is implied by $\mathcal{C} \cup \mathcal{C}'$.*

Proof. Let $\{(u, p(u)), (v, p(v)), (w, p(w)), e_L(w)\}$ be a 4-set that is marked in Line 14. This implies that $w \in W_0(M(v))$. Thus, we have that $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = M(v)$. Furthermore, since the condition in Line 13 is satisfied, we have that $L_1(w)$ is not a descendant of $\mathit{high}(v)$. This shows that $w \in W(v)$. Then, we also have that $u \in U_2(v)$, and w is a proper ancestor of v, $w \leq \mathit{low}(u)$ and $\mathit{bcount}(v) = \mathit{bcount}(u) + \mathit{bcount}(w) - 1$. Thus, Lemma 5.89 implies that $e_L(w) \notin B(u) \cup B(v)$ and $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$. In other words, we have that $\{(u, p(u)), (v, p(v)), (w, p(w)), e_L(w)\}$ is a Type-$3\beta ii$-2 4-cut. Thus, since $L_1(w)$ is not a descendant of $\mathit{high}(v)$, we have that this 4-cut is in $\mathcal{C}_{3\beta ii2}$. This shows that the collection $\mathcal{C}$ of 4-sets marked by Algorithm 41 is a collection of 4-cuts such that $\mathcal{C} \subseteq \mathcal{C}_{3\beta ii2}$.

According Proposition 3.6, we can compute the values $M(B(w) \setminus \{e_L(w)\})$, for all vertices $w \neq r$, in total linear time. Thus, the **for** loop in Line 1 can be performed in linear time. Then, the construction of the lists $W_0(x)$ in Line 4 can be performed in

Algorithm 41: Compute a collection of Type-3βii-2 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e_L(w)\}$, where u is a descendant of v, v is a descendant of w, and $L_1(w)$ is not a descendant of $high(v)$, so that all Type-3βii-2 4-cuts of this form are implied from this collection, plus that of the Type-2ii 4-cuts returned by Algorithm 24

```
1  foreach vertex w ≠ r do
2      compute M(B(w) \ {e_L(w)})
3  end
4  foreach vertex x do
5      let W_0(x) be the list of all vertices w ≠ r such that
         M(w) ≠ M(B(w) \ {e_L(w)}) = x, sorted in decreasing order
6  end
7  foreach vertex v do
8      compute the set U_2(v)
9  end
10 foreach vertex v such that U_2(v) ≠ ∅ do
11     foreach u ∈ U_2(v) do
12         let w be the greatest proper ancestor of v in W_0(M(v)) such that
             w ≤ low(u) and w ≤ firstW(v)
13         if bcount(v) = bcount(u) + bcount(w) − 1 and L_1(w) is not a descendant of
             high(v) then
14             mark {(u,p(u)),(v,p(v)),(w,p(w)),e_L(w)} as a Type-3βii-2 4-cut
15         end
16     end
17 end
```

$O(n)$ in total, for all vertices x, with bucket-sort. (Notice that all these lists are pairwise disjoint.) The sets $U_2(v)$, for all vertices v such that $W(v) \neq \emptyset$, can be computed in linear time in total, according to Lemma 5.88. For the remaining vertices v, we let $U_2(v) \leftarrow \emptyset$. Thus, the **for** loop in Line 7 can be performed in linear time. It remains to show how we can compute all w in Line 12. For this, we can use Algorithm 22. More specifically, for every vertex v such that $U_2(v) \neq \emptyset$, and for every $u \in U_2(v)$, we generate a query $q(W_0(M(v)), \mathit{min}\{p(v), \mathit{low}(u), \mathit{firstW}(v)\})$. This is to return the greatest $w \in W_0(M(v))$ such that $w \leq p(v)$, $w \leq \mathit{low}(u)$ and $w \leq \mathit{firstW}(v)$. Since $w \in W_0(M(v))$, we have that $M(B(w) \setminus \{e_L(w)\}) = M(v)$, and therefore w is an ancestor of $M(v)$. Thus, $M(v)$ is a common descendant of w and v, and therefore w and v are related as ancestor and descendant. Thus, $w \leq p(v)$ implies that w is a proper ancestor of v. Therefore, w is the greatest proper ancestor of v in $W_0(M(v))$ such that $w \leq \mathit{low}(u)$ and $w \leq \mathit{firstW}(v)$. Since the total size of the U_2 sets is $O(n)$, we can answer all those queries in $O(n)$ time using Algorithm 22 according to Lemma 5.27. Thus, we can see that Algorithm 41 runs in linear time.

Now let (u, v, w) be a triple of vertices that induces a 4-cut $C \in \mathcal{C}_{3\beta ii2}$. This means that C is a Type-3βii-2 4-cut such that $L_1(w)$ is not a descendant of $\mathit{high}(v)$. Therefore, Lemma 5.84 implies that $u \in U_2(v)$ and $w \in W(v)$. Furthermore, Lemma 5.79 implies that $w \leq \mathit{low}(u)$. So let w' be the greatest vertex in $W(v)$ such that $w' \leq \mathit{low}(u)$. Then, Lemma 5.85 implies that (u, v, w') induces a 4-cut $C' \in \mathcal{C}_{3\beta ii2}$. This implies that $B(v) = B(u) \sqcup (B(w') \setminus \{e_L(w')\})$, and therefore $\mathit{bcount}(v) = \mathit{bcount}(u) + \mathit{bcount}(w') - 1$. Since $w' \in W(v)$ we have $w' \in W_0(M(v))$ and $w' \leq \mathit{firstW}(v)$.

In what follows, let $\tilde{w} = \mathit{firstW}(v)$. Now let w'' be the greatest proper ancestor of v in $W_0(M(v))$ such that $w'' \leq \mathit{low}(u)$ and $w'' \leq \tilde{w}$. We will show that $w' = w''$. By Lemma 5.80 we have that $\tilde{w}$ is a proper ancestor of $\mathit{high}(v)$. Since $\tilde{w} \in W_0(M(v))$ and $w'' \in W_0(M(v))$, we have that $M(v)$ is a common descendant of $\tilde{w}$ and w'', and therefore $\tilde{w}$ and w'' are related as ancestor and descendant. Thus, $w'' \leq \tilde{w}$ implies that w'' is an ancestor of $\tilde{w}$. Now let us suppose, for the sake of contradiction, that $w'' \notin W(v)$. Since $w'' \in W_0(M(v))$, this implies that $L_1(w'')$ is a descendant of $\mathit{high}(v)$ (because otherwise w'' would satisfy all the conditions to be in $W(v)$). Since $w'' \in W_0(M(v))$, we have $M(v) = M(B(w'') \setminus \{e_L(w'')\}) \neq M(w'')$. Thus, $e_L(w'')$ is the only back-edge in $B(w'')$ whose higher endpoint is not a descendant of $M(v)$. Similarly, since $\tilde{w} \in W_0(M(v))$, we have that $e_L(\tilde{w})$ is the only back-edge in $B(\tilde{w})$ whose higher endpoint is not a descendant of $M(v)$. Now let $(x, y) = e_L(w'')$. Then $x = L_1(w'')$ is a

descendant of $high(v)$, and therefore a descendant of $\tilde{w}$. Furthermore, y is a proper ancestor of w'', and therefore a proper ancestor of $\tilde{w}$. This shows that $(x,y) \in B(\tilde{w})$. But since the higher endpoint of (x,y) is not a descendant of $M(v)$, we have that $(x,y) = e_L(\tilde{w})$, contradicting the fact that $L_1(\tilde{w})$ is not a descendant of $high(v)$ (which is implied by $\tilde{w} \in W(v)$). Thus, we have $w'' \in W(v)$, which is a strengthening of the condition $w'' \in W_0(M(v))$. Thus, w'' is the greatest vertex in $W(v)$ such that $w'' \leq low(u)$, and so we have $w'' = w'$.

Now, since $w' \in W(v)$, we have that $L_1(w')$ is not a descendant of $high(v)$. And since $w' = w''$, we have that w' will be the value of the variable "w" when we reach Line 12 during the processing of v and u. Thus, the 4-cut induced by (u,v,w') satisfies all the conditions to be marked in Line 14, and therefore we have that $C' \in \mathcal{C}$. If $w' = w$, then we have that $C = C'$, and thus it is trivially true that C is implied by $\mathcal{C}$. So let us assume that $w' \neq w$. Then, Lemma 5.85 implies that $B(w) \sqcup \{e_L(w')\} = B(w') \sqcup \{e_L(w)\}$. Since both w and w' are in $W(v)$, we have that $M(B(w) \setminus \{e_L(w)\}) = M(B(w') \setminus \{e_L(w')\}) = M(v)$. Thus, $M(v)$ is a common descendant of w and w'. Therefore, since the maximality of w' (w.r.t. to $w' \leq low(u)$ and $w' \in W(v)$) implies that $w' > w$, we have that w' is a proper descendant of w. Thus, since $B(w) \sqcup \{e_L(w')\} = B(w') \sqcup \{e_L(w)\}$, Lemma 5.28 implies that $C'' = \{(w,p(w)),(w',p(w')),e_L(w),e_L(w')\}$ is a Type-$2ii$ 4-cut. Since $C = \{(u,p(u)),(v,p(v)),(w,p(w)),e_L(w)\}$ and $C' = \{(u,p(u)),(v,p(v)),(w',p(w')),e_L(w')\}$, notice that C is implied by C' and C'' through the pair of edges $\{(w,p(w)),e_L(w)\}$. Let $\mathcal{C}'$ be the collection of Type-$2ii$ 4-cuts computed by Algorithm 24. Then, by Proposition 5.12 we have that C'' is implied by $\mathcal{C}'$ through the pair of edges $\{(w,p(w)),e_L(w)\}$. Thus, by Lemma 5.7 we have that C is implied by $\mathcal{C} \cup \mathcal{C}'$. □

The case where $L_1(w)$ is a descendant of $high(v)$

Lemma 5.90. *Let (u,v,w) be a triple of vertices that induces a Type-$3\beta ii$-2 4-cut, where $L_1(w)$ is a descendant of $high(v)$. Then w is the greatest ancestor of $high(v)$ such that $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = M(v)$ and $w \leq low(u)$.*

Proof. Let e be the back-edge in the 4-cut induced by (u,v,w). By the assumption we have made for the 4-cuts in this subsection, we have that $e = e_L(w)$. Then, by Lemma 5.79 we have that w is an ancestor of $high(v)$, $M(w) \neq M(B(w) \setminus \{e_L(w)\}) =$

$M(v)$ and $w \leq \mathit{low}(u)$. Now let us suppose, for the sake of contradiction, that there is an ancestor w' of $\mathit{high}(v)$ with $M(w') \neq M(B(w') \setminus \{e_L(w')\}) = M(v)$ and $w' \leq \mathit{low}(u)$, such that $w' > w$. Since w' and w have $\mathit{high}(v)$ as a common descendant, they are related as ancestor and descendant. Thus, $w' > w$ implies that w' is a proper descendant of w.

Since $L_1(w)$ is a descendant of $\mathit{high}(v)$, we have that the higher endpoint of $e_L(w)$ is a descendant of $\mathit{high}(v)$, and therefore a descendant of w'. Furthermore, the lower endpoint of $e_L(w)$ is a proper ancestor of w, and therefore a proper ancestor of w'. This shows that $e_L(w) \in B(w')$. Since $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = M(v)$, we have that $e_L(w)$ is the only back-edge in $B(w)$ whose higher endpoint is not a descendant of $M(v)$. Similarly, since $M(w') \neq M(B(w') \setminus \{e_L(w')\}) = M(v)$, we have that $e_L(w')$ is the only back-edge in $B(w')$ whose higher endpoint is not a descendant of $M(v)$. Thus, since $e_L(w) \in B(w')$, we have $e_L(w) = e_L(w')$.

Since (u, v, w) induces a Type-3βii-2 4-cut, we have that $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$. This implies that $B(w) \setminus \{e_L(w)\} = B(v) \setminus B(u)$. Now let (x, y) be a back-edge in $B(w)$. If $(x, y) = e_L(w)$, then $(x, y) \in B(w')$. Otherwise, $B(w) \setminus \{e_L(w)\} = B(v) \setminus B(u)$ implies that $(x, y) \in B(v)$. Therefore, x is a descendant of v, and therefore a descendant of $\mathit{high}(v)$, and therefore a descendant of w'. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of w'. This shows that $(x, y) \in B(w')$. Conversely, let (x, y) be a back-edge in $B(w')$. If $(x, y) = e_L(w')$, then $(x, y) \in B(w)$. Otherwise, $M(B(w') \setminus \{e_L(w')\}) = M(v)$ implies that x is a descendant of $M(v)$, and therefore a descendant of v. Furthermore, y is a proper ancestor of w', and therefore a proper ancestor of $\mathit{high}(v)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Since $w' \leq \mathit{low}(u)$, we have that $B(u) \cap B(w') = \emptyset$ (because no back-edge in $B(u)$ has low enough lower endpoint in order to leap over w'). Thus, we have $(x, y) \notin B(u)$, and therefore $(x, y) \in B(v) \setminus B(u)$. Since $B(w) \setminus \{e_L(w)\} = B(v) \setminus B(u)$, this implies that $(x, y) \in B(w)$. Thus, we have that $B(w') = B(w)$, in contradiction to the fact that the graph is 3-edge-connected. We conclude that w is the greatest ancestor of $\mathit{high}(v)$ such that $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = M(v)$ and $w \leq \mathit{low}(u)$. $\square$

Lemma 5.90 motivates the following definition. Let v be a vertex $\neq r$. Then we let $\widetilde{W}(v)$ denote the collection of all vertices w such that w is an ancestor of $\mathit{high}(v)$, $L_1(w)$ is a descendant of $\mathit{high}(v)$, and $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = M(v)$. Then, Lemma 5.90 implies that, if there is a triple of vertices (u, v, w) that induces a Type-

3βii-2 4-cut such that $L_1(w)$ is a descendant of *high*(v), then $w \in \widetilde{W}(v)$. Thus, the sets $\widetilde{W}(v)$ can guide us into the search for such 4-cuts.

Lemma 5.91. *Let v and v' be two distinct vertices $\neq r$. Then $\widetilde{W}(v) \cap \widetilde{W}(v') = \emptyset$.*

Proof. If $M(v) \neq M(v')$, then obviously $\widetilde{W}(v) \cap \widetilde{W}(v') = \emptyset$. (Because every vertex $w \in \widetilde{W}(v)$ has $M(B(w) \setminus \{e_L(w)\}) = M(v)$, and every vertex $w' \in \widetilde{W}(v')$ has $M(B(w') \setminus \{e_L(w')\}) = M(v')$.) Thus, we may assume that $M(v) = M(v')$.

Now let us suppose, for the sake of contradiction, that $\widetilde{W}(v) \cap \widetilde{W}(v') \neq \emptyset$. Since $v \neq v'$, we may assume w.l.o.g. that $v > v'$. Therefore, since v and v' have $M(v) = M(v')$ as a common descendant, we have that v is a proper descendant of v'. Let w be a vertex in $\widetilde{W}(v) \cap \widetilde{W}(v')$. Then, since $w \in \widetilde{W}(v)$, we have that $L_1(w)$ is a descendant of *high*(v). Furthermore, since $w \in \widetilde{W}(v')$, we have that w is an ancestor of *high*(v'). Notice that, since $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = M(v)$, we have that $L_1(w)$ (i.e., the higher endpoint of $e_L(w)$) is not a descendant of $M(v)$. Since v' is a proper ancestor of v with $M(v') = M(v)$, by Lemma 3.2 we have that $B(v') \subseteq B(v)$.

Let us suppose, for the sake of contradiction, that *high*(v) is a proper ancestor of v'. Let (x, y) be a back-edge in $B(v)$. Then x is a descendant of v, and therefore a descendant of v'. Furthermore, y is an ancestor of *high*(v), and therefore a proper ancestor of v'. This shows that $(x, y) \in B(v')$. Due to the generality of $(x, y) \in B(v)$, this implies that $B(v) \subseteq B(v')$. Thus, $B(v') \subseteq B(v)$ implies that $B(v) = B(v')$, in contradiction to the fact that the graph is 3-edge-connected. Thus, we have that *high*(v) is not a proper ancestor of v'. Since *high*(v) is a proper ancestor of v, and v' is also an ancestor of v, we have that *high*(v) and v' are related as ancestor and descendant. Thus, since *high*(v) is not a proper ancestor of v', we have that *high*(v) is a descendant of v'.

Now, since $L_1(w)$ is a descendant of *high*(v), it is a descendant of v'. Furthermore, since w is an ancestor of *high*(v'), we have that w is a proper ancestor of v'. Therefore, the lower endpoint of $e_L(w)$ is a proper ancestor of v' (since it is a proper ancestor of w). This shows that $e_L(w) \in B(v')$, and therefore $L_1(w)$ is a descendant of $M(v')$. But $L_1(w)$ is not a descendant of $M(v)$, in contradiction to the fact that $M(v') = M(v)$. Thus, we conclude that $\widetilde{W}(v) \cap \widetilde{W}(v') = \emptyset$. □

Lemma 5.92. *For every vertex x, let $\widetilde{L}(x)$ be the list of all w such that $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = x$, sorted in decreasing order. Let v be a vertex such that $M(v) = x$*

and $\widetilde{W}(v) \neq \emptyset$. Let $w = max(\widetilde{W}(v))$. Then, w is the greatest vertex in $\widetilde{L}(x)$ such that $w \leq high(v)$. Furthermore, $\widetilde{W}(v)$ is a segment of $\widetilde{L}(x)$.

Proof. By definition of $\widetilde{W}(v)$, we have that $w = max(\widetilde{W}(v))$ is an ancestor of $high(v)$, and therefore $w \leq high(v)$. Now let us suppose, for the sake of contradiction, that there is a vertex $w' \in \widetilde{L}(x)$ with $w' > w$, such that $w' \leq high(v)$. Since $M(B(w') \setminus \{e_L(w')\}) = M(B(w) \setminus \{e_L(w)\}) = x$, we have that w' and w have x as a common descendant. Thus, w' and w are related as ancestor and descendant. Therefore, $w' > w$ implies that w' is a proper descendant of w.

Since $w' \in \widetilde{L}(x)$, we have that $M(B(w') \setminus \{e_L(w')\}) = x = M(v)$. Thus, since $M(B(w') \setminus \{e_L(w')\})$ is a descendant of w' and $M(v)$ is a descendant of v, we have that w' and v have x as a common descendant, and therefore they are related as ancestor and descendant. Thus, since $w' \leq high(v)$ and $high(v)$ is an ancestor of v, we have that w' is an ancestor of v. Thus, since w' and $high(v)$ have v as a common descendant, we have that w' is related as ancestor and descendant with $high(v)$. Therefore, $w' \leq high(v)$ implies that w' is an ancestor of $high(v)$. Since $w = max(\widetilde{W}(v))$ and $w' > w$, we have that $w' \notin \widetilde{W}(v)$. Thus, since w' is an ancestor of $high(v)$, we have that $L_1(w')$ is not a descendant of $high(v)$ (because this is the only condition that prevents w' to be in $\widetilde{W}(v)$). Notice that, since $M(w') \neq M(B(w') \setminus \{e_L(w')\}) = x$, we have that $L_1(w')$ (i.e., the higher endpoint of $e_L(w')$) is not a descendant of x.

Since $w \in \widetilde{W}(v)$, we have that $L_1(w)$ is a descendant of $high(v)$. Thus, $e_L(w) \neq e_L(w')$. Since $L_1(w)$ is a descendant of $high(v)$, we have that $L_1(w)$ is a descendant of w'. And since the lower endpoint of $e_L(w)$ is a proper ancestor of w, we have that the lower endpoint of $e_L(w)$ is a proper ancestor of w'. This shows that $e_L(w) \in B(w')$. Notice that, since $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = x$, we have that $L_1(w)$ (i.e., the higher endpoint of $e_L(w)$) is not a descendant of x. Since $e_L(w) \neq e_L(w')$ and $e_L(w) \in B(w')$, we have that $e_L(w) \in B(w') \setminus \{e_L(w')\}$. Thus, $B(w') \setminus \{e_L(w')\}$ contains a back-edge whose higher endpoint is not a descendant of x, and therefore $M(B(w') \setminus \{e_L(w')\}) \neq x$, a contradiction. Thus, we have shown that there is no vertex $w' \in \widetilde{L}(x)$ with $w' > w$, such that $w' \leq high(v)$. This shows that w is the greatest vertex in $\widetilde{L}(x)$ such that $w \leq high(v)$.

By definition, we have that $\widetilde{W}(v) \subseteq \widetilde{L}(x)$. (Since $\widetilde{L}(x)$ contains every vertex w such that $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = M(v)$.) Now let us suppose, for the sake of contradiction, that $\widetilde{W}(v)$ is not a segment of $\widetilde{L}(x)$. Since $w = max(\widetilde{W}(v))$, this means that there are w' and w'' in $\widetilde{L}(x)$, with $w > w' > w''$, such that $w' \notin \widetilde{W}(v)$ and $w'' \in$

$\widetilde{W}(v)$. Since $w, w', w'' \in \widetilde{L}(x)$, we have that $M(B(w) \setminus \{e_L(w)\}) = M(B(w') \setminus \{e_L(w')\}) = M(B(w'') \setminus \{e_L(w'')\}) = x$. Thus, $\{w, w', w''\}$ have x as a common descendant, and therefore all three of them are related as ancestor and descendant. Thus, $w > w' > w''$ implies that w is a proper descendant of w', and w' is a proper descendant of w''.

Since $w \in \widetilde{W}(v)$, we have that w is an ancestor of $high(v)$. This implies that w' is also an ancestor of $high(v)$. Thus, since $w' \in \widetilde{L}(x)$ and $w' \notin \widetilde{W}(v)$, we have that $L_1(w')$ is not a descendant of $high(v)$. Now, since $w'' \in \widetilde{W}(v)$, we have that $L_1(w'')$ is a descendant of $high(v)$. Thus, $e_L(w') \neq e_L(w'')$. Since $L_1(w'')$ is a descendant of $high(v)$ and w' is an ancestor of $high(v)$, we have that $L_1(w'')$ is a descendant of w'. Furthermore, the lower endpoint of $e_L(w'')$ is a proper ancestor of w'', and therefore a proper ancestor of w'. This shows that $e_L(w'') \in B(w')$. As previously, notice that $e_L(w'')$ is not a descendant of x (since $M(w'') \neq M(B(w'') \setminus \{e_L(w'')\}) = x$). Since $e_L(w') \neq e_L(w'')$ and $e_L(w'') \in B(w')$, we have that $e_L(w'') \in B(w') \setminus \{e_L(w')\}$. Thus, $B(w') \setminus \{e_L(w')\}$ contains a back-edge whose higher endpoint is not a descendant of x, and therefore $M(B(w') \setminus \{e_L(w')\}) \neq x$ – in contradiction to $w' \in \widetilde{L}(x)$. Thus, we conclude that $\widetilde{W}(v)$ is a segment of $\widetilde{L}(x)$. □

Algorithm 42 shows how we can compute all sets $\widetilde{W}(v)$, for all $v \neq r$, in total linear time. The idea is to find, for every vertex v, the greatest w that has $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = M(v)$ and $w \leq high(v)$. According to Lemma 5.92, if $\widetilde{W}(v) \neq \emptyset$, then this w must satisfy $w \in \widetilde{W}(v)$ (i.e., it also has that $L_1(w)$ is a descendant of $high(v)$). Then, still according to Lemma 5.92, we have that $\widetilde{W}(v)$ is a segment of the decreasingly sorted list that consists of all vertices w' with $M(w') \neq M(B(w') \setminus \{e_L(w')\}) = M(v)$. Thus, we keep traversing this list, and we greedily insert as many vertices as we can into $\widetilde{W}(v)$, until we reach a w' that no longer satisfies that $L_1(w')$ is a descendant of $high(v)$. The full proof of correctness and linear complexity of Algorithm 42 is given in Lemma 5.93. The proof of linear complexity relies on Lemma 5.91: i.e., the sets in $\{\widetilde{W}(v) \mid v \text{ is a vertex } \neq r\}$ are pairwise disjoint.

Lemma 5.93. *Algorithm 42 correctly computes the sets $\widetilde{W}(v)$, for all vertices $v \neq r$. Furthermore, it has a linear-time implementation.*

Proof. It should be clear that, when we reach Line 8, we have that $\widetilde{L}(x)$, for every vertex x, is the decreasingly sorted list of all vertices w that have $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = x$. The idea in the **for** loop in Line 13 is to process all vertices $v \neq r$ in a bottom-up fashion. Then, for every vertex $v \neq r$, we start searching in $\widetilde{L}(M(v))$ for

Algorithm 42: Compute the sets $\widetilde{W}(v)$, for all $v \neq r$

```
1  compute M(B(w) \ {e_L(w)}), for every w ≠ r
2  initialize an empty list L̃(x) ← ∅, for every vertex x
3  for w ← n to w = 2 do
4  |   if M(w) ≠ M(B(w) \ {e_L(w)}) then
5  |   |   insert w into L̃(M(B(w) \ {e_L(w)}))
6  |   end
7  end
   // L̃(x) contains all vertices w with M(w) ≠ M(B(w) \ {e_L(w)}) = x,
      sorted in decreasing order, for every vertex x
8  initialize an array currentVertex[x], for every vertex x
9  foreach vertex x do
10 |   let currentVertex[x] ← first element of L̃(x)
11 end
12 initialize W̃(v) ← ∅, for every vertex v ≠ r
13 for v ← n to v = 2 do
14 |   let x ← M(v)
15 |   let w ← currentVertex[x]
16 |   while w > high(v) do
17 |   |   w ← next_{L̃(x)}(w)
18 |   end
19 |   while w ≠ ⊥ and L_1(w) is a descendant of high(v) do
20 |   |   insert w into W̃(v)
21 |   |   w ← next_{L̃(x)}(w)
22 |   end
23 |   currentVertex[x] ← w
24 end
```

the greatest vertex w that has $w \leq \mathit{high}(v)$ (see the **while** loop in Line 16). This search for this w starts from the last vertex that we accessed in $\widetilde{L}(M(v))$. This is ensured by the use of variable $\mathit{currentVertex}[x]$ (where $x = M(v)$), which we use in order to initialize w in Line 15, and then we update $\mathit{currentVertex}[x]$ in Line 23. We only have to explain why this is sufficient.

First, we have that, if v' and v are two vertices with $M(v') = M(v)$ and $v' < v$, then $\mathit{high}(v')$ is an ancestor of $\mathit{high}(v)$. To see this, suppose the contrary. Since $M(v') = M(v)$, we have that v' and v have $M(v)$ as a common descendant. Therefore, $v' < v$ implies that v' is a proper ancestor of v. Then, since $\mathit{high}(v')$ is a proper ancestor of v', we have that $\mathit{high}(v')$ is a proper ancestor of v. Thus, since $\mathit{high}(v)$ is also a proper ancestor of v, we have that $\mathit{high}(v')$ and $\mathit{high}(v)$ are related as ancestor and descendant. Then, since $\mathit{high}(v')$ is not an ancestor of $\mathit{high}(v)$, it must be a proper descendant of $\mathit{high}(v)$. This implies that $\mathit{high}(v') > \mathit{high}(v)$. Since v' is a proper ancestor of v with $M(v') = M(v)$, Lemma 3.2 implies that $B(v') \subseteq B(v)$. This implies that $\mathit{high}(v) \geq \mathit{high}(v')$, a contradiction. Therefore, we have indeed that $\mathit{high}(v')$ is an ancestor of $\mathit{high}(v)$. This implies that $\mathit{high}(v') \leq \mathit{high}(v)$.

Now we can see inductively that $\widetilde{W}(v)$ is computed correctly by the **for** loop in Line 13, for every vertex $v \neq r$. We use induction on the number of times that a vertex v with $M(v) = x$ is processed, for every fixed x. The first time that the **for** loop in Line 13 processes a vertex v with $M(v) = x$, we begin the search for vertices $w \in \widetilde{W}(v)$ from the beginning of $\widetilde{L}(x)$ (due to the initialization of $\mathit{currentVertex}[x]$ in Line 10). The **while** loop in Line 16 traverses the list $\widetilde{L}(x)$, until it reaches a vertex w such that $w \leq \mathit{high}(v)$. Thus, this is the greatest vertex in $\widetilde{L}(x)$ such that $w \leq \mathit{high}(v)$. According to Lemma 5.92, if $\widetilde{W}(v) \neq \emptyset$, then $w = \mathit{max}(\widetilde{W}(v))$. Since $w \in \widetilde{L}(x)$ and $w \leq \mathit{high}(v)$, it is sufficient to check whether $L_1(w)$ is a descendant of $\mathit{high}(v)$. If that is the case, then we correctly insert w into $\widetilde{W}(v)$, in Line 20. Furthermore, by Lemma 5.92 we have that $\widetilde{W}(v)$ is a segment of $\widetilde{L}(x)$. Thus, since w is the leftmost element of this segment, it is sufficient to keep traversing $\widetilde{L}(x)$, and keep inserting all the vertices that we meet into $\widetilde{W}(v)$, until we meet a vertex that is provably no longer in $\widetilde{W}(v)$. Since $\widetilde{L}(x)$ is sorted in decreasing order, all the vertices w' that we meet have $w' \leq w \leq \mathit{high}(v)$. Thus, the only reason that may prevent such a w' to be in $\widetilde{W}(v)$, is that $L_1(w')$ is not a descendant of $\mathit{high}(v)$. This shows that the **while** loop in Line 19 will correctly compute $\widetilde{W}(v)$. Then, in Line 23 we set "$\mathit{currentVertex}[x] \leftarrow w$". This implies that the next time that we meet a vertex v' that has $M(v') = x$, we

will start the search for $\widetilde{W}(v')$ from the last entry w of $\widetilde{L}(x)$ that we accessed while processing v. This is sufficient for the following reasons. First, since the **for** loop in Line 13 processes the vertices in a bottom-up fashion, we have that the next v' with $M(v') = x$ that it will process is a proper ancestor of v. Thus, as shown above, we have that $high(v') \leq high(v)$. Then, the search from $\widetilde{L}(x)$ will be picked up from the vertex w that is either the greatest that satisfies $w \leq high(v)$, or it is the lowest in $\widetilde{W}(v)$. In the first case, it is sufficient to start the search in $\widetilde{L}(x)$ for $\widetilde{W}(v')$ from w. In the second case, we note that by Lemma 5.91 we have that $\widetilde{W}(v) \cap \widetilde{W}(v') = \emptyset$. Thus, no vertex that is greater than w can be in $\widetilde{W}(v')$: because no vertex in $\widetilde{W}(v)$ is in $\widetilde{W}(v')$, and by Lemma 5.92 we have that the greatest vertex in $\widetilde{W}(v)$ is the greatest vertex w' in $\widetilde{L}(x)$ that has $w' \leq high(v)$. Thus, with the same argument that we used for v we can see that $\widetilde{W}(v')$ will be correctly computed, and the same is true for any future vertex v'' with $M(v'') = x$ that we will meet. This demonstrates the correctness of Algorithm 42.

By Proposition 3.6, we have that the values $M(B(w) \setminus \{e_L(w)\})$, for all vertices $w \neq r$, can be computed in linear time in total. Thus, Line 1 can be performed in linear time. All other steps take $O(n)$ time in total. In particular, observe that, in the worst case, the **for** loop in Line 13 may have to traverse the entire lists $M^{-1}(x)$ and $\widetilde{L}(x)$, for all vertices x. Still, since all lists $M^{-1}(x)$ are pairwise disjoint, and all lists $\widetilde{L}(x)$ are pairwise disjoint, we have that the **for** loop in Line 13 takes $O(n)$ time. Thus, Algorithm 42 runs in linear time. □

Lemma 5.94. *A triple of vertices (u, v, w) induces a Type-3βii-2 4-cut, where $L_1(w)$ is a descendant of $high(v)$, if and only if:* (1) *u is a proper descendant of v,* (2) *u and v belong to a segment of $H(high(v))$ that is maximal w.r.t. the property that all its elements are related as ancestor and descendant,* (3) *$w \in \widetilde{W}(v)$,* (4) *$w \leq low(u)$, and* (5) *$bcount(v) = bcount(u) + bcount(w) - 1$.*

Proof. ($\Rightarrow$) Due to the convention we have made in this subsection, we have that the back-edge in the 4-cut induced by (u, v, w) is $e_L(w)$. Since (u, v, w) induces a Type-3βii-2 4-cut, by definition we have that u is a proper descendant of v. Let u' be a vertex such that $u \geq u' \geq v$ and $high(u') = high(v)$. Then, Lemma 5.79 implies that u' is an ancestor of u. Thus, the segment from u to v in $H(high(v))$ consists of vertices that are related as ancestor and descendant (since all of them are ancestors of u). This implies (2). Lemma 5.90 implies that w is an ancestor of

$high(v)$ and $M(w) \neq M(B(w) \setminus \{e_L(w)\}) = M(v)$. By assumption we have that $L_1(w)$ is a descendant of $high(v)$. Thus, w satisfies all the conditions to be in $\widetilde{W}(v)$. (4) is an implication of Lemma 5.79. (5) is an immediate implication of the fact that $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$.

($\Leftarrow$) (2) implies that $high(u) = high(v)$. Since by (1) we have that u is a proper descendant of v, Lemma 3.3 implies that $B(u) \subseteq B(v)$. Let (x, y) be a back-edge in $B(w) \setminus \{e_L(w)\}$. Since $w \in \widetilde{W}(v)$, we have that $M(B(w) \setminus \{e_L(w)\}) = M(v)$. Thus, since $(x, y) \in B(w) \setminus \{e_L(w)\}$, we have that x is a descendant of $M(B(w) \setminus \{e_L(w)\})$, and therefore a descendant of $M(v)$. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of $high(v)$ (since $w \in \widetilde{W}(v)$ implies that w is an ancestor of $high(v)$). This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(w) \setminus \{e_L(w)\}$, this implies that $B(w) \setminus \{e_L(w)\} \subseteq B(v)$. Let (x, y) be a back-edge in $B(u)$. Then we have $low(u) \leq y$. Therefore, $w \leq low(u)$ implies that $w \leq y$. Thus, y cannot be a proper ancestor of w, and therefore $(x, y) \notin B(w)$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \cap B(w) = \emptyset$. Thus, since $B(u) \subseteq B(v)$ and $B(w) \setminus \{e_L(w)\} \subseteq B(v)$ and $B(u) \cap B(w) = \emptyset$ and $bcount(v) = bcount(u) + bcount(w) - 1$, we have that $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$. Thus, (u, v, w) induces a Type-3βii-2 4-cut. ☐

Lemma 5.95. *Let u and v be two distinct vertices with $high(u) = high(v)$ that are related as ancestor and descendant. Then, $bcount(u) \neq bcount(v)$.*

Proof. We may assume w.l.o.g. that v is a proper ancestor of u. Then, since $high(u) = high(v)$, Lemma 3.3 implies that $B(u) \subseteq B(v)$. Thus, if we suppose that $bcount(u) = bcount(v)$, then we have that $B(u) = B(v)$, in contradiction to the fact that the graph is 3-edge-connected. This shows that $bcount(u) \neq bcount(v)$. ☐

Now we will show how to compute all Type-3βii-2 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e_L(w)\}$, where $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$ and $L_1(w)$ is a descendant of $high(v)$. So let (u, v, w) be a triple of vertices that induces a 4-cut of this form. Then, by Lemma 5.94 we have that $w \in \widetilde{W}(v)$, and $u \in S(v)$ (i.e., u belongs to the segment of $H(high(v))$ that contains v and is maximal w.r.t. the property that all its elements are related as ancestor and descendant). Thus, given v and $w \in \widetilde{W}(v)$, it is sufficient to find all $u \in S(v)$ that provide a triple (u, v, w) that induces a 4-cut of this form. By Lemma 5.94 we have that $bcount(v) = bcount(u) + bcount(w) - 1$. Then, Lemma 5.95 implies that u is the unique

vertex in $S(v)$ that has $bcount(u) = bcount(v) - bcount(w) + 1$. Thus, the idea is to process separately all segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant, for every vertex x. Let S be such a segment. Then, we store the *bcount* values for all vertices in S. (By Lemma 5.95, all these are distinct.) Then, for every $v \in S$ such that $\widetilde{W}(v) \neq \emptyset$, and every $w \in \widetilde{W}(v)$, we seek the unique u in S that has $bcount(u) = bcount(v) - bcount(w) + 1$, and, if it exists, then we check whether all conditions in Lemma 5.94 are satisfied, in order to have that (u, v, w) induces a 4-cut of the desired form. This procedure is shown in Algorithm 43. The proof of correctness and linear complexity is given in Proposition 5.27.

Proposition 5.27. *Algorithm 43 correctly computes all Type-3βii-2 4-cuts of the form* $\{(u, p(u)), (v, p(v)), (w, p(w)), e_L(w)\}$, *where* $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$ *and* $L_1(w)$ *is a descendant of high*(v). *Furthermore, it has a linear-time implementation.*

Proof. For every vertex x, let $\mathcal{S}(x)$ be the collection of the segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant. Let $\{(u, p(u)), (v, p(v)), (w, p(w)), e_L(w)\}$ be a Type-3βii-2 4-cut such that $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$ and $L_1(w)$ is a descendant of *high*(v). Lemma 5.94 implies that u and v belong to the same segment S of $\mathcal{S}(high(v))$, $w \in \widetilde{W}(v)$, $w \leq low(u)$, and $bcount(v) = bcount(u) + bcount(w) - 1$. Let $x = high(v)$. Then, during the processing of $S \in \mathcal{S}(x)$ (in the **for** loop in Line 8), we will eventually reach Line 14 for this particular w (during the **for** loop in Line 13). We will show that the $bcount(v) - bcount(w) + 1$ entry of the A array is precisely u. Notice that the entries of A are filled with vertices in Line 10, for every segment $S \in \mathcal{S}(x)$, for every vertex x. And then, after the processing of S, the entries of A that were filled with vertices are set again to *null* in Line 21. Thus, when we reach Line 13, we have that the non-*null* entries of A contain vertices from S. More precisely, for every $z \in S$, we have that $A[bcount(z)] = z$. Thus, since all vertices in S are related as ancestor and descendant and have the same *high* point, Lemma 5.95 implies that $A[bcount(z)] = z$, for every $z \in S$, when the **for** loop in Line 9 is completed (during the processing of S); and also, we have $A[c] = \perp$, if there is no vertex $z \in S$ with $bcount(z) = c$. Therefore, since $bcount(v) = bcount(u) + bcount(w) - 1$, we have that $bcount(u) = bcount(v) - bcount(w) + 1$, and therefore $A[bcount(v) - bcount(w) + 1] = u$. Thus, when we reach Line 14, we have that the variable "u" contains the value u, and so the 4-cut $\{(u, p(u)), (v, p(v)), (w, p(w)), e_L(w)\}$ will be correctly marked in Line 16

Algorithm 43: Compute all Type-3βii-2 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e_L(w)\}$, where $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$ and $L_1(w)$ is a descendant of $high(v)$

```
1  compute the set W̃(v), for every vertex v ≠ r
2  for every vertex x, let H(x) be the list of all vertices z with high(z) = x, sorted
     in decreasing order
3  foreach vertex x do
4  |   compute the collection S(x) of the segments of H(x) that are maximal
   |     w.r.t. the property that their elements are related as ancestor and
   |     descendant
5  end
6  initialize an array A of size m
7  foreach vertex x do
8  |   foreach S ∈ S(x) do
9  |   |   foreach z ∈ S do
10 |   |   |   set A[bcount(z)] ← z
11 |   |   end
12 |   |   foreach v ∈ S do
13 |   |   |   foreach w ∈ W̃(v) do
14 |   |   |   |   let u ← A[bcount(v) − bcount(w) + 1]
15 |   |   |   |   if u ≠ ⊥ and u is a proper descendant of v and w ≤ low(u) then
16 |   |   |   |   |   mark {(u,p(u)),(v,p(v)),(w,p(w)),e_L(w)} as a Type-3βii-2
   |   |   |   |   |     4-cut
17 |   |   |   |   end
18 |   |   |   end
19 |   |   end
20 |   |   foreach z ∈ S do
21 |   |   |   set A[bcount(z)] ← ⊥
22 |   |   end
23 |   end
24 end
```

(since the condition in Line 15 is satisfied).

Conversely, whenever the condition in Line 15 is satisfied, we have that: (1) u is a proper descendant of v, (2) u and v belong to a segment of $H(high(v))$ that is maximal w.r.t. the property that all its elements are related as ancestor and descendant, (3) $w \in \widetilde{W}(v)$, (4) $w \leq low(u)$, and (5) $bcount(v) = bcount(u) + bcount(w) - 1$. Thus, Lemma 5.94 implies that $\{(u, p(u)), (v, p(v)), (w, p(w)), e_L(w)\}$ is a Type-3βii-2 4-cut such that $B(v) = B(u) \sqcup (B(w) \setminus \{e_L(w)\})$ and $L_1(w)$ is a descendant of $high(v)$. Therefore, it is correct to mark it in Line 16.

Now let us establish the linear-time complexity of Algorithm 43. First, by Lemma 5.93 we have that Line 1 has a linear-time implementation. In particular, Lemma 5.91 implies that the total size of all sets $\widetilde{W}(v)$, for $v \neq r$, is $O(n)$. Then, by Lemma 3.22 we have that the collections $\mathcal{S}(x)$, for every vertex x, can be computed in linear time in total. Thus, the **for** loop in Line 3 has a linear-time implementation. Finally, the **for** loop in Line 7 is completed in linear time, precisely because all segments in the collection $\{S \in \mathcal{S}(x) \mid x \text{ is a vertex }\}$ are pairwise disjoint, and so they have total size $O(n)$, and the sets $\widetilde{W}(v)$, for all $v \neq r$, have total size $O(n)$. □

5.8.2.3 Type-3βii-3 4-cuts

Now we consider case (3) of Lemma 5.57.

Let u, v, w be three vertices $\neq r$ such that w is proper ancestor of v, v is a proper ancestor of u, and there is a back-edge $e \in B(u)$ such that $e \notin B(v) \cup B(w)$, $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ and $M(w) = M(v)$. By Lemma 5.57, we have that $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ is a 4-cut, and we call this a Type-3βii-3 4-cut.

The following lemma provides some useful information concerning this type of 4-cuts.

Lemma 5.96. *Let (u, v, w) be a triple of vertices that induces a Type-3βii-3 4-cut, and let e be the back-edge of this 4-cut. Then $e = (highD(u), high(u))$. Furthermore, $low(u) \geq w$ and $high(u) \neq high_2(u) = high(v)$. Finally, if u' is a vertex such that $u \geq u' \geq v$ and either $high(u') = high(v)$ or $high_2(u') = high(v)$, then u' is an ancestor of u.*

Proof. Since (u, v, w) induces a Type-3βii-3 4-cut, we have that $e \in B(u)$, $e \notin B(v) \cup B(w)$, and $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ $(*)$. This implies that e is the only back-edge in $B(u)$ that is not in $B(v)$. Now let $(x_1, y_1), \dots, (x_k, y_k)$ be all the back-edges in $B(u)$ sorted in decreasing order w.r.t. their lower endpoint, so that we have $(x_i, y_i) =$

$(\mathit{highD}_i(u), \mathit{high}_i(u))$, for every $i \in \{1, \ldots, k\}$. Let $i \in \{1, \ldots, k\}$ be an index such that $(x_i, y_i) \in B(v)$. Then we have that y_i is a proper ancestor of v. This implies that all y_j, with $j \in \{i, \ldots, k\}$ are proper ancestors of v. Furthermore, we have that all x_j, for $j \in \{1, \ldots, k\}$, are descendants of u, and therefore all of them are descendants of v. This shows that all the back-edges $(x_i, y_i), \ldots, (x_k, y_k)$ are in $B(v)$. Thus, we cannot have that $(x_1, y_1) \in B(v)$, because otherwise we would have $B(u) \subseteq B(v)$. Since e is the unique back-edge in $B(u) \setminus B(v)$, this shows that $e = (x_1, y_1)$, and therefore $e = (\mathit{highD}(u), \mathit{high}(u))$. Furthermore, this argument also shows that $y_2 \neq y_1$, and therefore $\mathit{high}_2(u) \neq \mathit{high}(u)$.

Since $e = (\mathit{highD}(u), \mathit{high}(u))$ and $B(u) \setminus \{e\} \subseteq B(v)$ (and $B(u) \setminus \{e\}$ contains at least one back-edge, since the graph is 3-edge-connected), we have that $(x_2, y_2) \in B(v)$, and therefore $\mathit{high}(v) \geq \mathit{high}_2(u)$. Conversely, let (x, y) be a back-edge in $B(v)$ such that $y = \mathit{high}(v)$. We can use a similar argument as above, in order to conclude that, if $\mathit{high}(v) < w$, then $B(v) \subseteq B(w)$. But this is impossible to be the case, since there are back-edges from $B(u)$ in $B(v)$, and we have $B(u) \cap B(w) = \emptyset$. Thus, we have that $(x, y) \notin B(w)$. Then $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that $(x, y) \in B(u) \setminus \{e\}$. Since $e = (\mathit{highD}(u), \mathit{high}(u))$ and $\mathit{high}_2(u) \neq \mathit{high}(u)$, this implies that $y_2 \geq y$. Thus we have $\mathit{high}_2(u) \geq \mathit{high}(v)$. This shows that $\mathit{high}(v) = \mathit{high}_2(u)$.

Let us suppose, for the sake of contradiction, that $\mathit{low}(u) < w$. Consider a back-edge $(x, y) \in B(u)$ such that $y = \mathit{low}(u)$. Then x is a descendant of u, and therefore a descendant of v, and therefore a descendant of w. Furthermore, since $\mathit{low}(u) < w$, we have that y is a proper ancestor of w (because y and w are related as ancestor and descendant, since both of them are ancestors of u). This shows that $(x, y) \in B(w)$. But this contradicts $B(u) \cap B(w) = \emptyset$, which is a consequence of $(*)$. Thus we have shown that $\mathit{low}(u) \geq w$.

Now let u' be a vertex with $u \geq u' \geq v$ such that there is a back-edge $(x, y) \in B(u')$ with $y = \mathit{high}(v)$ (notice that this includes the cases $\mathit{high}(u') = \mathit{high}(v)$ and $\mathit{high}_2(u') = \mathit{high}(v)$). Since u is a descendant of v, $u \geq u' \geq v$ implies that u' is a descendant of v. Then, x is a descendant of v. Thus, since $y = \mathit{high}(v)$, we have that $(x, y) \in B(v)$. Since (u, v, w) induces a Type-3βii-3 4-cut, by definition we have that w is a proper ancestor of v with $M(w) = M(v)$. Then, Lemma 3.6 implies that $\mathit{high}(v)$ is a descendant of w. Thus, $y = \mathit{high}(v)$ is not a proper ancestor of w, and therefore $(x, y) \notin B(w)$. Thus, $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that $(x, y) \in B(u)$. Then we have that x is a common descendant of u and u', and therefore u and u' are related as ancestor and

descendant. Thus, $u \geq u'$ implies that u' is an ancestor of u. □

Recall that, for every vertex x, we let $\widetilde{H}(x)$ denote the list of all vertices z such that either $high_1(z) = x$ or $high_2(z) = x$, sorted in decreasing order. For every vertex $v \neq r$, let $\widetilde{S}_1(v)$ denote the segment of $\widetilde{H}(high(v))$ that contains v and is maximal w.r.t. the property that its elements are related as ancestor and descendant. Then, for every vertex $v \neq r$ with $nextM(v) \neq \bot$, we let $U_3(v)$ denote the collection of all $u \in \widetilde{S}_1(v)$ such that: (1) u is a proper descendant of v, (2) $high_1(u) \neq high_2(u) = high_1(v)$, (3) $low(u) \geq lastM(v)$, and (4) either $low(u) < nextM(v)$, or u is the lowest vertex in $\widetilde{S}_1(v)$ that satisfies (1), (2) and $low(u) \geq nextM(v)$.

Lemma 5.97. *Let (u, v, w) be a triple of vertices that induces a Type-3βii-3 4-cut. Then $U_3(v) \neq \emptyset$, and let $\tilde{u}$ be the greatest vertex in $U_3(v)$. Then, if $u \notin U_3(v)$, we have that $(\tilde{u}, v, w)$ induces a Type-3βii-3 4-cut, and $B(\tilde{u}) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e_{high}(\tilde{u})\}$.*

Proof. Let u' be a vertex in $\widetilde{H}(high(v))$ such that $u \geq u' \geq v$. Since $u' \in \widetilde{H}(high(v))$, we have that either $high_1(u') = high(v)$ or $high_2(u') = high(v)$. Thus, since (u, v, w) induces a Type-3βii-3 4-cut, by Lemma 5.96 we have that u' is an ancestor of u. This implies that all vertices from u to v in $\widetilde{H}(high(v))$ are related as ancestor and descendant (since all of them are ancestors of u). This shows that $u \in \widetilde{S}_1(v)$. Since (u, v, w) induces a Type-3βii-3 4-cut, we have that u is a proper descendant of v. Furthermore, by Lemma 5.96 we have that $high_1(u) \neq high_2(u) = high_1(v)$, and $low(u) \geq w \geq lastM(v)$. Thus, if $low(u) < nextM(v)$, then we have $u \in U_3(v)$. Otherwise, we have $low(u) \geq nextM(v)$, and therefore we have $U_3(v) \neq \emptyset$, because we can consider the lowest vertex $\tilde{u} \in \widetilde{S}_1(v)$ that is a proper descendant of v and satisfies $high_1(\tilde{u}) \neq high_2(\tilde{u}) = high_1(v)$ and $low(\tilde{u}) \geq nextM(v)$.

This shows that $U_3(v) \neq \emptyset$. Furthermore, this shows that, if $u \notin U_3(v)$, then we can define $\tilde{u}$ as previously, and we have $\tilde{u} < u$ (due to the minimality of $\tilde{u}$), and therefore $\tilde{u}$ is a proper ancestor of u (since both $\tilde{u}$ and u are in $\widetilde{S}_1(v)$, and therefore they are related as ancestor and descendant). Now we will show that $\tilde{u}$ is the greatest vertex in $U_3(v)$. Notice that $low(\tilde{u}) \geq nextM(v)$, and every other vertex $u' \in U_3(v)$ (except $\tilde{u}$, that is) satisfies $low(u') < nextM(v)$. So let us suppose, for the sake of contradiction, that $\tilde{u}$ is not the greatest vertex in $U_3(v)$. Thus, there is a vertex $u' \in U_3(v)$ such that $u' > \tilde{u}$. Since both u' and $\tilde{u}$ are in $\widetilde{S}_1(v)$, this implies that u' is a proper descendant of $\tilde{u}$. Now let (x, y) be a back-edge in $B(u')$ such that $y = low(u')$. Then x is a descendant of u', and therefore a descendant of $\tilde{u}$. We also have $y = low(u') < nextM(v) \leq low(\tilde{u})$. Since

(x,y) is a back-edge, we have that y is an ancestor of x. Furthermore, we have that $low(\tilde{u})$ is an ancestor of $\tilde{u}$, and therefore an ancestor of u', and therefore an ancestor of x. Thus, x is a common descendant of y and $low(\tilde{u})$, and therefore y and $low(\tilde{u})$ are related as ancestor and descendant, and therefore y is a proper ancestor of $low(\tilde{u})$ (since $y < low(\tilde{u})$). Since x is a descendant of $\tilde{u}$, this implies that $(x,y) \in B(\tilde{u})$. But y is lower than $low(\tilde{u})$, a contradiction. This shows that $\tilde{u}$ is the greatest vertex in $U_3(v)$.

Now let us assume that $u \notin U_3(v)$. Let us suppose, for the sake of contradiction, that $e_{high}(\tilde{u}) = e_{high}(u)$. Then, since $\tilde{u}$ is an ancestor of u, by Lemma 3.3 we have $B(u) \subseteq B(\tilde{u})$. This can be strengthened to $B(u) \subset B(\tilde{u})$, since the graph is 3-edge-connected. Thus, there is a back-edge $(x,y) \in B(\tilde{u}) \setminus B(u)$. In particular, we have $(x,y) \neq e_{high}(u) = e_{high}(\tilde{u})$. Since $\tilde{u} \in U_3(v)$, we have $high_2(\tilde{u}) = high(v)$. Thus, since $(x,y) \neq e_{high}(\tilde{u})$, we have that y is an ancestor of $high_2(\tilde{u}) = high(v)$, and therefore it is a proper ancestor of v. Furthermore, x is a descendant of $\tilde{u}$, and therefore a descendant of v. This shows that $(x,y) \in B(v)$. Then, since (u,v,w) induces a Type-3βii-3 4-cut, we have $B(v) = (B(u) \setminus \{e_{high}(u)\}) \sqcup B(w)$. Thus, since $(x,y) \in B(v)$ and $(x,y) \notin B(u)$, we have $(x,y) \in B(w)$. But we have $low(\tilde{u}) \geq nextM(v)$, and therefore $low(\tilde{u}) \geq w$, and therefore $y \geq w$ (since $(x,y) \in B(\tilde{u})$ implies that $y \geq low(\tilde{u})$). Thus, y cannot be a proper ancestor of w, a contradiction. This shows that $e_{high}(\tilde{u}) \neq e_{high}(u)$.

Now let us suppose, for the sake of contradiction, that $e_{high}(\tilde{u}) \in B(u)$. Since $\tilde{u} \in U_3(v)$, we have $high_1(\tilde{u}) \neq high_2(\tilde{u}) = high(v)$. This implies that the lower endpoint of $e_{high}(\tilde{u})$ is greater than $high(v)$. Since $high_1(u) \neq high_2(u) = high(v)$, we have that $e_{high}(u)$ is the only back-edge in $B(u)$ whose lower endpoint is greater than $high(v)$. Thus, since $e_{high}(\tilde{u}) \in B(u)$, we have $e_{high}(\tilde{u}) = e_{high}(u)$, a contradiction. This shows that $e_{high}(\tilde{u}) \notin B(u)$. Similarly, we can show that $e_{high}(u) \notin B(\tilde{u})$.

Now let (x,y) be a back-edge in $B(\tilde{u}) \setminus \{e_{high}(\tilde{u})\}$. Then we have that x is a descendant of $\tilde{u}$, and therefore a descendant of v. Furthermore, since $\tilde{u} \in U_3(v)$, we have $high_2(\tilde{u}) = high_1(v)$, and therefore y is an ancestor of $high_1(v)$. This shows that $(x,y) \in B(v)$. Then, $B(v) = (B(u) \setminus \{e_{high}(u)\}) \sqcup B(w)$ implies that either $(x,y) \in B(u) \setminus \{e_{high}(u)\}$, or $(x,y) \in B(w)$. The case $(x,y) \in B(w)$ is rejected, since $low(\tilde{u}) \geq nextM(v) \geq w$. Thus, we have $(x,y) \in B(u) \setminus \{e_{high}(u)\}$. Due to the generality of $(x,y) \in B(\tilde{u}) \setminus \{e_{high}(\tilde{u})\}$, this shows that $B(\tilde{u}) \setminus \{e_{high}(\tilde{u})\} \subseteq B(u) \setminus \{e_{high}(u)\}$. Conversely, let (x,y) be a back-edge in $B(u) \setminus \{e_{high}(u)\}$. Then we have that x is a descendant of u, and therefore a descendant of $\tilde{u}$. Furthermore, we have $(x,y) \in B(v)$, and therefore y is a proper ancestor of v, and therefore a proper ancestor of $\tilde{u}$. This shows

that $(x,y) \in B(\tilde{u})$. Since $(x,y) \in B(u)$ and $e_{high}(\tilde{u}) \notin B(u)$, we have $(x,y) \neq e_{high}(\tilde{u})$. Thus, $(x,y) \in B(\tilde{u}) \setminus \{e_{high}(\tilde{u})\}$. Due to the generality of $(x,y) \in B(u) \setminus \{e_{high}(u)\}$, this shows that $B(u) \setminus \{e_{high}(u)\} \subseteq B(\tilde{u}) \setminus \{e_{high}(\tilde{u})\}$. Thus, we have shown that $B(\tilde{u}) \setminus \{e_{high}(\tilde{u})\} = B(u) \setminus \{e_{high}(u)\}$.

Then, since $e_{high}(\tilde{u}) \notin B(u)$ and $e_{high}(u) \notin B(\tilde{u})$, we have $B(\tilde{u}) \sqcup \{e_{high}(u)\} = B(u) \sqcup \{e_{high}(\tilde{u})\}$. Furthermore, since $B(v) = (B(u) \setminus \{e_{high}(u)\}) \sqcup B(w)$, we have $B(v) = (B(\tilde{u}) \setminus \{e_{high}(\tilde{u})\}) \sqcup B(w)$. Finally, since $high_1(\tilde{u}) \neq high_2(\tilde{u}) = high(v)$, we have $high_1(\tilde{u}) > high(v)$, and therefore $e_{high}(\tilde{u}) \notin B(v)$. Then, by Lemma 5.57, we conclude that $(\tilde{u}, v, w)$ induces a Type-3βii-3 4-cut. $\square$

Lemma 5.98. *Let v and v' be two vertices with $nextM(v) \neq \bot$ and $nextM(v') \neq \bot$, such that v' is a proper descendant of v with $high(v) = high(v')$ and $\widetilde{S}_1(v) = \widetilde{S}_1(v')$. If $U_3(v') = \emptyset$, then $U_3(v) = \emptyset$. If $U_3(v') \neq \emptyset$, then the lowest vertex in $U_3(v)$ (if it exists) is greater than, or equal to, the greatest vertex in $U_3(v')$.*

Proof. Since $high(v) = high(v')$ and v' is a proper descendant of v, by Lemma 3.3 we have that $B(v') \subseteq B(v)$. Since the graph is 3-edge-connected, this can be strengthened to $B(v') \subset B(v)$. This implies that $M(v')$ is a descendant of $M(v)$. Since $high(v) = high(v')$, we cannot have $M(v') = M(v)$, because otherwise Lemma 3.7 implies that $B(v') = B(v)$. Thus, we have that $M(v')$ is a proper descendant of $M(v)$. Since $lastM(v)$ is an ancestor of $M(v)$ and $nextM(v')$ is an ancestor of $M(v')$, we have that $lastM(v)$ and $nextM(v')$ are related as ancestor and descendant. Let us suppose, for the sake of contradiction, that $nextM(v')$ is a descendant of $lastM(v)$. Since $M(v')$ is a proper descendant of $M(v)$ and $M(lastM(v)) = M(v)$, there is a back-edge $(x,y) \in B(lastM(v))$ such that x is not a descendant of $M(v')$. Then, we have that x is a descendant of $M(v)$, and therefore a descendant of v, and therefore a descendant of $high(v) = high(v')$. By Lemma 3.8, we have that $high(v')$ is a descendant of $nextM(v')$. Therefore, since x is a descendant of $high(v')$, we have that x is a descendant of $nextM(v')$. Furthermore, y is a proper ancestor of $lastM(v)$, and therefore a proper ancestor of $nextM(v')$. This shows that $(x,y) \in B(nextM(v'))$. But x is not a descendant of $M(v') = M(nextM(v'))$, a contradiction. Thus, we have that $nextM(v')$ is not a descendant of $lastM(v)$. Therefore, since $lastM(v)$ and $nextM(v')$ are related as ancestor and descendant, we have that $nextM(v')$ is a proper ancestor of $lastM(v)$.

Let us suppose, for the sake of contradiction, that $U_3(v') = \emptyset$ and $U_3(v) \neq \emptyset$. Let u be a vertex in $U_3(v)$. Let us suppose, for the sake of contradiction, that u

is not a proper descendant of v'. Since $u \in U_3(v)$, we have $u \in \widetilde{S}_1(v)$. Since v' is also in $\widetilde{S}_1(v)$, this implies that u and v' are related as ancestor and descendant. Thus, since u is not a proper descendant of v', we have that u is an ancestor of v'. Let (x, y) be a back-edge in $B(v')$ such that $y = \mathit{low}(v')$. Lemma 3.2 implies that $B(\mathit{nextM}(v')) \subseteq B(v')$. Thus, we have that $\mathit{low}(v')$ is an ancestor of $\mathit{low}(\mathit{nextM}(v'))$, and therefore a proper ancestor of $\mathit{nextM}(v')$. Since $\mathit{nextM}(v')$ is a proper ancestor of $\mathit{lastM}(v)$, this implies that $\mathit{low}(v')$ is a proper ancestor of $\mathit{lastM}(v)$. Now, since $(x, y) \in B(v')$, we have that x is a descendant of v', and therefore a descendant of u. Furthermore, $y = \mathit{low}(v')$ is a proper ancestor of $\mathit{lastM}(v)$, and therefore a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. But then we have that $\mathit{low}(u) \leq \mathit{low}(v') < \mathit{lastM}(v)$, in contradiction to the fact that $u \in U_3(v)$. Thus, we have that u is a proper descendant of v'. Then, since $u \in U_3(v)$, we have that $\mathit{high}_1(u) \neq \mathit{high}_2(u) = \mathit{high}(v) = \mathit{high}(v')$. Furthermore, we have that $\mathit{low}(u) \geq \mathit{lastM}(v)$, and therefore $\mathit{low}(u) \geq \mathit{nextM}(v')$. This implies that $U_3(v')$ is not empty (because we can consider the lowest proper descendant u' of v' in $\widetilde{S}_1(v') = \widetilde{S}_1(v)$ such that $\mathit{high}_1(u') \neq \mathit{high}_2(u') = \mathit{high}_1(v')$ and $\mathit{low}(u') \geq \mathit{nextM}(v')$). This contradicts our supposition that $U_3(v') \neq \emptyset$. Thus, we have shown that $U_3(v') = \emptyset$ implies that $U_3(v) = \emptyset$.

Now let us assume that $U_3(v) \neq \emptyset$. This implies that $U_3(v')$ is not empty. Let us suppose, for the sake of contradiction, that there is a vertex $u \in U_3(v)$ that is lower than the greatest vertex u' in $U_3(v')$. Since $u \in U_3(v)$ and $u' \in U_3(v')$, we have $u \in \widetilde{S}_1(v)$ and $u' \in \widetilde{S}_1(v')$, respectively. Thus, since $\widetilde{S}_1(v) = \widetilde{S}_1(v')$, this implies that u and u' are related as ancestor and descendant. Thus, since u is lower than u', we have that u is a proper ancestor of u'. Let us suppose, for the sake of contradiction, that $\mathit{low}(u')$ is a proper ancestor of $\mathit{nextM}(v')$. Then, since $\mathit{nextM}(v')$ is a proper ancestor of $\mathit{lastM}(v)$, we have that $\mathit{low}(u')$ is a proper ancestor of $\mathit{lastM}(v)$. Now let (x, y) be a back-edge in $B(u')$ such that $y = \mathit{low}(u')$. Then x is a descendant of u', and therefore a descendant of u. Furthermore, y is a proper ancestor of $\mathit{lastM}(v)$, and therefore a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Thus, we have $\mathit{low}(u) \leq y < \mathit{lastM}(v)$, in contradiction to the fact that $u \in U_3(v)$. Thus, our last supposition is not true, and therefore we have that $\mathit{low}(u')$ is not a proper ancestor of $\mathit{nextM}(v')$. Thus, since $u' \in U_3(v')$, we have that $\mathit{low}(u') \geq \mathit{nextM}(v')$, and u' is the lowest vertex in $\widetilde{S}_1(v')$ with $\mathit{high}_1(u') \neq \mathit{high}_2(u') = \mathit{high}_1(v')$ that has this property $(*)$.

Now we will trace the implications of $u \in U_3(v)$. First, we have that $u \in \widetilde{S}_1(v) =$

$\widetilde{S}_1(v')$. Then, we have $high_1(u) \neq high_2(u) = high_1(v) = high_1(v')$. Furthermore, we have that $low(u) \geq lastM(v)$, and therefore $low(u) > nextM(v')$ (since $nextM(v')$ is a proper ancestor of $lastM(v)$). Finally, we can show as above that u is a proper descendant of v' (the proof of this fact above did not rely on $U_3(v') = \emptyset$). But then, since u is lower than u', we have a contradiction to $(*)$. Thus, we have shown that every vertex in $U_3(v)$ is at least as great as the greatest vertex in $U_3(v')$. In particular, this implies that the lowest vertex in $U_3(v)$ is greater than, or equal to, the greatest vertex in $U_3(v')$. □

Based on Lemma 5.98, we can provide an efficient algorithm for computing the sets $U_3(v)$, for all vertices $v \neq r$ such that $nextM(v) \neq \bot$. The computation takes place on segments of $\widetilde{H}(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant. Specifically, let $v \neq r$ be a vertex such that $nextM(v) \neq \bot$. Then we have $U_3(v) \subset \widetilde{S}_1(v)$. In other words, $U_3(v)$ is a subset of the segment of $\widetilde{H}(high_1(v))$ that contains v and is maximal w.r.t. the property that its elements are related as ancestor and descendant. So let $z_1, \dots, z_k$ the vertices of $\widetilde{S}_1(v)$, sorted in decreasing order. Then, we have that $v = z_i$, for an $i \in \{1, \dots, k\}$. By definition, $U_3(v)$ contains every vertex u in $\{z_1, \dots, z_{i-1}\}$ such that either $high_1(u) \neq high_2(u) = high_1(v)$ and $nextM(v) > low(u) \geq lastM(v)$, or u is the lowest vertex in this set with $high_1(u) \neq high_2(u) = high_1(v)$ and $low(u) \geq nextM(v)$. As an implication of Lemma 3.24, we have that the vertices in $\{z_1, \dots, z_{i-1}\}$ are sorted in decreasing order w.r.t. their *low* point. Thus, it is sufficient to process the vertices from $\{z_1, \dots, z_{i-1}\}$ in reverse order, in order to find the first vertex u that has $low(u) \geq lastM(v)$. Then, we keep traversing this set in reverse order, and, as long as the *low* point of every vertex u that we meet is lower than $nextM(v)$, we insert u into $U_3(v)$, provided that it satisfies $high_1(u) \neq high_2(u) = high_1(v)$. Then, once we reach a vertex with *low* point no lower than $nextM(v)$, we keep traversing this set in reverse order, until we meet one more u that satisfies $high_1(u) \neq high_2(u) = high_1(v)$, which we also insert into $U_3(v)$, and we are done.

Now, if there is a proper ancestor v' of v in $\widetilde{S}_1(v)$ such that $high_1(v') = high_1(v)$, then we have that $\widetilde{S}_1(v') = \widetilde{S}_1(v)$. If $nextM(v') \neq \bot$, then we have that $U_3(v')$ is defined. Then we can follow the same process as above in order to compute $U_3(v')$. Furthermore, according to Lemma 5.98, it is sufficient to start from the greatest element of $U_3(v)$ (i.e., the one that was inserted last into $U_3(v)$). In particular, if $U_3(v) = \emptyset$, then it is certain that $U_3(v') = \emptyset$, and therefore we are done. Otherwise, we just pick up the

computation from the greatest vertex in $U_3(v)$. In order to perform efficiently those computations, first we compute, for every vertex x, the collection $\mathcal{S}(x)$ of the segments of $\widetilde{H}(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant. For every vertex x, this computation takes $O(|\widetilde{H}(x)|)$ time, according to Lemma 3.23. Since every vertex participates in at most two sets of the form $\widetilde{H}(x)$, we have that the total size of all $\mathcal{S}(x)$, for all vertices x, is $O(n)$. Then it is sufficient to process separately all segments of $\mathcal{S}(x)$, for every vertex x, as described above, by starting the computation each time from the first vertex v of the segment that satisfies $\mathit{nextM}(v) \neq \perp$ and $\mathit{high}_1(v) = x$. The whole procedure is shown in Algorithm 44. The result is formally stated in Lemma 5.99.

Lemma 5.99. *Algorithm 44 correctly computes the sets $U_3(v)$, for all vertices $v \neq r$ such that $\mathit{nextM}(v) \neq \perp$. Furthermore, it runs in $O(n)$ time.*

Proof. This was basically given in the main text, in the two paragraphs above Algorithm 44. □

Lemma 5.100. *Let (u, v, w) be a triple of vertices such that $u \in U_3(v)$. Then, (u, v, w) induces a Type-3βii-3 4-cut if and only if: (1) w is the greatest proper ancestor of v such that $M(w) = M(v)$ and $w \leq \mathit{low}(u)$, and (2) $\mathit{bcount}(v) = \mathit{bcount}(u) + \mathit{bcount}(w) - 1$.*

Proof. ($\Rightarrow$) By definition of Type-3βii-3 4-cuts, we have that w is a proper ancestor of v with $M(w) = M(v)$. Furthermore, we have $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$, where e is the back-edge of the 4-cut induced by (u, v, w). This implies that $\mathit{bcount}(v) = \mathit{bcount}(u) + \mathit{bcount}(w) - 1$. Lemma 5.96 implies that $w \leq \mathit{low}(u)$.

Now let us suppose, for the sake of contradiction, that there is a proper ancestor w' of v with $w' > w$ such that $M(w') = M(v)$ and $w' \leq \mathit{low}(u)$. Since $M(w') = M(w)$ and $w' > w$, we have that w' is a proper descendant of w, and Lemma 3.2 implies that $B(w) \subseteq B(w')$. Since the graph is 3-edge-connected, this can be strengthened to $B(w) \subset B(w')$. Thus, there is a back-edge $(x, y) \in B(w') \setminus B(w)$. Then we have that x is a descendant of $M(w') = M(v)$. Furthermore, we have that y is a proper ancestor of w', and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Then, since $(x, y) \notin B(w)$, $B(v) = (B(u) \setminus \{e\}) \sqcup B(w)$ implies that $(x, y) \in B(u) \setminus \{e\}$. Since y is a proper ancestor of w', we have that $y < w'$. Then, $w' \leq \mathit{low}(u)$ implies that $y < \mathit{low}(u)$, in contradiction to $(x, y) \in B(u)$. This shows that w is the greatest proper ancestor of v such that $M(w) = M(v)$ and $w \leq \mathit{low}(u)$.

Algorithm 44: Compute the sets $U_3(v)$, for all vertices $v \neq r$ such that $nextM(v) \neq \perp$

```
1  foreach vertex x do
2      compute the collection S(x) of the segments of H̃(x) that are maximal w.r.t. the
         property that their elements are related as ancestor and descendant
3  end
4  foreach vertex v ≠ r such that nextM(v) ≠ ⊥ do
5      set U3(v) ← ∅
6  end
7  foreach vertex x do
8      foreach segment S ∈ S(x) do
9          let v be the first vertex in S
10         while v ≠ ⊥ and (high1(v) ≠ x or nextM(v) = ⊥) do
11             v ← nextS(v)
12         end
13         if v = ⊥ then continue
14         let u ← prevS(v)
15         while v ≠ ⊥ do
16             while u ≠ ⊥ and low(u) < lastM(v) do
17                 u ← prevS(u)
18             end
19             while u ≠ ⊥ and low(u) < nextM(v) do
20                 if high1(u) ≠ high2(u) and high2(u) = x then
21                     insert u into U3(v)
22                 end
23                 u ← prevS(u)
24             end
25             while u ≠ ⊥ and (high1(u) = high2(u) or high2(u) ≠ x) do
26                 u ← prevS(u)
27             end
28             if u ≠ ⊥ then
29                 insert u into U3(v)
30             end
31             v ← nextS(v)
32             while v ≠ ⊥ and (high1(v) ≠ x or nextM(v) = ⊥) do
33                 v ← nextS(v)
34             end
35         end
36     end
37 end
```

($\Leftarrow$) Since $u \in U_3(v)$, we have that u is a proper descendant of v with $high_1(u) \neq high_2(u) = high(v)$. Let (x, y) be a back-edge in $B(u) \setminus \{e_{high}(u)\}$. Then, we have that x is a descendant of u, and therefore a descendant of v. Furthermore, we have that $y \leq high_2(u) = high(v)$, and therefore $y < v$. Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Thus, $y < v$ implies that y is a proper ancestor of v. Therefore, since x is a descendant of v, we have $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u) \setminus \{e_{high}(u)\}$, this implies that $B(u) \setminus \{e_{high}(u)\} \subseteq B(v)$. Since w is a proper ancestor of v with $M(w) = M(v)$, Lemma 3.2 implies that $B(w) \subseteq B(v)$. Let (x, y) be a back-edge in $B(w)$. Then we have that y is a proper ancestor of w, and therefore $y < w$. Thus, $w \leq low(u)$ implies that $y < low(u)$. Therefore, we cannot have $(x, y) \in B(u)$. This shows that $B(u) \cap B(w) = \emptyset$.

Thus, since $B(u) \setminus \{e_{high}(u)\} \subseteq B(v)$ and $B(w) \subseteq B(v)$ and $B(u) \cap B(w) = \emptyset$ and $bcount(v) = bcount(u) + bcount(w) - 1$, we have that $B(v) = (B(u) \setminus \{e_{high}(u)\}) \sqcup B(w)$. Furthermore, since $B(u) \cap B(w) = \emptyset$, we have that $e_{high}(u) \notin B(w)$, and therefore $B(v) = (B(u) \setminus \{e_{high}(u)\}) \sqcup B(w)$ implies that $e_{high}(u) \notin B(v) \cup B(w)$. Thus, since $M(w) = M(v)$, we have that (u, v, w) induces a Type-3βii-3 4-cut. ☐

Now we are ready to describe the algorithm for computing a collection of enough Type-3βii-3 4-cuts, so that the rest of them are implied from this collection, plus that computed by Algorithm 24. So let (u, v, w) be a triple of vertices that induces a Type-3βii-3 4-cut. Then, Lemma 5.97 implies that either $u \in U_3(v)$, or $(\tilde{u}, v, w)$ induces a Type-3βii-3 4-cut, where $\tilde{u}$ is the greatest vertex in $U_3(v)$. Furthermore, if $u \notin U_3(v)$, then $B(u) \sqcup \{e_{high}(\tilde{u})\} = B(\tilde{u}) \sqcup \{e_{high}(u)\}$. Thus, if $u \notin U_3(v)$, then it is sufficient to have computed the 4-cut C induced by $(\tilde{u}, v, w)$, because then the one induced by (u, v, w) is implied by C, plus some Type-2ii 4-cuts that are computed by Algorithm 24 (see Proposition 5.28). Now, if $u \in U_3(v)$, then by Lemma 5.100 we have that w is the greatest proper ancestor of v such that $w \leq low(u)$ and $M(w) = M(v)$. Thus, we can use Algorithm 22 in order to get w from v and u.

The full procedure for computing enough Type-3βii-3 4-cuts is shown in Algorithm 45. The proof of correctness and linear complexity is given in Proposition 5.28.

Proposition 5.28. *Algorithm 45 computes a collection $\mathcal{C}$ of Type-3βii-3 4-cuts, and it runs in $O(n)$ time. Furthermore, let $\mathcal{C}'$ be the collection of Type-2ii 4-cuts computed by Algorithm 24. Then, every Type-3βii-3 4-cut is implied by $\mathcal{C} \cup \mathcal{C}'$.*

Algorithm 45: Compute a collection of Type-3βii-3 4-cuts, so that all Type-3βii-3 4-cuts are implied from this collection, plus that of the Type-2ii 4-cuts returned by Algorithm 24

```
1  foreach vertex v ≠ r such that nextM(v) ≠ ⊥ do
2      compute U_3(v)
3  end
4  foreach vertex v ≠ r such that nextM(v) ≠ ⊥ do
5      foreach u ∈ U_3(v) do
6          let w be the greatest proper ancestor of v such that w ≤ low(u) and
            M(w) = M(v)
7          if bcount(v) = bcount(u) + bcount(w) − 1 then
8              mark {(u, p(u)), (v, p(v)), (w, p(w)), e_high(u)} as a Type-3βii-3 4-cut
9          end
10     end
11 end
```

Proof. When we reach Line 8, notice that the following conditions are true. (1) $u \in U_3(v)$, (2) w is the greatest proper ancestor of v such that $w \leq \mathit{low}(u)$ and $M(w) = M(v)$, and (3) $\mathit{bcount}(v) = \mathit{bcount}(u) + \mathit{bcount}(w) - 1$. Thus, Lemma 5.100 implies that (u, v, w) induces a Type-3βii-3 4-cut. By Lemma 5.96, we have that the back-edge in this 4-cut is $e_{\mathit{high}}(u)$. Thus, it is correct to mark $\{(u, p(u)), (v, p(v)), (w, p(w)), e_{\mathit{high}}(u)\}$ as a Type-3βii-3 4-cut. This shows that the collection $\mathcal{C}$ of the 4-element sets marked by Algorithm 45 is a collection of Type-3βii-3 4-cuts.

According to Lemma 5.99, the computation of all sets $U_3(v)$, for all vertices $v \neq r$ such that $\mathit{nextM}(v) \neq \bot$, can be performed in $O(n)$ time in total, using Algorithm 44. Thus, the **for** loop in Line 1 can be performed in $O(n)$ time. In particular, this implies that the total size of all U_3 sets is $O(n)$. It remains to explain how to compute the w in Line 6, for every vertex $u \in U_3(v)$, for every vertex $v \neq r$ such that $\mathit{nextM}(v) \neq \bot$. For this purpose, we can simply use Algorithm 22. First, we let $M^{-1}(x)$, for every vertex x, be the collection of all vertices w such that $M(w) = x$. Thus, if $x \neq x'$, then $M^{-1}(x) \cap M^{-1}(x') = \emptyset$. Then, for every vertex $v \neq r$ such that $\mathit{nextM}(v) \neq \bot$, and every vertex $u \in U_3(v)$, we generate a query $q(M^{-1}(M(v)), \mathit{min}\{\mathit{low}(u), p(v)\})$. This is to return the greatest w that has $M(w) = M(v)$, $w \leq \mathit{low}(u)$ and $w \leq p(v)$. In

particular, since $M(w)=M(v)$ and $w \leq p(v)$, we have that w is a proper ancestor of v. Thus, the w returned is the greatest proper ancestor of v with $M(w)=M(v)$ such that $w \leq \mathit{low}(u)$. The number of all those queries is bounded by the total size of the U_3 sets, which is bounded by $O(n)$. Thus, Lemma 5.27 implies that all these queries can be answered in $O(n)$ time in total.

Now let (u,v,w) be a triple of vertices that induces a Type-3βii-3 4-cut C. If $u \in U_3(v)$, then by Lemma 5.100 we have that w is uniquely determined by u and v, and therefore C has been marked at some point in Line 8. So let us assume that $u \notin U_3(v)$. Then, Lemma 5.97 implies that $(\tilde{u},v,w)$ induces a Type-3βii-3 4-cut C', where $\tilde{u}$ is the greatest vertex in $U_3(v)$. Thus, $C' \in \mathcal{C}$. Furthermore, Lemma 5.97 implies $B(u) \sqcup \{e_{high}(\tilde{u})\} = B(\tilde{u}) \sqcup \{e_{high}(u)\}$. Thus, by Lemma 5.28 we have that $C'' = \{(u,p(u)),(\tilde{u},p(\tilde{u})),e_{high}(u),e_{high}(\tilde{u})\}$ is a Type-2ii 4-cut. By Lemma 5.96, we have that the 4-cuts induced by (u,v,w) and $(\tilde{u},v,w)$ are $\{(u,p(u)),(v,p(v)),(w,p(w)),e_{high}(u)\}$ and $\{(\tilde{u},p(\tilde{u})),(v,p(v)),(w,p(w)),e_{high}(\tilde{u})\}$, respectively. Notice that C is implied by C' and C'' through the pair of edges $\{(u,p(u)),e_{high}(u)\}$. According to Proposition 5.12, we have that C'' is implied by $\mathcal{C}'$ through the pair of edges $\{(u,p(u)),e_{high}(u)\}$. Thus, by Lemma 5.7 we have that C is implied by $\mathcal{C} \cup \mathcal{C}'$. □

5.8.2.4 Type-3βii-4 4-cuts

Now we consider case (4) of Lemma 5.57.

Let u,v,w be three vertices $\neq r$ such that w is proper ancestor of v, v is a proper ancestor of u, and there is a back-edge $e \in B(v)$ such $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ and $M(B(v) \setminus \{e\}) = M(w)$. By Lemma 5.57, we have that $C = \{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ is a 4-cut, and we call this a Type-3βii-4 4-cut.

The following lemma provides some useful information concerning this type of 4-cuts.

Lemma 5.101. *Let (u,v,w) be a triple of vertices that induces a Type-3βii-4 4-cut, and let e be the back-edge in the 4-cut induced by (u,v,w). Then $w \leq \mathit{low}(u)$, and either $\mathit{high}_1(v) = \mathit{high}(u)$, or $\mathit{high}_1(v) > \mathit{high}(u)$ and $\mathit{high}_2(v) = \mathit{high}(u)$. If $\mathit{high}_1(v) \neq \mathit{high}(u)$, then $e = e_{high}(v)$.*

Proof. Since (u,v,w) induces a Type-3βii-4 4-cut, we have that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, where e is the back-edge in the 4-cut induced by (u,v,w). This implies that $B(u) \cap B(w) = \emptyset$. Since $\mathit{low}(u)$ and w have u as a common descendant, we have that

$low(u)$ and w are related as ancestor and descendant. Let us suppose, for the sake of contradiction, that $low(u) < w$. Then, we have that $low(u)$ is a proper ancestor of w. Let (x, y) be a back-edge in $B(u)$ such that $y = low(u)$. Then x is a descendant of u, and therefore a descendant of w. Furthermore, y is a proper ancestor of w. This shows that $(x, y) \in B(w)$, in contradiction to the fact that $B(u) \cap B(w) = \emptyset$. This shows that $low(u) \geq w$.

Now let us assume that $high_1(v) \neq high(u)$. (If $high_1(v) = high(u)$, then there is nothing further to show.) Since $B(u) \subseteq B(v)$, we have that $high_1(v) \geq high(u)$. Thus, $high_1(v) \neq high(u)$ implies that $high_1(v) > high(u)$. Let (x, y) be a back-edge in $B(v)$ such that $y > high(u)$. Then, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. Since $y > high(u)$, the case $(x, y) \in B(u)$ is rejected. Furthermore, since $w \leq low(u) \leq high(u) < y$, the case $(x, y) \in B(w)$ is rejected too. Thus, we have $(x, y) = e$. Therefore, since $high_1(v) > high(u)$, we have that $e = high(v)$.

Now let (x', y') be a back-edge in $B(v)$ such that $y' = high_2(v)$ and $(x', y') \neq e$. Then we cannot have $y' > high(u)$, because otherwise we would conclude as previously that $(x', y') = e$, which is impossible. Thus, we have $y' \leq high(u)$. Let us suppose, for the sake of contradiction, that $y' < high(u)$. Since $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, we have that the $high$-edge of $B(u)$ is in $B(v)$. Thus, there is a back-edge (x'', y'') in $B(v)$ such that $y'' = high(u)$. Then we have $high_1(v) > y'' > high_2(v)$, which contradicts the definition of the $high_2$ point. Thus, we conclude that $high_2(v) = high(u)$. □

Lemma 5.102. *Let (u, v, w) be a triple of vertices that induces a Type-3βii-4 4-cut, such that $high_1(v) \neq high(u)$. Let u' be a vertex in $\widetilde{H}(high_2(v))$ such that $u \geq u' \geq v$. Then u' is an ancestor of u.*

Proof. Since (u, v, w) induces a Type-3βii-4 4-cut, we have that u is a descendant of v. Thus, $u \geq u' \geq v$ implies that u' is also a descendant of v. Since $high_1(v) \neq high(u)$, by Lemma 5.101 we have that $high_2(v) = high(u)$ and $e = e_{high}(v)$, where e is the back-edge in the 4-cut induced by (u, v, w).

Since $u' \in \widetilde{H}(high_2(v))$, we have that either $high_1(u') = high_2(v)$, or $high_2(u') = high_2(v)$. In either case then, there is a back-edge $(x, y) \in B(u)$ such that $y = high_2(v)$. Then, we have that x is a descendant of u', and therefore a descendant of v. Furhermore, $y = high_2(v)$ is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Since (u, v, w) induces a Type-3βii-4 4-cut, by Lemma 5.101 we have that $w \leq low(u)$. Then, we have that $w \leq low(u) \leq high(u) = high_2(v) = y$. This implies that y cannot

be a proper ancestor of w, and therefore we have that $(x,y) \notin B(w)$. Since (u,v,w) induces a Type-3βii-4 4-cut, we have that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. Thus, since $(x,y) \in B(v)$ and $(x,y) \notin B(w)$, we have that either $(x,y) \in B(u)$, or $(x,y) = e$.

Let us suppose, for the sake of contradiction, that $(x,y) = e$. Since $e = e_{high}(v)$, we have that $y = high_1(v)$. Since $high_1(v) \neq high(u)$ and $high(u) = high_2(v)$, we have that $high_1(v) \neq high_2(v)$. But then $y = high_1(v)$ contradicts the fact that $y = high_2(v)$. This shows that $(x,y) \neq e$. Thus, we have that $(x,y) \in B(u)$. This implies that x is a common descendant of u and u'. Thus, we have that u and u' are related as ancestor and descendant. Then, $u \geq u'$ implies that u' is an ancestor of u. □

Lemma 5.103. *Let (u,v,w) be a triple of vertices that induces a Type-3βii-4 4-cut, such that $M(B(v) \setminus \{e\}) \neq M(v)$, where e is the back-edge in the 4-cut induced by (u,v,w). Let u' be a vertex in $H(high_1(v))$ such that $u \geq u' \geq v$. Then u' is an ancestor of u.*

Proof. Since (u,v,w) induces a Type-3βii-4 4-cut, we have that u is a descendant of v. Thus, $u \geq u' \geq v$ implies that u' is also a descendant of v. Lemma 5.101 implies that either $high_1(v) = high(u)$, or $high_2(v) = high(u)$. In any case, then, since $high_2(v) \leq high_1(v)$, we have that $high(u) \leq high_1(v)$.

Since $u' \in H(high_1(v))$, we have that $high(u') = high_1(v)$. Thus, there is a back-edge $(x,y) \in B(u)$ such that $y = high_1(v)$. Then, we have that x is a descendant of u', and therefore a descendant of v. Furhermore, $y = high_1(v)$ is a proper ancestor of v. This shows that $(x,y) \in B(v)$. Since (u,v,w) induces a Type-3βii-4 4-cut, by Lemma 5.101 we have that $w \leq low(u)$. Then, we have that $w \leq low(u) \leq high(u) \leq high_1(v) = y$. This implies that y cannot be a proper ancestor of w, and therefore we have that $(x,y) \notin B(w)$. Since (u,v,w) induces a Type-3βii-4 4-cut, we have that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. Thus, since $(x,y) \in B(v)$ and $(x,y) \notin B(w)$, we have that either $(x,y) \in B(u)$, or $(x,y) = e$. If $(x,y) \in B(u)$, then we have that x is a common descendant of u and u'. Thus, u and u' are related as ancestor and descendant. Then, $u \geq u'$ implies that u' is an ancestor of u. So let us assume that $(x,y) \notin B(u)$. This implies that $(x,y) = e$.

Let us suppose, for the sake of contradiction, that u' is not an ancestor of $M(v)$ $(*)$. Since $e = (x,y) \in B(u') \cap B(v)$, we have that x is a descendant of both u' and $M(v)$. Thus, u' and $M(v)$ are related as ancestor and descendant. Then, since u' is not an ancestor of $M(v)$, we have that u' is a proper descendant of $M(v)$. Let c be the child of $M(v)$ that is an ancestor of u'. Since $M(B(v) \setminus \{e\}) \neq M(v)$, we have that e is the

only back-edge in $B(v)$ whose higher endpoint is not a descendant of $M(B(v) \setminus \{e\})$. Furthermore, since $M(B(v) \setminus \{e\}) \neq M(v)$, we have that $M(B(v) \setminus \{e\})$ is a proper descendant of $M(v)$. Let c' be the child of $M(v)$ that is an ancestor of $M(B(v) \setminus \{e\})$.

Let (x', y') be a back-edge in $B(v)$. Then, if $(x', y') = e$, we have that $(x', y') = (x, y)$, and therefore x' is a descendant of u', and therefore a descendant of c. Otherwise, if $(x', y') \neq e$, then we have that $(x', y') \in B(v) \setminus \{e\}$, and therefore x' is a descendant of $M(B(v) \setminus \{e\})$, and therefore a descendant of c'. This shows that there is no back-edge of the form $(M(v), z) \in B(v)$. Let us suppose, for the sake of contradiction, that $c = c'$. Then, the previous argument shows that all back-edges in $B(v)$ stem from $T(c')$, and therefore $M(v)$ is a descendant of c', which is absurd. Thus, we have that $c \neq c'$.

Since the graph is 3-edge-connected, we have that $|B(u')| > 1$. Thus, there is a back-edge $(x', y') \in B(u') \setminus \{e\}$. Since u' is a descendant of v with $high(u') = high_1(v)$, by Lemma 3.3 we have that $B(u') \subseteq B(v)$. Thus, $(x', y') \in B(u')$ implies that $(x', y') \in B(v)$. Then, since $(x', y') \neq e$, we have that $(x', y') \in B(v) \setminus \{e\}$, and therefore x' is a descendant of $M(B(v) \setminus \{e\})$, and therefore a descendant of c'. But since $(x', y') \in B(u')$, we have that x' is a descendant of u', and therefore a descendant of c. Thus, x' is a common descendant of c and c', and therefore c and c' are related as ancestor and descendant, which is absurd. Thus, starting from $(*)$, we have arrived at a contradiction. This shows that u' is an ancestor of $M(v)$.

Since $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, we have that $B(u) \subseteq B(v)$, and therefore $M(u)$ is a descendant of $M(v)$. Since u' is an ancestor of $M(v)$, this implies that $M(u)$ is a descendant of u'. Thus, $M(u)$ is a common descendant of u and u', and therefore u and u' are related as ancestor and descendant. Thus, $u \geq u'$ implies that u' is an ancestor of u. $\square$

Lemma 5.104. *Let (u, v, w) be a triple of vertices that induces a Type-3βii-4 4-cut, such that $high_1(v) = high(u)$ and $y \neq high_1(v)$, where y is the lower endpoint of the back-edge in the 4-cut induced by (u, v, w). Let u' be a vertex in $H(high_1(v))$ such that $u \geq u' \geq v$. Then u' is an ancestor of u.*

Proof. Let $e = (x, y)$ be the back-edge in the 4-cut induced by (u, v, w). Since (u, v, w) induces a Type-3βii-4 4-cut, we have that u is a descendant of v. Thus, $u \geq u' \geq v$ implies that u' is also a descendant of v. Since $u' \in H(high_1(v))$, we have $high(u') = high_1(v)$. This implies that there is a back-edge $(x', y') \in B(u')$ such that $y' = high_1(v)$. Then, x' is a descendant of u', and therefore a descendant of v. Furthermore, $y =$

$high_1(v)$ is a proper ancestor of v. This shows that $(x',y') \in B(v)$. Since (u,v,w) induces a Type-3βii-4 4-cut, by Lemma 5.101 we have $w \leq low(u)$. Thus, we have $w \leq low(u) \leq high(u) = high_1(v) = y'$. This implies that y' cannot be a proper ancestor of w, and therefore we have $(x',y') \notin B(w)$. Since (u,v,w) induces a Type-3βii-4 4-cut, we have $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. Thus, since $(x',y') \in B(v)$ and $(x',y') \notin B(w)$, we have that either $(x',y') \in B(u)$, or $(x',y') = e$. The case $(x',y') = e$ is rejected, because it implies that $y' = y$, contradicting the fact that $y' = high_1(v)$ and $y \neq high_1(v)$. Thus, we have $(x',y') \in B(u)$. This implies that x' is a descendant of u. Thus, x' is a common descendant of u and u', and therefore u and u' are related as ancestor and descendant. Since $u \geq u'$, this implies that u' is an ancestor of u. ☐

Here, we distinguish the following four different cases, depending on the location of the endpoints of the back-edge e:

1. $M(B(v) \setminus \{e\}) \neq M(v)$ and $high_1(v) > high(u)$.
2. $M(B(v) \setminus \{e\}) \neq M(v)$ and $high_1(v) = high(u)$.
3. $M(B(v) \setminus \{e\}) = M(v)$ and $high_1(v) > high(u)$.
4. $M(B(v) \setminus \{e\}) = M(v)$ and $high_1(v) = high(u)$.

The case where $M(B(v) \setminus \{e\}) \neq M(v)$ and $high_1(v) > high(u)$

Let (u,v,w) be a triple of vertices that induces a Type-3βii-4 4-cut, such that $M(B(v) \setminus \{e\}) \neq M(v)$ and $high_1(v) > high(u)$, where e is the back-edge in the 4-cut induced by (u,v,w). Then, by Lemma 5.101 we have $high_1(v) \neq high_2(v)$ and $e = e_{high}(v)$.

Let $v \neq r$ be a vertex such that $high_1(v) \neq high_2(v)$ and $M(B(v) \setminus \{e_{high}(v)\}) \neq M(v)$. Then, we let $V(v)$ denote the collection of all vertices $v' \neq r$ such that $high_1(v') \neq high_2(v') = high_2(v)$ and $M(B(v) \setminus \{e_{high}(v)\}) = M(B(v') \setminus \{e_{high}(v')\}) \neq M(v')$. (We note that $v \in V(v)$.) If a vertex $v \neq r$ does not satisfy $high_1(v) \neq high_2(v)$ and $M(B(v) \setminus \{e_{high}(v)\}) \neq M(v)$, then we let $V(v) = \emptyset$.

Lemma 5.105. *Let v and v' be two distinct vertices such that $v' \in V(v)$. Then, $B(v) \sqcup \{e_{high}(v')\} = B(v') \sqcup \{e_{high}(v)\}$.*

Proof. Since $v' \in V(v)$, we have $high_1(v) \neq high_2(v) = high_2(v') \neq high_1(v')$ and $M(v) \neq M(B(v) \setminus \{e_{high}(v)\}) = M(B(v') \setminus \{e_{high}(v')\}) \neq M(v')$. Since $high_1(v) \neq high_2(v)$, we have $high_1(v) > high_2(v)$. Similarly, we have $high_1(v') > high_2(v')$. Furthermore, $e_{high}(v)$ is the unique back-edge in $B(v)$ whose lower endpoint is $high_1(v)$, and $e_{high}(v')$ is the unique back-edge in $B(v')$ whose lower endpoint is $high_1(v')$.

Let (x, y) be a back-edge in $B(v)$ such that $y = high_2(v)$. Then we have that x is a descendant of v. Furthermore, since $(x, y) \in B(v) \setminus \{e_{high}(v)\}$ and $M(B(v) \setminus \{e_{high}(v)\}) = M(B(v') \setminus \{e_{high}(v')\})$, we have that x is a descendant of $M(B(v') \setminus \{e_{high}(v')\})$, and therefore a descendant of $M(v')$, and therefore a descendant of v'. Thus, x is a common descendant of v and v', and therefore we have that v and v' are related as ancestor and descendant. Thus, we may assume w.l.o.g. that v' is a proper ancestor of v.

Let us suppose, for the sake of contradiction, that $e_{high}(v') \in B(v)$. Then, since $high_1(v') > high_2(v') = high_2(v)$, we have that $e_{high}(v) = e_{high}(v')$. This implies that $high_1(v') = high_1(v)$. Thus, since v' is a proper ancestor of v, Lemma 3.3 implies that $B(v) \subseteq B(v')$. Since the graph is 3-edge-connected, this can be strengthened to $B(v) \subset B(v')$. Thus, there is a back-edge $(x, y) \in B(v') \setminus B(v)$. Then y is a proper ancestor of v', and therefore a proper ancestor of v. Thus, x cannot be a descendant of v. Since $M(v')$ is an ancestor of x, this implies that $M(v')$ cannot be a descendant of v. Let (x', y') be a back-edge in $B(v)$. Then, x' is a descendant of v. Furthermore, $B(v) \subseteq B(v')$ implies that $(x', y') \in B(v')$, and therefore x' is a descendant of $M(v')$. Thus, x' is a common descendant of v and $M(v')$, and therefore v and $M(v')$ are related as ancestor and descendant. Since $M(v')$ is not a descendant of v, this implies that $M(v')$ is a proper ancestor of v. Let c be the child of $M(v')$ that is an ancestor of v. Since $e_{high}(v') \in B(v)$, we have that the higher endpoint of $e_{high}(v')$ is a descendant of v, and therefore a descendant of c. Now let (x', y') be a back-edge in $B(v')$. If $(x', y') = e_{high}(v')$, then we have that x' is a descendant of c. If $(x', y') \neq e_{high}(v')$, then we have $(x', y') \in B(v') \setminus \{e_{high}(v')\}$, and therefore x is a descendant of $M(B(v') \setminus \{e_{high}(v')\}) = M(B(v) \setminus \{e_{high}(v)\})$, and therefore a descendant of v, and therefore a descendant of c. Thus, in either case we have that x' is a descendant of c. Due to the generality of $(x', y') \in B(v')$, this implies that $M(v')$ is a descendant of c. But this is absurd, since c is a child of $M(v')$. This shows that $e_{high}(v') \notin B(v)$.

Notice that we cannot have $e_{high}(v) \in B(v')$. (Because otherwise, since $high_1(v) > high_2(v) = high_2(v')$, we would have $e_{high}(v) = e_{high}(v')$, contradicting the fact that $e_{high}(v') \notin B(v)$.) Now let (x, y) be a back-edge in $B(v) \setminus \{e_{high}(v)\}$. Then x is

a descendant of v, and therefore a descendant of v'. Furthermore, y is an ancestor of $high_2(v) = high_2(v')$, and therefore a proper ancestor of v'. This shows that $(x,y) \in B(v')$. Due to the generality of $(x,y) \in B(v) \setminus \{e_{high}(v)\}$, this implies that $B(v) \setminus \{e_{high}(v)\} \subseteq B(v')$. And since $e_{high}(v') \notin B(v)$, this can be strengthened to $B(v) \setminus \{e_{high}(v)\} \subseteq B(v') \setminus \{e_{high}(v')\}$. Conversely, let (x,y) be a back-edge in $B(v') \setminus \{e_{high}(v')\}$. Then, x is a descendant of $M(B(v') \setminus \{e_{high}(v')\})$, and therefore a descendant of $M(B(v) \setminus \{e_{high}(v)\})$, and therefore a descendant of $M(v)$. Furthermore, y is an ancestor of $high_2(v') = high_2(v)$, and therefore a proper ancestor of v. This shows that $(x,y) \in B(v)$. Due to the generality of $(x,y) \in B(v') \setminus \{e_{high}(v')\}$, this implies that $B(v') \setminus \{e_{high}(v')\} \subseteq B(v)$. And since $e_{high}(v) \notin B(v')$, this can be strengthened to $B(v') \setminus \{e_{high}(v')\} \subseteq B(v) \setminus \{e_{high}(v)\}$. Thus, we have shown that $B(v) \setminus \{e_{high}(v)\} = B(v') \setminus \{e_{high}(v')\}$. Since $e_{high}(v) \notin B(v')$ and $e_{high}(v') \notin B(v)$, this implies that $B(v) \sqcup \{e_{high}(v')\} = B(v') \sqcup \{e_{high}(v)\}$. $\square$

Lemma 5.106. *Let v and v' be two distinct vertices such that $v' \in V(v)$. Then, v and v' belong to the same segment of $\widetilde{H}(high_2(v))$ that is maximal w.r.t. the property that its elements are related as ancestor and descendant.*

Proof. Since $v' \in V(v)$, we have that $M(v) \neq M(B(v) \setminus \{e_{high}(v)\}) = M(B(v') \setminus \{e_{high}(v')\}) \neq M(v')$ and $high_1(v) \neq high_2(v) = high_2(v') \neq high_1(v')$. Thus, both v and v' belong to $\widetilde{H}(high_2(v))$. Furthermore, we have that $M(B(v) \setminus \{e_{high}(v)\}) = M(B(v') \setminus \{e_{high}(v')\})$ is a common descendant of $M(v)$ and $M(v')$, and therefore a common descendant of v and v'. Thus, v and v' are related as ancestor and descendant. We may assume w.l.o.g. that v is a proper descendant of v'. This implies that $v > v'$.

Let us suppose, for the sake of contradiction, that v and v' do not belong to a segment of $\widetilde{H}(high_2(v))$ with the property that its elements are related as ancestor and descendant. Since $\widetilde{H}$ is sorted in decreasing order, this means that there is a vertex $v'' \in \widetilde{H}(high_2(v))$ such that $v > v'' > v'$, and v'' is not an ancestor of v. Notice that, since $v > v'' > v'$ and v is a descendant of v', we have that v'' is also a descendant of v'.

Since $v'' \in \widetilde{H}(high_2(v))$ we have that either $high_1(v'') = high_2(v)$, or $high_2(v'') = high_2(v)$. Thus, in either case, there is a back-edge (x,y) in $B(v'')$ such that $y = high_2(v)$. Then, we have that x is a descendant of v'', and therefore a descendant of v'. Furthermore, we have $y = high_2(v) = high_2(v')$, and therefore y is a proper

ancestor of v'. This shows that $(x,y) \in B(v')$. Notice that we cannot have that x is a descendant of v (because otherwise, x is a common descendant of v and v'', and therefore v and v'' are related as ancestor and descendant, which is impossible, since $v'' < v$ and v'' is not an ancestor of v). This implies that $(x,y) \notin B(v)$. Thus, we have $(x,y) \in B(v') \setminus B(v)$. Lemma 5.105 implies that $B(v') \setminus B(v) = \{e_{high}(v')\}$. Thus, $(x,y) = e_{high}(v')$. But $y = high_2(v) = high_2(v')$ and $high_2(v') \neq high_1(v')$, a contradiction. This shows that v and v' belong to a segment of $\widetilde{H}(high_2(v))$ with the property that its elements are related as ancestor and descendant, and so they belong to a maximal such segment. □

Let $v \neq r$ be a vertex such that $high_1(v) \neq high_2(v)$ and $M(B(v) \setminus \{e_{high}(v)\}) \neq M(v)$. We let $\widetilde{W}(v)$ denote the collection of all ancestors w of $high_2(v)$ such that $M(w) = M(B(v) \setminus \{e_{high}(v)\})$. We also let $\widetilde{firstW}(v) := max(\widetilde{W}(v))$ and $\widetilde{lastW}(v) := min(\widetilde{W}(v))$. (If $\widetilde{W}(v) = \emptyset$, then we let $\widetilde{firstW}(v) := \bot$ and $\widetilde{lastW}(v) := \bot$.)

Lemma 5.107. *The values $\widetilde{firstW}(v)$ and $\widetilde{lastW}(v)$ can be computed in linear time in total, for all vertices $v \neq r$.*

Proof. Let $v \neq r$ be a vertex. If $M(B(v) \setminus \{e_{high}(v)\}) = M(v)$ or $high_1(v) = high_2(v)$, then by definition we have $\widetilde{W}(v) = \emptyset$, and therefore $\widetilde{firstW}(v) = \bot$ and $\widetilde{lastW}(v) = \bot$. So let us assume that $M(B(v) \setminus \{e_{high}(v)\}) \neq M(v)$ and $high_1(v) \neq high_2(v)$.

Let $x = M(B(v) \setminus \{e_{high}(v)\})$. Then, notice that $\widetilde{W}(v)$ consists of all vertices w with $M(w) = x$ that are ancestors of $high_2(v)$. Let $w = lastM(x)$. If $w > high_2(v)$, then we have $\widetilde{W}(v) = \emptyset$, because this implies that every vertex w' with $M(w') = x$ has $w' > high_2(v)$, and therefore it cannot be an ancestor of $high_2(v)$. Now suppose that $w \leq high_2(v)$. We have that x is a common descendant of w and v, and therefore w and v are related as ancestor and descendant. Then $w \leq high_2(v)$ implies that $w < v$, and therefore w is an ancestor of v. Then, since v is a common descendant of w and $high_2(v)$, we have that w and $high_2(v)$ are related as ancestor and descendant. Thus, $w \leq high_2(v)$ implies that w is an ancestor of $high_2(v)$. Thus, we have $w \in \widetilde{W}(v)$, and since $\widetilde{lastW}(v) = min(\widetilde{W}(v))$, we have $\widetilde{lastW}(v) = w$. Thus, $\widetilde{lastW}(v)$ can be easily computed in constant time, for every vertex v.

If we have established that $\widetilde{lastW}(v) \neq \bot$, then, in order to compute $\widetilde{firstW}(v)$, we use Algorithm 22. Specifically, we generate a query of the form $q(M^{-1}(x), high_2(v))$. This is to return the greatest w with $M(w) = M(B(v) \setminus \{e_{high}(v)\})$ such that $w \leq high_2(v)$. Then we can show as previously that w is an ancestor of $high_2(v)$, and

therefore we have that w is the greatest ancestor of $high_2(v)$ such that $M(w) = M(B(v)\setminus \{e_{high}(v)\})$. Thus, we have $w = \widetilde{firstW}(v)$. Since the number of all those queries is $O(n)$, Lemma 5.27 implies that Algorithm 22 can compute all of them in $O(n)$ time in total. $\square$

Let $v \neq r$ be a vertex such that $high_1(v) \neq high_2(v)$ and $M(B(v)\setminus\{e_{high}(v)\}) \neq M(v)$. Let $\widetilde{S}$ be the segment of $\widetilde{H}(high_2(v))$ that contains v and is maximal w.r.t. the property that all its elements are related as ancestor and descendant (i.e., $\widetilde{S} = \widetilde{S}_2(v)$). Then, $U_4^1(v)$ is the collection of all vertices $u \in \widetilde{S}$ such that: (1) u is a proper descendant of v with $high(u) = high_2(v)$, (2) $low(u) \geq \widetilde{lastW}(v)$, and (3) either $low(u) < \widetilde{firstW}(v)$, or u is the lowest vertex in $\widetilde{S}$ that satisfies (1), (2) and $low(u) \geq \widetilde{firstW}(v)$.

Lemma 5.108. *Let v and v' be two vertices $\neq r$ such that v' is a proper descendant of v with $high_1(v) \neq high_2(v) = high_2(v') \neq high_1(v')$, $M(B(v) \setminus \{e_{high}(v)\}) \neq M(v)$ and $M(B(v')\setminus\{e_{high}(v')\}) \neq M(v')$. Let us assume that v and v' belong to the same segment $\widetilde{S}$ of $\widetilde{H}(high_2(v))$ that is maximal w.r.t. the property that its elements are related as ancestor and descendant. Let us further assume that $v' \notin V(v)$, $\widetilde{W}(v) \neq \emptyset$ and $\widetilde{W}(v') \neq \emptyset$. If $U_4^1(v') = \emptyset$, then $U_4^1(v) = \emptyset$. If $U_4^1(v) \neq \emptyset$, then the lowest vertex in $U_4^1(v)$ is greater than, or equal to, the greatest vertex in $U_4^1(v')$.*

Proof. First we will show that $M(B(v) \setminus \{e_{high}(v)\})$ is a proper ancestor of v', and $\widetilde{firstW}(v')$ is a proper ancestor of $\widetilde{lastW}(v)$.

Since $high_1(v) \neq high_2(v) = high_2(v') \neq high_1(v')$, $M(B(v) \setminus \{e_{high}(v)\}) \neq M(v)$, $M(B(v')\setminus\{e_{high}(v')\}) \neq M(v')$, and $v' \notin V(v)$, we have $M(B(v)\setminus\{e_{high}(v)\}) \neq M(B(v')\setminus \{e_{high}(v')\})$ (because this is the only condition that prevents v' from being in $V(v)$).

Since $high_1(v) \neq high_2(v)$, we have that $high_1(v) > high_2(v)$, and $e_{high}(v)$ is the unique back-edge in $B(v)$ whose lower endpoint is $high_1(v)$. Similarly, we have that $high_1(v') > high_2(v')$, and $e_{high}(v')$ is the unique back-edge in $B(v')$ whose lower endpoint is $high_1(v')$.

Now let (x, y) be a back-edge in $B(v')$ such that $y = high_2(v')$. Then, x is a descendant of v', and therefore a descendant of v. Furthermore, $y = high_2(v') = high_2(v)$, and therefore y is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Since $high_2(v) \neq high_1(v)$, we have that $(x, y) \neq e_{high}(v)$. This implies that x is a descendant of $M(B(v) \setminus \{e_{high}(v)\})$. Thus, we have that x is a common descendant of v' and $M(B(v)\setminus\{e_{high}(v)\})$. This shows that v' and $M(B(v)\setminus\{e_{high}(v)\})$ are related as ancestor and descendant.

Let us suppose, for the sake of contradiction, that $M(B(v) \setminus \{e_{high}(v)\})$ is not a proper ancestor of v'. Then we have that $M(B(v) \setminus \{e_{high}(v)\})$ is a descendant of v'. Let (x, y) be a back-edge in $B(v) \setminus \{e_{high}(v)\}$. Then x is a descendant of $M(B(v) \setminus \{e_{high}(v)\})$, and therefore a descendant of v'. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of v'. This shows that $(x, y) \in B(v')$. Due to the generality of $(x, y) \in B(v) \setminus \{e_{high}(v)\}$, this implies that $B(v) \setminus \{e_{high}(v)\} \subseteq B(v')$. Notice that, if $e_{high}(v') \in B(v)$, then, since $high_1(v') > high_2(v') = high_2(v)$, we have that $e_{high}(v') = e_{high}(v)$. Thus, whether $e_{high}(v') \in B(v)$ or $e_{high}(v') \notin B(v)$, we infer that $B(v) \setminus \{e_{high}(v)\} \subseteq B(v')$ can be strengthened to $B(v) \setminus \{e_{high}(v)\} \subseteq B(v') \setminus \{e_{high}(v')\}$. Conversely, let (x, y) be a back-edge in $B(v') \setminus \{e_{high}(v')\}$. Then x is a descendant of v', and therefore a descendant of v. Furthermore, y is an ancestor of $high_2(v') = high_2(v)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(v') \setminus \{e_{high}(v')\}$, this implies that $B(v') \setminus \{e_{high}(v')\} \subseteq B(v)$. As previously, we can argue that $B(v') \setminus \{e_{high}(v')\} \subseteq B(v)$ can be strengthened to $B(v') \setminus \{e_{high}(v')\} \subseteq B(v) \setminus \{e_{high}(v)\}$. Thus, we have shown that $B(v') \setminus \{e_{high}(v')\} = B(v) \setminus \{e_{high}(v)\}$, and therefore we have $M(B(v') \setminus \{e_{high}(v')\}) = M(B(v) \setminus \{e_{high}(v)\})$, a contradiction. This shows that $M(B(v) \setminus \{e_{high}(v)\})$ is a proper ancestor of v'.

Now let w be a vertex in $\widetilde{W}(v)$, and let w' be a vertex in $\widetilde{W}(v')$. Then we have that w is an ancestor of $high_2(v)$, and w' is an ancestor of $high_2(v')$. Thus, since $high_2(v) = high_2(v')$, we have that w and w' have a common descendant, and therefore they are related as ancestor and descendant. Let us suppose, for the sake of contradiction, that w' is not a proper ancestor of w. Then we have that w' is a descendant of w. Since $w \in \widetilde{W}(v)$, we have that $M(w) = M(B(v) \setminus \{e_{high}(v)\})$. And since $w' \in \widetilde{W}(v')$, we have that $M(w') = M(B(v') \setminus \{e_{high}(v')\})$. Since $M(B(v) \setminus \{e_{high}(v)\})$ is a proper ancestor of v', and v' is an ancestor of $M(B(v') \setminus \{e_{high}(v')\})$, we have that $M(w)$ is a proper ancestor of $M(w')$. Thus, there is a back-edge $(x, y) \in B(w)$ such that x is not a descendant of $M(w')$. Then, x is a descendant of $M(w)$, and therefore a descendant of $M(B(v) \setminus \{e_{high}(v)\})$, and therefore a descendant of v, and therefore a descendant of $high_2(v) = high_2(v')$, and therefore a descendant of w' (since $w' \in \widetilde{W}(v')$ implies that w' is an ancestor of $high_2(v')$). Furthermore, we have that y is a proper ancestor of w, and therefore a proper ancestor of w'. This shows that $(x, y) \in B(w')$. But this implies that x is a descendant of $M(w')$, a contradiction. This shows that w' is a proper ancestor of w. Due to the generality of $w' \in \widetilde{W}(v')$, this implies that $\widetilde{firstW}(w')$ is a proper ancestor of w. And due to the generality of $w \in \widetilde{W}(v)$, this implies that

$\widetilde{firstW}(v')$ is a proper ancestor of $\widetilde{lastW}(v)$.

Now let us suppose, for the sake of contradiction, that there is a vertex $u \in U_4^1(v)$, and $U_4^1(v') = \emptyset$. Since $u \in U_4^1(v)$, we have that $u \in \widetilde{S}$. Thus, since $v' \in \widetilde{S}$, we have that u and v' are related as ancestor and descendant. Let us suppose, for the sake of contradiction, that u is not a descendant of v'. Then, we have that u is a proper ancestor of v'. Let (x, y) be a back-edge in $B(v')$ such that $y = low(v')$. Then, x is a descendant of v', and therefore a descendant of u. Furthermore, y is an ancestor of $high_2(v') = high_2(v)$, and therefore a proper ancestor of v, and therefore a proper ancestor of u (since $u \in U_4^1(v)$ implies that u is a proper descendant of v). This shows that $(x, y) \in B(u)$.

We have that $M(\widetilde{firstW}(v')) = M(B(v') \setminus \{e_{high}(v')\})$ and $M(B(v') \setminus \{e_{high}(v')\})$ is a descendant of $M(v')$. Thus, $M(\widetilde{firstW}(v'))$ is a descendant of $M(v')$. Furthermore, since $\widetilde{firstW}(v')$ is an ancestor of $high_2(v')$, we have that $\widetilde{firstW}(v')$ is a proper ancestor of v'. Thus, Lemma 3.2 implies that $B(\widetilde{firstW}(v')) \subseteq B(v')$, and therefore $low(v') < \widetilde{firstW}(v')$. Thus, since $y = low(v')$, we have $y < \widetilde{firstW}(v')$, and therefore $y < \widetilde{lastW}(v)$ (since $\widetilde{firstW}(v') < \widetilde{lastW}(v)$). Since $(x, y) \in B(u)$, we have that $low(u) \leq y$, and therefore $low(u) < \widetilde{lastW}(v)$. But this contradicts the fact that $low(u) \geq \widetilde{lastW}(v)$ (which is an implication of $u \in U_4^1(v)$). This shows that u is a descendant of v'. Since $high(u) = high_2(v) = high_2(v') \neq high_1(v')$, we have that $u \neq v'$. Thus, u is a proper descendant of v'. Thus, we have the following facts: $u \in \widetilde{S}$, u is a proper descendant of v', $high(u) = high_2(v) = high_2(v')$, and $low(u) \geq \widetilde{lastW}(v) > \widetilde{firstW}(v')$. But this implies that $U_4^1(v') \neq \emptyset$ (because we can consider the lowest u that has those properties). A contradiction. This shows that if $U_4^1(v') = \emptyset$, then $U_4^1(v) = \emptyset$.

Now let us assume that $U_4^1(v) \neq \emptyset$. This implies that $U_4^1(v') \neq \emptyset$. Let us suppose, for the sake of contradiction, that there is a vertex $u \in U_4^1(v)$ that is lower than the greatest vertex u' in $U_4^1(v')$. Since $u \in U_4^1(v)$, we have that $u \in \widetilde{S}$. Since $u' \in U_4^1(v')$ we have that $u' \in \widetilde{S}$. This implies that u and u' are related as ancestor and descendant. Thus, since u is lower than u', we have that u is a proper ancestor of u'.

Let us suppose, for the sake of contradiction, that $low(u')$ is a proper ancestor of $\widetilde{firstW}(v')$. Then, since $\widetilde{firstW}(v')$ is a proper ancestor of $\widetilde{lastW}(v)$, we have that $low(u')$ is a proper ancestor of $\widetilde{lastW}(v)$. Now let (x, y) be a back-edge in $B(u')$ such that $y = low(u')$. Then x is a descendant of u', and therefore a descendant of u. Furthermore, y is a proper ancestor of $\widetilde{lastW}(v)$, and therefore a proper ancestor of $high_2(v)$, and therefore a proper ancestor of v, and therefore a proper ancestor of u.

This shows that $(x, y) \in B(u)$. Thus, we have $low(u) \leq y < \widetilde{lastW}(v)$, in contradiction to the fact that $u \in U_4^1(v)$. Thus, our last supposition is not true, and therefore we have that $low(u')$ is not a proper ancestor of $\widetilde{firstW}(v')$.

Since $u' \in U_4^1(v')$, we have that u' is a descendant of v', and therefore a descendant of $high_2(v')$, and therefore a descendant of $\widetilde{firstW}(v')$. Thus, u' is a common descendant of $low(u')$ and $\widetilde{firstW}(v')$, and therefore $low(u')$ and $\widetilde{firstW}(v')$ are related as ancestor and descendant. Thus, since $low(u')$ is not a proper ancestor of $\widetilde{firstW}(v')$, we have that $low(u')$ is a descendant of $\widetilde{firstW}(v')$, and therefore $low(u') \geq \widetilde{firstW}(v')$. Therefore, since $u' \in U_4^1(v')$, we have that u' is the lowest proper descendant of v' in $\widetilde{S}$ such that $high(u') = high_2(v')$ and $low(u') \geq \widetilde{firstW}(v')$ $(*)$.

Now we will trace the implications of $u \in U_4^1(v)$. First, we have that $u \in \widetilde{S}$. Then, we have $high(u) = high_2(v) = high_2(v')$. Furthermore, we have that $low(u) \geq \widetilde{lastW}(v)$, and therefore $low(u) > \widetilde{firstW}(v')$ (since $\widetilde{firstW}(v')$ is a proper ancestor of $\widetilde{lastW}(v)$). Finally, we can show as above that u is a proper descendant of v' (the proof of this fact above did not rely on $U_4^1(v') = \emptyset$). But then, since u is lower than u', we have a contradiction to $(*)$. Thus, we have shown that every vertex in $U_4^1(v)$ is at least as great as the greatest vertex in $U_4^1(v')$. In particular, this implies that the lowest vertex in $U_4^1(v)$ is greater than, or equal to, the greatest vertex in $U_4^1(v')$. □

Due to the similarity of the definitions of the U_2 and the U_4^1 sets, and the similarity between Lemmata 5.87 and 5.108, we can use a similar procedure as Algorithm 40 in order to compute all U_4^1 sets in linear time. This is shown in Algorithm 46. Our result is summarized in Lemma 5.109.

Lemma 5.109. *Let $\mathcal{V}$ be a collection of vertices such that:*

(1) For every $v \in \mathcal{V}$, we have $v \neq r$ and $M(B(v) \setminus \{e_{high}(v)\}) \neq M(v)$ and $high_1(v) \neq high_2(v)$.

(2) For every $v, v' \in \mathcal{V}$ with $v \neq v'$, we have $v' \notin V(v)$.

(3) For every $v \in \mathcal{V}$, we have $\widetilde{W}(v) \neq \emptyset$, and the vertices $\widetilde{firstW}(v)$ and $\widetilde{lastW}(v)$ are computed.

Then, Algorithm 46 correctly computes the sets $U_4^1(v)$, for all vertices $v \in \mathcal{V}$. Furthermore, on input $\mathcal{V}$, Algorithm 46 runs in linear time.

Algorithm 46: Compute all sets $U_4^1(v)$, for a collection of vertices $\mathcal{V}$ that satisfies the properties described in Lemma 5.109

1. **foreach** *vertex* x **do**
2.   compute the collection $\mathcal{S}(x)$ of the segments of $\widetilde{H}(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant
3. **end**
4. **foreach** *vertex* $v \in \mathcal{V}$ **do**
5.   set $U_4^1(v) \leftarrow \emptyset$
6. **end**
7. **foreach** *vertex* x **do**
8.   **foreach** *segment* $S \in \mathcal{S}(x)$ **do**
9.   let v be the first vertex in S
10.   **while** $v \neq \bot$ ***and*** *(*$v \notin \mathcal{V}$ ***or*** $high_2(v) \neq x$*)* **do**
11. $v \leftarrow next_S(v)$
12.   **end**
13.   **if** $v = \bot$ **then continue**
14.   let $u \leftarrow prev_S(v)$
15.   **while** $v \neq \bot$ **do**
16.   **while** $u \neq \bot$ ***and*** $low(u) < \widetilde{lastW}(v)$ **do**
17. $u \leftarrow prev_S(u)$
18.   **end**
19.   **while** $u \neq \bot$ ***and*** $low(u) < \widetilde{firstW}(v)$ **do**
20.   **if** $high(u) = x$ **then**
21.   insert u into $U_4^1(v)$
22.   **end**
23. $u \leftarrow prev_S(u)$
24.   **end**
25.   **while** $u \neq \bot$ ***and*** $high(u) \neq x$ **do**
26. $u \leftarrow prev_S(u)$
27.   **end**
28.   **if** $u \neq \bot$ **then**
29.   insert u into $U_4^1(v)$
30.   **end**
31. $v \leftarrow next_S(v)$
32.   **while** $v \neq \bot$ ***and*** *(*$v \notin \mathcal{V}$ ***or*** $high_2(v) \neq x$*)* **do**
33. $v \leftarrow next_S(v)$
34.   **end**
35.   **end**
36.   **end**
37. **end**

Proof. The proof is the same as that of Lemma 5.88 (which is given in the main text, in the two paragraphs right above Algorithm 40). Notice that the difference between Algorithm 40 and Algorithm 46 is that the second algorithm has a different set $\mathcal{V}$ of vertices for which the U_4^1 sets are defined, and the occurrences of "*firstW*" and "*lastW*" are replaced with "$\widetilde{firstW}$" and "$\widetilde{lastW}$", respectively. Now we can use the argument of Lemma 5.88, by just replacing the references to Lemma 5.87 with references to Lemma 5.108. □

Lemma 5.110. *Let (u, v, w) be a triple of vertices that induces a Type-3βii-4 4-cut, such that $M(B(v) \setminus \{e\}) \neq M(v)$ and $high_1(v) > high(u)$, where e is the back-edge in the 4-cut induced by (u, v, w). Then $u \in U_4^1(v)$. Furthermore, for every $v' \in V(v)$ we have that (u, v', w) is a triple of vertices that induces a Type-3βii-4 4-cut.*

Proof. Since $high_1(v) > high(u)$, by Lemma 5.101 we have that $high(u) = high_2(v)$. Now let u' be a vertex with $u \geq u' \geq v$ such that $u' \in \widetilde{H}(high_2(v))$. Since $high_1(v) \neq high(u)$, Lemma 5.102 implies that u' is an ancestor of u. Thus, we have that u and v belong to the same segment $\widetilde{S}$ of $\widetilde{H}(high_2(v))$ that is maximal w.r.t. the property that its elements are related as ancestor and descendant.

Since (u, v, w) induces a Type-3βii-4 4-cut, we have that w is a proper ancestor of v with $M(w) = M(B(v) \setminus \{e\})$. By Lemma 5.101 we have that $e = e_{high}(v)$ and $w \leq low(u)$. Since $high_1(v) > high(u)$ and $high(u) = high_2(v)$, we have that $high_1(v) \neq high_2(v)$. Since w is a proper ancestor of v with $w \leq low(u) \leq high(u) = high_2(v)$, we have that w is an ancestor of $high_2(v)$. This shows that $w \in \widetilde{W}(v)$, and therefore $w \leq \widetilde{firstW}(v)$. Since $w \leq low(u)$ and $\widetilde{lastW}(v) \leq w$, we have $\widetilde{lastW}(v) \leq low(u)$. Now, if $low(u) < \widetilde{firstW}(v)$, then u satisfies enough conditions to be in $U_4^1(v)$. Otherwise, let us assume that $low(u) \geq \widetilde{firstW}(v)$.

Let us suppose, for the sake of contradiction, that u is not the lowest vertex in $\widetilde{S}$ that is a proper descendant of v such that $high(u) = high_2(v)$ and $low(u) \geq \widetilde{firstW}(v)$. Then, there is a vertex $u' \in \widetilde{S}$ that is a proper descendant of v such that $u' < u$, $high(u') = high_2(v)$ and $low(u') \geq \widetilde{firstW}(v)$. Since both u and u' are in $\widetilde{S}$, we have that u and u' are related as ancestor and descendant. Thus, $u' < u$ implies that u' is a proper ancestor of u. Now let (x, y) be a back-edge in $B(u)$. Then, x is a descendant of u, and therefore a descendant of u'. Furthermore, since $high(u) = high_2(v)$, we have that y is a proper ancestor of v, and therefore a proper ancestor of u'. This shows that $(x, y) \in B(u')$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(u')$.

Conversely, let (x, y) be a back-edge in $B(u')$. Then x is a descendant of u', and therefore a descendant of v. Furthermore, y is an ancestor of $high(u') = high_2(v)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Since (u, v, w) induces a Type-3βii-4 4-cut, we have that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. This implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. Since $y \geq low(u') \geq \widetilde{firstW}(v) \geq w$, we have that (x, y) cannot be in $B(w)$. Since $high_1(v) \neq high_2(v)$, we have that $high_1(v) > high_2(v)$. Thus, since $e = e_{high}(v)$ and $high_1(v) > high_2(v) = high(u')$, we have that $e \notin B(u')$. Thus, $(x, y) \in B(u)$ is the only viable option. Due to the generality of $(x, y) \in B(u')$, this implies that $B(u') \subseteq B(u)$. Thus, we have $B(u') = B(u)$, in contradiction to the fact that the graph is 3-edge-connected. This shows that u is the lowest vertex in $\widetilde{S}$ that is a proper descendant of v such that $high(u) = high_2(v)$ and $low(u) \geq \widetilde{firstW}(v)$. Thus, u satisfies enough conditions to be in $U_4^1(v)$.

Since $high_1(v) \neq high_2(v)$ and $M(B(v) \setminus \{e_{high}(v)\}) \neq M(v)$, we have that $v \in V(v)$. Now let v' be a vertex in $V(v)$ such that $v' \neq v$. Then, by Lemma 5.105 we have $B(v) \sqcup \{e_{high}(v')\} = B(v') \sqcup \{e_{high}(v)\}$. This implies that $B(v) \setminus \{e_{high}(v)\} = B(v') \setminus \{e_{high}(v')\}$. Also, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e_{high}(v)\}$ implies that $B(v) \setminus \{e_{high}(v)\} = B(u) \sqcup B(w)$. Thus, we infer that $B(v') \setminus \{e_{high}(v')\} = B(u) \sqcup B(w)$, and therefore $B(v') = (B(u) \sqcup B(w)) \cup \{e_{high}(v')\}$.

Since $v' \in V(v)$, we have $high_1(v') \neq high_2(v') = high_2(v)$. This implies that $high_1(v') > high_2(v')$, and therefore $high_1(v') > high(u)$ (since $high_2(v') = high(u)$). Thus, we cannot have $e_{high}(v') \in B(u)$. Furthermore, since $w \leq low(u) \leq high(u) = high_2(v) = high_2(v') < high_1(v')$, we have that $e_{high}(v') \notin B(w)$. Thus, $B(v') = (B(u) \sqcup B(w)) \cup \{e_{high}(v')\}$ can be strengthened to $B(v') = (B(u) \sqcup B(w)) \sqcup \{e_{high}(v')\}$. Finally, since $v' \in V(v)$, we have that $M(B(v') \setminus \{e_{high}(v')\}) = M(B(v) \setminus \{e_{high}(v)\}) = M(w)$. Thus, we conclude that (u, v', w) induces a Type-3βii-4 4-cut. □

Lemma 5.111. *Let (u, v, w) be a triple of vertices such that $high_1(v) \neq high_2(v)$, $M(B(v) \setminus \{e_{high}(v)\}) \neq M(v)$ and $u \in U_4^1(v)$. Then, (u, v, w) induces a Type-3βii-4 4-cut if and only if: $bcount(v) = bcount(u) + bcount(w) + 1$, and w is the greatest proper ancestor of v with $M(w) = M(B(v) \setminus \{e_{high}(v)\})$ such that $w \leq low(u)$.*

Proof. ($\Rightarrow$) Since (u, v, w) induces a Type-3βii-4 4-cut, we have $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, where e is the back-edge in the 4-cut induced by (u, v, w). Thus, we get $bcount(v) = bcount(u) + bcount(w) + 1$. Since $high_1(v) \neq high_2(v)$ we have $high_1(v) > high_2(v)$. Since $u \in U_4^1(v)$ we have $high(u) = high_2(v)$. This implies that $high_1(v) > high(u)$. Thus,

by Lemma 5.101 we have $e = e_{high}(v)$. Since (u, v, w) induces a Type-3βii-4 4-cut, we have $M(w) = M(B(v) \setminus \{e\})$. Thus, $M(w) = M(B(v) \setminus \{e_{high}(v)\})$. Furthermore, Lemma 5.101 implies that $w \leq low(u)$.

Now let us suppose, for the sake of contradiction, that there is an ancestor w' of v with $w' > w$, such that $M(w') = M(B(v) \setminus \{e_{high}(v)\})$ and $w' \leq low(u)$. Then, we have that v is a common descendant of w and w', and therefore w and w' are related as ancestor and descendant. Thus, $w' > w$ implies that w' is a proper descendant of w. Then, since $M(w') = M(w)$, Lemma 3.2 implies that $B(w) \subseteq B(w')$. Since the graph is 3-edge-connected, this can be strengthened to $B(w) \subset B(w')$. Thus, there is a back-edge $(x, y) \in B(w') \setminus B(w)$. Then, we have that x is a descendant of $M(w')$, and therefore a descendant of $M(B(v) \setminus \{e_{high}(v)\})$, and therefore a descendant of $M(v)$. Furthermore, we have that y is a proper ancestor of w', and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Then, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. The case $(x, y) \in B(u)$ is rejected, since $y < w' \leq low(u)$. Furthermore, the case $(x, y) = e$ is rejected, since $e = e_{high}(v)$, and $high_1(v) > high(u) \geq low(u)$ (but $y < w' \leq low(u)$). Thus, we have $(x, y) \in B(w)$, which is impossible, since $(x, y) \in B(w') \setminus B(w)$. This shows that w is the greatest proper ancestor of v with $M(w) = M(B(v) \setminus \{e_{high}(v)\})$ such that $w \leq low(u)$.

($\Leftarrow$) We have to show that there is a back-edge e such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, and $M(w) = M(B(v) \setminus \{e\})$.

Since $w \leq low(u)$, we have $B(u) \cap B(w) = \emptyset$ (because, if there existed a back-edge in $B(u) \cap B(w)$, its lower endpoint would be lower than w, and therefore lower than $low(u)$, which is impossible). Let (x, y) be a back-edge in $B(u)$. Since $u \in U_4^1(v)$, we have that u is a proper descendant of v with $high(u) = high_2(v)$. Thus, $(x, y) \in B(u)$ implies that x is a descendant of v. Furthermore, since $y \in B(u)$, we have that y is an ancestor of $high(u) = high_2(v)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(v)$.

Let (x, y) be a back-edge in $B(w)$. Then, x is a descendant of $M(w) = M(B(v) \setminus \{e_{high}(v)\})$, and therefore a descendant of v. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(w)$, this implies that $B(w) \subseteq B(v)$.

Since $high_1(v) \neq high_2(v)$, we have $high_1(v) > high_2(v)$. Then, since $u \in U_4^1(v)$, we have $high(u) = high_2(v)$, and therefore $high_1(v) > high(u)$, and therefore we cannot have $e_{high}(v) \in B(u)$ (because the lower endpoint of $e_{high}(v)$ is greater than $high(u)$).

Furthermore, since $w \leq low(u) \leq high(u) = high_2(v) < high_1(v)$, we have $e_{high}(v) \notin B(w)$ (because the lower endpoint of $e_{high}(v)$ is greater than w).

Thus, we have $B(u) \subseteq B(v)$, $B(w) \subseteq B(v)$, and the sets $B(u)$, $B(w)$, and $\{e_{high}(v)\}$ are mutually disjoint. Thus, $bcount(v) = bcount(u) + bcount(w) + 1$ implies that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e_{high}(v)\}$. By assumption, we have $M(w) = M(B(v) \setminus \{e_{high}(v)\})$. This shows that (u, v, w) induces a Type-3βii-4 4-cut. □

Algorithm 47: Compute a collection of Type-3βii-4 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where w is an ancestor of v, v is an ancestor of u, $M(B(v)\setminus\{e\}) \neq M(v)$ and $high_1(v) \neq high(u)$, such that all 4-cuts of this form are implied from this collection plus that returned by Algorithm 24

```
1  select a representative vertex for every non-empty set in
     {V(v) | v is a vertex ≠ r}; call this vertex a "marked" vertex
   // If V(v) ≠ ∅, for a vertex v ≠ r, then the representative vertex of
        V(v) is a vertex v' ∈ V(v), and so it has M(B(v') \ {e_high(v')}) ≠ M(v')
        and high_2(v') ≠ high_1(v')
2  foreach marked vertex v do
3      if W̃(v) ≠ ∅ then
4          compute the set U_4^1(v)
5      end
6  end
7  foreach marked vertex v do
8      foreach u ∈ U_4^1(v) do
9          let w be the greatest proper ancestor of v such that
             M(w) = M(B(v) \ {e_high(v)}) and w ≤ low(u)
10         if bcount(v) = bcount(u) + bcount(w) + 1 then
11             mark {(u,p(u)), (v,p(v)), (w,p(w)), e_high(v)} as a Type-3βii-4 4-cut
12         end
13     end
14 end
```

Proposition 5.29. *Algorithm 47 computes a collection $\mathcal{C}$ of Type-3βii-4 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where $M(B(v) \setminus \{e\}) \neq M(v)$ and $high_1(v) \neq high(u)$.*

Let $\mathcal{C}'$ be the collection of Type-2ii 4-cuts returned by Algorithm 24. Then, every Type-3βii-4 4-cut of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where $M(B(v)\setminus\{e\})\neq M(v)$ and $high_1(v)\neq high(u)$ is implied by $\mathcal{C}\cup\mathcal{C}'$. Finally, Algorithm 47 has a linear-time implementation.

Proof. Let $C=\{(u,p(u)),(v,p(v)),(w,p(w)),e_{high}(v)\}$ be a 4-element set that is marked in Line 11. Then we have that v is a marked vertex, and therefore it has $M(B(v)\setminus\{e_{high}(v)\})\neq M(v)$ and $high_2(v)\neq high_1(v)$. We also have that $u\in U_4^1(v)$, w is the greatest proper ancestor of v such that $M(w)=M(B(v)\setminus\{e_{high}(v)\})$ and $w\leq low(u)$, and $bcount(v)=bcount(u)+bcount(w)+1$. Thus, all the conditions of Lemma 5.111 are satisfied, and therefore we have that (u,v,w) induces a Type-3βii-4 4-cut. Let e be the back-edge in the 4-cut induced by (u,v,w). Since $u\in U_4^1(v)$ we have $high(u)=high_2(v)$. Therefore, $high_2(v)\neq high_1(v)$ implies that $high(u)\neq high_1(v)$. Thus, Lemma 5.101 implies that $e=e_{high}(v)$. Therefore, we have that C is the Type-3βii-4 4-cut induced by (u,v,w). So let $\mathcal{C}$ be the collection of all Type-3βii-4 4-cuts marked in Line 11.

Let $C=\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ be a Type-3βii-4 4-cut, where w is a proper ancestor of v, v is a proper ancestor of u, $M(B(v)\setminus\{e\})\neq M(v)$ and $high_1(v)\neq high(u)$. Since $high_1(v)\neq high(u)$, Lemma 5.101 implies that $high_1(v)>high(u)$ and $e=e_{high}(v)$. Thus, Lemma 5.110 implies that $u\in U_4^1(v)$. Since $u\in U_4^1(v)$ we have $high(u)=high_2(v)$. Therefore, $high_1(v)>high(u)$ implies that $high_1(v)\neq high_2(v)$. Thus, Lemma 5.111 implies that $bcount(v)=bcount(u)+bcount(w)+1$, and w is the greatest proper ancestor of v such that $M(w)=M(B(v)\setminus\{e_{high}(v)\})$ and $w\leq low(u)$. Thus, if v is one of the marked vertices, then C satisfies enough conditions to be marked in Line 11, and therefore $C\in\mathcal{C}$. So let us assume that v is not one of the marked vertices.

Since $high_1(v)\neq high_2(v)$ and $M(B(v)\setminus\{e_{high}(v)\})\neq M(v)$, we have that $V(v)\neq\emptyset$. Let v' be the marked vertex that was picked as a representative of $V(v)$ in Line 1. Then, Lemma 5.110 implies that (u,v',w) induces a Type-3βii-4 4-cut C'. Then, since $v'\in V(v)$ we have $high_1(v')\neq high_2(v')=high_2(v)=high(u)$ and $M(B(v')\setminus\{e_{high}(v')\})\neq M(v')$. Then, since $high_1(v')\neq high(u)$, Lemma 5.101 implies that the back-edge in the 4-cut induced by (u,v',w) is $e_{high}(v')$, and $high_1(v')>high(u)$. Then, Lemma 5.110 implies that $u\in U_4^1(v')$, and Lemma 5.111 implies that $bcount(v')=bcount(u)+bcount(w)+1$, and w is the greatest proper ancestor of v' such that $M(w)=M(B(v')\setminus\{e_{high}(v')\})$ and $w\leq low(u)$. Thus, C' satisfies enough conditions to be marked in Line 11, and therefore $C'\in\mathcal{C}$.

Since $v'\in V(v)$ and $v'\neq v$, Lemma 5.105 implies that $B(v)\sqcup\{e_{high}(v')\}=B(v')\sqcup$

$\{e_{high}(v)\}$. Then, Lemma 5.28 implies that $C'' = \{(v, p(v)), (v', p(v')), e_{high}(v), e_{high}(v')\}$ is a Type-$2ii$ 4-cut. Since $C = \{(u, p(u)), (v, p(v)), (w, p(w)), e_{high}(v)\}$ and $C' = \{(u, p(u)), (v', p(v')), (w, p(w)), e_{high}(v')\}$, notice that C is implied by C' and C'' through the pair of edges $\{(v, p(v)), e_{high}(v)\}$. Let $\mathcal{C}'$ be the collection of Type-$2ii$ 4-cuts computed by Algorithm 24. By Proposition 5.12 we have that C'' is implied by $\mathcal{C}'$ through the pair of edges $\{(v, p(v)), e_{high}(v)\}$. Thus, by Lemma 5.7 we have that C is implied by $\mathcal{C}' \cup \{C'\}$ through the pair of edges $\{(v, p(v)), e_{high}(v)\}$. Therefore, C is implied by $\mathcal{C}' \cup \mathcal{C}$.

Now we will argue about the complexity of Algorithm 47. By Proposition 3.6, the values $M(B(v) \setminus \{e_{high}(v)\})$ can be computed in linear time in total, for all vertices $v \neq r$. Then, for every vertex $v \neq r$ such that $high_1(v) \neq high_2(v)$ and $M(B(v) \setminus \{e_{high}(v)\}) \neq M(v)$, we generate a triple $(v, high_2(v), M(B(v) \setminus \{e_{high}(v)\}))$. Let L be the collection of all those triples. Then we sort L lexicographically w.r.t. the second and the third component of its elements. We note that this sorting can be performed in $O(n)$ time with bucket-sort. Then, every V set corresponds to a segment of L that is maximal w.r.t. the property that its elements coincide in their second and their third components. Then, we just pick a triple from every such segment, we extract its first component v, and we mark it, in order to get a marked representative of the corresponding V set. Thus, the collection of the marked vertices in Line 1 can be constructed in linear time.

By Lemma 5.107 we have that the vertices $\widetilde{firstW}(v)$ and $\widetilde{lastW}(v)$ can be computed in linear time in total, for all marked vertices v. Then, by Lemma 5.109 we have that the sets $U_4^1(v)$ can be computed in linear time in total, for all marked vertices v. Thus, the **for** loop in Line 2 can be performed in linear time.

In order to compute the vertex w in Line 9 we use Algorithm 22. Specifically, whenever we reach Line 9, we generate a query of the form $q(M^{-1}(M(B(v) \setminus \{e_{high}(v)\})), min\{p(v), low(u)\})$. This is to return the greatest vertex w with $M(w) = M(B(v) \setminus \{e_{high}(v)\})$ such that $w \leq min\{p(v), low(u)\}$. Since $M(B(v) \setminus \{e_{high}(v)\}) = M(w)$ is a common descendant of w and v, we have that w and v are related as ancestor and descendant. Then, $w \leq min\{p(v), low(u)\}$ implies that $w \leq p(v)$, and therefore w is a proper ancestor of v. Thus, w is the greatest proper ancestor of v with $M(w) = M(B(v) \setminus \{e_{high}(v)\})$ such that $w \leq low(u)$. Now, since the number of all those queries is $O(n)$, Lemma 5.27 implies that Algorithm 22 can answer all of them in $O(n)$ time in total. We conclude that Algorithm 47 runs in linear time. $\square$

The case where $M(B(v)\setminus\{e\})\neq M(v)$ and $high_1(v)=high(u)$

In this case, by Lemma 3.9 we have that either $e=e_L(v)$ or $e=e_R(v)$. In this subsection, we will focus on the case $e=e_L(v)$. Thus, whenever we consider a triple of vertices (u,v,w) that induces a Type-3βii-4 4-cut, such that $M(B(v)\setminus\{e\})\neq M(v)$ and $high_1(v)=high(u)$, we assume that $e=e_L(v)$, where e is the back-edge in the 4-cut induced by (u,v,w). We will show how to compute all 4-cuts of this type in linear time. The algorithms and the arguments for the case $e=e_R(v)$ are similar.

For every vertex $v\neq r$, we let $W_L(v)$ denote the collection of all proper ancestors w of v such that $M(w)=M(B(v)\setminus\{e_L(v)\})$. We also let $firstW_L(v):=max(W_L(v))$ and $lastW_L(v):=min(W_L(v))$. (If $W_L(v)=\emptyset$, then we let $firstW_L(v):=\bot$ and $lastW_L(v):=\bot$.)

Lemma 5.112. *For all vertices $v\neq r$, the values $firstW_L(v)$ and $lastW_L(v)$ can be computed in total linear time.*

Proof. First, we need to have the values $M(B(v)\setminus\{e_L(v)\})$ computed, for all vertices $v\neq r$. According to Proposition 3.6, this can be achieved in linear time in total. Now, in order to compute $lastW_L(v)$, we just need to know whether $w=lastM(M(B(v)\setminus\{e_L(v)\}))$ is a proper ancestor of v. If that is the case, then we set $lastW_L(v)\leftarrow w$. Otherwise, $lastW_L(v)$ is left to be $\bot$. Then, for every vertex $v\neq r$, we have that $firstW_L(v)$ is the greatest proper ancestor w of v that has $M(w)=M(B(v)\setminus\{e_L(v)\})$. Thus, we can compute all $firstW_L$ values using Algorithm 22. Specifically, let $v\neq r$ be a vertex, and let $x=M(B(v)\setminus\{e_L(v)\})$. Then we generate a query of the form $q(M^{-1}(x),p(v))$. This query returns the greatest vertex w in $M^{-1}(x)$ that has $w\leq p(v)$. Since $w\in M^{-1}(x)$, we have that $M(w)=M(B(v)\setminus\{e_L(v)\})$. Thus, $M(w)$ is an ancestor of $M(v)$, and therefore w is an ancestor of $M(v)$. Therefore, $M(v)$ is a common descendant of w and v, and so w and v are related as ancestor and descendant. Then, $w\leq p(v)$ implies that w is a proper ancestor of v. This shows that w is the greatest proper ancestor of v such that $w\in M^{-1}(x)$. In other words, $w=firstW_L(v)$. By Lemma 5.27 we have that all these queries can be answered in $O(n)$ time in total. $\square$

Now let $v\neq r$ be a vertex such that $M(B(v)\setminus\{e_L(v)\})\neq M(v)$, and let S be the segment of $H(high_1(v))$ that contains v and is maximal w.r.t. the property that all its elements are related as ancestor and descendant (i.e., $S=S(v)$). Then, we let $U_4^2(v)$

denote the collection of all vertices $u \in S$ such that: (1) u is a proper descendant of v, (2) $low(u) \geq lastW_L(v)$, and (3) either $low(u) < firstW_L(v)$, or u is the lowest vertex in S that satisfies (1), (2) and $low(u) \geq firstW_L(v)$.

Lemma 5.113. *Let v and v' be two vertices such that v' is a proper descendant of v with $high_1(v) = high_1(v')$, $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$ and $M(B(v') \setminus \{e_L(v')\}) \neq M(v')$. Suppose that $W_L(v) \neq \emptyset$, $W_L(v') \neq \emptyset$, and both v and v' belong to the same segment S of $H(high_1(v))$ that is maximal w.r.t. the property that its elements are related as ancestor and descendant. If $U_4^2(v') = \emptyset$, then $U_4^2(v) = \emptyset$. If $U_4^2(v) \neq \emptyset$, then the lowest vertex in $U_4^2(v)$ is greater than, or equal to, the greatest vertex in $U_4^2(v')$.*

Proof. First we will show that $M(B(v) \setminus \{e_L(v)\})$ is a proper ancestor of $M(B(v') \setminus \{e_L(v')\})$, and $firstW_L(v')$ is a proper ancestor of $lastW_L(v)$.

Since v' is a proper descendant of v such that $high_1(v') = high_1(v)$, Lemma 3.3 implies that $B(v') \subseteq B(v)$. Since the graph is 3-edge-connected, we have that $|B(v')| > 1$. Thus, there is a back-edge $(x, y) \in B(v') \setminus \{e_L(v)\}$. Then, since $B(v') \subseteq B(v)$, we have that $(x, y) \in B(v)$. Since $(x, y) \neq e_L(v)$, this can be strengthened to $(x, y) \in B(v) \setminus \{e_L(v)\}$. This implies that x is a descendant of $M(B(v) \setminus \{e_L(v)\})$. Thus, we have that x is a common descendant of v' and $M(B(v) \setminus \{e_L(v)\})$, and therefore v' and $M(B(v) \setminus \{e_L(v)\})$ are related as ancestor and descendant.

If $M(B(v) \setminus \{e_L(v)\})$ is a proper ancestor of v', then we have that $M(B(v) \setminus \{e_L(v)\})$ is a proper ancestor of $M(B(v') \setminus \{e_L(v')\})$, since $M(B(v') \setminus \{e_L(v')\})$ is a descendant of $M(v')$, and therefore a descendant of v'. So let us assume that $M(B(v) \setminus \{e_L(v)\})$ is a descendant of v'. Let (x, y) be a back-edge in $B(v) \setminus \{e_L(v)\}$. Then, we have that x is a descendant of $M(B(v) \setminus \{e_L(v)\})$, and therefore a descendant of v'. Furthermore, we have that y is a proper ancestor of v, and therefore a proper ancestor of v'. This shows that $(x, y) \in B(v')$. Due to the generality of $(x, y) \in B(v) \setminus \{e_L(v)\}$, this shows that $B(v) \setminus \{e_L(v)\}) \subseteq B(v')$. Then, since $B(v) \setminus \{e_L(v)\} \subseteq B(v') \subseteq B(v)$ and $B(v) \neq B(v')$ (since the graph is 3-edge-connected), we have that $B(v) \setminus \{e_L(v)\} = B(v')$. This implies that $M(B(v) \setminus \{e_L(v)\}) = M(v')$. Since $M(B(v') \setminus \{e_L(v')\}) \neq M(v')$, we have that $M(B(v') \setminus \{e_L(v')\})$ is a proper descendant of $M(v')$. Therefore, since $M(B(v) \setminus \{e_L(v)\}) = M(v')$, we have that $M(B(v') \setminus \{e_L(v')\})$ is a proper descendant of $M(B(v) \setminus \{e_L(v)\})$.

Now let w be a vertex in $W_L(v)$, and let w' be a vertex in $W_L(v')$. Then we have that $M(w) = M(B(v) \setminus \{e_L(v)\})$ and $M(w') = M(B(v') \setminus \{e_L(v')\})$. Thus, we have that

$M(w')$ is a proper descendant of $M(w)$, and therefore a proper descendant of w. Thus, since $M(w')$ is a common descendant of w' and w, we have that w' and w are related as ancestor and descendant. Let us suppose, for the sake of contradiction, that w' is not a proper ancestor of w. Then, w' is a descendant of w. Since $M(w')$ is a proper descendant of $M(w)$, there is a back-edge (x, y) in $B(w)$ such that x is not a descendant of $M(w')$. Then, we have that x is a descendant of $M(w) = M(B(v) \setminus \{e_L(v)\})$, and therefore a descendant of v, and therefore a descendant of $\mathit{high}_1(v) = \mathit{high}_1(v')$, and therefore a descendant of v', and therefore a descendant of w'. Furthermore, we have that y is a proper ancestor of w, and therefore a proper ancestor of w'. This shows that $(x, y) \in B(w')$, and therefore x is a descendant of $M(w')$, a contradiction. Thus, we have that w' is a proper ancestor of w. Due to the generality of $w' \in W_L(v')$, this implies that $\mathit{firstW}_L(v')$ is a proper ancestor of w. And due to the generality of $w \in W_L(v)$, this implies that $\mathit{firstW}_L(v')$ is a proper ancestor of $\mathit{lastW}_L(v)$.

Now let us suppose, for the sake of contradiction, that there is a vertex $u \in U_4^2(v)$, and $U_4^2(v') = \emptyset$. Since $u \in U_4^2(v)$, we have that $u \in S$. Thus, since $v' \in S$, we have that u and v' are related as ancestor and descendant. Let us suppose, for the sake of contradiction, that u is not a proper descendant of v'. Then, we have that u is an ancestor of v'. Since $W_L(v') \neq \emptyset$, we let $w' = \mathit{firstW}_L(v')$. Let (x, y) be a back-edge in $B(w')$ such that $y = \mathit{low}(w')$. Then, x is a descendant $M(w') = M(B(v') \setminus \{e_L(v')\})$, and therefore a descendant of $M(v')$, and therefore a descendant of v', and therefore a descendant of u. Furthermore, y is a proper ancestor of w', and therefore a proper ancestor of v'. Thus, since x is a descendant of v', this shows that $(x, y) \in B(v')$. Therefore, y is an ancestor of $\mathit{high}_1(v') = \mathit{high}_1(v)$, and therefore a proper ancestor of v, and therefore a proper ancestor of u (since $u \in U_4^2(v)$ implies that u is a proper descendant of v). Thus, since x is a descendant of u, this shows that $(x, y) \in B(u)$. Since $y = \mathit{low}(w')$ and $w' = \mathit{firstW}_L(v')$, we have that $y < \mathit{firstW}_L(v')$, and therefore $y < \mathit{lastW}_L(v)$ (since $\mathit{firstW}_L(v')$ is a proper ancestor of $\mathit{lastW}_L(v)$). Since $(x, y) \in B(u)$, we have that $\mathit{low}(u) \leq y$, and therefore $\mathit{low}(u) < \mathit{lastW}_L(v)$. But this contradicts the fact that $\mathit{low}(u) \geq \mathit{lastW}_L(v)$ (which is an implication of $u \in U_4^2(v)$). Thus, our last supposition is not true, and therefore u is a proper descendant of v'. Thus, we have the following facts: $u \in S$, u is a proper descendant of v', and $\mathit{low}(u) \geq \mathit{lastW}_L(v) > \mathit{firstW}_L(v')$. But this implies that $U_4^2(v') \neq \emptyset$ (because we can consider the lowest u that has those properties). A contradiction. This shows that if $U_4^2(v') = \emptyset$, then $U_4^2(v) = \emptyset$.

Now let us assume that $U_4^2(v) \neq \emptyset$. This implies that $U_4^2(v') \neq \emptyset$. Let us suppose, for

the sake of contradiction, that there is a vertex $u \in U_4^2(v)$ that is lower than the greatest vertex u' in $U_4^2(v')$. Since $u \in U_4^2(v)$, we have that $u \in S$. Since $u' \in U_4^2(v')$ we have that $u' \in S$. This implies that u and u' are related as ancestor and descendant. Thus, since u is lower than u', we have that u is a proper ancestor of u'. Let us suppose, for the sake of contradiction, that $low(u')$ is a proper ancestor of $firstW_L(v')$. Then, since $firstW_L(v')$ is a proper ancestor of $lastW_L(v)$, we have that $low(u')$ is a proper ancestor of $lastW_L(v)$. Now let (x, y) be a back-edge in $B(u')$ such that $y = low(u')$. Then x is a descendant of u', and therefore a descendant of u. Furthermore, y is a proper ancestor of $lastW_L(v)$, and therefore a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Thus, we have $low(u) \leq y < lastW_L(v)$, in contradiction to the fact that $u \in U_4^2(v)$. Thus, our last supposition is not true, and therefore we have that $low(u')$ is not a proper ancestor of $firstW_L(v')$.

Since $u' \in U_4^2(v')$, we have that u' is a proper descendant of v', and therefore a proper descendant of $firstW_L(v')$. Thus, u' is a common descendant of $low(u')$ and $firstW_L(v')$, and therefore $low(u')$ and $firstW_L(v')$ are related as ancestor and descendant. Thus, since $low(u')$ is not a proper ancestor of $firstW_L(v')$, we have that $low(u')$ is a descendant of $firstW_L(v')$, and therefore $low(u') \geq firstW_L(v')$. Thus, since $u' \in U_4^2(v')$, we have that u' is the lowest vertex in S that is a proper descendant of v' such that $low(u') \geq firstW_L(v')$ $(*)$.

Now we will trace the implications of $u \in U_4^2(v)$. First, we have that $u \in S$. Furthermore, we have that $low(u) \geq lastW_L(v)$, and therefore $low(u) > firstW_L(v')$ (since $firstW_L(v')$ is a proper ancestor of $lastW_L(v)$). Finally, we can show as above that u is a proper descendant of v' (the proof of this fact above did not rely on $U_4^2(v') = \emptyset$). But then, since u is lower than u', we have a contradiction to $(*)$. Thus, we have shown that every vertex in $U_4^2(v)$ is at least as great as the greatest vertex in $U_4^2(v')$. In particular, this implies that the lowest vertex in $U_4^2(v)$ is greater than, or equal to, the greatest vertex in $U_4^2(v')$. □

Due to the similarity of the definitions of the U_2 and the U_4^2 sets, and the similarity between Lemmata 5.87 and 5.113, we can use a similar procedure as Algorithm 40 in order to compute all U_4^2 sets in linear time. This is shown in Algorithm 48. Our result is summarized in Lemma 5.114.

Lemma 5.114. *Let $\mathcal{V}$ be the collection of all vertices $v \neq r$ such that $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$ and $W_L(v) \neq \emptyset$, and suppose that the vertices $firstW_L(v)$ and $lastW_L(v)$ are computed*

Algorithm 48: Compute the sets $U_4^2(v)$, for all vertices $v \neq r$ such that $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$ and $W_L(v) \neq \emptyset$

1. let $\mathcal{V}$ be the collection of all vertices $v \neq r$ such that $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$ and $W_L(v) \neq \emptyset$
2. **foreach** *vertex* x **do**
3.   compute the collection $\mathcal{S}(x)$ of the segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant
4. **end**
5. **foreach** $v \in \mathcal{V}$ **do**
6.   set $U_4^2(v) \leftarrow \emptyset$
7. **end**
8. **foreach** *vertex* x **do**
9.   **foreach** *segment* $S \in \mathcal{S}(x)$ **do**
10.   let v be the first vertex in S
11.   **while** $v \neq \bot$ ***and*** $v \notin \mathcal{V}$ **do**
12. $v \leftarrow next_S(v)$
13.   **end**
14.   **if** $v = \bot$ **then continue**
15.   let $u = prev_S(v)$
16.   **while** $v \neq \bot$ **do**
17.   **while** $u \neq \bot$ ***and*** $low(u) < lastW_L(v)$ **do**
18. $u \leftarrow prev_S(u)$
19.   **end**
20.   **while** $u \neq \bot$ ***and*** $low(u) < firstW_L(v)$ **do**
21.   insert u into $U_4^2(v)$
22. $u \leftarrow prev_S(u)$
23.   **end**
24.   **if** $u \neq \bot$ **then**
25.   insert u into $U_4^2(v)$
26.   **end**
27. $v \leftarrow next_S(v)$
28.   **while** $v \neq \bot$ ***and*** $v \notin \mathcal{V}$ **do**
29. $v \leftarrow next_S(v)$
30.   **end**
31.   **end**
32.   **end**
33. **end**

for every $v \in \mathcal{V}$. Then, Algorithm 48 correctly computes the sets $U_4^2(v)$, for all vertices $v \in \mathcal{V}$, in total linear time.

Proof. The proof is the same as that of Lemma 5.88 (which is given in the main text, in the two paragraphs right above Algorithm 40). Notice that the difference between Algorithm 40 and Algorithm 48 is that the second algorithm has a different set $\mathcal{V}$ of vertices for which the U_4^2 sets are defined, and the occurrences of "*firstW*" and "*lastW*" are replaced with "firstW_L" and "lastW_L", respectively. Now we can use the argument of Lemma 5.88, by just replacing the references to Lemma 5.87 with references to Lemma 5.113. □

Lemma 5.115. *Let $v \neq r$ be a vertex such that $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$, and let u be a proper descendant of v with $\mathit{high}(u) = \mathit{high}(v)$. Then, $e_L(v) \notin B(u)$.*

Proof. Since u is a proper descendant of v with $\mathit{high}(u) = \mathit{high}(v)$, Lemma 3.3 implies that $B(u) \subseteq B(v)$. This implies that $M(u)$ is a descendant of $M(v)$. Thus, since $M(u)$ is a common descendant of u and $M(v)$, we have that u and $M(v)$ are related as ancestor and descendant.

Let us suppose, for the sake of contradiction, that u is not a proper descendant of $M(v)$. Thus, we have that u is an ancestor of $M(v)$. Let (x, y) be a back-edge in $B(v)$. Then, x is a descendant of $M(v)$, and therefore a descendant of u. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Due to the generality of $(x, y) \in B(v)$, this implies that $B(v) \subseteq B(u)$. Thus, $B(u) \subseteq B(v)$ implies that $B(u) = B(v)$, in contradiction to the fact that the graph is 3-edge-connected. This shows that u is a proper descendant of $M(v)$.

Now let us suppose, for the sake of contradiction, that u is an ancestor of $L_1(v)$ (i.e., the higher endpoint of $e_L(v)$). Then, since u is a proper descendant of $M(v)$, we cannot have $M(v) = L_1(v)$. Thus, $L_1(v)$ is a proper descendant of $M(v)$. So let c be the child of $M(v)$ that is an ancestor of $L_1(v)$. Then, since $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$, we have that $e_L(v)$ is the unique back-edge in $B(v)$ whose higher endpoint is a descendant of c. Now, since u is an ancestor of $L_1(v)$ and a proper descendant of $M(v)$, we have that u is also a descendant of c. Since the graph is 3-edge-connected, we have that $|B(u)| > 1$. Thus, there is a back-edge $(x, y) \in B(u) \setminus \{e_L(v)\}$. Then, x is a descendant of u, and therefore a descendant of v. Furthermore, y is an ancestor of $\mathit{high}(u) = \mathit{high}(v)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Since $(x, y) \neq e_L(v)$, this can be strengthened to $(x, y) \in B(v) \setminus \{e_L(v)\}$. Thus, we have

that in $B(v) \setminus \{e_L(v)\}$ there is still a back-edge whose higher endpoint is a descendant of c (i.e., (x, y)), which is impossible. We conclude that u is not an ancestor of $L_1(v)$, and therefore $e_L(v) \notin B(u)$. □

Lemma 5.116. *Let (u, v, w) be a triple of vertices that induces a Type-3βii-4 4-cut, with back-edge $e_L(v)$, such that $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$ and $high_1(v) = high(u)$. Then $u \in U_4^2(v)$.*

Proof. Since (u, v, w) induces a 4-cut, we have that u is a proper descendant of v. Let u' be a vertex such that $u \geq u' \geq v$ and $high(u') = high(u)$. Since $high_1(v) = high(u)$, this implies that $high(u') = high_1(v)$, and therefore $u' \in H(high_1(v))$. Then, since (u, v, w) induces a Type-3βii-4 4-cut with back-edge $e_L(v)$ such that $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$, by Lemma 5.103 we have that u' is an ancestor of u. This shows that u and v belong to a segment S of $H(high_1(v))$ that is maximal w.r.t. the property that its elements are related as ancestor and descendant.

Since (u, v, w) induces a Type-3βii-4 4-cut with back-edge $e_L(v)$, we have that w is a proper ancestor of v with $M(w) = M(B(v) \setminus \{e_L(v)\})$. Thus, since $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$, we have that $w \in W_L(v)$, and therefore $w \geq lastW_L(v)$. By Lemma 5.101, we have that $w \leq low(u)$, and therefore $lastW_L(v) \leq low(u)$. Thus, if $low(u) < firstW_L(v)$, then u satisfies enough conditions to be in $U_4^2(v)$. So let us assume that $low(u) \geq firstW_L(v)$.

Let us suppose, for the sake of contradiction, that u is not the lowest vertex in S that is a proper descendant of v such that $low(u) \geq firstW_L(v)$. Then, there is a vertex u' in S that is lower than u, it is a proper descendant of v, and has $low(u') \geq firstW_L(v)$. Since both u and u' are in S, they are related as ancestor and descendant. Thus, since u' is lower than u, we have that u' is a proper ancestor of u. Since both u and u' are in S, we have that $high(u) = high(u')$. Thus, Lemma 3.3 implies that $B(u) \subseteq B(u')$. Now let (x, y) be a back-edge in $B(u')$. Then x is a descendant of u', and therefore a descendant of v. Furthermore, y is an ancestor of $high(u')$, and therefore an ancestor of $high(v)$ (because $u' \in S$, and therefore $high(u') = high(v)$), and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Since (u, v, w) induces a Type-3βii-4 4-cut with back-edge $e_L(v)$, we have that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e_L(v)\}$. Thus, $(x, y) \in B(v)$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e_L(v)$. Since $y \geq low(u') \geq firstW_L(v) \geq w$, we have that y cannot be a proper ancestor of w, and so the case $(x, y) \in B(w)$ is rejected. Then, since u' is a proper descendant of v with

$high(u') = high(v)$, Lemma 5.115 implies that $e_L(v) \notin B(u')$. Therefore $(x, y) \neq e_L(v)$. Thus, since $(x, y) \in B(w)$ is rejected, the only viable option is $(x, y) \in B(u)$. Due to the generality of $(x, y) \in B(u')$, this implies that $B(u') \subseteq B(u)$. Thus, since $B(u) \subseteq B(u')$, we have that $B(u') = B(u)$, in contradiction to the fact that the graph is 3-edge-connected. Thus, we have that u is the lowest vertex in S that is a proper descendant of v such that $low(u) \geq firstW_L(v)$. This means that u satisfies enough conditions to be in $U_4^2(v)$. □

Lemma 5.117. *Let (u, v, w) be a triple of vertices such that $e_L(v) \notin B(u)$, $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$ and $u \in U_4^2(v)$. Then, (u, v, w) induces a Type-3βii-4 4-cut with back-edge $e_L(v)$ if and only if: $bcount(v) = bcount(u) + bcount(w) + 1$, and w is the greatest proper ancestor of v with $M(w) = M(B(v) \setminus \{e_L(v)\})$ such that $w \leq low(u)$.*

Proof. (⇒) Since (u, v, w) induces a Type-3βii-4 4-cut with back-edge $e_L(v)$, we have that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e_L(v)\}$. Thus, we get $bcount(v) = bcount(u) + bcount(w) + 1$. Furthermore, we have $M(w) = M(B(v) \setminus \{e_L(v)\})$ (since $e_L(v)$ is the back-edge of the 4-cut induced by (u, v, w)). By Lemma 5.101, we have $w \leq low(u)$.

Now let us suppose, for the sake of contradiction, that there is a proper ancestor w' of v with $M(w') = M(B(v) \setminus \{e_L(v)\})$ and $w' > w$, such that $w' \leq low(u)$. Then, since $M(w) = M(B(v) \setminus \{e_L(v)\})$ and $M(w') = M(B(v) \setminus \{e_L(v)\})$, we have that $M(B(v) \setminus \{e_L(v)\})$ is a common descendant of w and w', and therefore w and w' are related as ancestor and descendant. Thus, $w' > w$ implies that w' is a proper descendant of w. Since w' is a proper descendant of w with $M(w') = M(w)$, Lemma 3.2 implies that $B(w) \subseteq B(w')$. This can be strengthened to $B(w) \subset B(w')$, since the graph is 3-edge-connected. Thus, there is a back-edge $(x, y) \in B(w') \setminus B(w)$. Then, x is a descendant of $M(w')$, and therefore a descendant of $M(B(v) \setminus \{e_L(v)\})$, and therefore a descendant of v. Furthermore, y is a proper ancestor of w', and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Since $B(v) = (B(u) \sqcup B(w)) \sqcup \{e_L(v)\}$, this implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e_L(v)$. The case $(x, y) \in B(u)$ is rejected, since $y < w'$ and $w' \leq low(u)$, and therefore $y < low(u)$. The case $(x, y) = e_L(v)$ is also rejected, because $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$, and therefore the higher endpoint of $e_L(v)$ is not a descendant of $M(B(v) \setminus \{e_L(v)\})$ $(= M(w'))$. Thus, we have that $(x, y) \in B(w)$, a contradiction. This shows that w is the greatest proper ancestor of v with $M(w) = M(B(v) \setminus \{e_L(v)\})$ such that $w \leq low(u)$.

(⇐) We have to show that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e_L(v)\}$, and $M(w) = M(B(v) \setminus$

$\{e_L(v)\})$. Since $w \leq low(u)$, we have that $B(u) \cap B(w) = \emptyset$ (because, if there existed a back-edge in $B(u) \cap B(w)$, its lower endpoint would be lower than $low(u)$, which is impossible).

Let (x, y) be a back-edge in $B(u)$. Since $u \in U_4^2(v)$, we have that u is a proper descendant of v with $high(u) = high_1(v)$. Thus, $(x, y) \in B(u)$ implies that x is a descendant of v. Furthermore, we have that y is an ancestor of $high(u) = high_1(v)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(v)$.

Let (x, y) be a back-edge in $B(w)$. Then, x is a descendant of $M(w) = M(B(v) \setminus \{e_L(v)\})$, and therefore a descendant of v. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(w)$, this implies that $B(w) \subseteq B(v)$.

Since $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$, we have that the higher endpoint of $e_L(v)$ is not a descendant of $M(B(v) \setminus \{e_L(v)\})$. Since $M(B(v) \setminus \{e_L(v)\}) = M(w)$, this implies that $e_L(v) \notin B(w)$. By assumption, we have $e_L(v) \notin B(u)$. Therefore, we have $e_L(v) \notin B(u) \cup B(w)$.

Thus, we have $B(u) \subseteq B(v)$, $B(w) \subseteq B(v)$, and $B(u) \cap B(w) = \emptyset$. This implies that $B(u) \sqcup B(v) \subseteq B(v)$. Therefore, since $bcount(v) = bcount(u) + bcount(w) + 1$, we have that there is a back-edge e, such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. Since $e_L(v) \notin B(u) \cup B(w)$, this implies that $e = e_L(v)$. By assumption, we have $M(w) = M(B(v) \setminus \{e_L(v)\})$.

□

Proposition 5.30. *Algorithm 49 correctly computes all Type-3βii-4 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e_L(v)\}$, where u is a descendant of v, v is a descendant of w, $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$ and $high_1(v) = high(u)$. Furthermore, it has a linear-time implementation.*

Proof. Let (u, v, w) be a triple of vertices that induces a Type-3βii-4 4-cut with back-edge $e_L(v)$ such that $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$ and $high_1(v) = high(u)$. By the definition of Type-3βii-4 4-cuts, this implies that $e_L(v) \notin B(u)$. Then, by Lemma 5.116 we have that $u \in U_4^2(v)$. Then, by Lemma 5.117 we have that w is the greatest proper ancestor of v with $M(w) = M(B(v) \setminus \{e_L(v)\})$ such that $w \leq low(u)$. This implies that $w \in W_L(v)$. Thus, since $M(B(v) \setminus \{e_L(v)\}) \neq M(v)$ and $W_L(v) \neq \emptyset$, we have that v is in the collection $\mathcal{V}$, computed in Line 1. Then, notice that

Algorithm 49: Compute all Type-3βii-4 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e_L(v)\}$, where u is a proper descendant of v, v is a proper descendant of w, $M(B(v)\setminus\{e_L(v)\}) \neq M(v)$ and $high_1(v) = high(u)$

```
1  let 𝒱 be the collection of all vertices v ≠ r such that M(B(v) \ {e_L(v)}) ≠ M(v)
     and W_L(v) ≠ ∅
2  foreach v ∈ 𝒱 do
3  |   compute U_4^2(v)
4  end
5  foreach v ∈ 𝒱 do
6  |   foreach u ∈ U_4^2(v) do
7  |   |   let w be the greatest proper ancestor of v with
   |   |     M(w) = M(B(v) \ {e_L(v)}) such that w ≤ low(u)
8  |   |   if bcount(v) = bcount(u) + bcount(w) + 1 and e_L(v) ∉ B(u) then
9  |   |   |   mark {(u,p(u)),(v,p(v)),(w,p(w)),e_L(v)} as a Type-3βii-4 4-cut
10 |   |   end
11 |   end
12 end
```

$\{(u,p(u)),(v,p(v)),(w,p(w)),e_L(v)\}$ (i.e., the 4-cut induced by (u,v,w)) will be correctly marked by Algorithm 49 in Line 9.

Conversely, suppose that a 4-element set $\{(u,p(u)),(v,p(v)),(w,p(w)),e_L(v)\}$ is marked by Algorithm 49 in Line 9. Then we have that: (1) v is a vertex such that $M(B(v)\setminus\{e_L(v)\})\neq M(v)$ (due to $v\in\mathcal{V}$), (2) $u\in U_4^2(v)$, (3) w is the greatest proper ancestor of v such that $M(w)=M(B(v)\setminus\{e_L(v)\})$ and $w\leq \mathit{low}(u)$, and (4) $\mathit{bcount}(v)=\mathit{bcount}(u)+\mathit{bcount}(w)+1$ and $e_L(v)\notin B(u)$. Thus, Lemma 5.117 implies that $\{(u,p(u)),(v,p(v)),(w,p(w)),e_L(v)\}$ is a Type-3βii-4 4-cut. Since $u\in U_4^2(v)$, we have that $\mathit{high}_1(v)=\mathit{high}(u)$.

Now we will show that Algorithm 49 has a linear-time implementation. First, by Proposition 3.6, we can compute $M(B(v)\setminus\{e_L(v)\})$ for all vertices $v\neq r$, in total linear time. By Lemma 5.112, we have that the values $\mathit{firstW}_L(v)$ and $\mathit{lastW}_L(v)$ can be computed in linear time in total, for all vertices $v\neq r$. Then, we can check in constant time whether $W_L(v)\neq\emptyset$, for a vertex $v\neq r$, by simply checking whether e.g. $\mathit{firstW}_L(v)\neq\perp$. Thus, the set $\mathcal{V}$ in Line 1 can be computed in linear time. By Lemma 5.114, we have that the sets $U_4^2(v)$ can be computed in total linear time, for all vertices $v\in\mathcal{V}$, using Algorithm 48. Thus, the **for** loop in Line 3 can be performed in linear time. In particular, we have that the total size of all U_4^2 sets is $O(n)$.

It remains to show how to find the vertex w in Line 7. To do this, we use Algorithm 22. Specifically, let v be a vertex in $\mathcal{V}$, let u be a vertex in $U_4^2(v)$, and let $x=M(B(v)\setminus\{e_L(v)\})$. Then we generate a query $q(M^{-1}(x),\mathit{min}\{\mathit{low}(u),p(v)\})$. This is to return the greatest vertex w such that $M(w)=M(B(v)\setminus\{e_L(v)\})$ and $w\leq\mathit{low}(u)$ and $w\leq p(v)$. Since $M(w)=M(B(v)\setminus\{e_L(v)\})$, we have that $M(w)$ is a descendant of $M(v)$, and therefore a descendant of v. Thus, $M(w)$ is a common descendant of v and w, and therefore v and w are related as ancestor and descendant. Thus, $w\leq p(v)$ implies that w is a proper ancestor of v, and so w is the greatest proper ancestor of v such that $M(w)=M(B(v)\setminus\{e_L(v)\})$ and $w\leq\mathit{low}(u)$. Now, using Algorithm 22 we can answer all those queries in total linear time, according to Lemma 5.27. Thus, the **for** loop in Line 5 can be performed in linear time. We conclude that Algorithm 49 runs in linear time. ☐

The case where $M(B(v)\setminus\{e\})=M(v)$ and $\mathit{high}_1(v)>\mathit{high}(u)$

Let (u,v,w) be a triple of vertices that induces a Type-3βii-4 4-cut, such that $M(w)=M(B(v)\setminus\{e\})=M(v)$ and $\mathit{high}_1(v)>\mathit{high}(u)$, where e is the back-edge in the 4-cut induced by (u,v,w). Then, by Lemma 5.101 we have that $\mathit{high}_1(v)\neq\mathit{high}_2(v)=\mathit{high}(u)$, and $e=e_{\mathit{high}}(v)$. Furthermore, by Lemma 5.101 we have that $w\leq \mathit{low}(u)$. Since $\mathit{high}_1(v)\neq\mathit{high}_2(v)$, we have that $\mathit{high}_2(v)<\mathit{high}_1(v)$. Thus, since $\mathit{low}(u)\leq\mathit{high}(u)=\mathit{high}_2(v)<\mathit{high}_1(v)$, we have $w\leq\mathit{high}_2(v)<\mathit{high}_1(v)$. Since $M(w)=M(v)$, this implies that w is an ancestor of $\mathit{high}_2(v)$, and a proper ancestor of $\mathit{high}_1(v)$.

Now let $v\neq r$ be a vertex such that $\mathit{high}_1(v)\neq\mathit{high}_2(v)$ and $\mathit{lastM}(v)\leq\mathit{high}_2(v)$. We let $\widetilde{\mathit{nextM}}(v)$ denote the greatest proper ancestor w of $\mathit{high}_1(v)$ such that $M(w)=M(v)$.

Lemma 5.118. *Let $v\neq r$ be a vertex such that $\mathit{high}_1(v)\neq\mathit{high}_2(v)$ and $\mathit{lastM}(v)\leq\mathit{high}_2(v)$. Then $\widetilde{\mathit{nextM}}(v)$ is either $\mathit{nextM}(v)$ or $\mathit{nextM}(\mathit{nextM}(v))$.*

Proof. Since $\mathit{high}_1(v)\neq\mathit{high}_2(v)$, we have that $\mathit{high}_2(v)<\mathit{high}_1(v)$. Thus, since $\mathit{high}_1(v)$ and $\mathit{high}_2(v)$ are both ancestors of v, we have that $\mathit{high}_2(v)$ is a proper ancestor of $\mathit{high}_1(v)$. Since $\mathit{lastM}(v)$ and v have the same M point, we have that $\mathit{lastM}(v)$ is related as ancestor and descendant with v. Due to the minimality of $\mathit{lastM}(v)$ in $M^{-1}(M(v))$, this implies that $\mathit{lastM}(v)$ is an ancestor of v. Thus, since v is a common descendant of $\mathit{high}_2(v)$ and $\mathit{lastM}(v)$, we have that $\mathit{high}_2(v)$ and $\mathit{lastM}(v)$ are related as ancestor and descendant. Thus, $\mathit{lastM}(v)\leq\mathit{high}_2(v)$ implies that $\mathit{lastM}(v)$ is an ancestor of $\mathit{high}_2(v)$. Therefore, $\mathit{lastM}(v)$ is a proper ancestor of $\mathit{high}_1(v)$. This shows that $\widetilde{\mathit{nextM}}(v)\neq\bot$.

Now suppose that $\widetilde{\mathit{nextM}}(v)\neq\mathit{nextM}(v)$. By Lemma 3.6 we have that $\mathit{nextM}(v)$ is an ancestor of $\mathit{high}_1(v)$. Thus, $\widetilde{\mathit{nextM}}(v)\neq\mathit{nextM}(v)$ implies that $\mathit{nextM}(v)=\mathit{high}_1(v)$. By definition, $\mathit{nextM}(\mathit{nextM}(v))$ is the greatest proper ancestor of $\mathit{nextM}(v)$ that has the same M point with v. Thus, since $\mathit{nextM}(v)=\mathit{high}_1(v)$, we have that $\mathit{nextM}(\mathit{nextM}(v))$ is the greatest proper ancestor of $\mathit{high}_1(v)$ that has the same M point as v. In other words, $\mathit{nextM}(\mathit{nextM}(v))=\widetilde{\mathit{nextM}}(v)$. $\square$

Now let S be the segment of $\widetilde{H}(\mathit{high}_2(v))$ that contains v and is maximal w.r.t. the property that all its elements are related as ancestor and descendant (i.e., we have $S=\widetilde{S}_2(v)$). Then, we let $U_4^3(v)$ denote the collection of all vertices $u\in S$ such that: (1) u is a proper descendant of v with $\mathit{high}(u)=\mathit{high}_2(v)$, (2) $\mathit{low}(u)\geq\mathit{lastM}(v)$, and (3) either $\mathit{low}(u)<\widetilde{\mathit{nextM}}(v)$, or u is the lowest vertex in S that satisfies (1), (2) and $\mathit{low}(u)\geq\widetilde{\mathit{nextM}}(v)$.

Lemma 5.119. *Let v and v' be two vertices $\neq r$ such that v' is a proper descendant of v with $\mathit{high}_1(v)\neq\mathit{high}_2(v)=\mathit{high}_2(v')\neq\mathit{high}_1(v')$. Let w and w' be two vertices such that*

$M(w) = M(v)$, $M(w') = M(v')$, and both w and w' are ancestors of $high_2(v) = high_2(v')$. Then w' is a proper ancestor of w.

Proof. Let (x, y) be a back-edge in $B(v')$ such that $y = high_2(v')$. Then x is a descendant of v', and therefore a descendant of v. Furthermore, $y = high_2(v') = high_2(v)$ is a proper ancestor of v. This shows that $(x, y) \in B(v)$, and therefore x is a descendant of $M(v)$. Thus, since x is a common descendant of $M(v)$ and v', we have that $M(v)$ and v' are related as ancestor and descendant.

Let us suppose, for the sake of contradiction, that $M(v)$ is not a proper ancestor of v'. Then we have that $M(v)$ is a descendant of v'. We have that $high_1(v')$ and v are related as ancestor and descendant, since both have v' as a common descendant. Let us suppose, for the sake of contradiction, that $high_1(v')$ is not a proper ancestor of v. Then we have that $high_1(v')$ is a descendant of v. Now consider a back-edge $(x, y) \in B(v)$ such that $y = high_1(v)$. Then we have that x is a descendant of $M(v)$, and therefore a descendant of v'. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of v'. This shows that $(x, y) \in B(v')$. Since $high_1(v)$ is a proper ancestor of v and $high_1(v')$ is a descendant of v, we have that $high_1(v')$ is a proper descendant of $high_1(v)$, and therefore $high_1(v') > high_1(v)$. Since $(x, y) \in B(v')$, this implies that $high_2(v') \geq high_1(v)$. Since $high_2(v') = high_2(v)$, this implies that $high_2(v) \geq high_1(v)$, and therefore $high_2(v) = high_1(v)$. But this contradicts $high_2(v) \neq high_1(v)$. This shows that our last supposition cannot be true, and therefore we have that $high_1(v')$ is a proper ancestor of v.

Now let (x, y) be a back-edge in $B(v')$. Then x is a descendant of v', and therefore a descendant of v. Furthermore, y is an ancestor of $high_1(v')$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(v')$, this implies that $B(v') \subseteq B(v)$. Conversely, let (x, y) be a back-edge in $B(v)$. Then x is a descendant of $M(v)$, and therefore a descendant of v'. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of v'. This shows that $(x, y) \in B(v')$. Due to the generality of $(x, y) \in B(v)$, this implies that $B(v) \subseteq B(v')$. Thus we have $B(v') = B(v)$, in contradiction to the fact that the graph is 3-edge-connected. This shows that $M(v)$ is a proper ancestor of v'.

Now let w and w' be two vertices such that $M(w) = M(v)$, $M(w') = M(v')$, and both w and w' are ancestors of $high_2(v) = high_2(v')$. Then w and w' are related as ancestor and descendant. Let us suppose, for the sake of contradiction, that w' is not a proper ancestor of w. Then we have that w' is a descendant of w. Since

$M(v) = M(w)$ is a proper ancestor of v', there is a back-edge $(x,y) \in B(w)$ such that x is not a descendant of v'. Then, x is a descendant of $M(w) = M(v)$, and therefore a descendant of v, and therefore a descendnant of $high_2(v)$, and therefore a descendant of w'. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of w'. This shows that $(x,y) \in B(w')$. But we have that $M(w') = M(v')$, and therefore x must be a descendant of $M(v')$, and therefore a descendant of v'. This contradicts the fact that x is not a descendant of v'. Thus, we have that w' is a proper ancestor of w. □

Lemma 5.120. *Let v and v' be two vertices $\neq r$ such that $high_1(v) \neq high_2(v) = high_2(v') \neq high_1(v')$, $lastM(v) \leq high_2(v)$, $lastM(v') \leq high_2(v')$, v' is a proper descendant of v, and both v and v' belong to the same segment $\widetilde{S}$ of $\widetilde{H}(high_2(v))$ that is maximal w.r.t. the property that all its elements are related as ancestor and descendant (i.e., we have $\widetilde{S} = \widetilde{S}(v) = \widetilde{S}(v')$). If $U_4^3(v') = \emptyset$, then $U_4^3(v) = \emptyset$. If $U_4^3(v) \neq \emptyset$, then the lowest vertex in $U_4^3(v)$ is at least as great as the greatest vertex in $U_4^3(v')$.*

Proof. Notice that, since $lastM(v) \leq high_2(v)$ and $lastM(v') \leq high_2(v')$, we have that both $\widetilde{nextM}(v)$ and $\widetilde{nextM}(v')$ are well-defined. Furthermore, Lemma 5.119 implies that $\widetilde{nextM}(v')$ is a proper ancestor of $lastM(v)$.

Let us suppose, for the sake of contradiction, that $U_4^3(v') = \emptyset$ and $U_4^3(v) \neq \emptyset$. Let u be a vertex in $U_4^3(v)$. Let us suppose, for the sake of contradiction, that u is not a proper descendant of v'. Since $u \in U_4^3(v)$, we have that $u \in \widetilde{S}$. Thus, since $v' \in \widetilde{S}$, we have that u and v' are related as ancestor and descendant. Since u is not a proper descendant of v', we have that u is an ancestor of v'. Let (x,y) be a back-edge in $B(v')$ such that $y = low(v')$. Lemma 3.4 implies that $low(v')$ is a proper ancestor of $\widetilde{nextM}(v')$. Thus, since $\widetilde{nextM}(v')$ is a proper ancestor of $lastM(v)$, we have that $low(v')$ is a proper ancestor of $lastM(v)$. Now, since $(x,y) \in B(v')$, we have that x is a descendant of v', and therefore a descendant of u. Furthermore, $y = low(v')$ is a proper ancestor of $lastM(v)$, and therefore a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x,y) \in B(u)$. But then we have that $low(u) \leq y = low(v') < lastM(v)$, in contradiction to the fact that $u \in U_4^3(v)$. Thus, our last supposition is not true, and therefore u is a proper descendant of v'. Then, since $u \in U_4^3(v)$, we have that $high(u) = high_2(v) = high_2(v')$. Furthermore, we have that $low(u) \geq lastM(v)$, and therefore $low(u) \geq \widetilde{nextM}(v')$. This implies that $U_4^3(v')$ is not empty (because we can consider the lowest proper descendant u' of v' in

$\widetilde{S}(v') = \widetilde{S}(v)$ such that $high(u') = high_2(v')$ and $low(u') \geq \widetilde{nextM}(v')$). This contradicts our supposition that $U_4^3(v') \neq \emptyset$. Thus, we have shown that $U_4^3(v') = \emptyset$ implies that $U_4^3(v) = \emptyset$.

Now let us assume that $U_4^3(v) \neq \emptyset$. This implies that $U_4^3(v')$ is not empty. Let us suppose, for the sake of contradiction, that there is a vertex $u \in U_4^3(v)$ that is lower than the greatest vertex u' in $U_4^3(v')$. Since $u \in U_4^3(v)$, we have that $u \in \widetilde{S}$. Since $u' \in U_4^3(v')$ we have that $u' \in \widetilde{S}$. This implies that u and u' are related as ancestor and descendant. Thus, since u is lower than u', we have that u is a proper ancestor of u'. Let us suppose, for the sake of contradiction, that $low(u')$ is a proper ancestor of $\widetilde{nextM}(v')$. Then, since $\widetilde{nextM}(v')$ is a proper ancestor of $lastM(v)$, we have that $low(u')$ is a proper ancestor of $lastM(v)$. Now let (x, y) be a back-edge in $B(u')$ such that $y = low(u')$. Then x is a descendant of u', and therefore a descendant of u. Furthermore, y is a proper ancestor of $lastM(v)$, and therefore a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Thus, we have $low(u) \leq y = low(u') < lastM(v)$, in contradiction to the fact that $u \in U_4^3(v)$. Thus, our last supposition is not true, and therefore we have that $low(u')$ is not a proper ancestor of $\widetilde{nextM}(v')$.

Since $u' \in U_4^3(v')$, we have that u' is a proper descendant of v', and therefore a proper descendant of $\widetilde{nextM}(v')$. Thus, since u' is a common descendant of $low(u')$ and $\widetilde{nextM}(v')$, we have that $low(u')$ and $\widetilde{nextM}(v')$ are related as ancestor and descendant. Therefore, since $low(u')$ is not a proper ancestor of $\widetilde{nextM}(v')$, we have that $low(u')$ is a descendant of $\widetilde{nextM}(v')$, and therefore $low(u') \geq \widetilde{nextM}(v')$. Thus, since $u' \in U_4^3(v')$, we have that u' is the lowest proper descendant of v' in $\widetilde{S}$ with $high(u') = high_2(v')$ and $low(u') \geq \widetilde{nextM}(v')$ $(*)$.

Now we will trace the implications of $u \in U_4^3(v)$. First, we have that $u \in \widetilde{S}$. Then, we have $high(u) = high_2(v) = high_2(v')$. Furthermore, we have that $low(u) \geq lastM(v)$, and therefore $low(u) > \widetilde{nextM}(v')$ (since $\widetilde{nextM}(v')$ is a proper ancestor of $lastM(v)$). Finally, we can show as above that u is a proper descendant of v' (the proof of this fact above did not rely on $U_4^3(v') = \emptyset$). But then, since u is lower than u', we have a contradiction to $(*)$. Thus, we have shown that every vertex in $U_4^3(v)$ is at least as great as the greatest vertex in $U_4^3(v')$. In particular, this implies that the lowest vertex in $U_4^3(v)$ is greater than, or equal to, the greatest vertex in $U_4^3(v')$. □

Due to the similarity of the definitions of the U_3 and the U_4^3 sets, and the similarity

between Lemmata 5.98 and 5.120, we can use a similar procedure as Algorithm 44 in order to compute all U_4^3 sets in linear time. This is shown in Algorithm 50. Our result is summarized in Lemma 5.121.

Lemma 5.121. *Algorithm 50 correctly computes the sets $U_4^3(v)$, for all vertices $v \neq r$ such that $\widetilde{nextM}(v) \neq \emptyset$. Furthermore, it runs in linear time.*

Proof. The proof here is similar as that of Lemma 5.99 (which was given in the main text, in the two paragraphs above Algorithm 44). The differences are the following. First, the set $\mathcal{V}$ of the vertices for which the sets U_4^3 are defined is given by all vertices $v \neq r$ such that $high_1(v) \neq high_2(v)$ and $\widetilde{nextM}(v) \neq \perp$. Due to Lemma 5.118, it is easy to collect all those vertices in $O(n)$ time. I.e., we have to check, for every vertex $v \neq r$ with $high_1(v) \neq high_2(v)$, whether $nextM(v)$ is a proper ancestor of $high_1(v)$. If that is the case, then $\widetilde{nextM}(v) = nextM(v)$. Otherwise, $\widetilde{nextM}(v) = nextM(nextM(v))$ (which may be *null*). Second, here we process the vertices v in their $\widetilde{S}_2(v)$ segment (instead of the $\widetilde{S}_1(v)$). Thus, in Lines 11 and 33, we check whether v satisfies $high_2(v) = x$ (where x is the vertex for which we process the $\widetilde{H}(x)$ list). And third, we have that every $u \in U_4^3(v)$ satisfies $high_1(u) = high_2(v)$. Thus, in Lines 21 and 26 we have the appropriate condition (where is it checked whether $high_1(u) = x$). Then the proof follows the same reasoning as in Lemma 5.99. The main difference in the argument here is that every reference to Lemma 5.98 is replaced with a reference to Lemma 5.120. □

Lemma 5.122. *Let (u, v, w) be a triple of vertices that induces a Type-3βii-4 4-cut, such that $high_1(v) > high(u)$ and $M(B(v) \setminus \{e_{high}(v)\}) = M(v)$. Then $u \in U_4^3(v)$.*

Proof. Since $high_1(v) > high(u)$, Lemma 5.101 implies that $high_2(v) = high(u)$. Thus, we may consider the segment $\widetilde{S}$ of $\widetilde{H}(high_2(v))$ from u to v. Let u' be a vertex in $\widetilde{S}$. Then, we have that $u \geq u' \geq v$, and therefore Lemma 5.102 implies that u' is an ancestor of u. Thus, we have that all elements of $\widetilde{S}$ are related as ancestor and descendant (since all of them are ancestors of u). Since $high_1(v) > high(u)$, Lemma 5.101 implies that $e = e_{high}(v)$, where e is the back-edge in the 4-cut induced by (u, v, w). Furthermore, Lemma 5.101 implies that $w \leq low(u)$. Thus, since $low(u) \leq high(u) = high_2(v)$, we have that $w \leq high_2(v)$. Then, since (u, v, w) induces a Type-3βii-4 4-cut, we have $M(w) = M(B(v) \setminus \{e\}) = M(v)$, and therefore $\widetilde{nextM}(v)$ is defined (and it is greater than, or equal to, w). Also, we have $w \geq lastM(v)$, and therefore $low(u) \geq lastM(v)$.

Algorithm 50: Compute the sets $U_4^3(v)$, for all vertices $v \neq r$ such that $high_1(v) \neq high_2(v)$ and $\widetilde{nextM}(v) \neq \bot$

```
1  let 𝒱 be the collection of all vertices v ≠ r such that high_1(v) ≠ high_2(v) and nextM~(v) ≠ ⊥
2  foreach vertex x do
3      compute the collection 𝒮(x) of the segments of H~(x) that are maximal w.r.t. the property
         that their elements are related as ancestor and descendant
4  end
5  foreach v ∈ 𝒱 do
6      set U_4^3(v) ← ∅
7  end
8  foreach vertex x do
9      foreach segment S ∈ 𝒮(x) do
10         let v be the first vertex in S
11         while v ≠ ⊥ and (high_2(v) ≠ x or v ∉ 𝒱) do
12             v ← next_S(v)
13         end
14         if v = ⊥ then continue
15         let u ← prev_S(v)
16         while v ≠ ⊥ do
17             while u ≠ ⊥ and low(u) < lastM(v) do
18                 u ← prev_S(u)
19             end
20             while u ≠ ⊥ and low(u) < nextM~(v) do
21                 if high_1(u) = x then
22                     insert u into U_4^3(v)
23                 end
24                 u ← prev_S(u)
25             end
26             while u ≠ ⊥ and high_1(u) ≠ x do
27                 u ← prev_S(u)
28             end
29             if u ≠ ⊥ then
30                 insert u into U_4^3(v)
31             end
32             v ← next_S(v)
33             while v ≠ ⊥ and (high_2(v) ≠ x or v ∉ 𝒱) do
34                 v ← next_S(v)
35             end
36         end
37     end
38 end
```

Thus, if $low(u) < \widetilde{nextM}(v)$, then u satisfies enough conditions to be in $U_4^3(v)$. So let us assume that $low(u) \geq \widetilde{nextM}(v)$.

Let us suppose, for the sake of contradiction, that u is not the lowest vertex in $\widetilde{S}$ that is a proper descendant of v such that $high(u) = high_2(v)$, $low(u) \geq lastM(v)$ and $low(u) \geq \widetilde{nextM}(v)$. Then, there is a vertex u' in $\widetilde{S}$, that is a proper ancestor of u and a proper descendant of v, such that $high(u') = high_2(v)$, $low(u') \geq lastM(v)$ and $low(u') \geq \widetilde{nextM}(v)$. Since (u, v, w) induces a Type-3βii-4 4-cut, we have that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. Now let (x, y) be a back-edge in $B(u)$. Then x is a descendant of u, and therefore a descendant of u'. Furthermore, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $(x, y) \in B(v)$, and therefore y is a proper ancestor of v, and therefore a proper ancestor of u'. This shows that $(x, y) \in B(u')$. Due to the generality of $(x, y) \in B(u)$, this shows that $B(u) \subseteq B(u')$. Conversely, let (x, y) be a back-edge in $B(u')$. Then we have that x is a descendant of u', and therefore a descendant of v. Furthermore, y is an ancestor of $high(u') = high_2(v)$, and therefore it is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Then, $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. The case $(x, y) \in B(w)$ is rejected, since $y = low(u') \geq \widetilde{nextM}(v) \geq w$. Furthermore, since $e = e_{high}(v)$ and $high_1(v) > high(u)$ and $high(u) = high_2(v) = high(u')$, we cannot have $e_{high}(v) \in B(u')$, and therefore the case $(x, y) = e$ is also rejected. Thus, we have that $(x, y) \in B(u)$. Due to the generality of $(x, y) \in B(u')$, this shows that $B(u') \subseteq B(u)$. Thus, we have $B(u) = B(u')$, in contradiction to the fact that the graph is 3-edge-connected. Thus, we have shown that u is the lowest vertex in $\widetilde{S}$ that is a proper descendant of v such that $high(u) = high_2(v)$, $low(u) \geq lastM(v)$ and $low(u) \geq \widetilde{nextM}(v)$. We conclude that u satisfies enough conditions to be in $U_4^3(v)$. □

Lemma 5.123. *Let (u, v, w) be a triple of vertices such that $high_1(v) > high(u)$, $M(B(v) \setminus \{e_{high}(v)\}) = M(v)$ and $u \in U_4^3(v)$. Then, (u, v, w) induces a Type-3βii-4 4-cut if and only if: $bcount(v) = bcount(u) + bcount(w) + 1$, and w is the greatest proper ancestor of v with $M(w) = M(v)$ such that $w \leq low(u)$.*

Proof. ($\Rightarrow$) Since (u, v, w) induces a Type-3βii-4 4-cut, we have $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, where e is the back-edge in the 4-cut induced by (u, v, w). Thus, we get $bcount(v) = bcount(u) + bcount(w) + 1$. Since $high_1(v) > high(u)$, by Lemma 5.101 we have $e = e_{high}(v)$. Furthermore, Lemma 5.101 implies that $w \leq low(u)$. Since (u, v, w) induces a Type-3βii-4 4-cut, we have $M(w) = M(B(v) \setminus \{e\})$. Thus, since $M(B(v) \setminus \{e_{high}(v)\}) = M(v)$,

we have $M(w) = M(v)$.

Now let us suppose, for the sake of contradiction, that there is a proper ancestor w' of v with $M(w') = M(v)$ and $w' \leq low(u)$, such that $w' > w$. Then, since $M(v) = M(w)$ and $M(v) = M(w')$, we have that $M(v)$ is a common descendant of w and w', and therefore w and w' are related as ancestor and descendant. Thus, $w' > w$ implies that w' is a proper descendant of w. Since w' is a proper descendant of w with $M(w') = M(w)$, Lemma 3.2 implies that $B(w) \subseteq B(w')$. This can be strengthened to $B(w) \subset B(w')$, since the graph is 3-edge-connected. Thus, there is a back-edge $(x, y) \in B(w') \setminus B(w)$. Then, x is a descendant of $M(w') = M(v)$. Furthermore, y is a proper ancestor of w', and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Since $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, this implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. The case $(x, y) \in B(u)$ is rejected, since $y < w'$ and $w' \leq low(u)$, and therefore $y < low(u)$. The case $(x, y) = e$ is also rejected, because $e = e_{high}(v)$, and $high_1(v) > high(u) \geq low(u) \geq w'$. Thus, we have that $(x, y) \in B(w)$, a contradiction. This shows that w is the greatest proper ancestor of v with $M(w) = M(v)$ such that $w \leq low(u)$.

($\Leftarrow$) We have to show that there is a back-edge e such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$, and $M(w) = M(B(v) \setminus \{e\})$.

Since $M(B(v) \setminus \{e_{high}(v)\}) = M(v)$ and $M(w) = M(v)$, we have that $M(w) = M(B(v) \setminus \{e_{high}(v)\})$. Since $high_1(v) > high(u)$, we have that $e_{high}(v) \notin B(u)$. And since $w \leq low(u) \leq high(u) < high_1(v)$, we have that $e_{high}(v) \notin B(w)$. Furthermore, since $w \leq low(u)$, we have that $B(u) \cap B(w) = \emptyset$ (because, if there existed a back-edge in $B(u) \cap B(w)$, its lower endpoint would be lower than $low(u)$, which is impossible). This shows that the sets $B(u)$, $B(w)$ and $\{e_{high}(v)\}$ are pairwise disjoint.

Let (x, y) be a back-edge in $B(u)$. Since $u \in U_4^3(v)$, we have that u is a proper descendant of v with $high(u) = high_2(v)$. Thus, $(x, y) \in B(u)$ implies that x is a descendant of v. Furthermore, we have that y is an ancestor of $high(u) = high_2(v)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(v)$.

Let (x, y) be a back-edge in $B(w)$. Then, x is a descendant of $M(w) = M(v)$, and therefore a descendant of v. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(w)$, this implies that $B(w) \subseteq B(v)$.

Thus, we have $B(u) \subseteq B(v)$, $B(w) \subseteq B(v)$, $e_{high}(v) \in B(v)$, and the sets $B(u)$, $B(w)$

and $\{e_{high}(v)\}$ are pairwise disjoint. Thus, $(B(u) \sqcup B(v)) \sqcup \{e_{high}(v)\} \subseteq B(v)$. Since $bcount(v) = bcount(u) + bcount(w) + 1$, this implies that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e_{high}(v)\}$. By assumption we have $M(B(v) \setminus \{e_{high}(v)\}) = M(v)$. Therefore, since $M(w) = M(v)$, we have $M(w) = M(B(v) \setminus \{e_{high}(v)\})$. □

Algorithm 51: Compute all Type-3βii-4 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$, where $M(B(v) \setminus \{e\}) = M(v)$ and $high_1(v) > high(u)$

---

1 let $\mathcal{V}$ be the collection of all vertices $v \neq r$ such that $M(B(v) \setminus \{e_{high}(v)\}) = M(v)$, $high_1(v) \neq high_2(v)$ and $lastM(v) \leq high_2(v)$
2 **foreach** $v \in \mathcal{V}$ **do**
3   compute $U_4^3(v)$
4 **end**
5 **foreach** $v \in \mathcal{V}$ **do**
6   **foreach** $u \in U_4^3(v)$ **do**
7   let w be the greatest proper ancestor of v with $M(w) = M(v)$ such that $w \leq low(u)$
8   **if** $bcount(v) = bcount(u) + bcount(w) + 1$ ***and*** $high_1(v) > high(u)$ **then**
9   mark $\{(u, p(u)), (v, p(v)), (w, p(w)), e_{high}(v)\}$ as a Type-3βii-4 4-cut
10   **end**
11   **end**
12 **end**

Proposition 5.31. *Algorithm 51 correctly computes all Type-3βii-4 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$, where u is a descendant of v, v is a descendant of w, $M(B(v) \setminus \{e\}) = M(v)$ and $high_1(v) > high(u)$. Furthermore, it has a linear-time implementation.*

Proof. Let (u, v, w) be a triple of vertices that induces a Type-3βii-4 4-cut with back-edge e such that $M(B(v) \setminus \{e\}) = M(v)$ and $high_1(v) > high(u)$. Since $high_1(v) > high(u)$, by Lemma 5.101 we have that $high_1(v) \neq high_2(v)$ and $e = e_{high}(v)$. Then, by Lemma 5.122 we have that $u \in U_4^3(v)$. This implies that $lastM(v) \leq high_2(v)$, and therefore v belongs to the collection $\mathcal{V}$ computed in Line 1. Then, by Lemma 5.123 we have that $bcount(v) = bcount(u) + bcount(w) + 1$, and w is the greatest proper

ancestor of v with $M(w) = M(v)$ such that $w \leq low(u)$. Then, notice that $\{(u,p(u)),(v,p(v)),(w,p(w)),e_{high}(v))\}$ (i.e., the 4-cut induced by (u,v,w)) will be correctly marked by Algorithm 51 in Line 9.

Conversely, suppose that a 4-element set $\{(u,p(u)),(v,p(v)),(w,p(w)),e_{high}(v)\}$ is marked by Algorithm 51 in Line 9. Then we have that: (1) $M(B(v)\setminus\{e_{high}(v)\}) = M(v)$, (2) $u \in U_4^3(v)$, (3) w is the greatest proper ancestor of v such that $M(w) = M(v)$ and $w \leq low(u)$, and (4) $bcount(v) = bcount(u) + bcount(w) + 1$ and $high_1(v) > high(u)$. Thus, Lemma 5.123 implies that $\{(u,p(u)),(v,p(v)),(w,p(w)),e_{high}(v)\}$ is a Type-3βii-4 4-cut.

Now we will show that Algorithm 51 has a linear-time implementation. Computing the values $M(B(v)\setminus\{e_{high}(v)\})$, for all vertices $v \neq r$, takes linear time in total, according to Proposition 3.6. Thus, the computation of the collection of vertices $\mathcal{V}$ in Line 1 can be performed in linear time. By Lemma 5.121, we have that the sets $U_4^3(v)$ can be computed in linear time, for all vertices $v \in \mathcal{V}$, using Algorithm 50. Thus, the **for** loop in Line 3 can be performed in linear time. In particular, we have that the total size of all sets U_4^3 is $O(n)$. In order to find the vertex w in Line 7, we can use Algorithm 22. Specifically, let v and u be two vertices such that $v \in \mathcal{V}$ and $u \in U_4^3(v)$. Then we generate a query $q(M^{-1}(M(v)), min\{low(u), p(v)\})$. This returns the greatest w such that $M(w) = M(v)$ and $w \leq low(u)$ and $w \leq p(v)$. Thus, we have that w is the greatest proper ancestor of v such that $M(w) = M(v)$ and $w \leq low(u)$. Since the number of all those queries is $O(n)$, Lemma 5.27 implies that all of them can be answered in linear time in total, using Algorithm 22. We conclude that Algorithm 51 has a linear-time implementation. □

The case where $M(B(v) \setminus \{e\}) = M(v)$ and $high_1(v) = high(u)$

Let (u,v,w) be a triple of vertices that induces a Type-3βii-4 4-cut such that $M(B(v) \setminus \{e\}) = M(v)$. Then, since $M(w) = M(B(v) \setminus \{e\})$, we have $M(w) = M(v)$.

Now let $v \neq r$ be a vertex such that $nextM(v) \neq \bot$. Then we let $U_4^4(v)$ denote the collection of all vertices $u \in S(v)$ such that: (1) u is a proper descendant of v, (2) $low(u) \geq lastM(v)$, and (3) either $low(u) < nextM(v)$, or u is the lowest vertex in $S(v)$ that satisfies (1) and $low(u) \geq nextM(v)$.

Lemma 5.124. *Let v and v' be two vertices $\neq r$ with $nextM(v) \neq \bot$ and $nextM(v') \neq \bot$, such that v' is a proper descendant of v with $high_1(v) = high_1(v')$. Then, $nextM(v')$ is a*

proper ancestor of $lastM(v)$.

Proof. Let (x, y) be a back-edge in $B(v')$ such that and $y = high_1(v')$. Then, we have that x is a descendant of v', and therefore a descendant of v. Furthermore, since $high_1(v') = high_1(v)$, we have that y is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Thus, we have that $M(v)$ is an ancestor of x. Therefore, since x is a common descendant of v' and $M(v)$, we have that v' and $M(v)$ are related as ancestor and descendant.

Let us suppose, for the sake of contradiction, that $M(v)$ is not a proper ancestor of v'. Then, we have that $M(v)$ is a descendant of v'. Let (x, y) be a back-edge in $B(v')$. Then x is a descendant of v', and therefore a descendant of v. Furthermore, y is an ancestor of $high_1(v') = high_1(v)$, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(v')$, this implies that $B(v') \subseteq B(v)$. Conversely, let (x, y) be a back-edge in $B(v)$. Then, x is a descendant of $M(v)$, and therefore a descendant of v'. Furthermore, y is a proper ancestor of v, and therefore a proper ancestor of v'. This shows that $(x, y) \in B(v')$. Due to the generality of $(x, y) \in B(v)$, this implies that $B(v) \subseteq B(v')$. Thus we have $B(v') = B(v)$, in contradiction to the fact that the graph is 3-edge-connected. This shows that $M(v)$ is a proper ancestor of v'.

Let w and w' be two vertices such that $M(w) = M(v)$, $M(w') = M(v')$, $w \leq nextM(v)$ and $w' \leq nextM(v')$. Then, Lemma 3.6 implies that w is an ancestor of $high_1(v)$ and w' is an ancestor of $high_1(v')$. Thus, since $high_1(v) = high_1(v')$, we have that w and w' have a common descendant, and therefore they are related as ancestor and descendant.

Let us suppose, for the sake of contradiction, that w' is not a proper ancestor of w. Then, we have that w' is a descendant of w. Since $M(w) = M(v)$ is a proper ancestor of v', there is a back-edge (x, y) in $B(w)$ such that x is not a descendant of v'. Therefore, x is not a descendant of $M(v') = M(w')$. Since x is a descendant of $M(v)$, we have that x is a descendant of v, and therefore a descendant of $high_1(v) = high_1(v')$, and therefore a descendant of w'. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of w'. This shows that $(x, y) \in B(w')$. But this implies that x is a descendant of $M(w') = M(v')$, a contradiction. This shows that w' is a proper ancestor of w. Due to the generality of w' and w, this implies that $nextM(v')$ is a proper ancestor of $lastM(v)$. □

Lemma 5.125. *Let v and v' be two vertices $\neq r$ with $nextM(v) \neq \bot$ and $nextM(v') \neq \bot$,*

such that v' is a proper descendant of v, $high_1(v) = high_1(v')$, and both v and v' belong to the same segment S of $H(high_1(v))$ that is maximal w.r.t. the property that all its elements are related as ancestor and descendant (i.e., we have $S = S(v) = S(v')$). If $U_4^4(v') = \emptyset$, then $U_4^4(v) = \emptyset$. If $U_4^4(v) \neq \emptyset$, then the lowest vertex in $U_4^4(v)$ is at least as great as the greatest vertex in $U_4^4(v')$.

Proof. By Lemma 5.124, we have that *nextM*(v') is a proper ancestor of *lastM*(v).

Let us suppose, for the sake of contradiction, that $U_4^4(v') = \emptyset$ and $U_4^4(v) \neq \emptyset$. Let u be a vertex in $U_4^4(v)$. Let us suppose, for the sake of contradiction, that u is not a proper descendant of v'. Since $u \in U_4^4(v)$, we have that $u \in S$. Thus, since $v' \in S$, we have that u and v' are related as ancestor and descendant. Since u is not a proper descendant of v', we have that u is an ancestor of v'. Let (x, y) be a back-edge in $B(v')$ such that $y = low(v')$. Lemma 3.4 implies that $low(v')$ is a proper ancestor of *nextM*(v'). Thus, since *nextM*(v') is a proper ancestor of *lastM*(v), we have that $low(v')$ is a proper ancestor of *lastM*(v). Now, since $(x, y) \in B(v')$, we have that x is a descendant of v', and therefore a descendant of u. Furthermore, $y = low(v')$ is a proper ancestor of *lastM*(v), and therefore a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. But then we have $low(u) \leq y = low(v') < lastM(v)$, in contradiction to the fact that $u \in U_4^4(v)$. Thus, our last supposition is not true, and therefore u is a proper descendant of v'. Then, since $u \in U_4^4(v)$, we have $high(u) = high_1(v) = high_1(v')$. Furthermore, we have $low(u) \geq lastM(v)$, and therefore $low(u) \geq nextM(v')$. This implies that $U_4^4(v')$ is not empty (because we can consider the lowest proper descendant u' of v' in $S(v') = S(v)$ such that $high(u') = high_1(v')$ and $low(u') \geq nextM(v')$). This contradicts our supposition that $U_4^4(v') \neq \emptyset$. Thus, we have shown that $U_4^4(v') = \emptyset$ implies that $U_4^4(v) = \emptyset$.

Now let us assume that $U_4^4(v) \neq \emptyset$. This implies that $U_4^4(v')$ is not empty. Let us suppose, for the sake of contradiction, that there is a vertex $u \in U_4^4(v)$ that is lower than the greatest vertex u' in $U_4^4(v')$. Since $u \in U_4^4(v)$, we have $u \in S$. Since $u' \in U_4^4(v')$ we have $u' \in S$. This implies that u and u' are related as ancestor and descendant. Thus, since u is lower than u', we have that u is a proper ancestor of u'. Let us suppose, for the sake of contradiction, that $low(u')$ is a proper ancestor of *nextM*(v'). Then, since *nextM*(v') is a proper ancestor of *lastM*(v), we have that $low(u')$ is a proper ancestor of *lastM*(v). Now let (x, y) be a back-edge in $B(u')$ such that $y = low(u')$. Then x is a descendant of u', and therefore a descendant of u. Furthermore, y is a proper

ancestor of $lastM(v)$, and therefore a proper ancestor of v, and therefore a proper ancestor of u. This shows that $(x, y) \in B(u)$. Thus, we have $low(u) \leq y < lastM(v)$, in contradiction to the fact that $u \in U_4^4(v)$. Thus, our last supposition is not true, and therefore we have that $low(u')$ is not a proper ancestor of $nextM(v')$.

Since $u' \in U_4^4(v')$, we have that u' is a proper descendant of v', and therefore a proper descendant of $nextM(v')$. Thus, u' is a common descendant of $low(u')$ and $nextM(v')$, and therefore $low(u')$ and $nextM(v')$ are related as ancestor and descendant. Thus, since $low(u')$ is not a proper ancestor of $nextM(v')$, we have that $low(u')$ is a descendant of $nextM(v')$, and therefore $low(u') \geq nextM(v')$. Thus, since $u' \in U_4^4(v')$, we have that u' is the lowest proper descendant of v' in S with $high(u') = high_1(v')$ such that $low(u') \geq nextM(v')$ $(*)$.

Now we will trace the implications of $u \in U_4^4(v)$. First, we have $u \in S$. Then, we have $high(u) = high_1(v) = high_1(v')$. Furthermore, we have $low(u) \geq lastM(v)$, and therefore $low(u) > nextM(v')$ (since $nextM(v')$ is a proper ancestor of $lastM(v)$). Finally, we can show as above that u is a proper descendant of v' (the proof of this fact above did not rely on $U_4^4(v') = \emptyset$). But then, since u is lower than u', we have a contradiction to $(*)$. Thus, we have shown that every vertex in $U_4^4(v)$ is at least as great as the greatest vertex in $U_4^4(v')$. In particular, this implies that the lowest vertex in $U_4^4(v)$ is greater than, or equal to, the greatest vertex in $U_4^4(v')$. ☐

Due to the similarity of the definitions of the U_1 and the U_4^4 sets, and their properties described in Lemmata 5.75 and 5.125, respectively, we can compute all U_4^4 sets with a procedure similar to Algorithm 37. This is shown in Algorithm 52. Our result is summarized in Lemma 5.126.

Lemma 5.126. *Algorithm 52 correctly computes the sets $U_4^4(v)$, for all vertices $v \neq r$ such that $nextM(v) \neq \bot$. Furthermore, it has a linear-time implementation.*

Proof. The argument is almost identical to that provided for Lemma 5.76 in the main text (in the three paragraphs above Algorithm 37). The only difference is that here we care about the *low* point of the vertices u (and not for their low_2 point). This does not affect the analysis of correctness, because the computation is performed in segments of $H(x)$ that are maximal w.r.t. the property that their elements are related as ancestor and descendant, and therefore all vertices in those segments are sorted in decreasing order w.r.t. their *low* point. ☐

Algorithm 52: Compute the sets $U_4^4(v)$, for all vertices $v \neq r$ such that $\mathit{nextM}(v) \neq \bot$

```
1  foreach vertex x do
2      compute the collection 𝒮(x) of the segments of H(x) that are maximal
         w.r.t. the property that their elements are related as ancestor and
         descendant
3  end
4  foreach v ≠ r such that nextM(v) ≠ ⊥ do
5      set U_4^4(v) ← ∅
6  end
7  foreach vertex x do
8      foreach segment S ∈ 𝒮(x) do
9          let v be the first vertex in S
10         while v ≠ ⊥ and nextM(v) = ⊥ do
11             v ← next_S(v)
12         end
13         if v = ⊥ then continue
14         let u = prev_S(v)
15         while v ≠ ⊥ do
16             while u ≠ ⊥ and low(u) < lastM(v) do
17                 u ← prev_S(u)
18             end
19             while u ≠ ⊥ and low(u) < nextM(v) do
20                 insert u into U_4^4(v)
21                 u ← prev_S(u)
22             end
23             if u ≠ ⊥ then
24                 insert u into U_4^4(v)
25             end
26             v ← next_S(v)
27             while v ≠ ⊥ and nextM(v) = ⊥ do
28                 v ← next_S(v)
29             end
30         end
31     end
32 end
```

Lemma 5.127. *Let (u, v, w) be a triple of vertices that induces a Type-3βii-4 4-cut, such that $high_1(v) = high(u)$ and $M(B(v) \setminus \{e\}) = M(v)$, where e is the back-edge in the 4-cut induced by (u, v, w). Suppose that the lower endpoint of e is distinct from $high_1(v)$. Then $U_4^4(v) \neq \emptyset$, and let $\tilde{u}$ be the greatest vertex in $U_4^4(v)$. If $u \notin U_4^4(v)$, then u is the predecessor of $\tilde{u}$ in $H(high_1(v))$.*

Proof. Since $high_1(v) = high(u)$ and u is a proper descendant of v, we may consider the segment S of $H(high_1(v))$ from u to v. Let u' be a vertex in S. Due to the sorting of $H(high_1(v))$, we have $u \geq u' \geq v$. Thus, since the lower endpoint of e is distinct from $high_1(v)$, Lemma 5.104 implies that u' is an ancestor of u. Thus, we have that all elements of S are related as ancestor and descendant (since all of them are ancestors of u), and therefore $S \subseteq S(v)$. Since (u, v, w) induces a Type-3βii-4 4-cut, we have $M(w) = M(B(v)\setminus\{e\})$. Therefore, since $M(B(v)\setminus\{e\}) = M(v)$, we have $M(w) = M(v)$. Since w is a proper ancestor of v, this implies that $nextM(v) \neq \perp$, and $w \leq nextM(v)$.

By Lemma 5.101 we have $w \leq low(u)$, and therefore $lastM(v) \leq low(u)$. Thus, if $low(u) < nextM(v)$, then u satisfies enough conditions to be in $U_4^4(v)$. Otherwise, we have $low(u) \geq nextM(v)$, and therefore $U_4^4(v)$ is not empty, because we can consider the lowest proper descendant u' of v in $S(v)$ such that $low(u') \geq nextM(v)$. So let $\tilde{u}$ be the greatest vertex in $U_4^4(v)$. Let us suppose, for the sake of contradiction, that $u \notin U_4^4(v)$, and u is not the predecessor of $\tilde{u}$ in $H(high_1(v))$.

Since u is a proper descendant of v in $S(v)$ such that $low(u) \geq nextM(v)$, we have that there is a proper descendant u' of v in $S(v)$ that is lower than u and has $low(u') \geq nextM(v)$ (because this is the only condition that prevents u from being in $U_4^4(v)$). Thus, we may consider the lowest vertex u' that has this property. Then, we have that $u' \in U_4^4(v)$, and every other vertex $u'' \in U_4^4(v)$ satisfies $low(u'') < nextM(v)$. Let us suppose, for the sake of contradiction, that u' is not the greatest vertex in $U_4^4(v)$. Then there is a vertex $u'' \in U_4^4(v)$ such that $u'' > u'$. Since $u' \in U_4^4(v)$ and $u'' \in U_4^4(v)$, we have $u' \in S(v)$ and $u'' \in S(v)$. Therefore, $u'' > u'$ implies that u'' is a proper descendant of u'. Furthermore, we have $high(u') = high_1(v) = high(u'')$. Thus, Lemma 3.3 implies that $B(u'') \subseteq B(u')$. This implies that $low(u') \leq low(u'')$. But we have $low(u') \geq nextM(v)$ and $low(u'') < nextM(v)$, a contradiction. This shows that u' is indeed the greatest vertex in $U_4^4(v)$, and therefore we have $u' = \tilde{u}$.

Let $\tilde{u}'$ be the predecessor of $\tilde{u}$ in $H(high_1(v))$. Then, since $\tilde{u}$ is lower than u, and u is neither $\tilde{u}$ nor $\tilde{u}'$, we have $u > \tilde{u}' > \tilde{u}$. Thus, since u and $\tilde{u}$ are in $S(v)$, we have that $\tilde{u}'$ is also in $S(v)$ (because $S(v)$ is a segment of $H(high_1(v))$). Thus, $\tilde{u}'$ is related as ancestor

and descendant with both u and $\tilde{u}$. Therefore, we have that u is a proper descendant of $\tilde{u}'$, and $\tilde{u}'$ is a proper descendant of $\tilde{u}$. Then, since $\mathit{high}(u) = \mathit{high}(\tilde{u}') = \mathit{high}(\tilde{u})$, Lemma 3.3 implies that $B(u) \subseteq B(\tilde{u}') \subseteq B(\tilde{u})$. Since the graph is 3-edge-connected, this can be strengthened to $B(u) \subset B(\tilde{u}') \subset B(\tilde{u})$.

Let (x, y) be a back-edge in $B(\tilde{u})$. Then we have that x is a descendant of $\tilde{u}$, and therefore a descendant of v. Since $\tilde{u} \in S(v)$, we have $\mathit{high}(\tilde{u}) = \mathit{high}_1(v)$. Thus, since $(x, y) \in B(\tilde{u})$, we have that y is an ancestor of $\mathit{high}(\tilde{u}) = \mathit{high}_1(v)$, and therefore y is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Since (u, v, w) induces a Type-3βii-4 4-cut, we have $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. This implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. The case $(x, y) \in B(w)$ is rejected, since $y \geq \mathit{low}(\tilde{u}) \geq \mathit{nextM}(v) \geq w$. Thus, we have that either $(x, y) \in B(u)$ or $(x, y) = e$. Due to the generality of $(x, y) \in B(\tilde{u})$, this implies that $B(\tilde{u}) \subseteq B(u) \sqcup \{e\}$. Thus, we have $B(u) \subset B(\tilde{u}') \subset B(\tilde{u}) \subseteq B(u) \sqcup \{e\}$. But this implies that $\mathit{bcount}(u) < \mathit{bcount}(\tilde{u}') < \mathit{bcount}(\tilde{u}) \leq \mathit{bcount}(u) + 1$, which is impossible (because those numbers are integers).

Thus, we conclude that either $u \in U_4^4(v)$, or u is the predecessor of $\tilde{u}$ in $H(\mathit{high}_1(v))$. □

Lemma 5.128. *Let (u, v, w) be a triple of vertices such that u is a proper descendant of v, v is a proper descendant of w, and $M(w) = M(v)$. Then there is a back-edge e such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ if and only if: (1) $\mathit{high}(u) < v$, (2) $\mathit{bcount}(v) = \mathit{bcount}(u) + \mathit{bcount}(w) + 1$, and (3) w is either the greatest or the second-greatest proper ancestor of v such that $M(w) = M(v)$ and $w \leq \mathit{low}(u)$.*

Proof. ($\Rightarrow$) $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that $B(u) \subseteq B(v)$. Let (x, y) be a back-edge in $B(u)$. Then $B(u) \subseteq B(v)$ implies that $(x, y) \in B(v)$, and therefore y is a proper ancestor of v, and therefore $y < v$. Due to the generality of $(x, y) \in B(u)$, this implies that $\mathit{high}(u) < v$. (2) is an immediate consequence of $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$.

Let us suppose, for the sake of contradiction, that $\mathit{low}(u) < w$. Let (x, y) be a back-edge in $B(u)$ such that $y = \mathit{low}(u)$. Then x is a descendant of u, and therefore a descendant of v, and therefore a descendant of w. Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of w and y, and therefore w and y are related as ancestor and descendant. Then, $y = \mathit{low}(u) < w$ implies that y is a proper ancestor of w. But this shows that $(x, y) \in B(w)$, in contradiction to $B(u) \cap B(w) = \emptyset$. This shows that $\mathit{low}(u) \geq w$.

Thus, it makes sense to consider the greatest proper ancestor w' of v such that

$M(w') = M(v)$ and $w' \leq \mathit{low}(u)$. If $w = w'$, then we are done. Otherwise, we can also consider the second-greatest proper ancestor w'' of v such that $M(w'') = M(v)$ and $w'' \leq \mathit{low}(u)$.

Let us suppose, for the sake of contradiction, that w is neither w' nor w''. Thus, we have $w < w'' < w'$. Then, since $M(w) = M(w'') = M(w')$, we have that w is a proper ancestor of w'', w'' is a proper ancestor of w', and Lemma 3.2 implies that $B(w) \subseteq B(w'') \subseteq B(w')$. Since the graph is 3-edge-connected, this can be strengthened to $B(w) \subset B(w'') \subset B(w')$. Notice that, since $w' \leq \mathit{low}(u)$, we have $B(u) \cap B(w') = \emptyset$ (because the lower endpoint of every back-edge in $B(u)$ is not low enough to be a proper ancestor of w').

Let (x, y) be a back-edge in $B(w')$. Then x is a descendant of $M(w')$, and therefore a descendant of $M(v)$, and therefore a descendant of v. Furthermore, y is a proper ancestor of w', and therefore y is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Then $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ implies that either $(x, y) \in B(u)$, or $(x, y) \in B(w)$, or $(x, y) = e$. The case $(x, y) \in B(u)$ is rejected, because $B(u) \cap B(w') = \emptyset$. Thus, we have that either $(x, y) \in B(w)$, or $(x, y) = e$. Due to the generality of $(x, y) \in B(w')$, this implies that $B(w') \subseteq B(w) \sqcup \{e\}$. Thus, we have $B(w) \subset B(w'') \subset B(w') \subseteq B(w) \sqcup \{e\}$. But this implies that $\mathit{bcount}(w) < \mathit{bcount}(w'') < \mathit{bcount}(w') \leq \mathit{bcount}(w) + 1$, which is impossible (because those number are integers).

Thus, we have that w is either the greatest or the second-greatest proper ancestor of v such that $M(w) = M(v)$ and $w \leq \mathit{low}(u)$.

($\Leftarrow$) Let (x, y) be a back-edge in $B(u)$. Then x is a descendant of u, and therefore a descendant of v. Furthermore, y is an ancestor of $\mathit{high}(u)$, and therefore $y \leq \mathit{high}(u)$, and therefore $y < v$ (due to (1)). Since (x, y) is a back-edge, we have that x is a descendant of y. Thus, x is a common descendant of v and y, and therefore v and y are related as ancestor and descendant. Then, $y < v$ implies that y is a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(u)$, this implies that $B(u) \subseteq B(v)$.

Let (x, y) be a back-edge in $B(w)$. Then x is a descendant of $M(w) = M(v)$. Furthermore, y is a proper ancestor of w, and therefore a proper ancestor of v. This shows that $(x, y) \in B(v)$. Due to the generality of $(x, y) \in B(w)$, this implies that $B(w) \subseteq B(v)$. Since $w \leq \mathit{low}(u)$, we infer that $B(u) \cap B(w) = \emptyset$ (because the lower endpoint of every back-edge in $B(u)$ is not low enough to be a proper ancestor of w).

Now, since $B(u) \subseteq B(v)$, $B(w) \subseteq B(v)$, $B(u) \cap B(w) = \emptyset$, and $\mathit{bcount}(v) = \mathit{bcount}(u) +$

$bcount(w)+1$, we infer that there is a back-edge e such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. $\square$

Proposition 5.32. *Algorithm 53 computes a collection of Type-3βii-4 4-cuts, which includes all Type-3βii-4 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where w is a proper ancestor of v, v is a proper ancestor of u, $M(B(v)\setminus\{e\}) = M(v)$, $high_1(v) = high(u)$, and the lower endpoint of e is distinct from $high_1(v)$. Furthermore, it has a linear-time implementation.*

Proof. Let $C = \{(u,p(u)),(v,p(v)),(w,p(w)),e\}$ be a Type-3βii-4 4-cut such that w is a proper ancestor of v, v is a proper ancestor of u, $M(B(v)\setminus\{e\}) = M(v)$, $high_1(v) = high(u)$, and the lower endpoint of e is distinct from $high_1(v)$. Lemma 5.67 implies that $e = e(u,v,w)$. Since C is a Type-3βii-4 4-cut, we have $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$ and $M(B(v)\setminus\{e\}) = M(w)$. Since $M(B(v)\setminus\{e\}) = M(v)$, this implies that $M(w) = M(v)$. Since the lower endpoint of e is distinct from $high_1(v)$, Lemma 5.127 implies that $U_4^4(v)$ is not empty, and either $u \in U_4^4(v)$, or u is the predecessor of $\tilde{u}$ in $H(high_1(v))$, where $\tilde{u}$ is the greatest vertex in $U_4^4(v)$. Thus, we have $u \in \widetilde{U}_4^4(v)$. By Lemma 5.128 we have that $bcount(v) = bcount(u) + bcount(w) + 1$, and w is either the greatest or the second-greatest proper ancestor of v such that $M(w) = M(v)$ and $w \leq low(u)$. Thus, if w is the greatest proper ancestor of v such that $M(w) = M(v)$ and $w \leq low(u)$, then C satisfies enough conditions to be marked in Line 20. Otherwise, let w' be the greatest proper ancestor of v such that $M(w') = M(v)$ and $w' \leq low(u)$. Consider the vertex $w'' = nextM(w')$. Then we have $w'' < w'$ and $M(w'') = M(w')$. Thus, w'' is a proper ancestor of w', and therefore w'' is a proper ancestor of v with $w'' < w' \leq low(u)$. Since w'' is the greatest vertex with $M(w'') = M(w')$ that is lower than w', this means that w'' is the second-greatest proper ancestor of v with $M(w'') = M(v)$ and $w'' \leq low(u)$. Thus, we have $w = w''$, and therefore C satisfies enough condition to be marked in Line 24.

Conversely, let $C = \{(u,p(u)),(v,p(v)),(w,p(w)),e(u,v,w)\}$ be a 4-element set that is marked in Line 20 or 24. In either case, we have that u is in $\widetilde{U}_4^4(v)$. This means that either $u \in U_4^4(v)$, or u is the predecessor of $\tilde{u}$ in $H(high_1(v))$, where $\tilde{u}$ is the greatest vertex in $U_4^4(v)$. Then, since $u \in H(high_1(v))$, we have $high(u) = high_1(v)$, and therefore $high(u) < v$. If $u \in U_4^4(v)$, then by definition we have that u is a proper descendant of v. Otherwise, since the condition in Line 11 is satisfied (during the processing of v), we have that u is a proper descendant of v.

Algorithm 53: Compute all Type-3βii-4 4-cuts of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where w is an ancestor of v, v is an ancestor of u, $M(B(v)\setminus\{e\})=M(v)$, $high_1(v)=high(u)$, and the lower endpoint of e is distinct from $high_1(v)$

```
1  foreach v ≠ r such that nextM(v) ≠ ⊥ do
2  |   compute U_4^4(v)
3  end
4  foreach v ≠ r such that nextM(v) ≠ ⊥ do
5  |   let Ũ_4^4(v) ← U_4^4(v)
6  end
7  foreach v ≠ r such that nextM(v) ≠ ⊥ do
8  |   if U_4^4(v) ≠ ∅ then
9  |   |   let ũ be the greatest vertex in U_4^4(v)
10 |   |   let u be the predecessor of ũ in H(high_1(v))
11 |   |   if u ≠ ⊥ and u is a proper descendant of v then
12 |   |   |   insert u into Ũ_4^4(v)
13 |   |   end
14 |   end
15 end
16 foreach v ≠ r such that nextM(v) ≠ ⊥ do
17 |   foreach u ∈ Ũ_4^4(v) do
18 |   |   let w be the greatest proper ancestor of v with M(w) = M(v) such that
   |   |     w ≤ low(u)
19 |   |   if bcount(v) = bcount(u) + bcount(w) + 1 then
20 |   |   |   mark {(u,p(u)),(v,p(v)),(w,p(w)),e(u,v,w)} as a 4-cut
21 |   |   end
22 |   |   w ← nextM(w)
23 |   |   if bcount(v) = bcount(u) + bcount(w) + 1 then
24 |   |   |   mark {(u,p(u)),(v,p(v)),(w,p(w)),e(u,v,w)} as a 4-cut
25 |   |   end
26 |   end
27 end
```

Let us suppose first that C is marked in Line 20. Then we have $\mathit{bcount}(v) = \mathit{bcount}(u)+\mathit{bcount}(w)+1$, and w is the greatest proper ancestor of v with $M(w) = M(v)$ and $w \leq \mathit{low}(u)$. Thus, all the conditions of Lemma 5.128 are satisfied, and so we have that there is a back-edge e such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. Then Lemma 5.67 implies that $e = e(u, v, w)$. Thus, Lemma 5.57 implies that C is a Type-3β 4-cut.

Now let us suppose that C is marked in Line 24. Let w' be the greatest proper ancestor of v with $M(w') = M(v)$ and $w' \leq \mathit{low}(u)$. Then we have $w = \mathit{nextM}(w')$. This means that w is the greatest vertex with $M(w) = M(w')$ and $w < w'$. This implies that w is a proper ancestor of w', and therefore a proper ancestor of v. Furthermore, we have $w < w' \leq \mathit{low}(u)$. This shows that w is the second-greatest proper ancestor of v such that $M(w) = M(v)$ and $w \leq \mathit{low}(u)$. Since we have met the condition in Line 23, we have $\mathit{bcount}(v) = \mathit{bcount}(u) + \mathit{bcount}(w) + 1$. Thus, all the conditions of Lemma 5.128 are satisfied, and so we have that there is a back-edge e such that $B(v) = (B(u) \sqcup B(w)) \sqcup \{e\}$. Then Lemma 5.67 implies that $e = e(u, v, w)$. Thus, Lemma 5.57 implies that C is a Type-3β 4-cut.

Now we will argue about the complexity of Algorithm 53. By Lemma 5.126 we have that the sets $U_4^4(v)$ can be computed in linear time in total, for all vertices $v \neq r$ such that $\mathit{nextM}(v) \neq \bot$. Thus, the **for** loop in Line 1 can be performed in linear time. In particular, we have that the total size of all U_4^4 sets is $O(n)$. Thus, the **for** loop in Line 7 takes $O(n)$ time. In order to compute the vertex w in Line 18, we use Algorithm 22. We have showed previously how to generate the appropriate queries that provide w (see e.g., the proof of Proposition 5.31). Since the number of all those queries is $O(n)$ (because it is bounded by the total size of all sets of the form $\widetilde{U}_4^4(v)$), by Lemma 5.27 we have that Algorithm 22 can answer all of them in $O(n)$ time. We conclude that Algorithm 53 runs in linear time. $\square$

According to Proposition 5.32, Algorithm 53 computes all Type-3βii-4 4-cuts of the form $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$, where u is a descendant of v, v is a descendant of w, $M(B(v) \setminus \{e\}) = M(v)$, $\mathit{high}_1(v) = \mathit{high}(u)$, and the lower endpoint of e is distinct from $\mathit{high}_1(v)$. It remains to show how to compute all such 4-cuts in the case where the lower endpoint of e is $\mathit{high}_1(v)$. For this case, we cannot use directly any of our techniques so far, because these rely on the fact that u and v belong to the same segment of $H(\mathit{high}_1(v))$ or $\widetilde{H}(\mathit{high}_2(v))$ that is maximal w.r.t. the property that its elements are related as ancestor and descendant. However, in this particular case,

this is not necessarily true. (See Figure 5.26 for an example.)

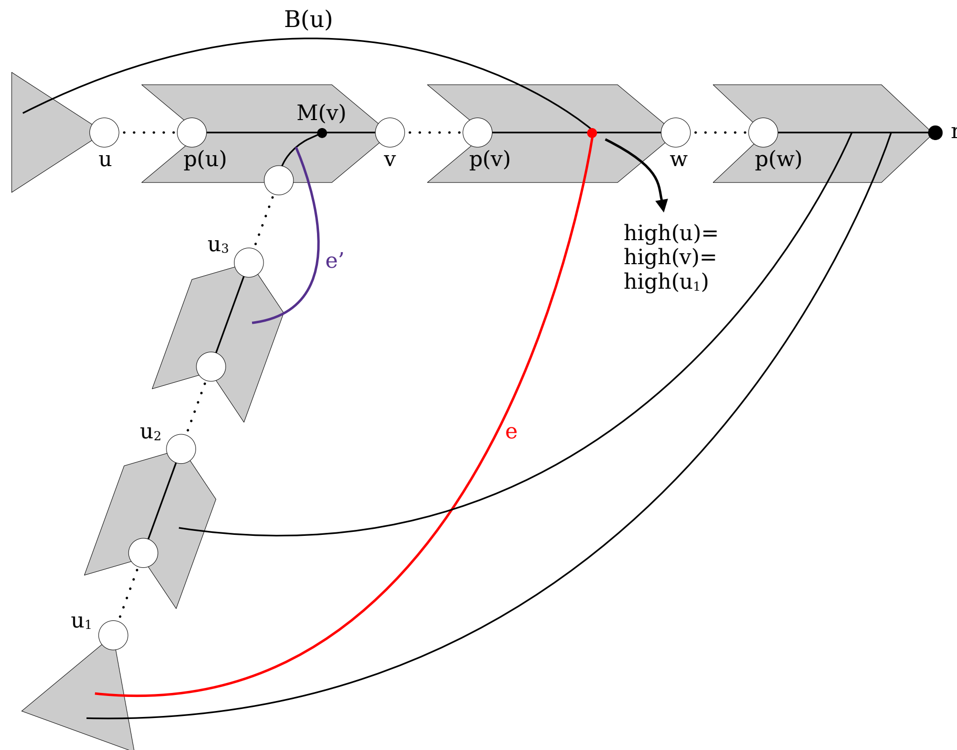


Figure 5.26: In this example we have that $\{(u, p(u)), (v, p(v)), (w, p(w)), e\}$ is a Type-3βii-4 4-cut, such that $M(v) = M(B(v) \setminus \{e\})$, $high_1(v) = high_1(u)$ and the lower endpoint of e is $high_1(v)$. Notice that $u \notin S(v)$, because u_1 and u_2 also have $high_1(u_1) = high_1(u_2) = high_1(v)$. (I.e., u does not belong to a segment of $H(high_1(v))$ that contains v and has the property that its elements are related as ancestor and descendant.) Also, it is not necessarily true that $high_2(u) = high_2(v)$, because we may have $high_2(u) < high_1(u)$, whereas $high_2(v) = high_1(v)$. However, even if we have $high_2(u) = high_2(v)$, then $u \notin \widetilde{S}(v)$, since $high_2(u_3) = high_2(v) = high_1(v)$. Thus, in this situation we cannot use the same techniques that we used so far in order to compute the Type-3βii 4-cuts.

Notice that, if such a 4-cut exists, then we have that there are two distinct back-edges with the same lower endpoint: i.e., the back-edge e, and one of the back-edges in $B(u)$ whose lower endpoint is $high(u) = high_1(v)$. Thus, if we had the property that no two back-edges have the same lower endpoint, then this case would not arise. We basically rely on this observation. Thus, we will perform the computation on a different – but "4-cut-equivalent" – graph, that has a DFS-tree in which no two back-edges that correspond to edges of the original graph can have the same lower endpoint. We construct this graph through repeated application of the following

vertex-splitting operation.

Definition 5.10 (Vertex Splitting). Let v be a vertex of G, and let (E_1, E_2) be an ordered bipartition of $\partial(v)$. Let G' be the graph that is formed from G by replacing v with two vertices v_1 and v_2, and by inserting five multiple edges of the form (v_1, v_2), one edge (v_1, z) for every z such that there is an edge $(v, z) \in E_1$, and one edge (v_2, z) for every z such that there is an edge $(v, z) \in E_2$. Then, G' is called *the graph that is derived from* G *by splitting* v *at* (E_1, E_2) *as* v_1 *and* v_2. (See Figure 5.27.)

We also define the corresponding mapping of edges $f : E(G) \to E(G')$ as follows. If (x, y) is an edge of G such that none of x, y is v, then $f((x, y)) = (x, y)$. Otherwise, if, say, $x = v$, then (x, y) belongs to one of the sets E_1 or E_2. If $(x, y) \in E_1$, then $f((x, y)) = (v_1, y)$. Otherwise, $f((x, y)) = (v_2, y)$.[6]

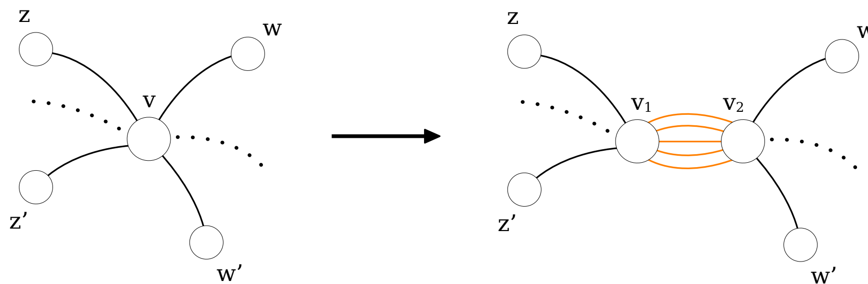


Figure 5.27: Splitting a vertex v at (E_1, E_2) as v_1 and v_2, where $E_1 = \{(v, z), \ldots, (v, z')\}$ and $E_2 = \{(v, w), \ldots, (v, w')\}$.

A graph G' that is derived from G by splitting a vertex maintains all 4-cuts, as shown in the following.

Lemma 5.129. *Let G' be the graph that is derived from G by splitting a vertex v at (E_1, E_2) as v_1 and v_2, and let $f : E(G) \to E(G')$ be the corresponding mapping of edges. Then, G' is 3-edge-connected. Furthermore, if C is a 4-cut of G, then $f(C)$ is a 4-cut of G'. Conversely, if C' is a 4-cut of G', then none of the edges in C' has the form (v_1, v_2), and $f^{-1}(C')$ is a 4-cut of G.*

[6]A more precise definition of f would require that it maintains the unique identifiers of the multiple edges. For the sake of simplicity, however, we omit this consideration from the definition and from the arguments that follow.

Proof. In order to prove this lemma, we establish a correspondence between paths in G and paths in G'. This basically works by replacing every part of a path in G that passes from v, with a part in G' that passes from v_1 or v_2. More precisely, we define the correspondence as follows.

Let $P = x_1, e_1, x_2, \ldots, e_{k-1}, x_k$ be a path in G. We perform the following substitutions.

- If $x_1 = v$, then e_1 has the form (v, z), and we have that either $e_1 \in E_1$ or $e_1 \in E_2$. If $e_1 \in E_1$, then we replace the part x_1, e_1 in P with $v_1, (v_1, z)$. Otherwise, if $e_1 \in E_2$, then we replace the part x_1, e_1 in P with $v_2, (v_2, z)$.
- If $x_k = v$, then e_{k-1} has the form (z, v), and we have that either $e_{k-1} \in E_1$ or $e_{k-1} \in E_2$. If $e_{k-1} \in E_1$, then we replace the part e_{k-1}, v in P with $(z, v_1), v_1$. Otherwise, if $e_{k-1} \in E_2$, then we replace the part e_{k-1}, v in P with $(z, v_2), v_2$.
- If there is a part e_i, v, e_{i+1} in P, for some $i \in \{1, \ldots, k-2\}$, then we have that the edge e_i has the form (z, v), the edge e_{i+1} has the form (v, z'), and there are four different cases to consider, depending on whether e_i and e_{i+1} are in E_1 or E_2. If $e_i \in E_1$ and $e_{i+1} \in E_1$, then we replace the part $(z, v), v, (v, z')$ with $(z, v_1), v_1, (v_1, z')$. If $e_i \in E_1$ and $e_{i+1} \in E_2$, then we replace the part $(z, v), v, (v, z')$ with $(z, v_1), v_1, (v_1, v_2), v_2, (v_2, z')$. If $e_i \in E_2$ and $e_{i+1} \in E_1$, then we replace the part $(z, v), v, (v, z')$ with $(z, v_2), v_2, (v_2, v_1), v_1, (v_1, z')$. And if $e_i \in E_2$ and $e_{i+1} \in E_2$, then we replace the part $(z, v), v, (v, z')$ with $(z, v_2), v_2, (v_2, z')$.

We denote the resulting sequence as P'. Observe the following facts.

1. P' is a path in G'.
2. Every occurrence of a vertex $x \neq v$ in P is maintained in P'.
3. Every occurrence of v in P, is substituted with either v_1, or v_2, or $v_1, (v_1, v_2), v_2$, or $v_2, (v_2, v_1), v_1$.
4. Every occurrence of an edge e in P, is substituted with $f(e)$.
5. Every part e, v, e' in P (where e and e' are edges), is substituted with either $f(e), v_1, f(e')$, or $f(e), v_2, f(e')$, or $f(e), v_1, (v_1, v_2), v_2, f(e')$, or $f(e), v_2, (v_2, v_1), v_1, f(e')$.

Conversely, let $Q = x_1, e_1, x_2, \ldots, e_{k-1}, x_k$ be a path in G'. We perform the following substitutions.

- If x_i is v_1 or v_2, and x_{i+1} is not v_1 or v_2, for some $i \in \{1, \ldots, k-2\}$, then x_i, e_i is replaced with $v, (v, x_{i+1})$.
- If x_i is v_1 or v_2, and x_{i-1} is not v_1 or v_2, for some $i \in \{2, \ldots, k-1\}$, then e_{i-1}, x_i is replaced with $(x_{i-1}, v), v$.
- Every maximal segment of the form $v_i, (v_i, v_j), v_j$, for $i, j \in \{1, 2\}$, is replaced with v.

We denote the resulting sequence as $\widetilde{Q}$. (Notice that after performing simultaneously the above substitutions, there may appear some segments of the form $v, v, v, \ldots$. We replace those maximal segments with v, so that we indeed have a path. For example, Q may contain the segment $v_1, (v_1, v_2), v_2, (v_2, z)$, where $z \notin \{v_1, v_2\}$. Then we replace this segment with $v, (v, z)$ in $\widetilde{Q}$.) Observe the following facts.

1. $\widetilde{Q}$ is a path in G.
2. Every occurrence of a vertex $x \notin \{v_1, v_2\}$ in Q is maintained in $\widetilde{Q}$.
3. Every occurrence of v_1 or v_2 in Q, is substituted with v.
4. Every occurrence of an edge $e \neq (v_1, v_2)$ in Q, is substituted with $f^{-1}(e)$.

Now it is easy to see why G' is connected. Let x and y be two distinct vertices in G', none of which is either v_1 or v_2. Then, since G is connected, there is a path P in G from x to y. Then, P' is a path in G' from x to y, and so x and y are connected in G'. The existence of the edges of the form (v_1, v_2) in G' shows that v_1 and v_2 are connected in G'. Finally, since $\{E_1, E_2\}$ is a bipartition of $\partial(v)$, we have that both E_1 and E_2 are non-empty. So let (v, z) be an edge in E_1. Then, there is an edge of the form (v_1, z) in G', and so v_1 is connected with the vertices in $G' \setminus \{v_1, v_2\}$. Furthermore, let (v, z) be an edge in E_2. Then, there is an edge of the form (v_2, z) in G', and so v_2 is connected with the vertices in $G' \setminus \{v_1, v_2\}$. This shows that G' is connected.

Now let us suppose, for the sake of contradiction, that G' is not 3-edge-connected. This means that there is a k-edge cut C of G', for some $k \leq 2$. This implies that the endpoints of any edge in C are not connected in $G' \setminus C$. Thus, we have that no edge of the form (v_1, v_2) is contained in C (because there are five edges of this form, and

so all of them must be removed in order to disconnect v_1 from v_2). Thus, the set of edges $f^{-1}(C)$ is defined. Then, since G is 3-edge-connected, we have that $G \setminus f^{-1}(C)$ is connected (since $|f^{-1}(C)| = |C| \leq 2$). Since C is a k-edge cut of G', we have that $G' \setminus C$ consists of two connected components S_1 and S_2. Let x be a vertex in S_1 and let y be a vertex in S_2. If $x \in \{v_1, v_2\}$ then we let x' denote v; otherwise, if $x \notin \{v_1, v_2\}$, then we let x' denote x. Similarly, if $y \in \{v_1, v_2\}$, then we let y' denote v; otherwise, if $y \notin \{v_1, v_2\}$, then we let y' denote y. Then, since $G \setminus f^{-1}(C)$ is connected, we have that there is a path P in $G \setminus f^{-1}(C)$ from x' to y'. Then, the path P' in G' avoids the edges in C, and demonstrates that x and y are connected in $G' \setminus C$. (To see this, distinguish the following cases. If none of x and y is in $\{v_1, v_2\}$, then P' is a path from x to y, and the contradiction is clear. If, say, x is v_1, then P' is a path from either v_1 or v_2; but this distinction has no effect, since the existence of the edges of the form (v_1, v_2) implies that v_1 is connected with v_2 in $G' \setminus C$. The same holds if x is v_2, or if y is either v_1 or v_2.) Thus, we have arrived at a contradiction. This shows that G' is 3-edge-connected.

Now let C be a 4-cut of G. This implies that $G \setminus C$ is split into two connected components S_1 and S_2, but $G \setminus C'$ is connected for every proper subset C' of C. Let x be a vertex in S_1, and let y be a vertex in S_2. Then, there is no path from x to y in $G \setminus C$. Let us suppose, for the sake of contradiction, that x' and y' are connected in $G' \setminus f(C)$, where we let x' denote v_1 if $x = v$, or x if $x \neq v$; and similarly, we let y' denote v_1 if $y = v$, or y if $y \neq v$. Then there is a path Q in $G' \setminus f(C)$ from x' to y'. Consider the path $\widetilde{Q}$ in G. First, observe that $\widetilde{Q}$ is a path in $G \setminus C$ (since Q is a path in $G' \setminus f(C)$). Furthermore, notice that $\widetilde{Q}$ is a path from x to y. But this is impossible, since x and y are not connected in $G \setminus C$. Thus, we have that $G' \setminus f(C)$ is disconnected. Now let C' be a proper subset of C. Let us suppose, for the sake of contradiction, that $G' \setminus f(C')$ is disconnected. Then, let x and y be two vertices that are not connected in $G' \setminus f(C')$. Notice that it cannot be that both x and y are in $\{v_1, v_2\}$, because there are five edges of the form (v_1, v_2), whereas $|f(C')| = |C'| < 4$. If none of x and y is either v_1 or v_2, then there is a path P from x to y in $G \setminus C'$, and therefore there is a path P' from x to y in $G' \setminus f(C')$, which is impossible. Thus, one of x and y is either v_1 or v_2. Let us assume w.l.o.g. that $x = v_1$. Then we have that $y \notin \{v_1, v_2\}$, and so there is a path P from v to y in $G \setminus C'$, and therefore P' is a path from either v_1 or v_2 to y in $G' \setminus f(C')$, which is also impossible (since v_1 is connected with v_2 in $G' \setminus f(C')$, but v_1 is not connected with y in $G' \setminus f(C')$). Thus, we have that

$G' \setminus f(C')$ is also connected. Since this is true for every proper subset C' of C, this shows that $f(C)$ is a 4-cut of G'.

Conversely, let C be a 4-cut of G'. This implies that the endpoints of every edge in C are disconnected in $G' \setminus C$. Thus, since $|C| = 4$, we have that C contains no edge of the form (v_1, v_2) (since there are five of them in G'), and v_1 is connected with v_2 in $G' \setminus C$. Thus, $f^{-1}(C)$ is a set of four edges of G. Let us suppose, for the sake of contradiction, that $G \setminus f^{-1}(C)$ is connected. Since C is a 4-cut of G', there are two vertices x and y that are disconnected in $G' \setminus C$. Let x' denote v if $x \in \{v_1, v_2\}$, and x if $x \notin \{v_1, v_2\}$. Similarly, let y' denote v if $y \in \{v_1, v_2\}$, and y if $y \notin \{v_1, v_2\}$. Then, since $G \setminus f^{-1}(C)$ is connected, there is a path P from x' to y' in $G \setminus f^{-1}(C)$. Then, observe that P' is a path in G', that avoids the edges in C. Furthermore, if $x' = x$ and $y' = y$, then P' is a path from x to y, which is impossible, since x and y are disconnected in $G' \setminus C$. Otherwise, let us assume w.l.o.g. that $x' = v$. Then, P' is a path from either v_1 or v_2 to y, which demonstrates that x is connected with y in $G' \setminus C$ (since v_1 is connected with v_2 in $G' \setminus C$). This is impossible. Thus, we have that $G \setminus f^{-1}(C)$ is disconnected. Now, since C is a 4-cut of G', we have that $G' \setminus C'$ is connected, for every proper subset C' of C. Let us suppose, for the sake of contradiction, that $G \setminus f^{-1}(C')$ is disconnected, for a proper subset C' of C. Then, there are two vertices x and y of G that are disconnected in $G \setminus f^{-1}(C')$. If $x = v$, let x' denote v_1; otherwise, let x' denote x. Similarly, if $y = v$, let y' denote v_1; otherwise, let y' denote y. Then, since $G' \setminus C'$ is connected, we have that there is a path Q from x' to y' in $G' \setminus C'$. Then, observe that $\widetilde{Q}$ is a path from x to y in $G \setminus f^{-1}(C')$, in contradiction to our supposition. Thus, we have that $G \setminus f^{-1}(C')$ is connected. Since this is true for every proper subset C' of C, this shows that $f^{-1}(C)$ is a 4-cut of G. □

Now the idea is to repeatedly split vertices on T, so that the resulting DFS-tree has the property that no two back-edges that correspond to back-edges of the original graph can have the same lower endpoint. So let v be a vertex, and let $(x_1, v), \dots, (x_k, v)$ be all the incoming back-edges to v. We may assume that $v \neq r$ (because this is sufficient for our purposes). If $k = 1$, then there is nothing to do, because there is only one back-edge whose lower endpoint is v. Otherwise, let $(c_1, v), \dots, (c_t, v)$ be the parent edges of the children of v (if it has any), and let $(v, y_1), \dots, (v, y_l)$ be the back-edges that stem from v. Then, we have $\partial(v) = \{(x_1, v), \dots, (x_k, v)\} \cup \{(c_1, v), \dots, (c_t, v)\} \cup \{(v, y_1), \dots, (v, y_l)\} \cup \{(v, p(v))\}$. Let $E_1 = \{(x_1, v)\} \cup \{(c_1, v), \dots, (c_t, v)\}$, and let $E_2 =$

$\partial(v)\setminus E_1$. Then, we split v at $P = (E_1, E_2)$ as v_1 and v_2, while maintaining the DFS-tree structure (see Figure 5.28). To be specific, we detach v from T, and we introduce the vertices v_1 and v_2 that replace v, such that $v_2 = p(v_1)$. Then, v_1 inherits the children of v and the back-edge (x_1, v) (as (x_1, v_1)), and v_2 inherits the remaining edges from $\partial(v)$. Thus, we introduce the parent edges $(c_1, v_1), \dots, (c_t, v_1)$, four back-edges of the form (v_1, v_2) (because we already have (v_1, v_2) as a parent edge), we set $p(v_2) \leftarrow p(v)$ (where $p(v)$ was the parent of v before its deletion), and we also put back the remaining back-edges from $\partial(v)$ as $(x_2, v_2), \dots, (x_k, v_2)$ and $(v_2, y_1), \dots, (v_2, y_l)$. We refer to Figure 5.28 for a depiction of this process.

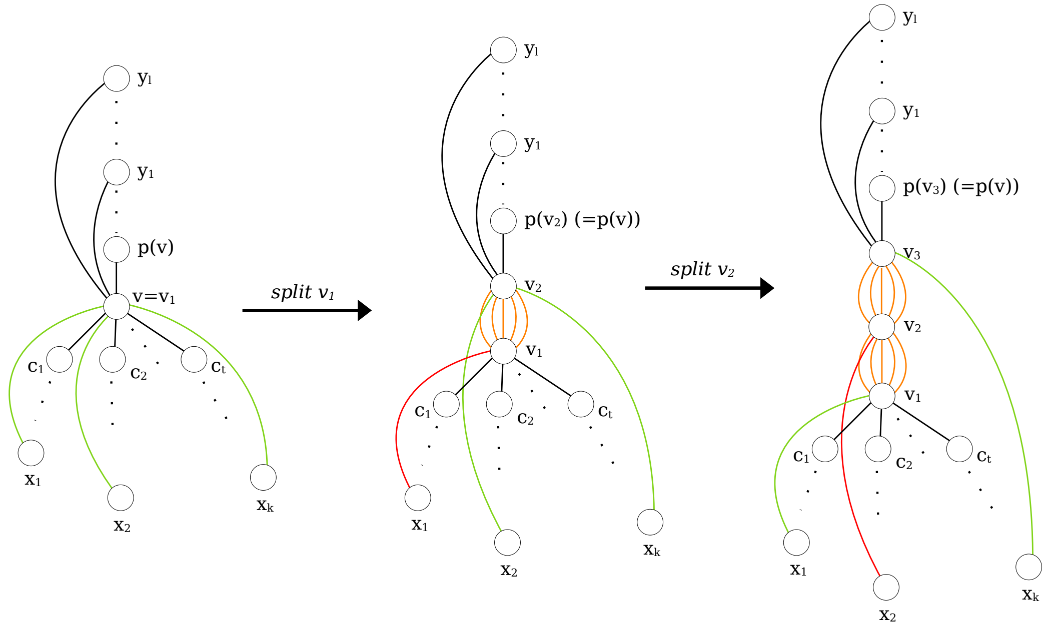


Figure 5.28: Splitting a vertex v on a DFS-tree, so that the number of back-edges with lower endpoint v is reduced by one. With green are shown the target back-edges (incoming to v) that we want to separate w.r.t. their lower endpoint. With red is shown the back-edge that was separated from the rest and is now unique with the property of having the lower endpoint that it has. With orange are shown the five multiple edges that join the two vertices into which the vertex was split. We note that one of those multiple edges is a tree-edge, and the rest are back-edges.

In this way, we have achieved the following things. First, we have maintained a DFS-tree for the graph that is derived from G by splitting v at P, and second, we have effectively reduced the number of back-edges with lower endpoint v by one (i.e., now all of these, except one, have v_2 as their lower endpoint). Now, if $k > 2$, then we

continue this process, by splitting v_2, until eventually we have separated the back-edges with lower endpoint v into back-edges with different lower endpoints. Then, we continue this process for all vertices $\neq r$. Let G' be the resulting graph, and let T' be the corresponding DFS-tree. Notice that, by Lemma 5.129, all 4-cuts of G that contain at least one back-edge (w.r.t. T) whose lower endpoint is not r, correspond to 4-cuts of G' that also have at least one back-edge (w.r.t. T'). Thus, we can compute all Type-3βii-4 4-cuts of G of the form $\{(u,p(u)),(v,p(v)),(w,p(w)),e\}$, where u is a descendant of v, v is a descendant of w, $M(B(v)\setminus\{e\}) = M(v)$, $high_1(v) = high(u)$, and the lower endpoint of e is $high_1(v)$, by simply computing all 4-cuts of G' that contain three tree-edges and one back-edge, using the algorithms that we have developed so far (since our analysis covers all cases, except this one, which may only arise due to the existence of at least two back-edges with the same lower endpoint).

It remains to show how we can efficiently perform all those splittings in linear time in total. This can be done easily, because it is essentially sufficient to determine the final vertices and the edges of G', and the parent relation of the vertices of G'. Then we may run a DFS with start vertex r, in order to assign a DFS numbering, determine the back-edges, and compute the DFS parameters that we need.

Let us describe in detail how to construct the vertices and the edges of the final graph G'. First, for every vertex v, let $in(v)$ denote the number of incoming back-edges to v. Then, for every vertex $v \neq r$ such that $in(v) > 1$, we will perform $in(v) - 1$ splittings of v. This will substitute v with $in(v)$ copies of it, which we denote as $v_1, v_2, \dots, v_{in(v)}$. Every one of the vertices $v_1, \dots, v_{in(v)}$ will be used in order to inherit one of the incoming back-edges to v. For every $i \in \{1, \dots, in(v) - 1\}$, we create five multiple edges of the form (v_i, v_{i+1}); one of these edges will be the parent edge of v_i. Among the $in(v)$ copies of v, we let v_1 inherit the children edges of v, and we let $v_{in(v)}$ inherit the outgoing back-edges from v and the parent edge $(v, p(v))$ (if $v \neq r$). Thus, let c be a child of v. Then the parent edge of c is replaced with $(c_{in(c)}, v_1)$ if $in(c) > 1$, or with (c, v_1) otherwise. Now let $e_1, \dots, e_{in(v)}$ be the list of the incoming back-edges to v. Let i be an index in $\{1, \dots, in(v)\}$, and let $e_i = (x, v)$. Then e_i is replaced with $(x_{in(x)}, v_i)$ if $in(x) > 1$, or with (x, v_i) otherwise. It is not difficult to see that this construction can be completed in linear time.

Chapter 6

Connectivity Queries under 4 Edge Failures

6.1 Introduction

6.2 E' contains zero tree-edges

6.3 E' contains one tree-edge

6.4 E' contains two tree-edges

6.5 E' contains three tree-edges

6.6 E' contains four tree-edges

6.7 The data structure

6.1 Introduction

Our goal in this chapter is to prove the following.

Proposition 6.1. *Let G be a connected graph with n vertices and m edges. Then there is a data structure with size $O(n)$, that we can use in order to answer connectivity queries in the presence of at most four edge-failures in constant time. Specifically, given an edge-set E' with $|E'| \leq 4$, and two vertices x and y, we can determine whether x and y are connected in $G \setminus E'$ in $O(1)$ time. The data structure can be constructed in $O(m+n)$ time.*

Let G be a connected graph, and let T be a rooted spanning tree of G. Let E' be a set of edges of G, and let k be the number of tree-edges in E'. Then $T \setminus E'$ is split

into $k+1$ connected components. Every connected component of $T \setminus E'$ is a subtree of T, and the connectivity in $G \setminus E'$ can be reduced to the connectivity of those subtrees as follows. Let x and y be two vertices of G, let C_1 be the connected component of $T \setminus E'$ that contains x, and let C_2 be the connected component of $T \setminus E'$ that contains y. Then we have that x and y are connected in $G \setminus E'$ if and only if C_1 and C_2 are connected in $G \setminus E'$. In particular, since C_1 and C_2 are subtrees of the rooted tree T, we can consider the roots r_1 and r_2 of C_1 and C_2 as representatives of C_1 and C_2, respectively. Then, we have that x and y are connected in $G \setminus E'$ if and only if r_1 and r_2 are connected in $G \setminus E'$. Thus, the connectivity relation of $G \setminus E'$ can be captured by the connectivity relation of the roots of the connected components of $T \setminus E'$ in $G \setminus E'$. In order to capture this connectivity relation, we introduce the concept of a *connectivity graph* for $G \setminus E'$.

Definition 6.1. Let T be a fixed rooted spanning tree of G, and let E' be a set of edges of G. Let $\{C_1, \dots, C_k\}$ be the set of the connected components of $T \setminus E'$. Thus, for every $i \in \{1, \dots, k\}$, C_i is a rooted subtree of T, and let r_i be its root. Then, a connectivity graph for $G \setminus E'$ is a graph $\mathcal{R}$ with vertex set $\{\bar{r_i} \mid i \in \{1, \dots, k\}\}$ such that: for every $i, j \in \{1, \dots, k\}$, r_i is connected with r_j in $G \setminus E'$ if and only if $\bar{r_i}$ is connected with $\bar{r_j}$ in $\mathcal{R}$.

We allow a connectivity graph $\mathcal{R}$ for $G \setminus E'$ to be a multigraph. (The important thing is that it captures the connectivity relation of the roots of the connected components of $T \setminus E'$.) In particular, if we shrink every connected component of $T \setminus E'$ with root z into a node $\bar{z}$, then the quotient graph of $G \setminus E'$ that we get is a connectivity graph for $G \setminus E'$. The main challenge is to compute a connectivity graph for $G \setminus E'$ without explicitly computing the connected components of $T \setminus E'$. We can achieve this if we assume that T is a DFS-tree of the graph. Then, with a creative use of the DFS-concepts that we defined in Section 3, we can determine enough edges of $G \setminus E'$ between the connected components of $T \setminus E'$, so that we can construct a connectivity graph for $G \setminus E'$ in constant time, if $|E'| \leq 4$.

We distinguish five different cases, depending on the number of tree-edges contained in E'. Then, for each case, we consider all the different possibilities for the edges from E' on T (i.e., all the different topologies of their endpoints w.r.t. the ancestry relation). In order to reduce the number of cases considered, we will use the following three facts (which we make precise and prove in the following paragraphs).

First, if we have established that the endpoints of a tree-edge e in E' remain connected in $G \setminus E'$, then it is sufficient to set $E' \leftarrow E' \setminus \{e\}$, and then revert to the previous case, where the number of tree-edges is that of E' minus 1 (see Lemma 6.2). Second, if there are at least two tree-edges in E' that are not descendants of other tree-edges in E', then we can handle the subtrees induced by those tree-edges separately, by reverting to previous cases (see Lemma 6.3). And third, if for a tree-edge $(u, p(u))$ in E' we have that $G \setminus E'$ contains no back-edges that leap over u, then we can handle the subtree induced by this tree-edge separately from the rest of the tree, by reverting to previous cases (see Lemma 6.4).

In the following, we assume that T is a fixed DFS-tree of G with root r. All connectivity graphs refer to this tree. First, we will need the following technical lemma.

Lemma 6.1. *Let E' be a set of edges, and let E'' be a subset of E' that contains all the back-edges from E', and has the property that no tree-edge from $E' \setminus E''$ is related as ancestor and descendant with a tree-edge from E''. Let U be the collection of the higher endpoints of the tree-edges in E'', and let z and z' be two vertices in $U \cup \{r\}$. Suppose that z and z' are connected in $G \setminus E''$. Then z and z' are connected in $G \setminus E'$.*

Proof. Since z and z' are connected in $G \setminus E''$, there is a path P from z to z' in $G \setminus E''$. If P does not use any tree-edge from $E' \setminus E''$, then we have that P is a path in $G \setminus E'$, and therefore z and z' are connected in $G \setminus E'$. So let us assume that P uses a tree-edge from $E' \setminus E''$. Let C be the connected component of $T \setminus E'$ that contains r. Then, we claim that P has the form $P_1 + Q + P_2$, where P_1 is path from z to a vertex $w \in C$ that does not use any tree-edge from $E' \setminus E''$, Q is a path from w to a vertex $w' \in C$ (that uses tree-edges from $E' \setminus E''$), and P_2 is a path from w' to z' that does not use any tree-edge from $E' \setminus E''$ $(*)$. This implies that P_1 is a path from z to w in $G \setminus E'$, and P_2 is a path from w' to z' in $G \setminus E'$. Then, since w and w' lie in the same connected component of $T \setminus E'$, we have that there is a path Q' from w to w' in $G \setminus E'$. Thus, $P_1 + Q' + P_2$ is a path from z to z' in $G \setminus E'$, and therefore we have that z and z' are connected in $G \setminus E'$.

Now we will prove $(*)$. Let U' be the collection of the higher endpoints of the tree-edges in $E' \setminus E''$. Then, we have that no vertex from U is related as ancestor and descendant with a vertex from U'. Now let $(v, p(v))$ be the first occurrence of a tree-edge from $E' \setminus E''$ that is used by P, and let $(v', p(v'))$ be the last occurrence of

a tree-edge from $E' \setminus E''$ that is used by P. Then, since P starts from z and visits v, Lemma 3.18 implies that P contains a subpath from an ancestor w of $nca\{z,v\}$ to v, and let w be the first vertex visited by P with this property. Let P_1 be the initial part of P from z to the first occurrence of w. Then, we have that P_1 does not use any tree-edge from $E' \setminus E''$. Notice that there is no $u \in U$ such that u is an ancestor of w, because otherwise we would have that u is an ancestor of v. Also, there is no $u' \in U'$ such that u' is an ancestor of w, because otherwise u' would be an ancestor of z. Thus, there is no tree-edge from E' on the tree-path $T[r,w]$. Similarly, since P visits v' and ends in z', Lemma 3.18 implies that P contains a subpath from v' to an ancestor w' of $nca\{v',z'\}$, and let w' be the last vertex visited by P with this property. Let P_2 be the final part of P from the last occurrence of w' to z'. Then, we have that P_2 does not use any tree-edge from $E' \setminus E''$. Again, we can see that there is no tree-edge from E' on the tree-path $T[r,w']$. Now we can consider the part Q of P from the first occurrence of w to the last occurrence of w', and the proof is complete, because $P = P_1 + Q + P_2$, and the endpoints of Q lie in the connected component of $T \setminus E'$ that contains r. □

Lemma 6.2. *Let E' be a set of edges, and let $(u,p(u))$ and $(v,p(v))$ be two distinct tree-edges in E'. Suppose that u, $p(u)$ and v are connected in $G \setminus E'$. Let $E'' = E' \setminus \{(u,p(u))\}$, and let $\mathcal{R}'$ be a connectivity graph for $G \setminus E''$. Then, $\mathcal{R}' \cup \{(\bar{u},\bar{v})\}$ is a connectivity graph for $G \setminus E'$.*

Proof. Let $(u_1,p(u_1)),\dots,(u_k,p(u_k))$ be the tree-edges in E', where $u_1 = u$. Let $E'' = E' \setminus \{(u,p(u))\}$, and let $\mathcal{R}'$ be a connectivity graph for $G \setminus E''$. Then we have that $V(\mathcal{R}') = \{\bar{u}_2,\dots,\bar{u}_k,\bar{r}\}$. Let $\mathcal{R}$ be the graph with $V(\mathcal{R}) = V(\mathcal{R}') \cup \{\bar{u}\}$ and $E(\mathcal{R}) = E(\mathcal{R}') \cup \{(\bar{u},\bar{v})\}$. We will show that $\mathcal{R}$ is a connectivity graph for $G \setminus E'$.

Let z and z' be two vertices in $\{u_2,\dots,u_k,r\}$. First, suppose that z and z' are connected in $G \setminus E'$. Then, since $E'' \subset E'$, we have that z and z' are connected in $G \setminus E''$. Thus, $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}'$, and therefore they are connected in $\mathcal{R}$. Conversely, suppose that $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}$. Then, it is easy to see that $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}'$. This implies that z and z' are connected in $G \setminus E''$. Thus, there is a path P from z to z' in $G \setminus E''$. If P does not use the edge $(u,p(u))$, then P is a path in $G \setminus E'$, and therefore z and z' are connected in $G \setminus E'$. Otherwise, since u and $p(u)$ are connected in $G \setminus E'$, there is a path Q from u to $p(u)$ in $G \setminus E'$ that avoids the edge $(u,p(u))$. Now replace every occurrence of $(u,p(u))$ in P with Q, and

let P' be the resulting path. Then, we have that P' is a path from z to z' in $G \setminus E'$. This shows that z and z' are connected in $G \setminus E'$.

Now let z be a vertex in $\{u_2, \dots, u_k, r\}$. First, suppose that u and z are connected in $G\setminus E'$. Then, since $E'' \subset E'$, we have that u and z are connected in $G\setminus E''$. Furthermore, since u and v are connected in $G \setminus E'$, we have that u and v are connected in $G \setminus E''$. Thus, v and z are connected in $G \setminus E''$, and therefore $\bar{v}$ and $\bar{z}$ are connected in $\mathcal{R}'$. Thus, the existence of the edge $(\bar{u}, \bar{v})$ in $\mathcal{R}$ implies that $\bar{u}$ and $\bar{z}$ are connected in $\mathcal{R}$. Conversely, suppose that $\bar{u}$ and $\bar{z}$ are connected in $\mathcal{R}$. Then, since $(\bar{u}, \bar{v})$ is the only edge of $\mathcal{R}$ that is incident to $\bar{u}$, we have that $\bar{v}$ is connected with $\bar{z}$ in $\mathcal{R}$ through a path that avoids $\bar{u}$. Therefore, $\bar{v}$ is connected with $\bar{z}$ in $\mathcal{R}'$. Thus, we have that v and z are connected in $G \setminus E''$. So let P be a path from v to z in $G \setminus E''$. Since u and $p(u)$ are connected in $G \setminus E'$, we have that there is a path Q from u to $p(u)$ in $G \setminus E'$. Now, if P uses the edge $(u, p(u))$, then we replace every occurrence of $(u, p(u))$ in P with the path Q. Let P' be the resulting path. Then, P' is a path from v to z in $G \setminus E'$, and therefore v is connected with z in $G \setminus E'$. Therefore, since u is connected with v in $G \setminus E'$, we have that u is connected with z in $G \setminus E'$.

Thus, we have shown that, for every pair of vertices z and z' in $\{u_1, \dots, u_k, r\}$, we have that z and z' are connected in $G \setminus E'$ if and only if $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}$. This means that $\mathcal{R}$ is a connectivity graph for $G \setminus E'$. □

Lemma 6.3. *Let E' be a set of edges, and let $(u, p(u))$ be a tree-edge in E' with the property that no tree-edge in E' is a proper ancestor of $(u, p(u))$. Let E_1 be the set of the tree-edges in E' that are descendants of $(u, p(u))$, plus the back-edges in E', and let E_2 be the set of the tree-edges in E' that are not descendants of $(u, p(u))$, plus the back-edges in E'. Let $\mathcal{R}_1$ be a connectivity graph for $G \setminus E_1$, and let $\mathcal{R}_2$ be a connectivity graph for $G \setminus E_2$. Then, $\mathcal{R}_1 \cup \mathcal{R}_2$ is a connectivity graph for $G \setminus E'$.*

Proof. Let $(u_1, p(u_1)), \dots, (u_t, p(u_t))$ be the tree-edges in E_1, and let $(v_1, p(v_1)), \dots, (v_s, p(v_s))$ be the tree-edges in E_2. (Notice that we may have $s = 0$, in which case the conclusion of the lemma follows trivially.) Then, we have that all vertices in $\{u_1, \dots, u_t\}$ are descendants of u, and none of the vertices in $\{v_1, \dots, v_s\}$ is related as ancestor and descendant with u. This implies that no vertex from $\{u_1, \dots, u_t\}$ is related as ancestor and descendant with a vertex from $\{v_1, \dots, v_s\}$. Let $\mathcal{R}_1$ be a connectivity graph for $G \setminus E_1$, and let $\mathcal{R}_2$ be a connectivity graph for $G \setminus E_2$. Then we have that $V(\mathcal{R}_1) = \{\bar{u}_1, \dots, \bar{u}_t, \bar{r}\}$ and $V(\mathcal{R}_2) = \{\bar{v}_1, \dots, \bar{v}_s, \bar{r}\}$. Let

$\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$. We will show that $\mathcal{R}$ is a connectivity graph for $G \setminus E'$.

Let z and z' be two vertices in $\{u_1, \ldots, u_t, r\}$. First, suppose that z and z' are connected in $G \setminus E'$. Then, since $E_1 \subseteq E'$, we have that z and z' are connected in $G \setminus E_1$. This implies that $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}_1$. Therefore, $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}$. Conversely, suppose that $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}$. Since $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}$ and are both in $\{u_1, \ldots, u_t, r\}$, it is easy to see that $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}_1$. This implies that z and z' are connected in $G \setminus E_1$. Then, Lemma 6.1 implies that z and z' are connected in $G \setminus E'$.

With the analogous argument we can show that, if z and z' are two vertices in $\{v_1, \ldots, v_s, r\}$, then z and z' are connected in $G \setminus E'$ if and only if they are connected in $\mathcal{R}$.

Now let z be a vertex in $\{u_1, \ldots, u_t\}$, and let z' be a vertex in $\{v_1, \ldots, v_s\}$. First, suppose that z and z' are connected in $G \setminus E'$. Then there is a path P from z to z' in $G \setminus E'$. By Lemma 3.18, we have that P passes from an ancestor w of $nca\{z, z'\}$. Notice that there is no $i \in \{1, \ldots, t\}$ such that u_i is an ancestor of w, because otherwise we would have that u_i is an ancestor of z'. Similarly, there is no $i \in \{1, \ldots, s\}$ such that v_i is an ancestor of w, because otherwise we would have that v_i is an ancestor of z. Thus, there is no tree-edge from E' on the tree-path $T[w, r]$, and therefore w is connected with r in $G \setminus E'$. Thus, both z and z' are connected with r in $G \setminus E'$. Since $E_1 \subseteq E'$, we have that z is connected with r in $G \setminus E_1$. This implies that $\bar{z}$ is connected with $\bar{r}$ in $\mathcal{R}_1$, and therefore $\bar{z}$ is connected with $\bar{r}$ in $\mathcal{R}$. Similarly, since $E_2 \subset E'$, we have that z' is connected with r in $G \setminus E_2$. This implies that $\bar{z}'$ is connected with $\bar{r}$ in $\mathcal{R}_2$, and therefore $\bar{z}'$ is connected with $\bar{r}$ in $\mathcal{R}$. Thus, we infer that $\bar{z}$ is connected with $\bar{z}'$ in $\mathcal{R}$.

Conversely, suppose that $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}$. Since $\bar{z} \in V(\mathcal{R}_1)$ and $\bar{z}' \in V(\mathcal{R}_2)$, this implies that $\bar{z}$ is connected with $\bar{r}$ in $\mathcal{R}_1$, and $\bar{z}'$ is connected with $\bar{r}$ in $\mathcal{R}_2$. Therefore, z is connected with r in $G \setminus E_1$, and z' is connected with r in $G \setminus E_2$. Then, Lemma 6.1 implies that z is connected with r in $G \setminus E'$, and z' is connected with r in $G \setminus E'$. Therefore, z is connected with z' in $G \setminus E'$.

Thus, we have shown that, for every pair of vertices z and z' in $\{u_1, \ldots, u_t, v_1, \ldots, v_s, r\}$, we have that z and z' are connected in $G \setminus E'$ if and only if $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}$. Since the tree-edges in E' are given by $\{(u_1, p(u_1)), \ldots, (u_t, p(u_t)), (v_1, p(v_1)), \ldots, (v_s, p(v_s))\}$, this means that $\mathcal{R}$ is a connectivity graph for $G \setminus E'$. $\square$

Lemma 6.4. *Let E' be a set of edges, and let $(u,p(u))$ be a tree-edge in E' with the property that $B(u)\setminus E'=\emptyset$. Let E_1 be the set of the tree-edges in E' that are proper descendants of $(u,p(u))$, plus the back-edges in E', and let E_2 be the set of the tree-edges in E' that are not descendants of $(u,p(u))$, plus the back-edges in E'. Let $\mathcal{R}_1$ be a connectivity graph for $G\setminus E_1$, and let $\mathcal{R}_2$ be a connectivity graph for $G\setminus E_2$. Let $\mathcal{R}_1'$ be the graph that is derived from $\mathcal{R}_1$ by replacing every occurrence of $\bar{r}$ with $\bar{u}$. Then, $\mathcal{R}_1'\cup\mathcal{R}_2$ is a connectivity graph for $G\setminus E'$.*

Proof. Let $(u_1,p(u_1)),\dots,(u_t,p(u_t))$ be the tree-edges in E_1, and let $(v_1,p(v_1)),\dots,(v_s,p(v_s))$ be the tree-edges in E_2. (We note that $u\notin\{u_1,\dots,u_t,v_1,\dots,v_s\}$, and both t and s may be 0.) Then, we have that all vertices in $\{u_1,\dots,u_t\}$ are proper descendants of u, and no vertex from $\{v_1,\dots,v_s\}$ is a descendant of u. Now let $\mathcal{R}_1$, $\mathcal{R}_1'$, $\mathcal{R}_2$, and $\mathcal{R}$ be as in the statement of the lemma. We will show that $\mathcal{R}$ is a connectivity graph for $G\setminus E'$.

Since $V(\mathcal{R}_1)=\{\bar{u}_1,\dots,\bar{u}_t,\bar{r}\}$ and $V(\mathcal{R}_2)=\{\bar{v}_1,\dots,\bar{v}_s,\bar{r}\}$, we have that $V(\mathcal{R}_1)\cap V(\mathcal{R}_2)=\{\bar{r}\}$. This implies that $V(\mathcal{R}_1')\cap V(\mathcal{R}_2)=\emptyset$. Therefore, it is easy to see that no vertex from $\{\bar{u}_1,\dots,\bar{u}_t,\bar{u}\}$ is connected with a vertex from $\{\bar{v}_1,\dots,\bar{v}_s,\bar{r}\}$ in $\mathcal{R}$.

Now let z be a vertex in $\{u_1,\dots,u_t,u\}$ and let z' be a vertex in $\{v_1,\dots,v_s,r\}$. Let us suppose, for the sake of contradiction, that z is connected with z' in $G\setminus E'$. Then there is a path P from z to z' in $G\setminus E'$. Lemma 3.18 implies that P passes from an ancestor w of *nca*$\{z,z'\}$. We have that z is a common descendant of w and u. Thus, u and w are related as ancestor and descendant. But u cannot be an ancestor of w, because otherwise it would be an ancestor of z'. Thus, we have that u is a proper descendant of w. Since P starts from a descendant of u and reaches a proper ancestor of u, by Lemma 3.19 we have that P must either use a back-edge that leaps over u, or it passes from the tree-edge $(u,p(u))$. But both of these cases are impossible, since $B(u)\setminus E'=\emptyset$ and $(u,p(u))\in E'$. This shows that z is not connected with z' in $G\setminus E'$.

Now let z and z' be two vertices in $\{u_1,\dots,u_t\}$. First, suppose that z and z' are connected in $G\setminus E'$. Then, since $E_1\subset E'$, we have that z and z' are connected in $G\setminus E_1$. This implies that $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}_1$, and therefore they are connected in $\mathcal{R}_1'$, and therefore they are connected in $\mathcal{R}$. Conversely, suppose that $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}$. Then, we have that $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}_1'$, and therefore they are connected in $\mathcal{R}_1$. This implies that there is a path P from z to z' in $G\setminus E_1$. If P does not use a tree-edge from $E_2'=E_2\cup\{(u,p(u))\}$, then we have that P is a path in $G\setminus E'$, and this shows that z is connected with z' in $G\setminus E'$. Otherwise, let

us suppose that P uses a tree-edge from E_2'. Then we claim that P has the form $P_1 + (u, p(u)) + Q + (p(u), u) + P_2$, where P_1 is a path from z to u that does not use tree-edges from E_2', Q is a path from $p(u)$ to $p(u)$ that may use tree-edges from E_2', and P_2 is a path from u to z' that does not use tree-edges from E_2' $(*)$. This implies that P_1 is a path from z to u in $G \setminus E'$, and P_2 is a path from u to z' in $G \setminus E'$. Thus, $P_1 + P_2$ is a path from z to z' in $G \setminus E'$, and therefore z is connected with z' in $G \setminus E'$.

Now we will prove $(*)$. Let $(v, p(v))$ be the first occurrence of a tree-edge from E_2' in P. Then, since P starts from z, by Lemma 3.18 we have that P passes from an ancestor w of $nca\{z, v\}$. Then, we have that z is a common descendant of u and w, and therefore u and w are related as ancestor and descendant. Notice that we cannot have that u is a proper ancestor of w, because this would imply that u is a proper ancestor of v (and we have that either $v \in \{v_1, \dots, v_s\}$, or $v = u$). Thus, u is a descendant of w. Then, since P starts from a descendant of u and reaches a proper ancestor of u (which is either w or $p(u)$), by Lemma 3.19 we have that P must either use a back-edge from $B(u)$, or the tree-edge $(u, p(u))$, before it leaves the subtree of u. Since P is a path in $G \setminus E_1$ and E_1 contains all the back-edges from E' and $B(u) \setminus E' = \emptyset$, the only viable option is that P uses the tree-edge $(u, p(u))$ in order to leave the subtree of u. Thus, there is a part P_1 of P from z to u that lies entirely within the subtree of u. In particular, we have that P_1 does not use a tree-edge from E_2'. Similarly, by considering the last occurrence of a tree-edge from E_2' in P, we can show that there is a part P_2 of P from u to z' that does not use a tree-edge from E_2'. This establishes $(*)$.

Now let z be a vertex in $\{u_1, \dots, u_t\}$. First, suppose that u and z are connected in $G \setminus E'$. Then there is a path P from u to z in $G \setminus E'$. Since $E_1 \subset E'$, we have that P is a path from u to z in $G \setminus E_1$. We have that there is no tree-edge from E_1 on the tree-path $T[r, u]$. Thus, $T[r, u] + P$ is a path from r to z in $G \setminus E_1$, and therefore r is connected with z in $G \setminus E_1$. This implies that $\bar{r}$ is connected with $\bar{z}$ in $\mathcal{R}_1$, and therefore $\bar{u}$ is connected with $\bar{z}$ in $\mathcal{R}_1'$. Thus, $\bar{u}$ is connected with $\bar{z}$ in $\mathcal{R}$. Conversely, suppose that $\bar{u}$ is connected with $\bar{z}$ in $\mathcal{R}$. This implies that $\bar{u}$ is connected with $\bar{z}$ in $\mathcal{R}_1'$, and therefore $\bar{r}$ is connected with $\bar{z}$ in $\mathcal{R}_1$. This implies that r is connected with z in $G \setminus E_1$. Thus, there is a path P from r to z in $G \setminus E_1$. Since r is a proper ancestor of u and z is a descendant of u, Lemma 3.19 implies that P must either use a back-edge from $B(u)$, or the tree-edge $(p(u), u)$, and then it finally lies entirely within the subtree of u. Thus, since E_1 contains all the back-edges from E' and $B(u) \setminus E' = \emptyset$, we have

that P eventually passes from the tree-edge $(p(u), u)$, and stays within the subtree of u. Thus, we have that u is connected with z in $G \setminus E_1$, through a path P' that lies entirely within the subtree of u. This implies that P' does not use tree-edges from E_2. Thus, P' is a path from u to z in $G \setminus E'$, and therefore u is connected with z in $G \setminus E'$.

Now let z and z' be two vertices in $\{v_1, \dots, v_s, r\}$. First, suppose that z and z' are connected in $G \setminus E'$. Then, since $E_2 \subset E'$, we have that z and z' are connected in $G \setminus E_2$. This implies that $\bar{z}$ is connected with $\bar{z}'$ in $\mathcal{R}_2$. Thus, we have that $\bar{z}$ is connected with $\bar{z}'$ in $\mathcal{R}$. Conversely, suppose that $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}$. Then we have that $\bar{z}$ and $\bar{z}'$ are connected in $\mathcal{R}_2$. This implies that z and z' are connected in $G \setminus E_2$. Thus, there is a path P from z to z' in $G \setminus E_2$. If P does not use any tree-edge from $E_1' = E_1 \cup \{(u, p(u))\}$, then P is a path in $G \setminus E'$, and therefore z and z' are connected in $G \setminus E'$. So let us assume that P uses a tree-edge from E_1'. Then we claim that P has the form $P_1 + Q + P_2$, where P_1 is a path from z to $p(u)$ that does not use tree-edges from E_1', Q is a path from $p(u)$ to $p(u)$ (that uses tree-edges from E_1), and P_2 is a path from $p(u)$ to z' that does not use tree-edges from E_1' $(**)$. This implies that $P_1 + P_2$ is a path from z to z' in $G \setminus E'$, and therefore z and z' are connected in $G \setminus E'$.

Now we will prove $(**)$. Since P uses a tree-edge from E_1', we may consider the first occurrence $(u', p(u'))$ of an edge from E_1' that is used by P. Then, we have that u' is a descendant of u. Since P starts from z, Lemma 3.18 implies that P contains a subpath from z to u' that passes from an ancestor w of *nca*$\{z, u'\}$, and let w be the first vertex visited by P with this property. Let P' be the initial part of P from z to the first occurrence of w. Then we have that P' does not use any tree-edge from E_1'. Since u' is a descendant of u but z is not, we have that *nca*$\{z, u'\}$ is a proper ancestor of u, and therefore w is a proper ancestor of u. Then, Lemma 3.19 implies that the part of P from w to u' must either use the tree-edge $(u, p(u))$, or a back-edge that leaps over u. Since P is a path in $G \setminus E_2$ and E_2 contains all the back-edges from E' and $B(u) \setminus E' = \emptyset$, we infer that the part of P from w to u' must use the tree-edge $(u, p(u))$ in order to enter the subtree of u. Now let P'' be the part of P from the first occurrence of w to the first occurrence of $p(u)$ that is followed by $(u, p(u))$. Then, we have that $P_1 = P' + P''$ is a path from z to $p(u)$ that does not use any tree-edge from E_1'. Similarly, since P ends in z', we can show that the final part of P is a subpath P_2 that starts from $p(u)$ and ends in z'. We let Q denote the middle part of P (i.e., the

one between P_1 and P_2), and this establishes $(**)$.

Thus, we have shown that, for any two vertices z and z' in $\{u_1, \ldots, u_t, u, v_1, \ldots, v_s, r\}$, we have that z is connected with z' in $G \setminus E'$ if and only if $\bar{z}$ is connected with $\bar{z}'$ in $\mathcal{R}$. This means that $\mathcal{R}$ is a connectivity graph for $G \setminus E'$. □

Given a set of edges E' with $|E'| \leq 4$, we will show how to construct a connectivity graph $\mathcal{R}$ for $G \setminus E'$. We distinguish five cases, depending on the number of tree-edges in E'. Whenever possible, we use Lemmata 6.2, 6.3 and 6.4, in order to revert to previous cases. This simplifies the analysis a lot, because the number of cases that may appear is very large. If A and B are two subtrees of T, we use (A, B) to denote the set of the back-edges that connect A and B.

6.2 E' contains zero tree-edges

In this case, $T \setminus E'$ is connected, and therefore $G \setminus E'$ is also connected. Thus, we know that every connectivity query in $G \setminus E'$ is positive. (The connectivity graph $\mathcal{R}$ of $G \setminus E'$ consists of a single vertex $\bar{r}$.)

6.3 E' contains one tree-edge

Let $(u, p(u))$ be the tree-edge contained in E'. The connected components of $T \setminus \{(u, p(u))\}$ are $A = T(u)$ and $B = T(r) \setminus T(u)$. Thus, $\mathcal{R}$ consists of the vertices $\{\bar{u}, \bar{r}\}$. We can see that A is connected with B in $G \setminus E'$ if and only if there is a back-edge in $B(u) \setminus E'$. Since E' contains at most three back-edges, it is sufficient to have collected at most four back-edges of $B(u)$ in a set $B_4(u)$, and then check whether $B_4(u) \setminus E' = \emptyset$. We may pick the four lowest *low*-edges of u in order to build $B_4(u)$, since these are easy to compute. (I.e., $B_4(u)$ consists of the non-null low_i-edges of u, for all $i \in \{1, 2, 3, 4\}$.) If $B_4(u) \setminus E' \neq \emptyset$, then we add the edge $(\bar{u}, \bar{r})$ to $\mathcal{R}$. Otherwise, $\bar{u}$ is disconnected from $\bar{r}$ in $\mathcal{R}$.

6.4 E' contains two tree-edges

Let $(u, p(u))$ and $(v, p(v))$ be the two tree-edges contained in E'. Then we have that $\mathcal{R}$ consists of the vertices $\{\bar{u}, \bar{v}, \bar{r}\}$. Let us assume w.l.o.g. that $u > v$. Suppose that u and v are not related as ancestor and descendant. Then, we set $E_1 = E' \setminus \{(u, p(u))\}$ and $E_2 = E' \setminus \{(v, p(v))\}$. Each of the sets E_1 and E_2 contains only one tree-edge, and therefore we can revert to the previous case in order to build a connectivity graph $\mathcal{R}_1$ and $\mathcal{R}_2$ for $G \setminus E_1$ and $G \setminus E_2$, respectively. Then, by Lemma 6.3 we have that $\mathcal{R}_1 \cup \mathcal{R}_2$ is a connectivity graph for $G \setminus E'$. So let us assume that the two tree-edges in E' are related as ancestor and descendant. Since $u > v$, this implies that u is a descendant of v. Let $A = T(u)$, $B = T(v) \setminus T(u)$, and $C = T(r) \setminus T(v)$.

First, we will check whether there is a back-edge from A to B in $G \setminus E'$. Since E' contains at most two back-edges, we have that $(A, B) \setminus E' \neq \emptyset$ if and only if at least one of the three highest *high*-edges of u is not in E' and has its lower endpoint in B. In other words, $(A, B) \setminus E' \neq \emptyset$ if and only if the $high_i$-edge of u (exists, and) is not in E', and $high_i(u) \in B$, for some $i \in \{1, 2, 3\}$. If that is the case, then we know that u is connected with $p(u)$ in $G \setminus E'$. Thus, we add the edge $(\bar{u}, \bar{v})$ to $\mathcal{R}$. Then we can set $E' \leftarrow E' \setminus \{(u, p(u))\}$, and revert to the previous case (where E' contains one tree-edge), according to Lemma 6.2.

Otherwise, we have that all the back-edges in $B(u) \setminus E'$ (if there are any), have their lower endpoint in C. In this case, we have that there is a back-edge from A to C if and only if $B(u) \setminus E' \neq \emptyset$. Since E' contains at most two back-edges, we have that $B(u) \setminus E' \neq \emptyset$ if and only if at least one of the low_i-edges of u, for $i \in \{1, 2, 3\}$, (exists and) does not lie in E'. If we have that $B(u) \setminus E' = \emptyset$, then we know that A is not connected with the rest of the graph in $G \setminus E'$. Thus, we can set $E' \leftarrow E' \setminus \{(u, p(u))\}$ and revert to the previous case (where E' contains one tree-edge), according to Lemma 6.4.

Thus, let us assume that there is a back-edge between A and C in $G \setminus E'$. Then we add the edge $(\bar{u}, \bar{r})$ to $\mathcal{R}$. Now it remains to check whether there is a back-edge from B to C in $G \setminus E'$. Since we assume that there is no back-edge from A to B in $G \setminus E'$, we can distinguish three possibilities: either (1) no edge from E' is in (A, B), or (2) precisely one edge e from E' is in (A, B), or (3) both the back-edges e and e' from E' are in (A, B). In case (1), we have that $B(u) \subseteq B(v)$. Thus, there is a back-edge from B to C in $G \setminus E'$ if and only if $\mathit{bcount}(v) - \mathit{bcount}(u) > |E' \cap (B(v) \setminus B(u))|$. We note that it is easy

to compute $|E' \cap (B(v) \setminus B(u))|$: we simply count how many back-edges from E' leap over v but not over u. Now, in case (2) we have $B(u) \setminus \{e\} \subseteq B(v)$. Thus, there is a back-edge from B to C in $G \setminus E'$ if and only if $\mathit{bcount}(v) - \mathit{bcount}(u) + 1 > |E' \cap (B(v) \setminus B(u))|$. Finally, in case (3) we have $B(u) \setminus \{e, e'\} \subseteq B(v)$, and E' contains no back-edge from $B(v) \setminus B(u)$. Thus, there is a back-edge from B to C in $G \setminus E'$ if and only if $\mathit{bcount}(v) - \mathit{bcount}(u) + 2 > 0$. We note that it is easy to determine which case $(1) - (3)$ applies: we simply count how many back-edges from E' (if E' contains back-edges) leap over u, but not over v. If we have determined that there is a back-edge from B to C in $G \setminus E'$, then we add the edge $(\bar{v}, \bar{r})$ to $\mathcal{R}$.

6.5 E' contains three tree-edges

Let $(u, p(u))$, $(v, p(v))$ and $(w, p(w))$ be the three tree-edges contained in E'. Then we have that $\mathcal{R}$ consists of the vertices $\{\bar{u}, \bar{v}, \bar{w}, \bar{r}\}$. Let us assume w.l.o.g. that $u > v > w$. We may also assume that one of the three tree-edges in E' is an ancestor of the other two. Otherwise, we can use Lemma 6.3 in order to revert to the previous case (where E' contains two tree-edges), because then there are at least two tree-edges in E' with the property that no tree-edge in E' is a proper ancestor of them.

Thus, since one of the three tree-edges in E' is an ancestor of the other two and $w < v < u$, we have that w is a common ancestor of u and v. Now there are two cases to consider: either (1) u and v are not related as ancestor and descendant, or (2) u and v are related as ancestor and descendant. Since $u > v$, in case (2) we have that v is an ancestor of u.

6.5.1 u and v are not related as ancestor and descendant

Let $A = T(u)$, $B = T(v)$, $C = T(w) \setminus (T(u) \cup T(v))$, and $D = T(r) \setminus T(w)$.

If there is no back-edge in $G \setminus E'$ that connects A, or B, or $A \cup B \cup C$, with the rest of the graph in $G \setminus E'$, then we can use Lemma 6.4 in order to revert to the previous case (where E' contains two tree-edges). Every one of those conditions is equivalent to [$\mathit{bcount}(u) = 0$ or $B(u) = \{e\}$, where e is the back-edge in E'], or [$\mathit{bcount}(v) = 0$ or $B(v) = \{e\}$, where e is the back-edge in E'], or [$\mathit{bcount}(w) = 0$ or $B(w) = \{e\}$, where e is the back-edge in E'], respectively, and so we can check them easily in constant time.

Otherwise, if there is a back-edge that connects A with C in $G \setminus E'$, or a back-edge that connects B with C in $G\setminus E'$, then we add the edge $(\bar{u}, \bar{w})$, or $(\bar{v}, \bar{w})$, respectively, to $\mathcal{R}$. Then we set $E' \leftarrow E' \setminus \{(u, p(u))\}$, or $E' \leftarrow E' \setminus \{(v, p(v))\}$, respectively, and we revert to the previous case (where E' contains two tree-edges) according to Lemma 6.2. Notice that there is a back-edge that connects A with C in $G \setminus E'$ if and only if: either (i) $high_1(u) \in C$ and the $high_1$-edge of u is not in E', or (ii) $high_2(u) \in C$ and the $high_2$-edge of u is not in E'. Similarly, we can easily check whether there is a back-edge that connects B with C in $G \setminus E'$.

So let us assume that none of the above is true. This means that, in $G \setminus E'$, there is a back-edge from A to D, a back-edge from B to D, no back-edge from A to C, and no back-edge from B to C. Then, we add the edges $(\bar{u}, \bar{r})$ and $(\bar{v}, \bar{r})$ to $\mathcal{R}$. Now it remains to determine whether there is a back-edge from C to D in $G \setminus E'$. Suppose first that E' contains a back-edge e, that connects either A and C, or B and C. Then, there is a back-edge from C to D in $G \setminus E'$ if and only if $bcount(w) > bcount(u) + bcount(v) - 1$. (This is because $B(u) \sqcup B(v) \subseteq B(w) \sqcup \{e\}$ in this case.) Thus, if this condition is satisfied, then we add the edge $(\bar{w}, \bar{r})$ to $\mathcal{R}$. Now, let us assume that E' contains a back-edge from C to D. Then, there is a back-edge from C to D in $G \setminus E'$ if and only if $bcount(w) - 1 > bcount(u) + bcount(v)$. Thus, if this condition is satisfied, then we add the edge $(\bar{w}, \bar{r})$ to $\mathcal{R}$. Finally, let us assume that the back-edge in E' (if it contains a back-edge) does not connect A and C, or B and C, or C and D. Then, there is a back-edge from C to D in $G \setminus E'$ if and only if $bcount(w) > bcount(u) + bcount(v)$. Thus, if this condition is satisfied, then we add the edge $(\bar{w}, \bar{r})$ to $\mathcal{R}$.

6.5.2 v is an ancestor of u

Let $A = T(u)$, $B = T(v) \setminus T(u)$, $C = T(w) \setminus T(v)$, and $D = T(r) \setminus T(w)$.

If there is no back-edge in $G \setminus E'$ that connects A, or $A \cup B$, or $A \cup B \cup C$, with the rest of the graph in $G \setminus E'$, then we can use Lemma 6.4 in order to revert to the previous case (where E' contains either one or two tree-edges). Every one of those conditions is equivalent to [$bcount(u) = 0$ or $B(u) = \{e\}$, where e is the back-edge in E'], or [$bcount(v) = 0$ or $B(v) = \{e\}$, where e is the back-edge in E'], or [$bcount(w) = 0$ or $B(w) = \{e\}$, where e is the back-edge in E'], respectively, and so we can check them easily in constant time.

If in $G \setminus E'$ there is a back-edge that connects A and B, then we add the edge

$(\bar{u},\bar{v})$ to $\mathcal{R}$. Then we set $E' \leftarrow E' \setminus \{(u,p(u))\}$, and we revert to the previous case (where E' contains two tree-edges) according to Lemma 6.2. Notice that there is a back-edge in $G \setminus E'$ that connects A and B if and only if: either (i) $high_1(u) \in B$ and the $high_1$-edge of u is not in E', or (ii) $high_2(u) \in B$ and the $high_2$-edge of u is not in E'. Thus, we can easily check this condition in constant time.

So let us assume that there is no back-edge that connects A and B in $G \setminus E'$. If in $G \setminus E'$ there is a back-edge that connects A and C, and a back-edge that connects A and D, then the parts C and D are connected in $G \setminus E'$ through the mediation of A. Thus, we add the edge $(\bar{w},\bar{r})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(w,p(w))\}$, and we revert to the previous case (where E' contains two tree-edges) according to Lemma 6.2. Since there is no back-edge that connects A and B in $G \setminus E'$, notice that there is a back-edge in $G \setminus E'$ that connects A and C if and only if: either (i) $high_1(u) \in C$ and the $high_1$-edge of u is not in E', or (ii) $high_2(u) \in C$ and the $high_2$-edge of u is not in E'. Also, there is a back-edge in $G \setminus E'$ that connects A and D if and only if: either (i) $low_1(u) \in D$ and the low_1-edge of u is not in E', or (ii) $low_2(u) \in D$. Thus, we can easily check those conditions in constant time.

So let us assume that none of the above is true. Thus, there are two cases to consider in $G \setminus E'$: either (1) there is a back-edge from A to C, but no back-edge from A to D, or (2) there is a back-edge from A to D, but no back-edge from A to C.

Let us consider case (1) first. Then, we add the edge $(\bar{u},\bar{w})$ to $\mathcal{R}$. First, we will determine whether there is a back-edge from B to D in $G \setminus E'$. Since there is no back-edge from A to D in $G \setminus E'$, notice that there is a back-edge from B to D in $G \setminus E'$ if and only if: either $low_1(v) \in D$ and the low_1-edge of v is not in E', or $low_2(v) \in D$ and the low_2-edge of v is not in E'. Thus, we can check in constant time whether there is a back-edge from B to D in $G \setminus E'$. If we have determined that there is no back-edge from B to D in $G \setminus E'$, then, since we have supposed that there is a back-edge in $B(w) \setminus E'$, we have that there is a back-edge from C to D in $G \setminus E'$ (since $(A,D) \setminus E' = (B,D) \setminus E' = \emptyset$). This implies that w remains connected with $p(w)$ in $G \setminus E'$. Thus, we add the edge $(\bar{w},\bar{r})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(w,p(w))\}$, and we revert to the previous case (where E' contains two tree-edges) according to Lemma 6.2. So let us suppose that there is a back-edge from B to D in $G \setminus E'$. Then, we add the edge $(\bar{v},\bar{r})$ to $\mathcal{R}$. Now it remains to determine if $(B,C) \setminus E' \neq \emptyset$ or $(C,D) \setminus E' \neq \emptyset$. Notice that either of those cases implies that $G \setminus E'$ is connected.

We have $B(u) = (A,B) \cup (A,C) \cup (A,D)$, $B(v) = (A,C) \cup (A,D) \cup (B,C) \cup (B,D)$ and

$B(w) = (A,D) \cup (B,D) \cup (C,D)$. Thus, we have $N = \mathit{bcount}(w) - \mathit{bcount}(v) + \mathit{bcount}(u) = |(C,D)| - |(B,C)| + |(A,B)| + |(A,D)|$ and $s = \mathit{SumAnc}(w) - \mathit{SumAnc}(v) + \mathit{SumAnc}(u) = \mathit{SumAnc}((C,D)) - \mathit{SumAnc}((B,C)) + \mathit{SumAnc}((A,B)) + \mathit{SumAnc}((A,D))$, where we let $\mathit{SumAnc}(S)$, for a set S of back-edges, be the sum of the lower endpoints of the back-edges in S.

Here we distinguish two cases: either (1.1) the back-edge in E' (if it exists) does not lie in $(A,B) \cup (A,D)$, or (1.2) the back-edge e in E' (exists and) lies in $(A,B) \cup (A,D)$. (It is easy to determine in constant time which case applies.)

First, let us consider case (1.1). Then we have $N = |(C,D)| - |(B,C)|$ and $s = \mathit{SumAnc}((C,D)) - \mathit{SumAnc}((B,C))$. Here we distinguish two cases: either (1.1.1) the back-edge in E' (if it exists) does not lie in $(C,D) \cup (B,C)$, or (1.1.2) E' contains a back-edge e that lies in $(C,D) \cup (B,C)$. (Again, it is easy to determine in constant time which case applies.) So let us consider case (1.1.1) first. Thus, if we have $N \neq 0$, then at least one of (C,D) and (B,C) is not empty, and therefore $G \setminus E'$ is connected. Thus, it is sufficient to add one more edge to $\mathcal{R}$ in order to make it connected (e.g., we may add $(\bar{w}, \bar{r})$). Otherwise, suppose that $N = 0$. Then we have $|(C,D)| = |(B,C)|$. Since the lower endpoint of every back-edge in (C,D) is lower than the lower endpoint of every back-edge in (B,C), this implies that $s = \mathit{SumAnc}((C,D)) - \mathit{SumAnc}((B,C)) < 0$ if and only if $(C,D) \neq \emptyset$ (and $(B,C) \neq \emptyset$). Thus, it is sufficient to add e.g. the edge $(\bar{w}, \bar{r})$ to $\mathcal{R}$ if and only if $s < 0$.

Now let us consider case (1.1.2). Let z be the lower endpoint of e. First, suppose that $e \in (C,D)$. Thus, if $N > 1$, then $(C,D) \setminus E'$ is not empty, and therefore $G \setminus E'$ is connected. Similarly, if $N < 1$, then (B,C) is not empty, and therefore $G \setminus E'$ is connected. Thus, in those cases, it is sufficient to add one more edge to $\mathcal{R}$ in order to make it connected (e.g., we may add $(\bar{w}, \bar{r})$). Otherwise, suppose that $N = 1$. Then we have $|(C,D)| = |(B,C)| + 1$. Thus, if $|(B,C)| = 0$, then we have $(C,D) = \{e\}$, and therefore s coincides with z. Otherwise, if $|(B,C)| > 0$, then, since the lower endpoint of every back-edge in (C,D) is lower than the lower endpoint of every back-edge in (B,C), we have that $s = \mathit{SumAnc}((C,D)) - \mathit{SumAnc}((B,C)) < z$. Thus, it is sufficient to add e.g. the edge $(\bar{w}, \bar{r})$ to $\mathcal{R}$ if and only if $s < z$. Now let us suppose that $e \in (B,C)$. Thus, if $N > -1$, then (C,D) is not empty, and therefore $G \setminus E'$ is connected. Similarly, if $N < -1$, then $(B,C) \setminus E'$ is not empty, and therefore $G \setminus E'$ is connected. Thus, in those cases, it is sufficient to add one more edge to $\mathcal{R}$ in order to make it connected (e.g., we may add $(\bar{w}, \bar{r})$). Otherwise, suppose that $N = -1$. Then

we have $|(C,D)|+1 = |(B,C)|$. Thus, if $|(C,D)| = 0$, then we have $(B,C) = \{e\}$, and therefore s coincides with $-z$. Otherwise, if $|(C,D)| > 0$, then, since the lower endpoint of every back-edge in (C,D) is lower than the lower endpoint of every back-edge in (B,C), we have that $s = \mathit{SumAnc}((C,D)) - \mathit{SumAnc}((B,C)) < -z$. Thus, it is sufficient to add e.g. the edge $(\bar{w},\bar{r})$ to $\mathcal{R}$ if and only if $s < -z$.

Now let us consider case (1.2). Then, since there is no back-edge from A to B in $G\setminus E'$, and no back-edge from A to D in $G\setminus E'$, we have that one of (A,B) and (A,D) coincides with $\{e\}$, and the other is empty. Then, we have $N = |(C,D)| - |(B,C)| + 1$ and $s = \mathit{SumAnc}((C,D)) - \mathit{SumAnc}((B,C)) + \mathit{SumAnc}(\{e\})$. Thus, if we have $N \neq 1$, then at least one of (C,D) and (B,C) is not empty, and therefore $G\setminus E'$ is connected. Thus, it is sufficient to add one more edge to $\mathcal{R}$ in order to make it connected (e.g., we may add $(\bar{w},\bar{r})$). Otherwise, suppose that $N = 1$. Then we have $|(C,D)| - |(B,C)| = 0$. Since the lower endpoint of every back-edge in (C,D) is lower than the lower endpoint of every back-edge in (B,C), this implies that $\mathit{SumAnc}((C,D)) - \mathit{SumAnc}((B,C)) < 0$ if and only if $(C,D) \neq \emptyset$ (and $(B,C) \neq \emptyset$). Thus, it is sufficient to add e.g. the edge $(\bar{w},\bar{r})$ to $\mathcal{R}$ if and only if $s < \mathit{SumAnc}(\{e\})$. (Notice that it is easy to compute $\mathit{SumAnc}(\{e\})$: this is just the (DFS number of the) lower endpoint of e.)

Now let us consider case (2). Then, we add the edge $(\bar{u},\bar{r})$ to $\mathcal{R}$. Now, if there is a back-edge that connects B and C in $G\setminus E'$, then we add the edge $(\bar{v},\bar{w})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(v,p(v)\}$, and we revert to the previous case (where E' contains two tree-edges) according to Lemma 6.2. Notice that, since there is no back-edge from A to C in $G\setminus E'$, this condition is equivalent to: either $\mathit{high}_1(v) \in C$ and the high_1-edge of v is not in E', or $\mathit{high}_2(v) \in C$ and the high_1-edge of v is not in E'. Thus, we can easily check whether there is a back-edge that connects B and C in $G\setminus E'$, in constant time.

Now let us assume that there is no back-edge that connects B and C in $G\setminus E'$. Then, if there is a back-edge that connects B and D in $G\setminus E'$ $(*)$, then we have that A and B remain connected in $G\setminus E'$ through the mediation of D. Thus, we add the edge $(\bar{u},\bar{v})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(u,p(u))\}$, and we revert to the previous case (where E' contains two tree-edges) according to Lemma 6.2. In order to check condition $(*)$, i.e., whether there is a back-edge that connects B and D in $G\setminus E'$, we distinguish the following cases. First, suppose that the back-edge in E' (if it exists), is neither in $B(u)$ nor in $B(v)$. Then, since all the back-edges in $B(u)$ connect A and D, and since all the back-edges in $B(v)$ connect either A and D or B and D, we have that

$(*)$ is true if and only if *bcount*$(v) >$ *bcount*(u). Now, suppose that the back-edge e in E' is in $B(u)$, but not in $B(v)$. Then, all the back-edges in $B(u)$, except e, connect A and D, and all back-edges in $B(v)$ connect either A and D or B and D. Thus, we have that $(*)$ is true if and only if *bcount*$(v) \geq$ *bcount*(u). Now, suppose that the back-edge e in E' is in both $B(u)$ and $B(v)$. Then, we have that all the back-edges in $B(u)$, except possibly e, connect A and D, and all the back-edges in $B(v)$, except possibly e, connect A and D or B and D. Thus, we have that $(*)$ is true if and only if *bcount*$(v) >$ *bcount*(u). Finally, suppose that the back-edge e in E' is in $B(v)$, but not in $B(u)$. Then, all the back-edges in $B(u)$ connect A and D, and all the back-edges in $B(v)$, except possibly e, connect A and D or B and D. Thus, we have that $(*)$ is true if and only if *bcount*$(v) >$ *bcount*$(u) + 1$. Thus, we can easily check whether there is a back-edge that connects B and D in $G \setminus E'$, in constant time.

Finally, suppose that neither of the above two is the case (i.e., we have $(B, C) \setminus E' = \emptyset$ and $(B, D) \setminus E' = \emptyset$). Then we only have to check whether there is a back-edge in $G \setminus E'$ that connects C and D. We distinguish the following cases: either (2.1) E' contains a back-edge $e \in (C, D)$, or (2.2) E' contains a back-edge $e \in (A, C) \cup (B, C)$, or (2.3) none of the previous is true. (Notice that it is easy to determine in constant time which case holds.) In case (2.1) we have $B(w) = (A, D) \cup ((C, D) \setminus \{e\}) \cup \{e\}$. Thus, there is a back-edge from C to D in $G \setminus E'$ if and only if *bcount*$(w) >$ *bcount*$(u) + 1$. In case (2.2) we have $B(w) = (B(v) \setminus \{e\}) \cup (C, D)$, and $e \in B(v) \setminus (C, D)$. Thus, there is a back-edge from C to D in $G \setminus E'$ if and only if *bcount*$(w) >$ *bcount*$(v) - 1$. In case (2.3) we have $B(w) = B(v) \cup (C, D)$ and $e \notin (C, D)$. Thus, there is a back-edge from C to D in $G \setminus E'$ if and only if *bcount*$(w) >$ *bcount*(v). Thus, in either of those cases, if we determine that there is a back-edge from C to D in $G \setminus E'$, then we add the edge $(\bar{w}, \bar{r})$ to $\mathcal{R}$.

6.6 E' contains four tree-edges

Let $(u, p(u))$, $(v, p(v))$, $(w, p(w))$ and $(z, p(z))$ be the tree-edges that are contained in E'. Then we have that $\mathcal{R}$ consists of the vertices $\{\bar{u}, \bar{v}, \bar{w}, \bar{z}, \bar{r}\}$. We may assume w.l.o.g. that $u > v > w > z$. Suppose that there are at least two distinct edges $(u', p(u'))$ and $(v', p(v'))$ in E' with the property that no edge in E' is a proper ancestor of them. Let E_1 be the subset of E' that consists of the descendants of $(u', p(u'))$, and let E_2 be the

subset of E' that consists of the non-descendants of $(u', p(u'))$. Then, we have that $(v', p(v')) \notin E_1$, and $(u', p(u')) \notin E_2$. Thus, we can construct connectivity graphs $\mathcal{R}_1$ and $\mathcal{R}_2$ for $G \setminus E_1$ and $G \setminus E_2$, respectively, by reverting to the previous cases. Then, by Lemma 6.3, we have that $\mathcal{R}_1 \cup \mathcal{R}_2$ is a connectivity graph for $G \setminus E'$.

Thus, we may assume that one of the tree-edges in E' is an ancestor of the other three, because otherwise we can construct the graph $\mathcal{R}$ by reverting to the previous cases. Then, since $u > v > w > z$, we have that z is a common ancestor of all vertices in $\{u, v, w\}$. Now, there are four cases to consider: either (1) no two vertices in $\{u, v, w\}$ are related as ancestor and descendant, or (2) only two among $\{u, v, w\}$ are related as ancestor and descendant, or (3) one of $\{u, v, w\}$ is an ancestor of the other two, but the other two are not related as ancestor and descendant, or (4) every two vertices in $\{u, v, w\}$ are related as ancestor and descendant.

In case (2), we can either have that w is an ancestor of v (and u is not related as ancestor and descendant with w and v), or v is an ancestor of u (and w is not related as ancestor and descendant with u and v). Both of these cases can be handled with essentially the same argument (it is just that the roles of w, v and u in the first case are exchanged with v, u and w, respectively, in the second case), and so we will assume w.l.o.g. that v is an ancestor of u in this case. In cases (3) and (4), we have that the ancestry relation between the vertices in $\{u, v, w\}$ is fixed by the assumption $u > v > w$. Thus, in case (3) we have that w is an ancestor of both v and u (but u, v are not related as ancestor and descendant), and in case (4) we have that w is an ancestor of v, and v is an ancestor of u.

In any case, notice that there are no back-edges in E' (since we have assumed that $|E'| \leq 4$, and therefore E' consists of four tree-edges).

6.6.1 No two vertices in $\{u, v, w\}$ are related as ancestor and descendant

Let $A = T(u)$, $B = T(v)$, $C = T(w)$, $D = T(z) \setminus (T(u) \cup T(v) \cup T(w))$ and $E = T(r) \setminus T(z)$.

If we have that either $\mathit{high}_1(u) = \bot$, or $\mathit{high}_1(v) = \bot$, or $\mathit{high}_1(w) = \bot$, then there is no back-edge in $G \setminus E'$ that connects A, or B, or C, respectively, with the rest of the graph. Therefore, we can use Lemma 6.4 in order to revert to the previous case (where E' contains three tree-edges).

Otherwise, if we have that either $\mathit{high}_1(u) \in D$, or $\mathit{high}_1(v) \in D$, or $\mathit{high}_1(w) \in D$,

then we have that A is connected with D, or B is connected with D, or C is connected with D, respectively, through a back-edge in $G \setminus E'$. Therefore, we add the edge $(\bar{u}, \bar{z})$, or $(\bar{v}, \bar{z})$, or $(\bar{w}, \bar{z})$, respectively, to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(u, p(u))\}$, or $E' \leftarrow E' \setminus \{(v, p(v))\}$, or $E' \leftarrow E' \setminus \{(w, p(w))\}$, respectively, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2.

Thus, let us assume that $high_1(u) \in E$, and $high_1(v) \in E$, and $high_1(w) \in E$. Then, we add the edges $(\bar{u}, \bar{r})$, $(\bar{v}, \bar{r})$ and $(\bar{w}, \bar{r})$ to $\mathcal{R}$. Now, if D is connected with the rest of the graph in $G \setminus E'$, then this can only be through a back-edge that starts from D and ends in E (since $high_1(u) \in E$ and $high_1(v) \in E$ and $high_1(w) \in E$, implies that $(A, D) = (B, D) = (C, D) = \emptyset$). Then, the existence of a back-edge in (D, E) is equivalent to the condition $bcount(z) > bcount(u) + bcount(v) + bcount(w)$. Thus, if this condition is satisfied, then we add the edge $(\bar{z}, \bar{r})$ to $\mathcal{R}$.

6.6.2 w and v are not related as ancestor and descendant, and v is an ancestor of u

Let $A = T(u)$, $B = T(v) \setminus T(u)$, $C = T(w)$, $D = T(z) \setminus (T(v) \cup T(w))$ and $E = T(r) \setminus T(z)$.

If we have that either $high_1(u) = \bot$, or $high_1(w) = \bot$, then there is no back-edge in $G \setminus E'$ that connects A, or C, respectively, with the rest of the graph. Therefore, we can use Lemma 6.4 in order to revert to the previous case (where E' contains three tree-edges).

Otherwise, if we have that either $high_1(u) \in B$, or $high_1(w) \in D$, then we have that A is connected with B, or C is connected with D, respectively, through a back-edge in $G \setminus E'$. Therefore, we add the edge $(\bar{u}, \bar{v})$, or $(\bar{w}, \bar{z})$, respectively, to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(u, p(u))\}$, or $E' \leftarrow E' \setminus \{(w, p(w))\}$, respectively, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2.

Similarly, if we have $high_1(u) \in D$ and $low_1(u) \in E$, then we have that A is connected with both D and E, through back-edges in $G \setminus E'$. Therefore, we add the edge $(\bar{z}, \bar{r})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(z, p(z))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2.

So let us assume that none of the above holds. This means that $high_1(w) \in E$, and either (1) $high_1(u) \in D$ and $low_1(u) \in D$, or (2) $high_1(u) \in E$.

Let us consider case (1) first. Then we add the edges $(\bar{w}, \bar{r})$ and $(\bar{u}, \bar{z})$ to $\mathcal{R}$, and it remains to determine the connectivity between B and D with the rest of the graph.

First, we check whether there is a back-edge that stems from B and ends in either D or E. This is equivalent to checking whether $bcount(v) > bcount(u)$ (since $high_1(u) \in D$ implies that $B(u) \subseteq B(v)$). If we have $bcount(v) = bcount(u)$, then we have that B is isolated from the rest of the graph in $G \setminus E'$. Thus, it remains to determine whether D is connected with E through a back-edge. This is equivalent to the condition $bcount(z) > bcount(w)$ (since $low_1(u) \notin E$ implies that $B(u) \cap B(z) = \emptyset$, and then $high_1(w) \in E$ implies that $B(w) \subseteq B(z)$). Thus, if this is satisfied, then we add the edge $(\bar{z}, \bar{r})$ to $\mathcal{R}$.

Otherwise, suppose that $bcount(v) > bcount(u)$. Let us assume, first, that $low(v) \in D$. Then we have that $B(v) = B(u) \sqcup (B, D)$. Thus, there is a back-edge from B to D, and therefore we add the edge $(\bar{v}, \bar{z})$ to $\mathcal{R}$. Then we set $E' \leftarrow E' \setminus \{(v, p(v))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. Now let us assume that $low(v) \in E$. Then, since $low(u) \in D$, we have that there is a back-edge from B to E. Thus, we insert the edge $(\bar{v}, \bar{r})$ to $\mathcal{R}$. Now it remains to determine whether $(B, D) \neq \emptyset$ or $(D, E) \neq \emptyset$. Notice that either of those cases implies that $G \setminus E'$ is connected (because $\mathcal{R}$ already contains the edges $(\bar{u}, \bar{z})$, $(\bar{w}, \bar{r})$ and $(\bar{v}, \bar{r})$, and either of those cases implies that we have to add the edge $(\bar{v}, \bar{z})$ or $(\bar{z}, \bar{r})$, respectively). Since $high_1(u) \in D$ and $low_1(u) \in D$, we have $B(u) = (A, D)$, $B(v) = (A, D) \cup (B, D) \cup (B, E)$, and $B(z) = (B, E) \cup (C, E) \cup (D, E)$. Since $high_1(w) \in E$, we have $B(w) = (C, E)$. Thus, we have the following:

- $bcount(u) = |(A, D)|$.
- $bcount(v) = |(A, D)| + |(B, D)| + |(B, E)|$.
- $bcount(w) = |(C, E)|$.
- $bcount(z) = |(B, E)| + |(C, E)| + |(D, E)|$.

This implies that $N = bcount(z) - bcount(w) - bcount(v) + bcount(u) = |(D, E)| - |(B, D)|$. Also, we have $s = SumAnc(z) - SumAnc(w) - SumAnc(v) + SumAnc(u) = SumAnc((D, E)) - SumAnc((B, D))$. Now, if $N \neq 0$, then at least one of (D, E) and (B, D) is not empty, and therefore $G \setminus E'$ is connected. Thus, it is sufficient to add one more edge to $\mathcal{R}$ in order to make it connected (e.g., we may add $(\bar{v}, \bar{z})$). Otherwise, we have $|(D, E)| = |(B, D)|$. Then, since the lower endpoint of every back-edge in (D, E) is lower than the lower endpoint of every back-edge in (B, D), we have that

$s < 0$ if and only if $|(D, E)| > 0$ (and $|(B, D)| > 0$). Thus, we add one more edge to $\mathcal{R}$ in order to make it connected (e.g., $(\bar{v}, \bar{z})$), if and only if $s < 0$.

Now let us consider case (2). Then we add the edges $(\bar{w}, \bar{r})$ and $(\bar{u}, \bar{r})$ to $\mathcal{R}$. Since $high_1(u) \in E$, we have that there is a back-edge from B to D if and only if $high_1(v) \in D$. In this case, we add the edge $(\bar{v}, \bar{z})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(v, p(v))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. So let us suppose that $high_1(v) \in E$. In this case, it may be that there is a back-edge from B to E. Since $high_1(u) \in E$ and $high_1(v) \in E$, the existence of a back-edge from B to E is equivalent to $bcount(v) > bcount(u)$. If that is the case, then we have that A is connected with B in $G \setminus E'$ through the mediation of E. Thus, we add the edge $(\bar{u}, \bar{v})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(u, p(u))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. So let us assume that there is no back-edge from B to E. Since $high_1(u) \in E$ and $high_1(v) \in E$, this implies that B is isolated from the rest of the graph in $G \setminus E'$. Then, it remains to check whether there is a back-edge from D to E. Since $high_1(u) \in E$ and $high_1(w) \in E$ (and $B(v) = B(u)$), we have that $B(z) = B(u) \sqcup B(w)$. Thus, there is a back-edge from D to E if and only if $bcount(z) > bcount(u) + bcount(w)$. In this case, we simply add the edge $(\bar{z}, \bar{r})$ to $\mathcal{R}$.

6.6.3 w is an ancestor of both u and v, and $\{u, v\}$ are not related as ancestor and descendant

Let $A = T(u)$, $B = T(v)$, $C = T(w) \setminus (T(u) \cup T(v))$, $D = T(z) \setminus T(w)$, and $E = T(r) \setminus T(z)$.

If we have that either $high_1(u) = \bot$, or $high_1(v) = \bot$, then there is no back-edge in $G \setminus E'$ that connects A, or B, respectively, with the rest of the graph. Therefore, we can use Lemma 6.4 in order to revert to the previous case (where E' contains three tree-edges).

Otherwise, if we have that either $high_1(u) \in C$, or $high_1(v) \in C$, then we have that A is connected with C, or B is connected with C, respectively, through a back-edge in $G \setminus E'$. Therefore, we add the edge $(\bar{u}, \bar{w})$, or $(\bar{v}, \bar{w})$, respectively, to $\mathcal{R}$. Then we set $E' \leftarrow E' \setminus \{(u, p(u))\}$, or $E' \leftarrow E' \setminus \{(v, p(v))\}$, respectively, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2.

Similarly, if we have that either $high_1(u) \in D$ and $low_1(u) \in E$, or $high_1(v) \in D$ and $low_1(v) \in E$, then we have that D remains connected with E in $G \setminus E'$, through the mediation of A or B, respectively. Therefore, we add the edge $(\bar{z}, \bar{r})$ to $\mathcal{R}$, we set

$E' \leftarrow E' \setminus \{(z, p(z))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. Also, if we have that $M(z) \in D$, then D is connected with E through a back-edge in $G \setminus E'$. Therefore, we add the edge $(\bar{z}, \bar{r})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(z, p(z))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2.

Thus, we may assume that none of the above is true. Therefore, we have that $M(z) \notin D$, and there are four possibilities to consider: either (1) $high_1(u) \in D$, $low_1(u) \in D$, $high_1(v) \in D$ and $low_1(v) \in D$, or (2) $high_1(u) \in D$, $low_1(u) \in D$, $high_1(v) \in E$ and $low_1(v) \in E$, or (3) $high_1(u) \in E$, $low_1(u) \in E$, $high_1(v) \in D$ and $low_1(v) \in D$, or (4) $high_1(u) \in E$, $low_1(u) \in E$, $high_1(v) \in E$ and $low_1(v) \in E$. In either of those cases, the problem will be to determine whether C is connected with either D or E through back-edges.

Let us consider case (1) first. In this case, we add the edges $(\bar{u}, \bar{z})$ and $(\bar{v}, \bar{z})$ to $\mathcal{R}$. If we have $low_1(w) \in D$, then there are no back-edges from C to E. Thus, in order to check whether there is a back-edge from C to D, it is sufficient to check whether $bcount(w) > bcount(u) + bcount(v)$ (because $high_1(u) \in D$ and $high_1(v) \in D$ imply that $B(u) \cup B(v) \subseteq B(w)$; this can be strengthened to $B(u) \sqcup B(v) \subseteq B(w)$, since u and v are not related as ancestor and descendant). If that is the case, then we add the edge $(\bar{w}, \bar{z})$ to $\mathcal{R}$. (Otherwise, there is nothing to do.) On the other hand, if we have $low_1(w) \in E$, then there is a back-edge from C to E (since $low_1(u) \in D$ and $low_1(v) \in D$). Thus, we add the edge $(\bar{w}, \bar{r})$ to $\mathcal{R}$. Now it remains to determine whether there is a back-edge that connects C with D. We claim that this is equivalent to checking whether $bcount(w) > bcount(u) + bcount(v) + bcount(z)$. To see this, first notice that $B(u) \sqcup B(v) \subseteq B(w)$ (since $high_1(u) \in D$ and $high_1(v) \in D$). We also have that $(B(u) \cup B(v)) \cap B(z) = \emptyset$ (since $low_1(u) \in D$ and $low_1(v) \in D$). We also have $B(z) \subseteq B(w)$ (since $M(z) \notin D$). This shows that $B(u) \sqcup B(v) \sqcup B(z) \subseteq B(w)$. Finally, notice that $B(w) \setminus (B(u) \cup B(v) \cup B(z))$ contains precisely the back-edges that connect C with D. Thus, it is sufficient to check whether $bcount(w) > bcount(u) + bcount(v) + bcount(z)$. If that is the case, then we add the edge $(\bar{w}, \bar{z})$ to $\mathcal{R}$.

Now let us consider case (2). In this case, we add the edges $(\bar{u}, \bar{z})$ and $(\bar{v}, \bar{r})$ to $\mathcal{R}$. Now we have to determine whether there is a back-edge from C to D, or from C to E. We claim that there is a back-edge from C to E if and only if $M(z) \in C$. The necessity is obvious (since $M(z) \notin D$). To see the sufficiency, notice that, since $low_1(u) \in D$, we have that all back-edges in $B(z)$ have their higher endpoint either

in B, or in C, or in D. The last case is rejected, since $M(z) \notin D$. If all the back-edges in $B(z)$ have their higher endpoint in B, then $M(z) \in B$. Thus, if we have $M(z) \in C$, then at least one back-edge in $B(z)$ must stem from C. Thus, if we have $M(z) \in C$, then we add the edge $(\bar{w}, \bar{r})$ to $\mathcal{R}$. Now it remains to determine whether there is a back-edge from C to D. Notice that $B(w)$ can be partitioned into: the back-edges in $B(u)$ (since $high_1(u) \in D$), the back-edges in $B(v)$ (since $high_1(v) \in E$), the back-edges from C to D, and the back-edges from C to E. Since $low_1(u) \in D$ and $M(z) \notin D$, we have $(C, E) = B(z) \setminus B(v)$. Thus, in order to determine whether there is a back-edge from C to D, it is sufficient to check whether $bcount(w) > bcount(u) + bcount(v) + (bcount(z) - bcount(v)) = bcount(u) + bcount(z)$. If that is the case, then we add the edge $(\bar{w}, \bar{z})$ to $\mathcal{R}$. (Otherwise, there is nothing to do.) On the other hand, if $M(z) \notin C$, then we know that there is no back-edge from C to E. Thus, in order to determine whether there is a back-edge from C to D, it is sufficient to check whether $bcount(w) > bcount(u) + bcount(v)$. If that is the case, then we add the edge $(\bar{w}, \bar{z})$ to $\mathcal{R}$. Case (3) is treated with a similar argument.

Finally, let us consider case (4). In this case, we add the edges $(\bar{u}, \bar{r})$ and $(\bar{v}, \bar{r})$ to $\mathcal{R}$. Then, notice that there is a back-edge from C to D if and only if $high_1(w) \in D$ (since $high_1(u) \in E$ and $high_1(v) \in E$). If that is the case, then we add the edge $(\bar{w}, \bar{z})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(w, p(w))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. Otherwise, all the back-edges in $B(w)$ that stem from C (if there are any), end in E. Thus, in order to determine whether such a back-edge exists, we simply check whether $bcount(w) > bcount(u) + bcount(v)$ (because $B(u) \sqcup B(v) \subseteq B(w)$). If that is the case, then we add the edge $(\bar{w}, \bar{r})$ to $\mathcal{R}$.

6.6.4 w is an ancestor of v, and v is an ancestor of u

Let $A = T(u)$, $B = T(v) \setminus T(u)$, $C = T(w) \setminus T(v)$, $D = T(z) \setminus T(w)$, and $E = T(r) \setminus T(z)$.

If we have that either $high_1(u) \in B$, or $M(z) \in D$, then we have that A is connected with B, or D is connected with E, respectively, through a back-edge. Thus, we insert the edge $(\bar{u}, \bar{v})$, or $(\bar{z}, \bar{r})$, respectively, to $\mathcal{R}$. Then we set $E' \leftarrow E' \setminus \{(u, p(u))\}$, or $E' \leftarrow E' \setminus \{(z, p(z))\}$, respectively, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2.

If we have that either $high_1(u) \in C$ and $low_1(u) \in D$, or $high_1(u) \in D$ and $low_1(u) \in E$, then we have that C is connected with D, or D is connected with E, respectively,

through the mediation of A. Thus, we insert the edge $(\bar{w},\bar{z})$, or $(\bar{z},\bar{r})$, respectively, to $\mathcal{R}$. Then we set $E' \leftarrow E' \setminus \{(w,p(w))\}$, or $E' \leftarrow E' \setminus \{(z,p(z))\}$, respectively, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2.

So let us assume that neither of the above is true. (In particular, we have $M(z) \notin D$, and therefore $B(z) = (A,E) \cup (B,E) \cup (C,E)$.) Then there are four cases to consider. Either (1) $high_1(u) \in C$ and $low_1(u) \in C$, or (2) $high_1(u) \in C$ and $low_1(u) \in E$, or (3) $high_1(u) \in D$ and $low_1(u) \in D$, or (4) $high_1(u) \in E$ and $low_1(u) \in E$.

Let us consider case (1) first. Then we add the edge $(\bar{u},\bar{w})$ to $\mathcal{R}$. Notice that $high_1(v) \in C$. Thus, there are three different cases to consider. Either (1.1) $low_1(v) \in C$, or (1.2) $low_1(v) \in D$, or (1.3) $low_1(v) \in E$. Let us consider case (1.1). Then, there is a back-edge from B to C if and only if $bcount(v) > bcount(u)$. If that is the case, then we add the edge $(\bar{v},\bar{w})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(v,p(v))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. Otherwise, we have that B is isolated from the rest of the graph in $G \setminus E'$. It remains to determine whether C is connected with D or E. If $bcount(z) = 0$, then there is no back-edge from C to E. Then, since $low(v) \in C$, we have that $B(w) = (C,D)$. Thus, there is a back-edge from C to D if and only if $bcount(w) > 0$. If that is the case, then we add the edge $(\bar{w},\bar{z})$ to $\mathcal{R}$ (and we are done). So let us assume that $bcount(z) > 0$. Then, since $M(z) \notin D$ and $low(v) \in C$, we have that $M(z) \in C$. Thus, there is a back-edge from C to E, and so we add the edge $(\bar{w},\bar{r})$ to $\mathcal{R}$. Since $M(z) \in C$ and $low(v) \in C$, we have that $B(z) = (C,E)$. Furthermore, we have $B(w) = (C,D) \cup (C,E)$. Thus, there is a back-edge from C to D if and only if $bcount(w) > bcount(z)$. If that is the case, then we add the edge $(\bar{w},\bar{z})$ to $\mathcal{R}$.

Now let us consider case (1.2). This means that B is connected with D with a back-edge, and so we add the edge $(\bar{v},\bar{z})$ to $\mathcal{R}$. Notice that $high_1(w) \in D$. Thus, there are two cases to consider. Either (1.2.1) $low_1(w) \in D$, or (1.2.2) $low_1(w) \in E$. Let us consider case (1.2.1). Then, since $M(z) \notin D$, we have that there are no back-edges from D to E, and therefore $B(z) = \emptyset$ (since $low(w) \in D$). This means that E is isolated from the rest of the graph in $G \setminus E'$. Thus, we can use Lemma 6.4 in order to revert to the previous case (where E' contains three tree-edges). Now let us consider case (1.2.2). Then there is a back-edge from C to E, and so we add the edge $(\bar{w},\bar{r})$ to $\mathcal{R}$. Thus far, $\mathcal{R}$ contains the edges $(\bar{u},\bar{w})$, $(\bar{v},\bar{z})$ and $(\bar{w},\bar{r})$. It remains to determine whether $(B,C) \neq \emptyset$ or $(C,D) \neq \emptyset$. Observe that either of those cases implies that $G \setminus E'$

is connected, and therefore it is sufficient to add one more edge to $\mathcal{R}$ in order to make it connected (e.g., we may add $(\bar{v},\bar{w})$). Since $high_1(u) \in C$ and $low_1(u) \in C$, we have $B(u) = (A,C)$. Then, since $low_1(v) \in D$, we have $B(v) = (A,C) \cup (B,C) \cup (B,D)$. We also have $B(w) = (B,D) \cup (C,D) \cup (C,E)$, and $B(z) = (C,E)$ (since $M(z) \notin D$). Thus, we have the following:

- $bcount(u) = |(A,C)|$.
- $bcount(v) = |(A,C)| + |(B,C)| + |(B,D)|$.
- $bcount(w) = |(B,D)| + |(C,D)| + |(C,E)|$.
- $bcount(z) = |(C,E)|$.

This implies that $N = bcount(w) - bcount(z) - bcount(v) + bcount(u) = |(C,D)| - |(B,C)|$. Furthermore, we have $s = SumAnc(w) - SumAnc(z) - SumAnc(v) + SumAnc(u) = SumAnc((C,D)) - SumAnc((B,C))$. Thus, if $N \neq 0$, then at least one of (C,D) and (B,C) is not empty, and therefore it is sufficient to add e.g. $(\bar{v},\bar{w})$ to $\mathcal{R}$ in order to make it connected. Otherwise, if $N = 0$, then we have $|(C,D)| = |(B,C)|$. Then, since the lower endpoint of every back-edge in (C,D) is lower than the lower endpoint of every back-edge in (B,C), we have that $s = SumAnc((C,D)) - SumAnc((B,C)) < 0$ if and only if $(C,D) \neq \emptyset$ (and $(B,C) \neq \emptyset$). Thus, in the case $N = 0$, we add the edge $(\bar{v},\bar{w})$ to $\mathcal{R}$ if and only if $s < 0$.

Now let us consider case (1.3). Then, there is a back-edge from B to E, and so we add the edge $(\bar{v},\bar{r})$ to $\mathcal{R}$. Now there are two cases to consider. Either (1.3.1) $high_1(w) \in D$, or (1.3.2) $high_1(w) \in E$. Let us consider case (1.3.1). Then, since $low_1(u) \in C$, we have that the $high_1$-edge of w either stems from B or from C. If the $high_1$-edge of w stems from B, then we have that D and E remain connected in $G \setminus E'$ through the mediation of B. Thus, we add the edge $(\bar{z},\bar{r})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(z,p(z))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. Otherwise, if the $high_1$-edge of w stems from C, then we have that this is a back-edge from C to D. Thus, we add the edge $(\bar{w},\bar{z})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(w,p(w))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. Now let us consider case (1.3.2). In this case, we have that D is isolated from the rest of the graph in $G \setminus E'$, and therefore $\bar{z}$ should be isolated in $\mathcal{R}$. Furthermore, all the back-edges in $B(w)$ that stem from C (if there are any), end in E. If $M(z) \in C$, then there is a back-edge from C to E. Thus, we add the edge $(\bar{w},\bar{r})$

to $\mathcal{R}$, and we are done, because now $\mathcal{R}$ has enough edges to make the vertices $\bar{u}$, $\bar{v}$, $\bar{w}$ and $\bar{r}$ connected. Otherwise, let us assume that $M(z) \notin C$. Then there is no back-edge from C to E. Since $\mathit{high}(w) \in E$, there is no back-edge from B to D, or from C to D. Notice that we have $B(u) = (A, C)$, $B(v) = (A, C) \cup (B, C) \cup (B, E)$ and $B(w) = (B, E)$. Thus, there is a back-edge from B to C, if and only if $\mathit{bcount}(v) > \mathit{bcount}(u) + \mathit{bcount}(w)$. If that is the case, then we add the edge $(\bar{v}, \bar{w})$ to $\mathcal{R}$. Thus, we have exhausted all possibilities for case (1.3).

Now let us consider case (2). Then, we have that A is connected with both C and E in $G \setminus E'$, and so we add the edges $(\bar{u}, \bar{w})$ and $(\bar{u}, \bar{r})$ to $\mathcal{R}$. First, we check whether there is back-edge from B to E. Such a back-edge exists if and only if there is a back-edge $(x, y) \in B(v) \setminus B(u)$ such that $y \leq p(z)$. Thus, we can determine the existence of such a back-edge in constant time, using the data structure from Lemma 3.20 (we assume that we have performed the linear-time preprocessing that is required in order to build this data structure). If there is a back-edge from B to E, then we have that A and B are connected in $G \setminus E'$ through the mediation of E. Thus, we add the edge $(\bar{u}, \bar{v})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(u, p(u))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. Otherwise, suppose that there is no back-edge from B to E. Then, we check whether there is a back-edge $(x, y) \in B(v) \setminus B(u)$ such that $y \leq p(w)$. (Again, we can perform this check in constant time using Lemma 3.20.) If that is the case, then, since there is no back-edge from B to E, we have that (x, y) is a back-edge from B to D. Otherwise, we have that there is no back-edge from B to D. So here we can distinguish in constant time two cases: either (2.1) there is a back-edge from B to D, or (2.2) there is no back-edge from B to D.

Let us consider case (2.1). Then, we add the edge $(\bar{v}, \bar{z})$ to $\mathcal{R}$. We will determine whether there is a back-edge from A to D, or a back-edge from B to C, or a back-edge from C to D. Notice that, in either of those cases, we have that $G \setminus E'$ is connected, and so it is sufficient to return a connected graph $\mathcal{R}$. Otherwise, we have that $\mathcal{R}$ is a connectivity graph for $G \setminus E'$. (It is irrelevant whether there is a back-edge from C to E, because this does not add any new connectivity information, since we know that the parts C and E are connected in $G \setminus E'$ through the mediation of A.) Since $\mathit{high}_1(u) \in C$, we have that $B(u) = (A, C) \cup (A, D) \cup (A, E)$. Since there is no back-edge from B to E, we have that $B(v) = (A, C) \cup (A, D) \cup (A, E) \cup (B, C) \cup (B, D)$ and $B(w) = (A, D) \cup (A, E) \cup (B, D) \cup (C, D) \cup (C, E)$. And since $M(z) \notin D$ and there is

no back-edge from B to E, we have that $B(z) = (A, E) \cup (C, E)$. Thus, we have the following:

- $\mathit{bcount}(u) = |(A, C)| + |(A, D)| + |(A, E)|$.
- $\mathit{bcount}(v) = |(A, C)| + |(A, D)| + |(A, E)| + |(B, C)| + |(B, D)|$.
- $\mathit{bcount}(w) = |(A, D)| + |(A, E)| + |(B, D)| + |(C, D)| + |(C, E)|$.
- $\mathit{bcount}(z) = |(A, E)| + |(C, E)|$.

This implies that $\mathit{bcount}(z) - \mathit{bcount}(w) + \mathit{bcount}(v) - \mathit{bcount}(u) = |(B, C)| - |(C, D)| - |(A, D)|$. Thus, if we have that $\mathit{bcount}(z) - \mathit{bcount}(w) + \mathit{bcount}(v) - \mathit{bcount}(u) \neq 0$, then at least one of (B, C), (C, D) or (A, D) is not empty. Thus, it suffices to return a connected graph $\mathcal{R}$. Otherwise, suppose that $\mathit{bcount}(z) - \mathit{bcount}(w) + \mathit{bcount}(v) - \mathit{bcount}(u) = 0$. Then we have that $|(B, C)| = |(C, D)| + |(A, D)|$. Consider the value $\mathit{SumAnc}(z) - \mathit{SumAnc}(w) + \mathit{SumAnc}(v) - \mathit{SumAnc}(u) = \mathit{SumAnc}(B, C) - \mathit{SumAnc}(C, D) - \mathit{SumAnc}(A, D)$. If we have that $|(B, C)| = 0$, then we also have that $|(C, D)| + |(A, D)| = 0$, and therefore $\mathit{SumAnc}(B, C) - \mathit{SumAnc}(C, D) - \mathit{SumAnc}(A, D) = 0$. Otherwise, if $|(B, C)| > 0$, then we have that $\mathit{SumAnc}(B, C) > \mathit{SumAnc}(C, D) + \mathit{SumAnc}(A, D)$, because $|(B, C)| = |(C, D)| + |(A, D)|$ and the lower endpoint of every back-edge in (B, C) is greater than, or equal to, w, whereas the lower endpoint of every back-edge in $(C, D) \cup (A, D)$ is lower than w. This implies that $\mathit{SumAnc}(B, C) - \mathit{SumAnc}(C, D) - \mathit{SumAnc}(A, D) > 0$. Thus, we have shown that $|(B, C)| > 0$ if and only if $\mathit{SumAnc}(B, C) - \mathit{SumAnc}(C, D) - \mathit{SumAnc}(A, D) > 0$. Thus, if $\mathit{bcount}(z) - \mathit{bcount}(w) + \mathit{bcount}(v) - \mathit{bcount}(u) = 0$, then it is sufficient to check whether $\mathit{SumAnc}(z) - \mathit{SumAnc}(w) + \mathit{SumAnc}(v) - \mathit{SumAnc}(u) > 0$. If that is the case, then we only have to add one more edge to $\mathcal{R}$ in order to make it connected (e.g., $(\bar{u}, \bar{z})$). Otherwise, we have that all of the sets (A, D), (B, C) and (C, D) are empty, and $\mathcal{R}$ is already a connectivity graph for $G \setminus E'$.

Now let us consider case (2.2). Since there is no back-edge from B to E or from B to D, it remains to check whether there is a back-edge from B to C. Since $\mathit{high}_1(u) \in C$, we have that $B(u) \subseteq B(v)$. Thus, there is a back-edge in $B(v) \setminus B(u)$ (and therefore a back-edge from B to C), if and only if $\mathit{bcount}(v) > \mathit{bcount}(u)$. If that is the case, then we add the edge $(\bar{v}, \bar{w})$ to $\mathcal{R}$. Then we set $E' \leftarrow E' \setminus \{(v, p(v))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. Otherwise, let us assume that $B(v) = B(u)$. This implies that B is isolated from the rest of the graph in $G \setminus E'$. Now it remains to determine whether there is a back-edge

from A to D, or from C to D. Since $B(v) = B(u)$, this is equivalent to $high_1(w) \in D$. If that is the case, then at least one of (A, D) and (C, D) is not empty. Thus, we have that the parts A, C, D and E, are connected in $G \setminus E'$. Therefore, it is sufficient to add the edge $(\bar{u}, \bar{z})$ to $\mathcal{R}$, in order to have a connectivity graph for $G \setminus E'$. Otherwise, if $high_1(w) \notin D$, then $\mathcal{R}$ is already a connectivity graph for $G \setminus E'$.

Now let us consider case (3). Then we add the edge $(\bar{u}, \bar{z})$ to $\mathcal{R}$. Since $high_1(u) \in D$, we have that, if $high_1(v) \in C$, then there is a back-edge from B to C. Then, we add the edge $(\bar{v}, \bar{w})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(v, p(v))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. So let us assume that $high_1(v) \notin C$. Here we distinguish two cases: either (3.1) $low(v) \in D$, or (3.2) $low(v) \in E$.

Let us consider case (3.1). Then we have that $B(v) = (A, D) \cup (B, D)$. Thus, we have that there is a back-edge from B to D if and only if $bcount(v) > bcount(u)$. If that is the case, then we have that A remains connected with B in $G \setminus E'$ through the mediation of D. Thus, we add the edge $(\bar{u}, \bar{v})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(u, p(u))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. So let us assume that there is no back-edge from B to D. Thus, B is isolated from the rest of the graph in $G \setminus E'$. Now it remains to determine whether there is a back-edge from C to D, or from C to E. We distinguish two cases: (3.1.1) $low(w) \in D$, or (3.1.2) $low(w) \in E$. Let us consider case (3.1.1). Then, we have that $B(w) = (A, D) \cup (C, D)$. Thus, there is a back-edge from (C, D) if and only if $bcount(w) > bcount(u)$. If that is the case, then we simply add the edge $(\bar{w}, \bar{z})$ to $\mathcal{R}$. Now let us consider case (3.1.2). Then, since $low(v) \in D$ and $M(z) \notin D$, we have that there is a back-edge from C to E. Thus, we add $(\bar{w}, \bar{r})$ to $\mathcal{R}$. Notice that we have $B(z) = (C, E)$, and $B(w) = (A, D) \cup (C, D) \cup (C, E)$. Thus, there is a back-edge from C to D if and only if $bcount(w) > bcount(u) + bcount(z)$. If that is the case, then we add the edge $(\bar{w}, \bar{z})$ to $\mathcal{R}$.

Now let us consider case (3.2). Then, since $low(u) \in D$, there is a back-edge from B to E. Thus, we add the edge $(\bar{v}, \bar{r})$ to $\mathcal{R}$. If $M(z) \in C$, then there is a back-edge from C to E, and therefore B and C remain connected in $G \setminus E'$ through the mediation of E. Thus, we add the edge $(\bar{v}, \bar{w})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(v, p(v))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. So let us suppose that $M(z) \notin C$. Then it remains to determine whether there is a back-edge from B to D, or from C to D. Since $high_1(u) \in D$

and $low_1(u) \in D$, we have $B(u) = (A, D)$. Since $M(z) \notin C$ and $low(v) \in E$, we have that $B(z) = (B, E)$. Furthermore, we have $B(v) = (A, D) \cup (B, D) \cup (B, E)$. Thus, there is a back-edge from B to D if and only if $bcount(v) > bcount(u) + bcount(z)$. If that is the case, then we add the edge $(\bar{v}, \bar{z})$ to $\mathcal{R}$. Since $M(z) \notin C$, we have that $B(w) = (A, D) \cup (B, D) \cup (B, E) \cup (C, D)$. Thus, there is a back-edge from C to D if and only if $bcount(w) > bcount(v)$. If that is the case, then we add the edge $(\bar{w}, \bar{z})$ to $\mathcal{R}$.

Now let us consider case (4). Then there is a back-edge from A to E, and so we add the edge $(\bar{u}, \bar{r})$ to $\mathcal{R}$. If we have that $high_1(v) \in C$, then B is connected with C in $G \setminus E'$, and so we add the edge $(\bar{v}, \bar{w})$ to $\mathcal{R}$. Then we set $E' \leftarrow E' \setminus \{(v, p(v))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. Thus, we may assume that $high_1(v) \notin C$. Then there are two cases to consider: either (4.1) $high_1(v) \in D$, or (4.2) $high_1(v) \in E$.

Now let us consider case (4.1). Then there is a back-edge from B to D, and so we add the edge $(\bar{v}, \bar{z})$ to $\mathcal{R}$. Now we distinguish three cases, depending on the location of $M(z)$. Since $M(z) \notin D$, we have that either $(4.1.1)$ $M(z) \in A$, or $(4.1.2)$ $M(z) \in B$, or $(4.1.3)$ $M(z) \in C$. Let us consider case $(4.1.1)$. Then we have that $B(v) = (A, E) \cup (B, D)$. Furthermore, there is no back-edge from C to E. Thus, it remains to determine whether there is a back-edge from C to D. Notice that we have $B(w) = (C, D) \cup (B, D) \cup (A, E)$. Thus, $(C, D) = B(w) \setminus B(v)$. Thus, there is a back-edge from C to D, if and only if $bcount(w) > bcount(v)$. In this case, we simply add the edge $(\bar{w}, \bar{z})$ to $\mathcal{R}$. Now let us consider case $(4.1.2)$. This implies that there is a back-edge from B to E, and therefore A and B are connected in $G \setminus E'$ through the mediation of E. Thus, we add the edge $(\bar{u}, \bar{v})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(u, p(u))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. Now let us consider case $(4.1.3)$. This implies that there is a back-edge from C to E, and so we add the edge $(\bar{w}, \bar{r})$ to $\mathcal{R}$. It remains to determine whether there is a back-edge from B to E, or a back-edge from C to D. Notice that either case implies that $G \setminus E'$ is connected. Thus, we only need to check whether $(B, E) \cup (C, D) \neq \emptyset$. Since $high_1(u) \in E$, we have that $B(u) = (A, E)$. Since $high_1(v) \in D$, we have that $B(v) = (A, E) \cup (B, D) \cup (B, E)$. Furthermore, we have $B(w) = (A, E) \cup (B, D) \cup (B, E) \cup (C, D) \cup (C, E)$, and $B(z) = (A, E) \cup (B, E) \cup (C, E)$. Thus, we have the following:

- $|B(u)| = |(A, E)|$.

- $|B(v)| = |(A,E)| + |(B,D)| + |(B,E)|$.
- $|B(w)| = |(A,E)| + |(B,D)| + |(B,E)| + |(C,D)| + |(C,E)|$.
- $|B(z)| = |(A,E)| + |(B,E)| + |(C,E)|$.

This implies that $|B(z)|-|B(w)|+|B(v)|-|B(u)| = |(B,E)|-|(C,D)|$. Thus, if we have that $\mathit{bcount}(z) - \mathit{bcount}(w) + \mathit{bcount}(v) - \mathit{bcount}(u) \neq 0$, then we can be certain that one of (B,E) and (C,D) is not empty, and therefore $G \setminus E'$ is connected. Thus, it suffices to add either $(\bar{v},\bar{r})$ or $(\bar{w},\bar{z})$ to $\mathcal{R}$, in order to make it connected. Otherwise, if $|(B,E)| - |(C,D)| = 0$, then we must use other means in order to determine whether one of (B,E) and (C,D) is non-empty. For this purpose, we consider the values *SumAnc* of those sets. Thus, we have that $\mathit{SumAnc}(B,E) - \mathit{SumAnc}(C,D) = \mathit{SumAncB}(z) - \mathit{SumAncB}(w) + \mathit{SumAncB}(v) - \mathit{SumAncB}(u)$. Then, if both (B,E) and (C,D) are empty, we have that $\mathit{SumAnc}(B,E) - \mathit{SumAnc}(C,D) = 0$. Otherwise, since (B,E) and (C,D) have the same number of back-edges, and since the lower endpoints of all back-edges in (B,E) are lower than the lower endpoints of all back-edges in (C,D), we have that $\mathit{SumAnc}(B,E) - \mathit{SumAnc}(C,D) < 0$. Thus, we have that both (B,E) and (C,D) are non-empty if and only if $\mathit{SumAnc}(B,E) - \mathit{SumAnc}(C,D) < 0$. If that is the case, then it is sufficient to return a connected graph $\mathcal{R}$.

Now let us consider case (4.2). Since $\mathit{high}_1(u) \in E$, we have that $B(u) \subseteq B(v)$. And since $\mathit{high}_1(v) \in E$, we have that $B(v) = B(u) \sqcup (B,E)$. Thus, there is a back-edge from B to E if and only if $\mathit{bcount}(v) > \mathit{bcount}(u)$. If that is the case, then we have that A is connected with B in $G \setminus E'$ through the mediation of E. Thus, we add the edge $(\bar{u},\bar{v})$ to $\mathcal{R}$, we set $E' \leftarrow E' \setminus \{(u,p(u))\}$, and we revert to the previous case (where E' contains three tree-edges), according to Lemma 6.2. So let us assume that there is no back-edge from B to E. This implies that B is isolated from the remaining parts in $G \setminus E'$. Now, since $M(z) \notin D$, we have that either (4.2.1) $M(z) \in A$, or (4.2.2) $M(z) \in C$. (The case $M(z) \in B$ is rejected, since $(B,E) = \emptyset$.) Let us consider case (4.2.1). Then, there is no back-edge from C to E, and it just remains to determine whether there is a back-edge from C to D. We have that $B(u) = (A,E)$ and $B(w) = (A,E) \cup (C,D)$. Thus, there is a back-edge from C to D if and only if $\mathit{bcount}(w) > \mathit{bcount}(u)$. If that is the case, then we simply add the edge $(\bar{w},\bar{z})$ to $\mathcal{R}$. Now let us consider case (4.2.2). In this case, there is a back-edge from C to E, and so we add the edge $(\bar{w},\bar{r})$ to $\mathcal{R}$. It remains to determine whether there is a back-edge from C to D. Since $\mathit{high}_1(u) \in E$

and $high_1(v) \in E$, we have that there is a back-edge from C to D if and only if $high_1(w) \in D$. If that is the case, then we simply add the edge $(\bar{w}, \bar{z})$ to $\mathcal{R}$.

6.7 The data structure

According to the preceding analysis, in order to be able to answer connectivity queries in the presence of at most four edge-failures in constant time, it is sufficient to have computed the following items:

- A DFS-tree of the graph rooted at a vertex r.
- The values $ND(v)$, $bcount(v)$, $M(v)$ and $SumAnc(v)$, for all vertices $v \neq r$.
- The low_1, low_2, low_3, low_4, $high_1$, $high_2$ and $high_3$ edges of v, for every vertex $v \neq r$.
- The data structure in Lemma 3.20 for answering back-edge queries.

Thus, we need $O(n)$ space to store all these items, and the results of Section 3 imply that we can compute all of them in linear time in total.

Finally, let us describe how to answer the queries. First, given the set of edges E' that failed (with $|E'| \leq 4$), we build a connectivity graph $\mathcal{R}$ of $G \setminus E'$, by going through the case analysis that is described in the preceding sections. This takes $O(1)$ time in total. Now let x and y be the two query vertices. Then we determine the root of the connected component of $T \setminus E'$ that contains x. This is given either by the largest higher endpoint of the tree-edges in E' that is an ancestor of x, or by r if there is no tree-edge in E' whose higher endpoint is an ancestor of x. Thus, retrieving this root takes $O(1)$ time. Then we do the same for the other query vertex too, and let r_1 and r_2 be the two roots that we have gathered. Then we have that x is connected with y in $G \setminus E'$ if and only if $\bar{r}_1$ is connected with $\bar{r}_2$ in $\mathcal{R}$. Since the connected components of $\mathcal{R}$ can be computed in $O(1)$ time, we can have the answer in constant time.

Chapter 7

Connectivity Queries under Vertex Failures

7.1 Introduction

7.2 Preliminaries

7.3 The algorithm for vertex failures

7.1 Introduction

In this chapter we deal with the following problem. Given an undirected graph G with n vertices and m edges, and a fixed integer $d_\star$ ($d_\star \ll n$), the goal is to construct a data structure $\mathcal{D}$ that can be used in order to answer connectivity queries in the presence of at most $d_\star$ vertex-failures. More precisely, given a set of vertices F, with $|F| \leq d_\star$, we must be able to efficiently derive an oracle from $\mathcal{D}$, which can efficiently answer queries of the form "are the vertices x and y connected in $G \setminus F$?". In this problem, we want to simultaneously optimize the following parameters: (1) the construction time of $\mathcal{D}$ (preprocessing time), (2) the space usage of $\mathcal{D}$, (3) the time to derive the oracle from $\mathcal{D}$ given F (update time), and (4) the time to answer a connectivity query in $G \setminus F$. This problem is very well motivated; it has attracted the attention of researchers for more than a decade now, and it has many interesting variations. For more details, we refer to [26] and [49].

7.1.1 Previous work

Despite being extensively studied, it is only very recently that an almost optimal solution was provided by Long and Saranurak [49]. Specifically, they provided a deterministic algorithm that has $\hat{O}(m)+\tilde{O}(d_\star m)$ preprocessing time, uses $O(m \log^* n)$ space, and has $\hat{O}(d^2)$ update time and $O(d)$ query time.[1] This improves on the previous best deterministic solution by Duan and Pettie [26], that has $O(mn \log n)$ preprocessing time, uses $O(d_\star m \log n)$ space, and has $O(d^3 \log^3 n)$ update time and $O(d)$ query time. We note that there are more solutions to this problem, that optimize some parameters while sacrifising others (e.g., in the solution of Pilipczuk et al. [55], there is no dependency on n in the update time, but this is superexponential in $d_\star$, and the preprocessing time is $O(mn^2 2^{2^{O(d_\star)}})$). We refer to Table 1 in reference [49] for more details on the best known (upper) bounds for this problem. We also refer to Theorem 1.2 in [49] for a summary of known (conditional) lower bounds, that establish the optimality of [49].

7.1.2 Our contribution

The bounds that we mentioned are the best known for a deterministic solution. In practice, one would prefer the solution of Long and Saranurak [49], because that of Duan and Pettie [26] has preprocessing time $O(mn \log n)$, which can be prohibitively slow for large enough graphs. However, the solution in [49] is highly complicated, and it seems very difficult to be implemented efficiently. This is a huge gap between theory and practice. Furthermore, the (hidden) dependence on n in the time-bounds of [49] is not necessarily optimal if we assume that $d_\star$ is a constant for our problem. We note that this is a problem with various parameters, and thus it is very difficult to optimize all of them simultaneously.

Considering that this is a fundamental connectivity problem, we believe that it is important to have a solution that is relatively simple to describe and analyze, compares

[1]The symbol $\hat{O}$ hides subpolynomial (i.e. $n^{o(1)}$) factors, and $\tilde{O}$ hides polylogarithmic factors. The hidden expressions in the time-bounds are not specified by the authors in their overview. Also, the description for the $\log^* n$ function that appears in the space complexity is that it “can be substituted with any slowly growing function”. One thing that is explicitly stated, however, is that the hidden subpolynomial factors are worse than polylogarithmic. We must emphasize that the difficulty in stating the precise bounds is partly due to there being various trade-offs in the functions involved, and is partly indicative of the complexity of the techniques that are used.

very well with the best known bounds (even improves them in some respects), opens a new direction to settle the complexity of the problem, and can be readily implemented efficiently.

In this chapter, we exhibit a solution that has precisely those characteristics. We present a deterministic algorithm that has preprocessing time $O(d_\star m \log n)$, uses space $O(d_\star m \log n)$, and has $O(d^4 \log n)$ update time and $O(d)$ query time.[2] Our approach is arguably the simplest that has been proposed for this problem. The previous solutions rely on sophisticated tree decompositions of the original graph. Here, instead, we basically rely on a single DFS-tree, and we simply analyze its connected components after the removal of a set of vertices. It turns out that there is enough structure to allow for an efficient solution (see Section 7.3.2).

Table 7.1: Comparison of the best-known deterministic bounds. We note that m can be replaced with $\bar{m} = min\{m, d_\star n\}$, using the sparsification of Nagamochi and Ibaraki [51]. The data structure of Pilipczuk et al. does not support an update phase, but answers queries directly, given a set of (at most $d_\star$) failed vertices and two query vertices.

	Preprocessing	Space	Update	Query
Pilipczuk et al. [55]	$O(2^{2^{O(d_\star)}} mn^2)$	$O(2^{2^{O(d_\star)}} m)$	–	$O(2^{2^{O(d_\star)}})$
Duan and Pettie [26]	$O(mn \log n)$	$O(d_\star m \log n)$	$O(d^3 \log^3 n)$	$O(d)$
Long and Saranurak [49]	$\hat{O}(m) + \tilde{O}(d_\star m)$	$O(m \log^* n)$	$\hat{O}(d^2)$	$O(d)$
This chapter	$O(d_\star m \log n)$	$O(d_\star m \log n)$	$O(d^4 \log n)$	$O(d)$

The bounds that we provide compare very well with the previous best, especially considering the simplicity of our approach. (See Tables 8.1 and 8.2.) In fact, as we can see in Table 8.1, our solution is the best choice for implementations, considering that the algorithm of Long and Saranurak is very difficult to be implemented within the claimed time-bounds. Furthermore, if we assume that $d_\star$ is a constant ($d_\star \geq 4$), then, as we can see in Table 8.2, our algorithm provides some trade-offs, that improve the state of the art in some respects.

[2]The log factors in the space usage and the time for the updates can be improved with the use of more sophisticated 2D-range-emptiness data structures, such as those in [13].

Table 7.2: Comparison of the best-known deterministic bounds, when $d_\star$ is a fixed (small) constant. Although the algorithm of Pilipczuk et al. has the best space and query-time bounds, it has very large preprocessing time. Our solution has the best preprocessing time, and also better update time compared to the solutions of [26] and [49]. Furthermore, our space usage is almost linear.

	Preprocessing	Space	Update	Query
Pilipczuk et al. [55]	$O(mn^2)$	$O(m)$	–	$O(1)$
Duan and Pettie [26]	$O(mn \log n)$	$O(m \log n)$	$O(\log^3 n)$	$O(1)$
Long and Saranurak [49]	$\hat{O}(m) + \tilde{O}(m)$	$O(m \log^* n)$	$\hat{O}(1)$	$O(1)$
This chapter	$O(m \log n)$	$O(m \log n)$	$O(\log n)$	$O(1)$

Finally, the data structure that we provide is flexible with respect to $d_\star$: it can be adapted to increases and decreases, in time and space that are almost proportional to the change in $d_\star$ and the size of the graph (see Corollary 7.2). We do not know if any of the previous solutions has this property. It is a natural question whether we can efficiently update the data structure so that it can handle more failures (or less, and thereby free some space). As far as we know, we are the first to take notice of this aspect of the problem.

7.2 Preliminaries

We assume that the reader is familiar with standard graph-theoretical terminology (see, e.g., [20]). The notation that we use is also standard. Since we deal with connectivity under *vertex* failures, it is sufficient to consider simple graphs as input to our problem (because the existence of parallel edges does not affect the connectivity relation). However, during the update phase, we construct a multigraph that represents the connectivity relationship between some connected components after removing the failed vertices (Definition 7.1). The parallel edges in this graph are redundant, but they may be introduced by the algorithm that we use, and it would be costly to check for redundancy throughout.

It is also sufficient to assume that the input graph G is connected. Because, otherwise, we can initialize a data structure on every connected component of G; the updates, for a given set of failures, are distributed to the data structures on the connected components, and the queries for pairs of vertices that lie in different connected components of G are always *false*. We use G to denote the input graph throughout; n and m denote its number of vertices and edges, respectively. For any two integers x, y, we use the interval notation $[x, y]$ to denote the set $\{x, x+1, \ldots, y\}$. (If $x > y$, then $[x, y] = \emptyset$.)

Finally, we note that this chapter is self-contained. In particular, the "*low*" points that we consider throughout this chapter have a different meaning than those introduced in Section 3.1.

7.2.1 DFS-based concepts

Let T be a DFS-tree of G, with start vertex r [63]. We use $p(v)$ to denote the parent of every vertex $v \neq r$ in T (v is a child of $p(v)$). For any two vertices u, v, we let $T[u, v]$ denote the simple tree path from u to v on T. For every two vertices u and v, if the tree path $T[r, u]$ uses v, then we say that v is an ancestor of u (equivalently, u is a descendant of v). In particular, a vertex is considered to be an ancestor (and also a descendant) of itself. It is very useful to identify the vertices with their order of visit during the DFS, starting with $r \leftarrow 1$. Thus, if v is an ancestor of u, we have $v < u$. For any vertex v, we let $T(v)$ denote the subtree rooted at v, and we let $ND(v)$ denote the number of descendants of v (i.e., $ND(v) = |T(v)|$). Thus, we have that $T(v) = [v, v + ND(v) - 1]$, and therefore we can check the ancestry relation in constant time. Two children c and c' of a vertex v are called *consecutive children* of v (in this order), if c' is the minimum child of v with $c' > c$. Notice that, in this case, we have $T(c) \cup T(c') = [c, c' + ND(c') - 1]$.

A DFS-tree T has the following extremely convenient property: the endpoints of every non-tree edge of G are related as ancestor and descendant on T [63], and so we call those edges *back-edges*. Our whole approach is basically an exploitation of this property, which does not hold in general rooted spanning trees of G (unless they are derived from a DFS traversal, and only then [63]). To see why this is relevant for our purposes, consider what happens when we remove a vertex $f \neq r$ from T. Let $c_1, \ldots, c_k$ be the children of f in T. Then, the connected components of $T \setminus f$ are given

by $T(c_1),\ldots,T(c_k)$ and $T(r)\setminus T(f)$. A subtree $T(c_i)$, $i\in\{1,\ldots,k\}$, is connected with the rest of the graph in $G\setminus f$ if and only if there is a back-edge that stems from $T(c_i)$ and ends in a proper ancestor of f. Now, this problem has an algorithmically elegant solution. Suppose that we have computed, for every vertex $v\neq r$, the *lowest* proper ancestor of v that is connected with $T(v)$ through a back-edge. We denote this vertex as $low(v)$. Then, we may simply check whether $low(c_i)<f$, in order to determine whether $T(c_i)$ is connected with $T(r)\setminus T(f)$ in $G\setminus f$.

We extend the concept of the *low* points, by introducing the low_k points, for any $k\in\mathbb{N}$. These are defined recursively, for any vertex $v\neq r$, as follows. $low_1(v)$ coincides with $low(v)$. Then, supposing that we have defined $low_k(v)$ for some $k\in\mathbb{N}$, we define $low_{k+1}(v)$ as $min(\{y \mid \exists$ a back-edge (x,y) such that $x\in T(v)$ and $y<v\}\setminus\{low_1(v),\ldots,low_k(v)\})$. Notice that $low_k(v)$ may not exist for some $k\in\mathbb{N}$ (and this implies that $low_{k'}(v)$ does not exist, for any $k'>k$). If, however, $low_k(v)$ exists, then $low_{k'}(v)$, for any $k'<k$, also exists, and we have $low_1(v)<low_2(v)<\cdots<low_k(v)$. Notice that the existence of $low_k(v)$ implies that there is a back-edge $(x,low_k(v))$, where x is a descendant of v.

Proposition 7.1. *Let T be a DFS-tree of a simple graph G, and assume that the adjacency list of every vertex of G is sorted in increasing order w.r.t. the DFS numbering. Suppose also that, for some $k\in\{0,\ldots,n-1\}$, we have computed the $low_1,\ldots,low_k$ points of all vertices (w.r.t. T), and the set $\{low_1(v),\ldots,low_k(v)\}$ is stored in an increasingly sorted array for every $v\neq r$. Then we can compute the low_{k+1} points of all vertices in $O(n\log(k+1))$ time.*[3]

Proof. For every $v\neq r$, let $lowArray(v)$ be the array that contains $\{low_1(v),\ldots,low_k(v)\}$ in increasing order, plus one more entry which is *null*. Now we process the vertices in a bottom-up fashion (e.g., in reverse DFS order). We will make sure that, when we start processing a vertex, the $low_1,\ldots,low_{k+1}$ points of its children are correctly computed $(*)$.

The processing of a vertex $v\neq r$ is done as follows. First, we perform a binary search within the first $k+1$ entries of the adjacency list of v, in order to find the smallest vertex that is greater than $low_k(v)$; if it exists, we insert it in the $k+1$ entry of $lowArray(v)$. Now we process the children of v. For every child c of v, if the $k+1$ entry of $lowArray(v)$ is *null*, then we perform a binary search in $lowArray(c)$, in order to find the smallest vertex that is greater than $low_k(v)$ and lower than v. If it exists, then we

[3]We make the convention that $\log(1)=1$, so that the time to compute the low_1 points is $O(n)$.

insert it in the $k+1$ entry of *lowArray*(v). Otherwise, if the $k+1$ entry of *lowArray*(v) is not *null*, then we perform a binary search in *lowArray*(c), in order to find the smallest vertex y that is greater than $low_k(v)$ and lower than the $k+1$ entry of *lowArray*(v). If it exists, then we replace the vertex at the $k+1$ entry of *lowArray*(v) with y. Notice that, for the processing of v, we need $O((1+nChilden_v)\log(k+1))$ time, where $nChilden_v$ is the number of children of v. Thus, the whole algorithm takes $O(n\log(k+1))$ time in total.

Now we have to argue about the correctness of this procedure. Suppose that $(*)$ is true for a vertex v right when we start processing it. (If v is a leaf, then $(*)$ is trivially true.) Let us also suppose that $low_k(v)$ is exists, because otherwise $low_{k+1}(v)$ does not exist and we are done. Consider the segment $y_1,\dots,y_{k+1}$ of the first $k+1$ entries of the adjacency list of v. Then, notice that $low_{k+1}(v)\leq y_{k+1}$ (where we let this inequality be trivially true if y_{k+1} is *null*). This is because $low_1(v)$ is at least as low as y_1, therefore $low_2(v)$ is at least as low as y_2, and so on. Thus, if $low_{k+1}(v)$ exists in the adjacency list of v, it coincides with the lowest among $y_1,\dots,y_{k+1}$ that is greater than $low_k(v)$. Otherwise, after the search in the adjacency list of v, we just have that the $k+1$ entry of *lowArray*(v) (if it is not *null*) contains a vertex that is greater than $low_{k+1}(v)$. Now we check the low_i points of the children of v, for $i\in\{1,\dots,k+1\}$. (By $(*)$, these are correctly computed, and they are stored in the *lowArray* arrays.) First, we notice, as previously, that $low_{k+1}(v)$ is at least as low as the $k+1$ entry in *lowArray*(c), for any child c of v. Thus, if there is a child c of v such that *lowArray*(c) contains $low_{k+1}(v)$, then this is precisely the smallest vertex in *lowArray*(c) that is greater than $low_k(v)$, and we correctly insert it in the $k+1$ entry of *lowArray*(v).

We conclude that, when we finish processing v, either the $k+1$ entry of *lowArray*(v) is *null* (from which we infer that $low_{k+1}(v)$ does not exist), or it contains a vertex that is greater than $low_k(v)$, but at least as low as any of the first $k+1$ entries of the adjacency list of v that are greater than $low_k(v)$, or the first $k+1$ *low* points of any of its children that are greater than $low_k(v)$. Thus, $low_{k+1}(v)$ has been correctly computed in the $k+1$ entry of *lowArray*(v). □

Corollary 7.1. *For any $k\in\{1,\dots,n-1\}$, the $low_1,\dots,low_k$ points of all vertices can be computed in $O(m+kn\log k)$ time.*

Proof. An immediate appplication of Proposition 7.1: we first sort the adjacency lists of all vertices with bucket-sort, and then we just compute the $low_1,\dots,low_k$ points, for all

vertices, in this order. This will take time $O(m+n)+O(n\log 1+n\log 2+\cdots+n\log k) = O(m+kn\log k)$. $\square$

7.3 The algorithm for vertex failures

7.3.1 Initializing the data structure

We will need the following ingredients in order to be able to handle at most $d_\star$ failed vertices.

(i) A DFS-tree T of G rooted at a vertex r. The values *ND* and *depth* (w.r.t. T) must be computed for all vertices. We identify the vertices of G with the DFS numbering of T.

(ii) A level-ancestor data structure on T.

(iii) A 2D-range-emptiness data structure on the set of the back-edges of G w.r.t. T.

(iv) The low_i points of all vertices, for every $i \in \{1,\dots,d_\star\}$.

(v) For every $i \in \{1,\dots,d_\star\}$, a DFS-tree T_i of T rooted at r, where the adjacency lists of the vertices are given by their children lists sorted in increasing order w.r.t. the low_i point.

(vi) For every $i \in \{1,\dots,d_\star\}$, a 2D-range-emptiness data structure on the set of the back-edges of G w.r.t. T_i.

The *depth* value in (i) refers to the depths of the vertices in T. This is defined for every vertex v as the size of the tree path $T[r,v]$. (Thus, e.g., $depth(r) = 1$.) It takes $O(n)$ additional time to compute the *depth* values during the DFS.

The level-ancestor data structure in (ii) is used in order to answer queries of the form `QueryLA`$(v,\delta) \equiv$ "return the ancestor of v that lies at depth δ". We use those queries in order to find the children of vertices that are ancestors of other vertices. (I.e., given that u is a descendant of v, we want to know the child of v that is an ancestor of u.) For our purposes, it is sufficient to use the solution in Section 3 of [10], that preprocesses T in $O(n\log n)$ time so that it can answer level-ancestor queries in (worst-case) $O(1)$ time.

The 2D-range-emptiness data structure in (iii) is used in order to answer queries of the form `2D_range`$([X_1, X_2] \times [Y_1, Y_2]) \equiv$ "is there a back-edge (x, y) with $x \in [X_1, X_2]$ and $y \in [Y_1, Y_2]$?".[4] We can use a standard implementation for this data structure, that has $O(m \log n)$ space and preprocessing time complexity, and can answer a query in (worst-case) $O(\log n)$ time (see, e.g., Section 5.6 in [19]). The m factor here is unavoidable, because the number of back-edges can be as large as $m-n+1$. However, we note that we can improve the $\log n$ factor in the space and the query time if we use a more sophisticated solution, such as [13].

The $low_1, \dots, low_{d_\star}$ points of all vertices can be computed in $O(m + d_\star n \log d_\star) = O(m + d_\star n \log n)$ time (Corollary 7.1). We obviously need $O(d_\star n)$ space to store them.

For (v), we just perform $d_\star$ DFS's on T, starting from r, where each time we use a different arrangement of the children lists of T as adjacency lists. This takes $O(d_\star n)$ time in total, but we do not need to actually store the trees. (In fact, the parent pointer is the same for all of them.) What we actually need here is the DFS numbering of the i-th DFS traversal, for every $i \in \{1, \dots, d_\star\}$, which we denote as DFS_i. We keep those DFS numberings stored, and so we need $O(d_\star n)$ additional space. The usefulness of performing all those DFS's will become clear in Section 7.3.4. Right now, we only need to mention that, for every $i \in \{1, \dots, d_\star\}$, the ancestry relation in T_i is the same as that in T. Thus, the $low_1, \dots, low_{d_\star}$ points for all vertices w.r.t. T_i are the same as those w.r.t. T.

The 2D-range-emptiness data structures in (vi) are used in order to answer queries of the form `2D_range_i`$([X_1, X_2] \times [Y_1, Y_2]) \equiv$ "is there a back-edge (x, y) with $x \in [X_1, X_2]$ and $y \in [Y_1, Y_2]$?", where the endpoints of the query rectangle refer to the DFS_i numbering, for $i \in \{1, \dots, d_\star\}$. Since the ancestry relation is the same for T_i and T, we have that the queries `2D_range`$([X_1, X_2] \times [Y_1, Y_2])$ and `2D_range_i`$([X_1, X_2]_i \times [Y_1, Y_2]_i)$ are equivalent, where the i index below the brackets means that we have translated the endpoints in the DFS_i numbering.

The construction of the 2D-range-emptiness data structures w.r.t. the DFS-trees $T_1, \dots, T_{d_\star}$ takes $O(d_\star m \log n)$ time in total. In order to keed those data structures stored, we need $O(d_\star m \log n)$ space. Thus, the construction and the storage of the 2D-range-emptiness data structures dominate the space-time complexity overall.

It is easy to see that the list of data structures from (i) to (vi) is flexible w.r.t. $d_\star$.

[4]The input to `2D_range` is just the endpoints X_1, X_2, Y_1, Y_2 of the query rectangle; we use brackets around them, and the symbol $\times$, just for readability.

Thus, if $d_\star$ increases by 1, then we need to additionally compute the $low_{d_\star+1}$ points of all vertices, the $T_{d_\star+1}$ DFS-tree, and the corresponding 2D-range-emptiness data structure. Computing the $low_{d_\star+1}$ points takes $O(n \log(d_\star + 1)) = O(n \log n)$ time, and demands an additional $O(n)$ space, assuming that we have sorted the adjacency lists of G in increasing order, and that we have stored the $low_1, \ldots, low_{d_\star}$ points, for every vertex, in an increasingly sorted array (see Proposition 7.1).

Corollary 7.2. *Suppose that we have initialized our data structure for some $d_\star$, and we want to get a data structure for $d_\star + k$. Then we can achieve this in $O(km \log n)$ time, using extra $O(km \log n)$ space.*

If $d_\star$ decreases by k, then we just have to discard the $low_{d_\star-k+1}, \ldots, low_{d_\star}$ points, the $T_{d_\star-k+1}, \ldots, T_{d_\star}$ DFS-trees, and the corresponding 2D-range-emptiness data structures. This will free $O(km \log n)$ space.

7.3.2 The general idea

Let F be a set of failed vertices. Then $T \setminus F$ may consist of several connected components, all of which are subtrees of T. It will be necessary to distinguish two types of connected components of $T \setminus F$. Let C be a connected component of $T \setminus F$. If no vertex in F is a descendant of C, then C is called a *hanging subtree* of $T \setminus F$. Otherwise, C is called an *internal component* of $T \setminus F$. (See Figure 7.1 for an illustration.) Observe that, while the number of connected components of $T \setminus F$ may be as large as $n - 1$ (even if $|F| = 1$), the number of internal components of $T \setminus F$ is at most $|F|$. This is an important observation, that allows us to reduce the connectivity of $G \setminus F$ to the connectivity of the internal components.

More precisely, we can already provide a high level description of our strategy for answering connectivity queries between pairs of vertices. Let x, y be two vertices of $G \setminus F$. Suppose first that x belongs to an internal component C_1 and y belongs to an internal component C_2. Then it is sufficient to know whether C_1 and C_2 are connected in $G \setminus F$. Otherwise, if either x or y lies in a hanging subtree C, then we can substitute C with any internal component that is connected with C in $G \setminus F$. If no such internal component exists, then x and y are connected in $G \setminus F$ if and only if they lie in the same hanging subtree.

Thus, after the deletion of F from G, it is sufficient to make provisions so as to be able to efficiently answer the following:

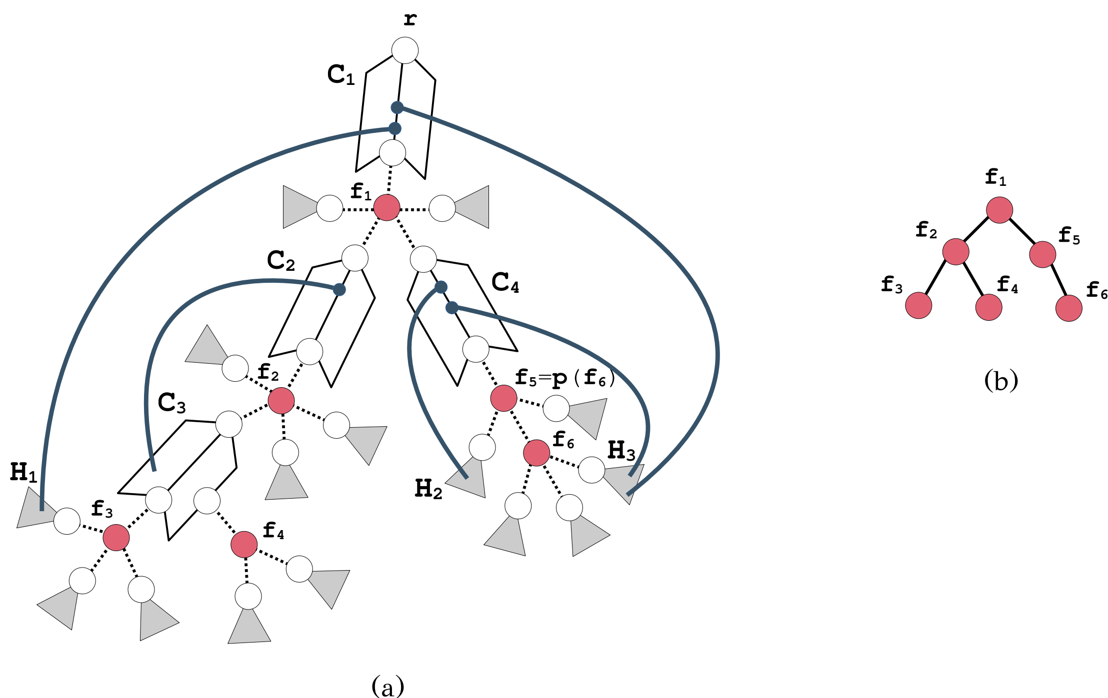


Figure 7.1: (a) A set of failed vertices $F = \{f_1, \ldots, f_6\}$ on a DFS-tree T, and (b) the corresponding F-forest, which shows the $parent_F$ relation between failed vertices. Notice that $T \setminus F$ is split into several connected components, but there are only four internal components, C_1, C_2, C_3 and C_4. The hanging subtrees of $T \setminus F$ are shown with gray color (e.g., H_1, H_2 and H_3). The internal components C_2 and C_3 remain connected in $G \setminus F$ through a back-edge that connects them directly. C_1 and C_4 remain connected through the hanging subtree H_3 of f_6. We have $\partial(C_1) = \{f_1\}$, $\partial(C_2) = \{f_2\}$, $\partial(C_3) = \{f_3, f_4\}$ and $\partial(C_4) = \{f_5\}$. Notice that f_6 is the only failed vertex that is not a boundary vertex of an internal component, and it has $parent_F(f_6) = p(f_6)$.

(1) Given a vertex x, determine the connected component of $T \setminus F$ that contains x.

(2) Given two internal components C_1 and C_2 of $T \setminus F$, determine whether C_1 and C_2 are connected in $G \setminus F$.

(3) Given a hanging subtree C of $T \setminus F$, find an internal component of $T \setminus F$ that is connected with C in $G \setminus F$, or report that no such internal component exists.

Actually, the most difficult task, and the only one that we provide a preprocessing for (during the update phase), is (2). We explain how to perform (1) and (3) during

the process of answering a query, in Section 7.3.5. An efficient solution for (2) is provided in Section 7.3.4.

The general idea is that, since there are at most $d = |F|$ internal components of $T\backslash F$, we can construct a graph with $O(d)$ nodes, representing the internal components of $T\backslash F$, that captures the connectivity relation among them in $G\backslash F$ (see Lemma 7.5). This is basically done with the introduction of some artificial edges between the (representatives of the) internal components. In the following subsection, we state some lemmata concerning the structure of the internal components, and their connectivity relationship in $G \setminus F$.

7.3.3 The structure of the internal components

We will use the roots of the connected components of $T\backslash F$ (viewed as rooted subtrees of T) as representantives of them. Now we introduce some terminology and notation. If C is a connected component of $T \setminus F$, we denote its root as r_C. If C is a hanging subtree of $T \setminus F$, then $p(r_C) = f$ is a failed vertex, and we say that C is *a hanging subtree of* f. If C, C' are two distinct connected components of $T \setminus F$ such that $r_{C'}$ is an ancestor of r_C, then we say that C' is an ancestor of C. Furthermore, if v is a vertex not in C such that v is an ancestor (resp., a descendant) of r_C, then we say that v is an ancestor (resp., a descendant) of C. If C is an internal component of $T \setminus F$ and f is a failed vertex such that $p(f) \in C$, then we say that f is a boundary vertex of C. The collection of all boundary vertices of C is denoted as $\partial(C)$. Notice that any vertex $b \in \partial(C)$ has the property that there is no failed vertex on the tree path $T[p(b), r_C]$. Conversely, a failed vertex b such that there is no failed vertex on the tree path $T[p(b), r_C]$ is a boundary vertex of C. Thus, if $b_1, \dots, b_k$ is the collection of all the boundary vertices of C, then $C = T(r_C) \setminus (T(b_1) \cup \dots \cup T(b_k))$.

The following lemma is a collection of properties that are satisfied by the internal components.

Lemma 7.1. *Let C be an internal component of $T \setminus F$. Then:*

(1) Either $r_C = r$, or $p(r_C) \in F$.

(2) For every vertex v that is a descendant of C, there is a unique boundary vertex of C that is an ancestor of v.

(3) Let $f_1, \ldots, f_k$ be the boundary vertices of C, sorted in increasing order. Then C is the union of the following subsets of consecutive vertices: $[r_C, f_1 - 1], [f_1 + ND(f_1), f_2 - 1], \ldots, [f_{k-1} + ND(f_{k-1}), f_k - 1], [f_k + ND(f_k), r_C + ND(r_C) - 1]$. (We note that some of those sets may be empty.)

Proof. (1) If $r_C \neq r$, then $p(r_C)$ is defined. Since r_C is the root of a connected component of $T \setminus F$, we have that $r_C \notin F$. If $p(r_C) \notin F$, then r_C is connected with $p(r_C)$ in $T \setminus F$ through the parent edge $(r_C, p(r_C))$, contradicting the fact that r_C is the root of a connected component of $T \setminus F$. Thus, $p(r_C) \in F$.

(2) Since v is a descendant of C, we have that $v \notin C$ and v is a descendant of r_C. Since v is a descendant of r_C, we have that all vertices on the tree path $T[v, r_C]$ are ancestors of v. (Notice that only a vertex on $T[v, r_C]$ can be both an ancestor of v and a boundary vertex of C, because all other ancestors of v are lower than r_C.) Since $v \notin C$, there must exist at least one failed vertex on $T[v, r_C]$. Take the lowest such failed vertex b. Then we have that none of the vertices on the tree path $T[p(b), r_C]$ is a failed vertex, and so $p(b)$ is connected with r_C in $T \setminus F$, and therefore b is a boundary vertex of C.

Now let us suppose, for the sake of contradiction, that there is another vertex b' on $T[v, r_C]$ that is a boundary vertex of C. Since b is the lowest with this property, we have that b' is a proper descendant of b. Since $b' \in \partial(C)$, we have that there cannot be a failed vertex on the tree path $T[p(b'), r_C]$, contradicting the fact that $b \in T[p(b'), r_C]$. Thus, we have that b is the unique vertex in $\partial(C)$ that is an ancestor of v.

(3) The subtree rooted at r_C consists of the vertices in $[r_C, r_C + ND(r_C) - 1]$. Since $f_1, \ldots, f_k$ are the boundary vertices of C, we have that $C = T(r_C) \setminus (T(f_1) \cup \cdots \cup T(f_k))$. Therefore, $C = [r_C, r_C + ND(r_C) - 1] \setminus ([f_1, f_1 + ND(f_1) - 1] \cup \cdots \cup [f_k, f_k + ND(f_k) - 1])$. Thus, since $f_1, \ldots, f_k$ are sorted in increasing order, we have $C = [r_C, f_1 - 1] \cup [f_1 + ND(f_1), f_2 - 1] \cup \cdots \cup [f_{k-1} + ND(f_{k-1}), f_k - 1] \cup [f_k + ND(f_k), r_C + ND(r_C) - 1]$. □

We represent the ancestry relation between failed vertices using a forest which we call the *failed vertex forest* (*F-forest*, for short). The F-forest consists of the following two elements. First, for every failed vertex f, there is a pointer $parent_F(f)$ to the nearest ancestor of f (in T) that is also a failed vertex. If there is no ancestor of f that is a failed vertex, then we let $parent_F(f) = \perp$. And second, every failed vertex f has a pointer to its list of children in the F-forest.

The F-forest can be easily constructed in $O(d^2)$ time: we just have to find, for every

failed vertex f, the maximum failed vertex f' that is a proper ancestor of f; then we set $\mathit{parent}_F(f) = f'$, and we append f to the list of the children of f' in the F-forest.

The next lemma shows how we can check in constant time whether a failed vertex belongs to the boundary of an internal component, and how to retrieve the root of this component.

Lemma 7.2. *A failed vertex f is a boundary vertex of an internal component if and only if $\mathit{parent}_F(f) \neq p(f)$. Now let f be a boundary vertex of an internal component C. Then, if $\mathit{parent}_F(f)$ exists, we have that the root of C is the child of $\mathit{parent}_F(f)$ that is an ancestor of f. Otherwise, the root of C is r.*

Proof. Let C be an internal component such that $f \in \partial(C)$. Then there is no failed vertex on the tree path $T[p(f), r_C]$. In particular, $p(f) \neq \mathit{parent}_F(f)$. Conversely, suppose that $\mathit{parent}_F(f) \neq p(f)$. (We can reject the case $f = r$, because then none of the expressions $\mathit{parent}_F(f), p(f)$ is defined.) If $\mathit{parent}_F(f)$ is not defined, then there is no failed vertex on the tree path $T[p(f), r]$ (i.e., on the path of the ancestors of f), and therefore f is a boundary vertex of the internal component with root r. Otherwise, if $\mathit{parent}_F(f)$ is defined, then we have that $p(f)$ cannot be a failed vertex (because otherwise we would have $\mathit{parent}_F(f) = p(f)$, because $\mathit{parent}_F(f)$ is the nearest proper ancestor of f that is a failed vertex). Thus, $p(f)$ belongs to a connected component of $T \setminus F$, to which f is a boundary vertex.

Now let f be a boundary vertex of an internal component C. This means that there is no failed vertex on the tree path $T[p(f), r_C]$. If $\mathit{parent}_F(f)$ exists, then it must be a proper ancestor of r_C. Thus, $r_C \neq r$, and therefore, by Lemma 7.1(1), we have that $p(r_C)$ is a failed vertex. Since $\mathit{parent}_F(f)$ is the nearest ancestor of f that is a failed vertex, we thus have that $\mathit{parent}_F(f) = p(r_C)$, and therefore r_C is the child of $\mathit{parent}_F(f)$ that is an ancestor of f. Otherwise, if $\mathit{parent}_F(f)$ does not exist, this implies that there is no failed vertex on the tree path $T[p(f), r]$. Thus, f is a boundary vertex of the internal component with root r. □

Thus, according to Lemma 7.2, if f is a boundary vertex of an internal component C with $r_C \neq r$, we can retrieve r_C in constant time using a level-ancestor query: i.e., we ask for the ancestor of f (in T) whose depth equals that of $\mathit{parent}_F(f) + 1$. We may use this fact throughout without mention.

The following lemma shows that there are two types of edges that determine the connectivity relation in $G \setminus F$ between the connected components of $T \setminus F$.

Lemma 7.3. *Let e be an edge of $G \setminus F$ whose endpoints lie in different connected components of $T \setminus F$. Then e is a back-edge and either (i) both endpoints of e lie in internal components, or (ii) one endpoint of e lies in a hanging subtree H, and the other endpoint lies in an internal component C that is an ancestor of H.*

Proof. Let $e = (x, y)$, let C be the connected component of $T \setminus F$ that contains x, and let C' be the connected component of $T \setminus F$ that contains y. We have that e cannot be a tree-edge, because otherwise x and y would be connected in $T \setminus F$. Thus, e is a back-edge. Since x, y are the endpoints of a back-edge, they are related as ancestor and descendant. We may assume w.l.o.g. that $r_C > r_{C'}$. We will show that this implies that x is a descendant of y. So let us suppose, for the sake of contradiction, that x is an ancestor of y. Since $y \in C'$, we have that y is a descendant of $r_{C'}$. Since x is an ancestor of y that does not lie in C', we have that x does not lie on the tree path $T[y, r_{C'}]$. Thus, x is a proper ancestor of $r_{C'}$, and so $x < r_{C'}$. Since $x \in C$, we have that $x \geq r_C$. Thus, we have $r_C \leq x < r_{C'}$, which contradicts the assumption $r_C > r_{C'}$. Thus, we have shown that x is a descendant of y. Now, since y does not lie in C, we have that y cannot lie on the tree path $T[x, r_C]$. Therefore, since y is an ancestor of x, it must be a proper ancestor of r_C. And since $y \in C'$, we have that y is a descendant of $r_{C'}$. Therefore, r_C is a descendant of $r_{C'}$.

Thus we have shown that C is a descendant of C'. This implies that C' cannot be a hanging subtree of $T \setminus F$. To see this, suppose the contrary. Since $r_{C'}$ is a proper ancestor of r_C, we have that $r_{C'}$ is an ancestor of $p(r_C)$. ($p(r_C)$ is defined, precisely because r_C has a proper ancestor, and therefore $r_C \neq r$.) Notice that $p(r_C)$ is a failed vertex (otherwise, r_C would be connected with $p(r_C)$ through the parent edge $(r_C, p(r_C))$, contradicting the fact that r_C is the root of a connected component of $T \setminus F$). But then we have that r'_C is an ancestor of a failed vertex, contradicting our supposition that C' is a hanging subtree of $T \setminus F$. We conclude that, among C and C', only C can be a hanging subtree of $T \setminus F$. □

Corollary 7.3. *Let C, C' be two distinct connected components of $T \setminus F$ that are connected with an edge e of $G \setminus F$. Assume w.l.o.g. that $r_{C'} < r_C$. Then C' is an ancestor of C.*

Proof. Lemma 7.3 implies that e is a back-edge. Let $e = (x, y)$, and assume w.l.o.g. that x is a descendant of y. We know that either $x \in C$ and $y \in C'$, or reversely. Let us suppose, for the sake of contradiction, that $x \in C'$ (and thus $y \in C$). This implies that x is a descendant of $r_{C'}$. Thus, x is a common descendant of $r_{C'}$ and y. This

implies that $r_{C'}$ and y are related as ancestor and descendant. We have that y cannot be a descendant of $r_{C'}$, because this would imply that $y \in T[r_{C'}, x]$ (but y lies outside of C'). Thus, we have that y is a proper ancestor of $r_{C'}$, and therefore $y < r_{C'}$. Since $r_{C'} < r_C$, this implies that $y < r_C$, and therefore y cannot be a descendant of r_C – contradicting the fact that $y \in C$.

Thus we have shown that $x \in C$ and $y \in C'$. $x \in C$ implies that x is a descendant of r_C. Thus, x is a common descendant of r_C and y. This implies that r_C and y are related as ancestor and descendant. We have that y cannot be a descendant of r_C, because this would imply $y \in T[r_C, x]$ (but y lies outside of C). Thus, r_C is a descendant of y. Also, $y \in C'$ implies that y is a descendant of $r_{C'}$. Thus, we conclude that r_C is a descendant of $r_{C'}$. □

The following lemma provides an algorithmically useful criterion to determine whether a connected component of $T \setminus F$ – a hanging subtree or an internal component – is connected with an internal component of $T \setminus F$ through a back-edge.

Lemma 7.4. *Let C, C' be two connected components of $T \setminus F$ such that C' is an internal component that is an ancestor of C, and let b be the boundary vertex of C' that is an ancestor of C. Then there is a back-edge from C to C' if and only if there is a back-edge from C whose lower end lies in $[r_{C'}, p(b)]$.*

Proof. First, let us explain the existence of b. Since C' is an ancestor of C, we have that $r_{C'}$ is an ancestor of r_C. Therefore, Lemma 7.1(2) implies that there is a unique boundary vertex b of C' that is an ancestor of r_C. Thus, b is an ancestor of C.

($\Rightarrow$) Let $e = (x, y)$ be a back-edge from C to C', and assume w.l.o.g. that x lies in C. Since e is a back-edge, we have that either x is a descendant of y, or reversely. Let us suppose, for the sake of contradiction, that y is a descendant of x. Since $x \in C$, we have that $x \geq r_C$. Since y is a descendant of x, we have that $y > x$. Thus, $y > r_C$. Since (x, y) is a back-edge from C to C' and $x \in C$, we have that $y \in C'$. This implies that there must be a failed vertex on the tree path $T[y, x]$. (Otherwise, y would be connected with x, and therefore C' would be connected with C, which is absurd.) Let f be the maximum failed vertex on the tree path $T[y, x]$. Then, the connected component of $T \setminus F$ that contains y has a child of f as a root. But this root is $r_{C'}$, and therefore we have $r_{C'} > f > x \geq r_C$, in contradiction to the assumption that C' is an ancestor of C. Thus we have shown that x is a descendant of y. Since y is an ancestor of x that does not lie in C, we have that y does not lie on the tree path $T[x, r_C]$, and

therefore it must be a proper ancestor of r_C. Thus, since $y \in C'$, we have that y lies on the tree path $T[p(b), r_{C'}]$. This implies that $y \in [r_{C'}, p(b)]$.

($\Leftarrow$) Let $e = (x, y)$ be a back-edge from C whose lower end lies in $[r_{C'}, p(b)]$. We may assume w.l.o.g. that $x \in C$. Thus, we have that $y \in [r_{C'}, p(b)]$, and that y is an ancestor of x. Since $x \in C$, we have that x is a descendant of r_C. Since b is an ancestor of C, we have that b is an ancestor of r_C. Thus, x is a descendant of b, and therefore a descendant of $p(b)$. This means that the tree path $T[p(b), r_{C'}]$ consists of ancestors of x. Thus, since y is an ancestor of x with $y \in [r_{C'}, p(b)]$, we have that $y \in T[p(b), r_{C'}]$. Since b is a boundary vertex of C', we have that all vertices on the tree path $T[p(b), r_{C'}]$ lie in C'. In particular, we have $y \in C'$. □

Definition 7.1. Let $\mathcal{R}$ be a multigraph where $V(\mathcal{R})$ is the set of the roots of the internal components of $T \setminus F$, and $E(\mathcal{R})$ satisfies the following three properties:

(1) For every back-edge connecting two internal components C and C', there is an edge $(r_C, r_{C'})$ in $\mathcal{R}$.

(2) Let H be a hanging subtree of a failed vertex f, and let $C_1, \dots, C_k$ be the internal components that are connected with H through a back-edge. (By Lemma 7.3, all of $C_1, \dots, C_k$ are ancestors of H.) Assume w.l.o.g. that C_k is an ancestor of all $C_1, \dots, C_{k-1}$. Then $\mathcal{R}$ contains the edges $(r_{C_1}, r_{C_k}), (r_{C_2}, r_{C_k}), \dots, (r_{C_{k-1}}, r_{C_k})$.

(3) Every edge of $\mathcal{R}$ is given by either (1) or (2), or it is an edge of the form $(r_C, r_{C'})$, where C, C' are two internal components that are connected in $G \setminus F$.

Then $\mathcal{R}$ is called a *connectivity graph of the internal components* of $T \setminus F$. The edges of (1) and (2) are called Type-1 and Type-2, respectively.

The following lemma shows that this graph captures the connectivity relationship of the internal components of $T \setminus F$ in $G \setminus F$.

Lemma 7.5. *Let $\mathcal{R}$ be a connectivity graph of the internal components of $T \setminus F$. Then, two internal components C, C' of $T \setminus F$ are connected in $G \setminus F$ if and only if $r_C, r_{C'}$ are connected in $\mathcal{R}$.*

Proof. ($\Rightarrow$) Let C, C' be two internal components of $T \setminus F$ that are connected in $G \setminus F$. This means that there is a sequence $C_1, \dots, C_k$ of pairwise distinct connected components of $T \setminus F$, and a sequence of back-edges $e_1, \dots, e_{k-1}$, such that: $C_1 = C$,

$C_k = C'$, and e_i connects C_i and C_{i+1}, for every $i \in \{1, \dots, k-1\}$. By Lemma 7.3, we have that, for every $i \in \{1, \dots, k-1\}$, either (1) C_i and C_{i+1} are internal components that are related as ancestor and descendant, or (2) one of C_i, C_{i+1} is a hanging subtree, and the other is an internal component that is an ancestor of it.

Let i be an index in $\{1, \dots, k-1\}$. If (1) is true, then there is a Type-1 edge $(r_{C_i}, r_{C_{i+1}})$ in $\mathcal{R}$. If (2) is true, then one of C_i, C_{i+1} is a hanging subtree. Let us assume that C_i is a hanging subtree. Since there is a back-edge connecting C_i with C_{i+1}, we may consider the lowest internal component $\tilde{C}$ that is an ancestor of C_i and is connected with it through a back-edge. If $C_{i+1} = \tilde{C}$, then we imply nothing at this point. Otherwise, we have that $\mathcal{R}$ contains the Type-2 edge $(r_{C_{i+1}}, r_{\tilde{C}})$. Now, since C_1 is an internal component, we have that $C_i \neq C_1$, and therefore C_{i-1} is defined. By Lemma 7.3, we have that C_{i-1} is also an internal component, that is connected with C_i through a back-edge. Again, if $C_{i-1} = \tilde{C}$, then we imply nothing at this point. Otherwise, we have that $\mathcal{R}$ contains the Type-2 edge $(r_{C_{i-1}}, r_{\tilde{C}})$. Thus, there are three possibilities to consider: either $C_{i-1} = \tilde{C}$ and $C_{i+1} \neq \tilde{C}$, or $C_{i-1} \neq \tilde{C}$ and $C_{i+1} = \tilde{C}$, or $C_{i-1} \neq \tilde{C}$ and $C_{i+1} \neq \tilde{C}$. In any case, we can see that $r_{C_{i-1}}$ is connected with $r_{C_{i+1}}$ in $\mathcal{R}$ – either directly, or through $r_{\tilde{C}}$. Similarly, if we assume that C_{i+1} is a hanging subtree (and C_i is an internal component), then we have that r_{C_i} is connected with $r_{C_{i+2}}$ in $\mathcal{R}$.

From all this we infer that, if $C_{i(1)}, \dots, C_{i(t)}$ is the subsequence of $C_1, \dots, C_k$ that consists of the internal components, then $r_{C_{i(1)}}, \dots, r_{C_{i(t)}}$ are connected in $\mathcal{R}$. In particular, we have that r_C and $r_{C'}$ are connected in $\mathcal{R}$.

($\Leftarrow$) Let $e = (r_C, r_{C'})$ be an edge of $\mathcal{R}$. If e is a Type-1 edge, then there is a back-edge that connects C and C' in $G \setminus F$. Otherwise, there is a hanging subtree of $T \setminus F$ that is connected with both C and C' in $G \setminus F$ (through back-edges). In any case, we have that C, C' are connected in $G \setminus F$. Since this is true for any edge of $\mathcal{R}$, we conclude that, if $r_C, r_{C'}$ are two vertices connected in $\mathcal{R}$, then C, C' are connected in $G \setminus F$. □

7.3.4 Handling the updates: construction of a connectivity graph for the internal components of $T \setminus F$

Given a set of failed vertices F, with $|F| = d \leq d_\star$, we will show how we can construct a connectivity graph $\mathcal{R}$ for the internal components of $T \setminus F$, using $O(d^4)$ calls to

2D-range-emptiness queries. Recall that $V(\mathcal{R})$ is the set of the roots of the internal components of $T \setminus F$.

Algorithm 54 shows how we can find all Type-1 edges of $\mathcal{R}$. The idea is basically to perform 2D-range-emptiness queries for every pair of internal components, in order to determine the existence of a back-edge that connects them. More precisely, we work as follows. Let C be an internal component of $T \setminus F$. Then it is sufficient to check every ancestor component C' of C, in order to determine whether there is a back-edge from C to C' (see Corollary 7.3). Let $f_1, \dots, f_k$ be the boundary vertices of C, sorted in increasing order. Let also f' be the boundary vertex of C' that is an ancestor of C, and let $I = [r_{C'}, p(f')]$. Then we perform 2D-range-emptiness queries for the existence of a back-edge on the rectangles $[r_C, f_1 - 1] \times I, [f_1 + ND(f_1), f_2 - 1] \times I, \dots, [f_k + ND(f_k), r_C + ND(r_C) - 1] \times I$. We know that there is a back-edge connecting C and C' if and only if at least one of those queries is positive (see Lemma 7.1(3) and Lemma 7.4). If that is the case, then we add the edge $(r_C, r_{C'})$ to $\mathcal{R}$.

Observe that the total number of 2D-range-emptiness queries that we perform is $O(d^2)$, because every one of them corresponds to a triple (C, f, C'), where C, C' are internal components, C' is an ancestor of C, and f is a boundary vertex of C, or r_C. And if $C_1, \dots, C_k$ are all the internal components of $T \setminus F$, then the number of those triples is bounded by $(|\partial(C_1)|+1)\cdot d + \dots + (|\partial(C_k)|+1)\cdot d = (|\partial(C_1)| + \dots + |\partial(C_k)| + k)\cdot d \leq (d+k)\cdot d \leq (d+d)\cdot d = O(d^2)$.

Proposition 7.2. *Algorithm 54 correctly computes all Type-1 edges to construct a connectivity graph for the internal components of $T \setminus F$. The running time of this algorithm is $O(d^2 \log n)$.*

Proof. First, we need to provide a method to efficiently iterate over the collection of the internal components and their boundary vertices (Lines 1 and 2), and then we have to prove that the **while** loop in Line 5 is sufficient to access all internal components that are ancestors of C. Then, the correctness and the $O(d^2 \log n)$ time-bound follow from the analysis above (in the main text).

Every internal component C of $T \setminus F$ is determined by its root r_C. By Lemma 7.1(1), we have that either $r_C = r$, or $p(r_C)$ is a failed vertex. If $r_C = r$ then C has no ancestor internal components, and therefore we may ignore this case. So let $p(r_C) = f$ be a failed vertex. Then, by Lemma 7.2, the boundary vertices of C are given by the children of f in the F-forest that are descendants (in T) of r_C.

Algorithm 54: Compute all Type-1 edges to construct a connectivity graph $\mathcal{R}$ for the internal components of $T \setminus F$

```
1  foreach internal component C of T \ F do
2      let f_1, ..., f_k be the boundary vertices of C, sorted in increasing order
3      // process every internal component C' that is an ancestor of C
4      set f' ← p(r_C)
5      while f' ≠ ⊥ do
6          if p(f') ≠ parent_F(f') then
7              let C' be the internal component of T \ F with f' ∈ ∂(C')
8              set I ← [r_{C'}, p(f')]
9              if at least one of the following queries is positive:
10             2D_range([r_C, f_1 − 1] × I)
11             2D_range([f_1 + ND(f_1), f_2 − 1] × I)
12             ...
13             2D_range([f_{k−1} + ND(f_{k−1}), f_k − 1] × I)
14             2D_range([f_k + ND(f_k), r_C + ND(r_C) − 1] × I) then
15                 add the Type-1 edge (r_C, r_{C'}) to R
16             end
17         end
18         f' ← parent_F(f')
19     end
20 end
```

Thus, we may work as follows. First, we sort the children of every failed vertex f in the F-forest in increasing order. This takes $O(d \log d)$ time in total. Then, for every failed vertex f, we traverse its list of children L (in the F-forest) in order. For every maximal segment S of L that consists of descendants of the same child c of f in T, we know that either c is the root of an internal component with boundary S, or $c \in F$. (For every $f' \in L$ that we meet, we can use a level-ancestor query to find in constant time the child of f in T that is an ancestor of f'.) Thus, Lines 1 and 2 need $O(d \log d)$ time in total.

Now let C be an internal component with $r_C \neq r$. Then we have that $p(r_C)$ is a failed vertex. Now let C' be an internal component that is a proper ancestor of C.

This means that $r_{C'}$ is a proper ancestor of r_C, and therefore $r_{C'}$ is an ancestor of $f = p(r_C)$. Then, by Lemma 7.1(2) we have that there is a boundary vertex b of C' that is an ancestor of f (in T). Since the set of failed vertices that are ancestors of f (in T) coincide with the set of ancestors of f in the F-forest, we have that the **while** loop in Line 5 will eventually reach b. Then we can retrieve C' (more precisely: $r_{C'}$) in constant time using Lemma 7.2. The purpose of Line 6 is to apply the criterion of Lemma 7.2, in order to check whether f' is a boundary vertex of an internal component. □

The construction of Type-2 edges is not so straightforward. For every failed vertex f, and every two internal components C and C', such that C is an ancestor of f and C' is an ancestor of C, we would like to know whether there is a hanging subtree of f, from which stem a back-edge e with an endpoint in C and a back-edge e' with an endpoint in C'. The straightforward way to determine this is the following. Let b (resp., b') be the boundary vertex of C (resp., C') that is an ancestor of f. Then, for every hanging subtree of f with root c, we perform 2D-range-emptiness queries on the rectangles $[c, c + ND(c) - 1] \times [r_C, p(b)]$ and $[c, c + ND(c) - 1] \times [r_{C'}, p(b')]$. If both queries are positive, then we know that C and C' are connected in $G \setminus F$ through the hanging subtree with root c.

Obviously, this method is not efficient in general, because the number of hanging subtrees of f can be very close to n. However, it is the basis for our more efficient method. The idea is to perform a lot of those queries at once, for large batches of hanging subtrees. More specifically, we perform the queries on *consecutive* hanging subtrees of f (i.e., their roots are consecutive children of f), for which we know that the answer is positive on C' (i.e., for every one of those subtrees, there certainly exists a back-edge that connects it with C'). In order for this idea to work, we have to rearrange properly the lists of children of all vertices. (Otherwise, the hanging subtrees of f that are connected with C' through a back-edge may not be consecutive in the list of children of f.) In effect, we maintain several DFS trees (specifically: $d_\star$), and several 2D-range-emptiness data structures, one for every different arrangement of the children lists.

Let us elaborate on this idea. Let H be a hanging subtree of f that connects some internal components, and let C' be the lowest one among them (i.e., the one that is an ancestor of all the others). Then we have that the lower ends of all back-edges

that stem from H and end in ancestors of C' are failed vertices that are ancestors of C'. Thus, since there are at most d failed vertices in total, we have that at least one among $low_1(r_H), \dots, low_d(r_H)$ is in C'. In other words, r_H is one of the children of f whose low_i point is in C', for some $i \in \{1, \dots, d\}$. Now, assume that for every $i \in \{1, \dots, d_\star\}$, we have a copy of the list of the children of f sorted in increasing order w.r.t. the low_i point; let us call this list $L_i(f)$, and let it be stored in way that allows for binary search w.r.t. the low_i point. Then, for every internal component C that is an ancestor of f, we can find the segment $S_i(C)$ of $L_i(f)$ that consists of the children of f whose low_i point lies in C, by searching for the leftmost and the righmost child in $L_i(f)$ whose low_i point lies in $[r_C, p(b)]$, where b is the boundary vertex of C that is an ancestor of f.

Now let $i \in \{1, \dots, d\}$ be such that $low_i(r_H) \in C'$. Then we have that $r_H \in S_i(C')$. Furthermore, we have that every child of f that lies in $S_i(C')$ and is the root of a hanging subtree H' of f has the property that H' is also connected with C' through a back-edge. Thus, we would like to be able to perform 2D-range-emptiness queries as above on the subset S of $S_i(C')$ that consists of roots of hanging subtrees, in order to determine the connectivity (in $G \setminus F$) of C' with all internal components C that are ancestors of f and descendants of C'. We could do this efficiently if we had the guarantee that S consists of large segments of consecutive children of f. We can accommodate for that during the preprocessing phase: for every $i \in \{1, \dots, d_\star\}$, we perform a DFS of T, starting from r, where the adjacency list of every vertex v is given by $L_i(v)$.[5] Let T_i be the resulting DFS tree, and let DFS_i be the corresponding DFS numbering. Then, with the DFS numbering of T_i, we initialize a data structure *2D_range_i*, for answering 2D-range-emptiness queries for back-edges w.r.t. T_i in subrectangles of $[1, n] \times [1, n]$.

Now let us see how everything is put together. Let H be a hanging subtree of f that connects two internal components C_1 and C_2, and let b_1 and b_2 be the boundary vertices of C_1 and C_2, respectively, that are ancestors of f. Let C' be the lowest internal component that is connected through a back-edge with H. Then there is an $i \in \{1, \dots, d\}$ such that $low_i(r_H) \in C'$. Let S be the maximal segment of $S_i(C')$ that contains r_H and consists of roots of hanging subtrees, let L be the minimum of S and let R be the maximum of S.[6] Then the 2D-range-emptiness queries on

[5] I.e., it is necessary that the vertices in the adjacency list of v appear in the same order as in $L_i(v)$.

[6] Notice that, due to the construction of T_i, we have that $DFS_i(L)$ and $DFS_i(R)$ are also the minimum

$[L, R+ND(R)-1]_i \times [r_{C_1}, p(b_1)]_i$ and $[L, R+ND(R)-1]_i \times [r_{C_2}, p(b_2)]_i$ with *2D_range_i* are both positive, and so we will add the edges $(r_{C_1}, r_{C'})$ and $(r_{C_2}, r_{C'})$ in $\mathcal{R}$. This will maintain in $\mathcal{R}$ the information that C', C_1 and C_2, are connected with the same hanging subtree of f.

The algorithm that constructs enough Type-2 edges to make $\mathcal{R}$ a connectivity graph of the internal components of $T \setminus F$ is given in Algorithm 55. The proof of correctness and time complexity is given in Proposition 7.3.

Proposition 7.3. *Algorithm 55 computes enough Type-2 edges to construct a connectivity graph $\mathcal{R}$ for the internal components of $T \setminus F$ (supposing that $\mathcal{R}$ contains all Type-1 edges). The running time of this algorithm is $O(d^4 \log n)$.*

Proof. By definition, it is sufficient to prove the following: for every failed vertex f, and every hanging subtree H of f, let C' be the lowest internal component that is connected with H through a back-edge; then, for every internal component $C \neq C'$ that is connected with H through a back-edge, there is an edge $(r_C, r_{C'})$ added to $\mathcal{R}$. And conversely: that these are all the Type-2 edges that are added to $\mathcal{R}$, and that any other edge $(r_C, r_{C'})$ that is added to $\mathcal{R}$ with Algorithm 55 has the property that C and C' are connected with the same hanging subtree through back-edges.

So let f be a failed vertex, let H be a hanging subtree of f, and let C' be the lowest internal component that is connected with H through a back-edge. Let us assume that $f \notin \partial(C')$. (Otherwise, there is no internal component C that is an ancestor of H and a descendant of C', and therefore H does not induce any Type-2 edges.) Let $C \neq C'$ be an internal component that is connected with H through a back-edge. Since C' is the lowest internal component that is connected with H through a back-edge, by the analysis above (in the main text) we have that there is an $i \in \{1, \dots, d\}$ such that $low_i(r_H) \in C'$. Thus, we may consider the maximal segment S of $L_i(f)$ that contains r_H and consists of roots of hanging subtrees whose low_i point is in C'. Let L and R be the minimum and the maximum, respectively, of S. By construction of T_i, we have that S is sorted in increasing order w.r.t. the DFS_i numbering. Thus, the interval $[L, R + ND(R) - 1]_i$ consists of the descendants of the vertices in S in T_i. Since the vertices in T_i have the same ancestry relation as in T, we have that the set $DFS_i(S)$ consists of children of f in T_i that are roots of hanging subtrees with low_i in C'.

Now let f'' be the boundary vertex of C that is an ancestor of f. Then we also

and the maximum, respectively, of $DFS_i(S)$.

Algorithm 55: Compute enough Type-2 edges to construct a connectivity graph for the internal components of $T \setminus F$

```
1  foreach failed vertex f do
2      // process all pairs of internal components that are ancestors of f
3      set f' ← parent_F(f)
4      while f' ≠ ⊥ do
5          let C' be the internal component with f' ∈ ∂(C')
6          // skip the following if C' does not exist, and go immediately to
              Line 26
7          foreach i ∈ {1,...,d} do
8              let S_i be the collection of all maximal segments of L_i(f) that consist
                 of roots of hanging subtrees with their low_i point in C'
9          end
10         // process all internal components C that are ancestors of f and
              descendants of C'
11         set f'' ← f
12         while f'' ≠ f' do
13             let C be the internal component with f'' ∈ ∂(C)
14             // skip the following if C does not exist, and go immediately
                  to Line 24
15             // check if C is connected with C' through at least one hanging
                  subtree of f
16             foreach i ∈ {1,...,d} do
17                 foreach S ∈ S_i do
18                     let L ← min(S) and R ← max(S)
19                     if 2D_range_i([L, R + ND(R) − 1]_i × [r_C, p(f'')]_i) = true then
20                         add the Type-2 edge (r_C, r_C') to R
21                     end
22                 end
23             end
24             f'' ← parent_F(f'')
25         end
26         f' ← parent_F(f')
27     end
28 end
```

have that f'' is the boundary vertex of C that is an ancestor of r_H (since $f = p(r_H)$). Thus, Lemma 7.4 implies that there is a back-edge from H to $[r_C, p(f'')]$. Therefore, there is also a back-edge (w.r.t. T_i) from $DFS_i(H)$ to $[DFS_i(r_C), DFS_i(p(f''))]$. This implies that the 2D-range query in Line 19 is true, and therefore the Type-2 edge $(r_C, r_{C'})$ will be correctly added to $\mathcal{R}$. It is not difficult to see that the converse is also true: whenever the 2D-range query in Line 19 is true, we can be certain that there is a hanging subtree of f that is connected through a back-edge with both C' and C.

Let us analyze the running time of Algorithm 55. First, we will provide a method to implement Line 8, i.e., how to find, for every internal component C' that is an ancestor of f, and every $i \in \{1, \dots, d\}$, the collection $\mathcal{S}_i$ of the maximal segments of $L_i(f)$ that consist of roots of hanging subtrees whose low_i points lie in C'. There are many ways to do this, but for the sake of simplicity we will provide a relatively straightforward method that incurs total cost $O(d^3 + d^2 \log n)$. The idea is to collect the children of f, at the beginning of the **for** loop in Line 1, that are ancestors of failed vertices. To do this, we collect the failed vertices $f_1, \dots, f_k$ that are children of f in the F-forest, and then we perform a level-ancestor query (in T) for every f_i to find the child c_i of f that is an ancestor of f_i. Then we keep d copies, $\mathcal{C}_1, \dots, \mathcal{C}_d$, of the collection $\{c_1, \dots, c_k\}$. (We note that some c_i, c_j, for $i, j \in \{1, \dots, k\}$ with $i \neq j$, may coincide. We ignore those repetitions.) For every $i \in \{1, \dots, d\}$, we let $\mathcal{C}_i$ be sorted in increasing order w.r.t. the low_i points. Now, at the beginning of the **for** loop in Line 7, we can use binary search to find in $O(\log n)$ time the (endpoints of the) segment S of $L_i(f)$ that consists of all children of f with their low_i point in C'. Let L and R be the minimum and the maximum, respectively, of S. Then we traverse the list $\mathcal{C}_i$, and, for every $c \in \mathcal{C}_i$ that we meet, we check whether c is in S. (This is done by simply checking whether $L \leq c \leq R$.) If that is the case, then we collect the (possibly empty) subsegment $[L, c-1]$ (i.e., the pair of its endpoints), and we remove $[L, c]$ from S (i.e., we set $L \leftarrow c + 1$). We repeat this process while traversing $\mathcal{C}_i$ until we reach its end, and finally we collect the (possibly empty) remainder of S (i.e., the pair of its endpoints). The collection of the non-empty subsegments we have gathered is precisely $\mathcal{S}_i$.

The cost of this method is as follows. For every failed vertex f, we need time analogous to its number of children in the F-forest to create the collection $\{c_1, \dots, c_k\}$. Then we make d copies of this collection, and we perform a sorting in every one of them. This takes time $O(d \cdot k \log k)$, where k is the number of children of f in the

F-forest. Since this is performed for every failed vertex, it incurs total cost $O(d^2 \log d)$. Now, for every failed vertex f, and every internal component C' that is an ancestor of f, we need $O(\log n)$ time to find the segment S, as described above. This is how we get an additional $O(d^2 \log n)$ cost in total. Now, for this f and C', and for every $i \in \{1, \dots, d\}$, we have to traverse the list $\mathcal{C}_i$ as above (while performing operations that take constant time). Since the size of $\mathcal{C}_i$ equals the number of children of f in the F-forest, this incurs cost $O(d^3)$ in total.

By the analysis above, we have that the total cost of Line 8 is $O(d^3 + d^2 \log n)$.[7] Thus, it remains to upper bound the times that the 2D-range queries in Line 19 are performed. To do this, we introduce the following notation. Let f be a failed vertex, and let $\mathcal{C}$ denote the collection of the internal components that are ancestors of f. Then, for every $C' \in \mathcal{C}$, and every $i \in \{1, \dots, d\}$, we let $\mathcal{S}_i(C')$ denote the collection of the maximal segments of $L_i(f)$ that consist of roots of hanging subtrees of f whose low_i point lies in C'. Then we can see that the number of 2D-range queries in Line 19 during the processing of f (during the outer **for** loop in Line 1) is precisely $\sum_{C' \in \mathcal{C}} \sum_{C} \sum_{i \in \{1,\dots,d\}} |\mathcal{S}_i(C')|$ $(*)$, where the second sum is indexed over the internal components C that are ancestors of f and descendants of C'.

Now fix an $i \in \{1, \dots, d\}$. For every $C' \in \mathcal{C}$, let $S_i(C')$ denote the the maximal segment of $L_i(f)$ that consists of children of f whose low_i point lies in C'. Notice that every segment in $\mathcal{S}_i(C')$ is contained entirely within $S_i(C')$. Since the internal components in $\mathcal{C}$ are pairwise disjoint, we have that the segments in $\{S_i(C') \mid C' \in \mathcal{C}\}$ are pairwise disjoint, and therefore their total number is bounded by $|\mathcal{C}| \leq d$. Since the number of failed vertices is d, the number of childen of f that are ancestors of failed vertices is at most d. It is precisely the existence of those children that may force the segments in $\{S_i(C') \mid C' \in \mathcal{C}\}$ to be partitioned further in order to get $\bigcup\{\mathcal{S}_i(C') \mid C' \in \mathcal{C}\}$. But every such child breaks the segment $S_i(C')$, for a $C' \in \mathcal{C}$, into at most two subsegments. (Recall the analysis above that concerns the implementation of Line 8.) Thus, the segments in $\{S_i(C') \mid C' \in \mathcal{C}\}$ must be partitioned at most d times in order to get $\bigcup\{\mathcal{S}_i(C') \mid C' \in \mathcal{C}\}$. Thus, we have $\sum_{C' \in \mathcal{C}} |\mathcal{S}_i(C')| \leq d + d = O(d)$. This implies that $\sum_{i \in \{1,\dots,d\}} \sum_{C' \in \mathcal{C}} |\mathcal{S}_i(C')| = O(d^2)$, and therefore the expression $(*)$ can be bounded by $O(d^3)$. Since this is true for every failed vertex f, we can bound

[7] In RAM machines with $O(\log n)$ word size, we can use van Emde Boas trees in order to perform the binary searches above as predecessor/successor queries, and so we can reduce the "$\log n$" factor to "$\log \log n$".

the number of the 2D-range queries in Line 19 by $O(d^4)$.

□

7.3.5 Answering the queries

Assume that we have constructed a connectivity graph $\mathcal{R}$ for the internal components of $T \setminus F$, and that we have computed its connected components. Thus, given two internal components C and C', we can determine in constant time whether C and C' are connected in $G \setminus F$, by simply checking whether r_C and $r_{C'}$ are in the same connected component of $\mathcal{R}$ (see Lemma 7.5).

Now let x, y be two vertices in $V(G) \setminus F$. In order to determine whether x, y are connected in $G \setminus F$, we try to substitute x, y with roots of internal components of $T \setminus F$, and then we reduce the query to those roots. Specifically, if x (resp., y) belongs to an internal component C of $T \setminus F$, then the connectivity between x, y is the same as that between r_C, y (resp., x, r_C). Otherwise, if x (resp., y) belongs to a hanging subtree H of $T \setminus F$, then we try to find an internal component that is connected with H through a back-edge. If such an internal component C exists, then we can substitute x (resp., y) with r_C. Otherwise, x, y are connected in $G \setminus F$ if and only if they belong to the same hanging subtree of $T \setminus F$. This idea is shown in Algorithm 56.

Proposition 7.4. *Given two vertices x, y in $V(G) \setminus F$, Algorithm 56 correctly determines whether x, y are connected in $G \setminus F$. The running time of Algorithm 56 is $O(d)$.*

Proof. To prove correctness, we only have to deal with the case that x lies in a hanging subtree H of $T \setminus F$ (Line 6). In this case, we simply have to check whether there is an edge in $G \setminus F$ that connects H with another connected component of $T \setminus F$. Thus, according to Lemma 7.3, we have to check whether H is connected with an internal component of $T \setminus F$ through a back-edge. If such an internal component exists, let C be the lowest among them. Since $p(r_H)$ is a failed vertex and the number of failed vertices that are ancestors of H is bounded by d, we have that there is at least one $i \in \{1, \ldots, d\}$ such that $low_i(r_H)$ lies in C. Thus, the connectivity query for x, y in $G \setminus F$ is equivalent to that for $low_i(r_H), y$. Since $low_i(r_H)$ belongs to the internal component C, eventually the algorithm will terminate, and it will produce the correct result.

Otherwise, if there is no internal component that is connected with H through a back-edge, then H is a connected component of $G \setminus F$, and so y is connected with x in $G \setminus F$ if and only if y also lies within H. Now, if the **for** loop in Line 8 has exhausted

Algorithm 56: query(x, y)

```
1  if x lies in an internal component C and y lies in an internal component C′ then
2      if r_C is connected with r_C′ in R then return true
3      return false
4  end
5  // at least one of x, y lies in a hanging subtree
6  if x lies in a hanging subtree H then
7      // check whether H is connected with an internal component through a
           back-edge
8      for i ∈ {1, . . . , d} do
9          if low_i(r_H) ≠ ⊥ and low_i(r_H) ∉ F then
10             return query(low_i(r_H), y)
11         end
12     end
13     // there is no internal component that is connected with H in G \ F
14     if y lies in H then return true
15     return false
16 end
17 return query(y, x)
```

the search and either $low_d(r_H)$ does not exist, or $low_d(r_H)$ is a failed vertex, then we can be certain that there is no back-edge that connects H with the rest of the graph $G \setminus F$.

Now we will establish the $O(d)$ time-bound. First, given a vertex $x \notin F$, we can determine the connected component of $T \setminus F$ that contains x by finding the nearest failed vertex f that is an ancestor of x. This is done in $O(d)$ time by finding the maximum failed vertex that is an ancestor of x. If no such vertex exists, then x belongs to the internal component with root r. Otherwise, the root of the connected component C of $T \setminus F$ that contains x is given by the child of f that is an ancestor of x. This child is determined in constant time with a level-ancestor query for the ancestor of x whose depth equals $depth(f) + 1$. Now, given the root r_C of a connected component C of $T \setminus F$, we can determine in $O(d)$ time whether C is an internal component or a hanging subtree of $T \setminus F$ by checking whether there is a failed vertex

that is a descendant of r_C. Finally, we can perform the search in Line 8 in $O(d)$ time, if we have the list of failed vertices sorted in increasing order. (We can have this done with an extra cost of $O(d \log d)$ during the update phase.) Then, we can easily check in $O(d)$ time whether there is an $i \in \{1, \ldots, d\}$ such that $low_i(r_C) \in F$, because the list $low_1(r_C), \ldots, low_d(r_C)$ is also sorted in increasing order. $\square$

Chapter 8

On Maximal k-Edge-Connected Subgraphs of Undirected Graphs

8.1 Introduction

8.2 Preliminaries

8.3 The decomposition tree of the maximal k-edge-connected subgraphs

8.4 Maintaining the decomposition tree after insertions

8.5 Data structures for trees and cactuses

8.6 Improved data structures for trees and cactuses

8.7 Sparse certificates for the maximal k-edge-connected subgraphs

8.8 Computing the maximal k-edge-connected subgraphs

8.9 A fully dynamic algorithm for maximal k-edge-connectivity

8.10 Conclusions

8.1 Introduction

A dynamic graph algorithm aims to maintain the solution of a given problem after each update faster than recomputing it from scratch. An algorithm is *fully dynamic* if it supports both insertions and deletions of edges, while it is *incremental* (resp. *decremental*) if it only supports insertions (resp. deletions) of edges. In this chapter, we are

particularly interested in providing algorithms for maintaining the maximal k-edge-connected subgraphs in dynamic graphs.

Determining or testing various notions of edge connectivity of undirected graphs, as well as computing edge-connected components or subgraphs, is a fundamental graph problem, and indeed k-edge connectivity has received much attention in the literature due to its importance in applications. As a matter of fact, finding the maximal k-edge-connected subgraphs of a graph is of significant interest in several areas, such as in the field of databases, social networks, graph visualization etc. [5, 11, 15, 14, 48, 62, 68, 70] (some of those papers refer to the maximal k-edge-connected subgraphs as k-edge-connected components).

The problem of computing maximal k-edge-connected subgraphs appears to be harder than computing k-edge-connected components. In more detail, it is known how to compute the k-edge-connected components in linear time for $k \leq 4$ [63, 67, 36, 50]. On the other hand, linear-time bounds for computing the maximal k-edge-connected subgraphs are known only in the trivial case $k \leq 2$, simply because in these cases the k-edge-connected components coincide with the maximal k-edge-connected subgraphs. As mentioned in [16], the maximal 3-edge-connected subgraphs can be computed with a simple-minded algorithm in $O(mn)$ time in the worst case. Henzinger et al. [42] provided a deterministic algorithm for computing the maximal k-edge-connected subgraphs, for constant k, that runs in $O(n^2 \log n)$ time. Chechik et al. [16] presented an algorithm with running time $O(k^{O(k)}(m + n\log n)\sqrt{n}\,)$ ($O(m\sqrt{n}\,)$ for $k = 3$), which improves the bound of [42] for sparse graphs. Forster et al. [29] gave a Las Vegas randomized algorithm for computing the maximal k-edge-connected subgraphs in $O(km \log^2 n + k^3 n\sqrt{n} \log n)$ expected running time. Recently, Nalam and Saranurak [53] provided a randomized algorithm for computing the maximal k-edge-connected subgraphs in *weighted* undirected graphs in $\widetilde{O}(m \cdot \min\{m^{3/4}, n^{4/5}\})$ time. Also, very recently, Saranurak and Yuan [61] gave a deterministic $O(m + n^{1+o(1)})$-time algorithm for computing the maximal k-edge-connected subgraphs of an undirected graph, for any $k = \log^{o(1)} n$. To achieve this result, Saranurak and Yuan [61] provide a black-box reduction to any decremental algorithm for answering k-edge-connectivity queries, and then they apply the fully dynamic algorithm of Jin and Sun [46]. In the special case of planar graphs, Holm et al. [44] showed how to compute the maximal 3-edge-connected subgraphs in $O(n)$ time.

8.1.1 Overview of our results

In this chapter we present several new results on maximal k-edge-connected subgraphs of undirected graphs. In particular, we provide the following results.

(1) A general framework for maintaining the maximal k-edge-connected subgraphs upon insertions of edges or vertices, by successively partitioning the graph into its k-edge-connected components. This defines a decomposition tree, which can be maintained by using algorithms for the incremental maintenance of the k-edge-connected components as black boxes at every level of the tree. As an application of our new framework, we provide two algorithms for the incremental maintenance of the maximal 3-edge-connected subgraphs, which constitute the main results of this article. These algorithms allow for vertex and edge insertions, interspersed with queries asking whether two vertices belong to the same maximal 3-edge-connected subgraph, and provide a trade-off between time- and space-complexity. The first algorithm has $O(m\alpha(m,n) + n^2 \log^2 n)$ total running time and uses asymptotically optimal $O(n)$ space, where m is the number of edge insertions and queries, and n is the total number of vertices inserted starting from an empty graph. The second algorithm improves the total running time to $O(m\alpha(m,n) + n^2\alpha(n,n))$ (i.e., almost optimal for dense graphs) at the expense of using $O(n^2)$ space. We note that those are the first incremental algorithms for this problem, and thus provide significant improvements over recomputing the maximal 3-edge-connected subgraphs after every insertion.

(2) Building up on results from Benczúr and Karger [8], we provide efficient constructions of (almost) sparse spanning subgraphs that have the same maximal k-edge-connected subgraphs as the original graph. We refer to such subgraphs as *k-certificates*. These are useful in speeding up computations involving the maximal k-edge-connected subgraphs in dense undirected graphs. In particular, we use those certificates to speed up the computation of the maximal k-edge-connected subgraphs. As another application of our k-certificates, we use them to provide a fully dynamic algorithm for maintaining information about the maximal k-edge-connected subgraphs for fixed k. Using the sparsification technique of Eppstein et al. [27], we show the existence of a fully dynamic algorithm with update times that are independent of the number of edges and with constant time for maximal k-edge-connected subgraph queries.

(3) Finally, we give a simple reduction for the computation of the maximal k-edge-connected subgraphs to fully dynamic mincut. By using Thorup's fully dynamic mincut algorithm [66], we obtain a deterministic algorithm that computes the maximal k-edge-connected subgraphs in $O(m + k^{O(1)} n\sqrt{n} \log^{O(1)} n)$ time, for $k = \log^{O(1)} n$.

We believe that our main technical contribution is given by (1). To achieve this result, we provide a structural characterization of the maximal 3-edge-connected subgraphs of an undirected graph by introducing a decomposition tree $\mathcal{T}$ into maximal 3-edge-connected subgraphs, and show how to maintain it efficiently under edge and vertex insertions. $\mathcal{T}$ is a rooted tree whose root corresponds to the whole graph G and is defined recursively by computing the (subgraphs induced by the) k-edge-connected components, for $k \in \{1, 2, 3\}$, of the graphs of the previous level. We proceed recursively in this decomposition until we reach a graph that is 3-edge-connected, which is a maximal 3-edge-connected subgraph of G (and a leaf of $\mathcal{T}$). Therefore, the nodes of $\mathcal{T}$ correspond to subgraphs of G, and the parent relation is given by (vertex) set inclusion. (See Section 8.3.3, and also Figure 8.2.)

Although maintaining the entire tree $\mathcal{T}$ may seem more challenging than maintaining only the maximal 3-edge-connected subgraphs, we show that it is in fact easier to maintain the decomposition tree. This is because $\mathcal{T}$ contains enough information to facilitate its efficient update after new insertions to G, as its nodes corresponds to all successive partitions of G into k-edge-connected components, for $k \in \{1, 2, 3\}$. In fact, if a new edge e is inserted to G, then we only have to locate the deepest subgraph N on $\mathcal{T}$ that contains the endpoints of e, and all changes to $\mathcal{T}$ due to this insertion apply to the subtree of N. However, we do not explicitly maintain the correspondence between nodes of $\mathcal{T}$ and subgraphs of G, as this would require as much as $\Omega(mn)$ space (where m is the number of edges and n is the number of vertices of G), and a much worse time to maintain $\mathcal{T}$ during a sequence of insertions. Instead, we associate with the nodes of $\mathcal{T}$ some data structures that represent the interconnections between the 2- and 3-edge-connected components of the subgraphs that (abstractly) correspond to the nodes of $\mathcal{T}$. To be more precise, we associate to every node of $\mathcal{T}$ that corresponds to a connected component C of its parent (a representation of) the tree of the 2-edge-connected components of C; and to every node of $\mathcal{T}$ that corresponds to a 2-edge-connected component of its parent, we associate (a representation of) the cactus of its 3-edge-connected components. All this information reveals to be

useful for maintaining our tree decomposition with the help of previous incremental approaches [35, 59, 58, 69]. We describe in detail, in Sections 8.5 and 8.6, how we can augment the previous incremental approaches with enough information to suit our purposes.

Notice that in the bounds provided in (1), there is a trade-off between space- and time-complexity. The second algorithm is more time-efficient (at least asymptotically), due to the following two reasons. First, we find a way to use the more sophisticated data structures of La Poutré [58], in order to efficiently maintain the 2- and 3-edge-connected components of the graphs in the various levels of $\mathcal{T}$. Secondly, we use an alternative and more intriguing method for answering nearest common ancestor and level ancestor queries, that are needed in the algorithm for maintaining $\mathcal{T}$. This method relies on the structure of those queries (i.e., they are not completely arbitrary but they depend on modifications already made on $\mathcal{T}$). However, in order to facilitate the efficient answering of those queries, we pay an extra $O(n^2)$ in space. In any case, the n^2 factor in the time-bound of both algorithms is an inherent bottleneck of the basic procedure that we use, for any sequence of insertions, and a lower bound on the worst-case time for a single insertion (see Figure 8.1 and Algorithm 57).

Both our algorithms can efficiently answer queries concerning the maximal 3-edge-connected subgraphs in asymptotically optimal time, plus the time to perform one or two calls to a *find* operation in an underlying disjoint set union (DSU) data structure [64] that maintains the vertex sets of the maximal 3-edge-connected subgraphs. (For the details on this DSU data structure, see Section 8.4.) For instance, given two query vertices x and y, we can report whether x and y belong to the same maximal 3-edge-connected subgraph using two calls to a *find* operation, or given a query vertex x we can report the maximal 3-edge-connected subgraph that contains x in time proportional to its size plus a call to a *find* operation. (See Section 8.10.)

We note that these algorithms can be seen as applications of a more general framework for maintaining the maximal k-edge-connected subgraphs, by relying on algorithms that maintain the k-edge-connected components. We think that it is possible that one can develop a similar framework for maintaining the maximal k-vertex-connected subgraphs, by relying on efficient algorithms for maintaining the k-vertex-connected components. (In particular, for the case $k = 3$, one may rely on the algorithms of [7, 57] for maintaining the 3-vertex-connected components.)

For (2), we show that it is sufficient to compute (a superset of) all the edges

whose endpoints lie in different maximal k-edge-connected subgraphs. Benczúr and Karger [8] provide efficient algorithms that achieve this. From those algorithms we get two different constructions for spanning subgraphs that have the same maximal k-edge-connected subgraphs as the original graph, and there is a trade-off between the time complexity and the size of the output subgraph. Following the terminology of [1], we call such a subgraph a k-certificate. Then, we have a linear-time algorithm for computing a k-certificate with $O(kn \log n)$ edges, and an $O(m \log^2 n)$-time algorithm for computing a k-certificate with $O(kn)$ edges (where m and n denote the number of edges and vertices of the graph, respectively). A key component in those algorithms is the certificates for k-edge connectivity of Nagamochi and Ibaraki [51]. For the details, see Section 8.7. We believe that it is an interesting question whether a k-certificate of $O(kn)$ size can be computed in linear time. This would be trivial if we had a linear-time algorithm for computing the maximal k-edge-connected subgraphs of a graph, but it is still an open problem whether this can be done for $k \geq 3$. Thus, we have to perform the construction of the certificates without explicitly computing the maximal k-edge-connected subgraphs, and this seems to be a challenging task. Then, we give two applications of our k-certificates. First, we use them to speed up previous algorithms for computing the maximal k-edge-connected subgraphs in dense undirected graphs. Furthermore, we apply our k-certificates within the sparsification technique of Eppstein et al. [27] to give a fully dynamic algorithm for maintaining information about the maximal k-edge-connected subgraphs for fixed k. This way, we achieve update times that are independent of the number of edges and constant time for maximal k-edge-connected subgraph queries. For the details, see Section 8.9.

Finally, for (3), we repeatedly find and remove all k'-edge cuts, for $k' < k$. For this purpose, we can rely on the fully dynamic min-cut algorithm of Thorup (Theorem 26 in [66]). Thus, we use this algorithm in order to successively find and remove all k'-edge cuts, for $k' < k$, until we are left with the maximal k-edge-connected subgraphs. This is how we get a deterministic $O(m + k^{O(1)} n\sqrt{n}\,\text{polylog}(n))$-time algorithm for computing the maximal k-edge-connected subgraphs of a graph with m edges and n vertices, for $k = \log^{O(1)} n$. This algorithm is described in Section 8.8.

8.1.2 Organization

The rest of the chapter is organized as follows. First, we provide some preliminaries in Section 8.2. We describe our decomposition tree of the maximal 3-edge-connected subgraphs in Section 8.3.3. This can be seen as an application of a general framework for maintaining the maximal k-edge-connected subgraphs by using algorithms for maintaining the k-edge-connected components, which is described in Sections 8.3.1 and 8.3.2. The details on maintaining the decomposition tree of the maximal 3-edge-connected subgraphs under insertions of new edges and vertices is given in Section 8.4. In Sections 8.5 and 8.6 we present efficient implementations for the data structures that are associated with the nodes of the decomposition tree, in order to efficiently update it. In Section 8.7 we present constructions of sparse certificates for maximal k-edge-connected subgraphs. From this we derive a deterministic $O(m + k^{O(k)} n\sqrt{n}\log n)$-time algorithm for computing the maximal k-edge-connected subgraphs. In Section 8.8 we provide an $O(m + k^{O(1)} n\sqrt{n}\text{polylog}(n))$ deterministic algorithm for computing the maximal k-edge-connected subgraphs, using Thorup's fully dynamic mincut algorithm. Section 8.9 presents our fully dynamic algorithm for maximal k-edge-connected subgraphs. We conclude in Section 8.10 with suggestions for further applications of our decomposition tree.

8.2 Preliminaries

In the sequel, we assume that the reader is familiar with standard graph terminologies, as presented e.g. in [18, 20, 52]. For the sake of completeness, we provide some definitions and state some results that will be used throughout concerning the structure of the 2- and 3-edge-connected components in undirected graphs. Consider the quotient graph Q^k of G that is formed by shrinking every k-edge-connected component into a single vertex, maintaining all inter-connection edges and discarding self-loops. Then the quotient map $\nu : V(G) \rightarrow V(Q^k)$ induces a natural correspondence between the edges of Q^k and some edges of G: that is, for every edge (u, v) of Q^k, there is an edge $(x, y) \in E(G)$ such that $\nu(x) = u$ and $\nu(y) = v$. Now, for $k = 2$, we have that Q^2 is a tree T. The nodes of T correspond to the 2-edge-connected components of G, and the edges of T correspond to the bridges of G. Now let C be a 2-edge-connected component of G. Then we have that $G[C]$ is 2-edge-connected,

and every k-edge-connected component of G that lies in C, for $k \geq 2$, is also a k-edge-connected component of $G[C]$. For $k = 3$, the quotient graph Q^3 of $G[C]$ is a *cactus* S (that is, a connected graph in which every edge belongs to a unique cycle) [21, 35, 59]. The nodes of S correspond to the 3-edge-connected components of $G[C]$ and the 2-edge cuts of Q^3 correspond to the 2-edge cuts of $G[C]$. (We note that these properties generalize to the *cactus of the minimum cuts* of an undirected graph [22].)

Now let us consider how the insertion of a new edge (x, y) to G affects its k-edge-connected components, for $k \leq 3$. (In general, observe that the edge-connectivity for any pair of vertices increases at most by one.) We distinguish four different cases, depending on whether $\lambda(x, y) \in \{0, 1, 2\}$ or $\lambda(x, y) \geq 3$, prior to the insertion of (x, y).

(1) If $\lambda(x, y) = 0$, then x and y belong to two different connected components C_1 and C_2, respectively. Thus the only change that occurs is that $\lambda(u, v) = 1$, for any two vertices $u \in C_1$ and $v \in C_2$.

(2) If $\lambda(x, y) = 1$, then x and y belong to the same connected component C, and lie in two different 2-edge-connected components X and Y, respectively. Let T be the tree of the 2-edge-connected components of $G[C]$, and let $P = X_1, \ldots, X_k$ be the simple path on T with endpoints X and Y (i.e., we have $X_1 = X$ and $X_k = Y$). Then, after inserting (x, y), we have $\lambda(u, v) \geq 2$, for any pair of vertices $u, v \in X_1 \cup \cdots \cup X_k$, and $\lambda(u, v)$ stays the same for any other pair of vertices. In particular, this implies that $X_1 \cup \cdots \cup X_k$ is a new 2-edge-connected component. Now, for every edge (X_i, X_{i+1}), $i \in \{1, \ldots, k-1\}$, of T, let (x_i, y_i) be the edge of G that corresponds to (X_i, X_{i+1}), and let $(x, y) = (y_0, x_k)$ (this is for notational convenience). Then, for every $i \in \{1, \ldots, k\}$, we have $y_{i-1}, x_i \in X_i$, and the remaining changes in the k-edge-connected components of G, after inserting (x, y), are given by supposing that we introduce a virtual edge (y_{i-1}, x_i) to G; thus they are described sufficiently in the following two cases.

(3) If $\lambda(x, y) = 2$, then x and y belong to the same 2-edge-connected component C, and lie in two different 3-edge-connected components of $G[C]$. Let S be the cactus of the 3-edge-connected components of $G[C]$. Then, after inserting (x, y), we have that at least X, and Y are now united into a larger 3-edge-connected component (together with some other nodes of S). To describe precisely the nodes of S that get united into a new 3-edge-connected component, we introduce the concept of the *cycle-path* on S connecting X and Y. Let $Z_1, \ldots, Z_k$ be a simple

path on S with $Z_1 = X$ and $Z_k = Y$. Then the cycle-path Q connecting X and Y on S is the set of nodes consisting of X, Y, and all Z_i, $i \in \{2, \ldots, k-1\}$, such that the edges $(Z_{i-1}, Z_i), (Z_i, Z_{i+1})$ belong to different cycles of S. (Notice that this definition is independent of the choice of the simple path $Z_1, \ldots, Z_k$.) Then, after inserting (x, y), all the nodes of Q are merged into a new maximal 3-edge-connected component. The edge-connectivity between any pair of vertices u and v, not both of which lie in nodes of Q, stays the same. Observe that the new cactus of the 3-edge-connected components of $G[C]$ is given by S where we have "squeezed" every cycle c that contains two consecutive nodes Z and W of Q, by merging Z and W into a new node U. (If Z and W are not connected with an edge on S, then the squeezing of c at Z and W produces two new cycles that meet at U.)

(4) If $\lambda(x, y) \geq 3$, then all the k-edge-connected components of G, for $k \leq 3$, stay the same [23].

8.3 The decomposition tree of the maximal k-edge-connected subgraphs

In this section we provide a general framework for maintaining the maximal k-edge-connected subgraphs of an undirected graph upon insertions of edges or vertices. This framework relies on a structural characterization of the maximal k-edge-connected subgraphs that is derived from the repeated decomposition of the graph into its k-edge-connected components. Thus we get a decomposition tree, which can be maintained by using any algorithm for the incremental maintenance of the k-edge-connected components (that satisfies some properties) as a black box. We provide a concrete application of this framework: two algorithms for maintaining the maximal 3-edge-connected subgraphs. These algorithms provide a trade-off between time- and space-bounds, because they rely on different algorithms for maintaining the 3-edge-connected components, and they utilize the structure of the decomposition tree in different ways.

8.3.1 A general framework for maintaining the k-edge-connected subgraphs

In what follows we let "k-ecc" mean "k-edge-connected component". We will not distinguish between the vertex set of a k-ecc and the subgraph induced by it. Let G be an undirected multigraph with m edges and n vertices. We consider the decomposition tree $\mathcal{T}$ of the maximal k-edge-connected subgraphs of G, which is a rooted tree whose nodes correspond to subgraphs of G. It is defined recursively as follows. (1) The root of $\mathcal{T}$ corresponds to G. (2) Every node of $\mathcal{T}$ that corresponds to a k-edge-connected subgraph of G is a leaf. (3) The children of every node N of $\mathcal{T}$ that is not a leaf correspond to the k-eccs of the subgraph of G corresponding to N. This completes the recursive definition of $\mathcal{T}$. Observe that the size of $\mathcal{T}$ is $O(n)$, because the leaves of $\mathcal{T}$ correspond precisely to the maximal k-edge-connected subgraphs of G (which are at most n), and every node of $\mathcal{T}$ is either a leaf or it has at least two children.

We observe that $\mathcal{T}$ has the following useful property.

Property 8.1. *Let G be an undirected graph, and consider the quotient graph Q that is formed by shrinking a maximal k-edge-connected subgraph of G into a single vertex. Then Q has the same decomposition tree into maximal k-edge-connected subgraphs as G.*

We use this decomposition tree because we want to maintain the maximal k-edge-connected subgraphs of G while new edges or vertices are inserted to it. The definition using successive partitions into k-edge-connected components is convenient, because every maximal k-edge-connected subgraph lies entirely within a k-edge-connected component, and we can utilize algorithms for the incremental maintenance of the k-edge-connected components (that satisfy some properties) in order to maintain the decomposition tree under new insertions.

Now let us describe the changes that take place in $\mathcal{T}$ after a new edge is inserted to G. Suppose a new edge $e = (x, y)$ is inserted to G. Let X and Y be the maximal k-edge-connected subgraphs of G that contain x and y, respectively. If $X = Y$, there is nothing to do. Otherwise, we find the nearest common ancestor N of X and Y on $\mathcal{T}$, and perform the insertion of e to N. Observe that N is the deepest node of $\mathcal{T}$ that contains both x and y. It is sufficient to perform the insertion of e to N, because N is separated from its siblings by a k'-edge cut, for some $k' < k$, and this does not change even if we insert e to N. Thus, all changes in $\mathcal{T}$ after the insertion of e to G take place in the subtree of N. If x and y do not become k-edge-connected in N, then

no change takes place in $\mathcal{T}$. Otherwise, some k-eccs of N get merged into a larger k-ecc. If all vertices of N become k-edge-connected, then N becomes a leaf node. Otherwise, we have to find its children that correspond to the k-eccs of N that get merged, and substitute them with a single child that corresponds to the bigger k-ecc that is formed. Then we have to push down to the new child all the edges that have their endpoints in different k-eccs that got merged. In effect, we re-insert those edges to the child of N that represents the new k-ecc that is formed. This completes the recursive description of the changes that take place in $\mathcal{T}$ after inserting a new edge to G.

In order to efficiently maintain the decomposition tree, we will have to attach more information to it. We discuss this in the following section, where we use as a black box any algorithm that maintains the k-edge-connected components (and satisfies some conditions). In any case, it is interesting to note - and it is essential in proving our time bounds - that the number of edge insertions that can affect the decomposition tree, in any sequence of edges insertions, is $O(kn)$. This is formally stated and proved in Theorem 8.1.

Lemma 8.1. [1] Let G be a graph with n vertices and m edges, such that every maximal k-edge-connected subgraph of G is trivial. Then $m \leq (k-1)(n-1)$.

Theorem 8.1. *Let $k \geq 3$, and let $G_0 = (V, E_0)$ be an empty graph with n vertices and no edges ($E_0 = \emptyset$). Let $e_1, \ldots, e_m$ be a sequence of edges joining vertices of V, and define $E_i = \{e_1, \ldots, e_i\}$ and $G_i = (V, E_i)$, for $i = 1, \ldots, m$. Then, $|\{i \in \{1, \ldots, m\} \mid e_i$ joins two different maximal k-edge-connected subgraphs of $G_{i-1}\}| \leq k(n-1)$.*

Proof. We proceed by induction on the number n of vertices. For $n = 1$, the theorem trivially holds, as there are no possible edge insertions. Assume that the theorem holds for every graph with at most n vertices, for some $n \geq 1$. Given an empty graph with $n+1$ vertices, we will show that at most kn edges can affect the decomposition tree during any sequence $e_1, \ldots, e_m$ of edge insertions.

Let ℓ, $0 \leq \ell \leq m-1$, be the lowest index such that $G_{\ell+1} = (V, E_{\ell+1})$ contains a non-trivial maximal k-edge-connected subgraph (i.e., $G_i = (V, E_i)$, $0 \leq i \leq \ell$ contain only trivial maximal k-edge-connected subgraphs). If no such index ℓ exists, then the final graph G_m has only trivial maximal k-edge-connected subgraphs and the theorem holds by Lemma 8.1. Otherwise, let S be the non-trivial maximal k-edge-connected subgraph that appears in $G_{\ell+1}$ after the edge $e_{\ell+1}$ has been inserted to G_ℓ,

and let S contain d vertices ($d \geq 2$) and t edges. Note that $e_{\ell+1}$ must necessarily be in S. Let $S' = S \setminus e_{\ell+1}$: then S' has $d \geq 2$ vertices and $t-1$ edges, and contains only trivial maximal k-edge-connected subgraphs. By Lemma 8.1 applied to S', we have $t - 1 \leq (k-1)(d-1)$, and therefore $t \leq (k-1)(d-1)+1$.

Now let us consider the quotient graph of $G_{\ell+1}/S$ that is formed by contracting S into one single vertex. This quotient graph has $\ell+1-t$ edges and $(n+1)-d+1 = n-d+2 \leq n$ vertices (since $d \geq 2$). Since we are contracting a maximal k-edge.connected subgraph, the quotient graph has exactly the same decomposition tree as $G_{\ell+1}$. By induction, there are at most $k((n-d+2)-1)-(\ell+1-t) = kn-kd+k-\ell-1+t$ new edges that can be added to the quotient graph and affect its decomposition tree. This number is at most $kn-\ell-d+1$ (since t is at most $kd-k-d+2$). Since we have already added $\ell+1$ edges to the initial graph, we have that, in total, at most $kn-d+2$ edges can affect the decomposition tree. The fact that $d \geq 2$ yields the desired result. □

8.3.2 Maintaining the decomposition tree

In order to be able to update $\mathcal{T}$ after a new insertion to the graph, we need a k-edge-connected components algorithm that satisfies the following properties. The algorithm works on an incremental graph G and maintains a unique label for every k-ecc of G. Furthermore, it supports the following two operations.

1. When a new edge is inserted to G, return the labels of the k-eccs of G that get merged due to this insertion, the label of the new k-ecc that is formed, and the set of the interconnection edges between those k-eccs that get merged.

2. Given a collection of vertex-disjoint graphs $G_1, \ldots, G_t$, join the underlying data structures that the algorithm uses for those graphs into a new data structure for the graph $G = G_1 \cup \cdots \cup G_t$.

We call an algorithm (plus the underlying data structure) that supports these operations an "incremental k-edge-connected components algorithm" (or "a data structure for maintaining the k-edge-connected components").

Now suppose that we are equipped with an incremental k-edge-connected components algorithm. In order to efficiently maintain the decomposition tree $\mathcal{T}$, we

associate with every non-leaf node N a data structure for maintaining the k-edge-connected components. Furthermore, for every k-ecc of N, we have a two-way correspondence between its label maintained by the data structure and the child of N that corresponds to it. Now suppose that an edge e is inserted to N. Then we use operation (1) in order to find the labels $C_1, \ldots, C_t$ of the k-eccs of N that get merged due to the insertion of e. If there is only one such k-ecc returned, there is nothing to do. If all the k-eccs of N are returned, then we make N a leaf node. Otherwise, we get from those labels the corresponding children of N, we merge those children into a new child, and we establish a two-way correspondence between this child and the new k-ecc of N that is formed. Furthermore, we retrieve the associated data structures of the children of N that get merged, we join them into a new data structure using operation (2), and we associate the new data structure with the new child of N. Finally, we insert the interconnection edges between $C_1, \ldots, C_t$ that we got from operation (1) into the new child of N. This completes the recursive description of the update of $\mathcal{T}$ using an incremental k-edge-connected components algorithm as a black box.

We will describe two different methods with which we can perform the merging of nodes of $\mathcal{T}$, with a trade-off in time- and space-complexity. The first method is the simplest one. Let N be a node of $\mathcal{T}$, and let $C_1, \ldots, C_t$ be the children of N that we have to merge. Then we will use the node with the greatest number of children among $C_1, \ldots, C_t$ as the new node in which $C_1, \ldots, C_t$ get merged. So let C be a node among $C_1, \ldots, C_t$ with maximum number of children. Then we discard all nodes in $\{C_1, \ldots, C_t\} \setminus \{C\}$, and we redirect the parent pointer of their children to C. Since this involves at most n nodes at every level, and a parent pointer can be redirected at most $\log n$ times, we get at most $O(n \log n)$ redirections of parent pointers at every level, and $O(n^2 \log n)$ redirections in total. In the second method we maintain all nodes of $\mathcal{T}$ (possibly in a "deactivated" mode) throughout the sequence of all insertions, and thus we may need as much as $\Omega(n^2)$ space (since there are at most $O(n)$ insertions that can affect the decomposition tree). The idea is to use a disjoint-set union data structure DSU_i [64] on the nodes of level i of the tree, for all $i \geq 1$. In order to merge nodes at level i we unite all of them using DSU_i and we choose one of them as the representative. We consider the representative to be an *active* node, in the sense that we can use its parent pointer to move to level $i-1$, whereas the parent pointer of the other nodes that got united will never be used again. Thus, in order to access

the parent of a node N at level i, we use $\mathit{find}_{i-1}(\mathit{parent}(N))$, where *parent* denotes the parent pointer in $\mathcal{T}$, and find_{i-1} is the *find* operation of DSU_{i-1}. Notice that this approach has the advantage that it does not explicitly redirect the parent pointers. Using an optimal implementation for DSU_i [64], we can perform any sequence of m operations find_i or unite_i in $O(m\alpha(m,n))$ time in total. Since at most n *unite* operations can take place at each level, we have an $O(n\alpha(n,n))$ time-bound at every level for the mergings, and an $O(n^2\alpha(n,n))$ time-bound in total.

We note that the n^2 expression in the time-bound is an inherent bottleneck of this approach for maintaining the maximal k-edge-connected subgraphs, for any $k \geq 3$. In order to demonstrate this, let us introduce some concepts. We call the edges that connect two different maximal k-edge-connected subgraphs of a graph *k-interconnection* edges. Then we partition the k-interconnection edges into levels as follows. If an edge participates in a k'-edge cut of the graph, for $k' < k$, we define the level of this edge to be 1. Now suppose that we have defined all d-level edges, for some $d \geq 1$. Then an edge that participates in a k'-edge cut of (a connected component of) the graph that remains after we remove all d'-level edges, for $d' \leq d$, is assigned level $d+1$. Now we observe that our method for updating the decomposition tree explicitly maintains the levels of the k-interconnection edges (because every d-level k-interconnection edge is essentially maintained in the associated data structure of a node of depth d of the decomposition tree). There are sequences of insertions of edges in which $\Omega(n)$ insertions may force $\Omega(n)$ edges to change their level, and thus we need at least $\Omega(n^2)$ time in total to maintain the decomposition tree. An example for $k=3$ is shown in Figure 8.1. (This generalizes easily to any $k > 3$.)

8.3.3 The decomposition tree of the maximal 3-edge-connected subgraphs

Here we apply the framework outlined in the previous section for the special case $k=3$. We focus on the case $k=3$ for the following reasons. (1) There exist very efficient/asymptotically optimal algorithms for the incremental maintenance of the 3-edge-connected components that we can use for maintaining the decomposition tree for the case $k=3$. (2) Although there exist very efficient algorithms for maintaining also the 4- and 5-edge-connected components [25, 23], and they seem applicable in our framework, they involve a lot of technicalities and it would be very tedious to

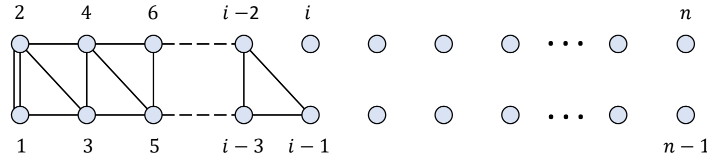


Figure 8.1: An example of a sequence of insertions that can force $\Omega(n)$ 3-interconnection edges to increase their level by one $\Omega(n)$ times. The sequence of insertions, starting from the empty graph with vertices in $\{1, \ldots, n\}$ is $(1,2),(1,2),(3,1),(3,2),(4,2),(4,3),\ldots,(i,i-2),(i,i-1),\ldots,(n,n-2),(n,n-1)$. After the insertion of $(i,i-1)$, we have that $(i,i-1)$ and $(i,i-2)$ are the only 1-level edges, and the level of the other edges $(1,2),\ldots,(i-1,i-2)$ increases by one. Note that $\{1,\ldots,i-1\}$ is a 3-edge-connected component at this point, whereas all the maximal 3-edge-connected subgraphs are trivial.

show how to extend them in order to apply them for our purposes. For $k > 5$ we are not even aware of any efficient incremental algorithms that we could apply.[1] (3) The case $k = 3$ is the simplest one to consider. And yet, even here, we have to extend appropriately the existing data structures and algorithms, and this involves various technicalities.

For the case $k = 3$ we incorporate the algorithms of [35, 59, 58, 69], for maintaining the 3-eccs, in the decomposition tree. Thus, since these algorithms basically maintain the 2-eccs of the 1-eccs, and the 3-eccs of the 2-eccs, we augment the decomposition tree with more levels in order to capture the decomposition into all k'-eccs, for $k' \leq 3$. To be more precise, we consider the decomposition tree $\mathcal{T}$ of the maximal 3-edge-connected subgraphs of G that is formed by repeatedly removing all 1- and 2-edge cuts. The nodes of $\mathcal{T}$ correspond to subgraphs of G. First, the root of $\mathcal{T}$ corresponds to the entire graph. If G is 3-edge-connected, then this is the only node of $\mathcal{T}$, and the decomposition is over. Otherwise, the children of the root correspond to the connected components of G, which we call 1-ecc nodes. Then the children of every 1-ecc node correspond to its 2-edge-connected components, and we call them 2-ecc nodes. Finally, the children of every 2-ecc node correspond to its 3-edge-connected components and we call them 3-ecc nodes. Now, if a 3-ecc node corresponds to a 3-

[1] The fully dynamic algorithm of Jin and Sun [46] seems very difficult to be usable in our framework, because it does not explicitly maintain the k-edge-connected components.

edge-connected subgraph of G, then it is a leaf of $\mathcal{T}$, and a maximal 3-edge-connected subgraph of G. Otherwise, it becomes the root of a subtree of $\mathcal{T}$ that is produced recursively with the same procedure. In this way, we compute the decomposition tree $\mathcal{T}$ of the maximal 3-edge-connected subgraphs of G. (See Figure 8.2 for an example.)

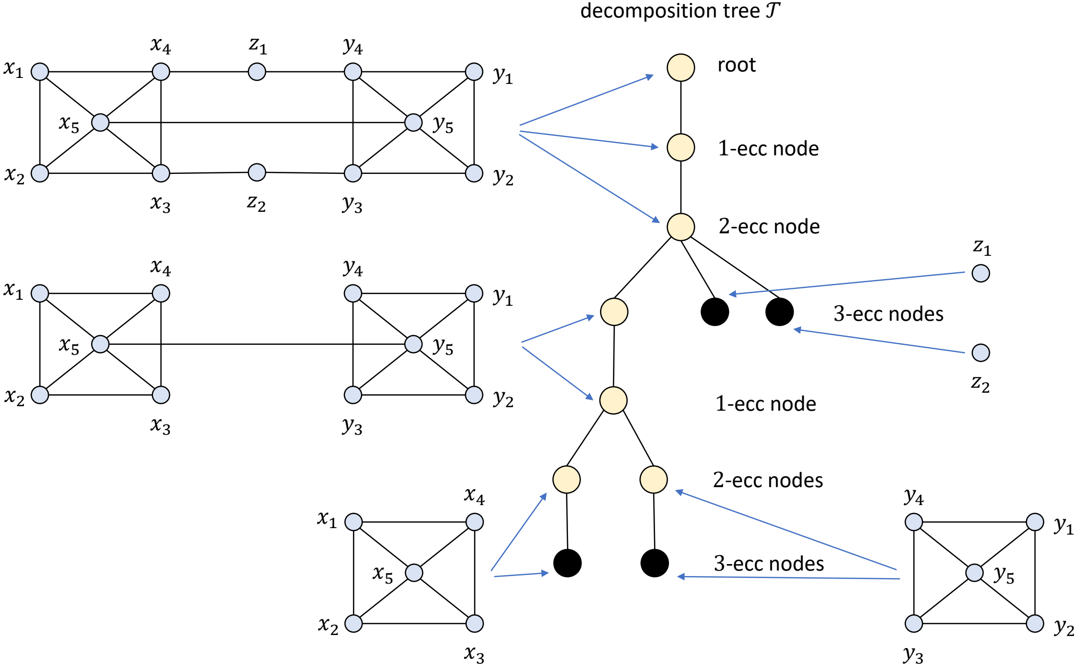


Figure 8.2: The decomposition tree $\mathcal{T}$ of the graph G of Figure 2.1. Notice the correspondence between the nodes of $\mathcal{T}$ and some subgraphs of G. The leaves of $\mathcal{T}$ (coloured in black) correspond to the maximal 3-edge-connected subgraphs of G.

Observe that the size of $\mathcal{T}$ is $O(n)$,[2] since the children of every node N of $\mathcal{T}$ correspond to subgraphs of the graph corresponding to N that partition its vertex set (and although some nodes have only one child, by construction this child must have at least two children or two grandchildren, unless it is a leaf). For convenience, and since no ambiguity arises, we identify every node of $\mathcal{T}$ with the subgraph of G that corresponds to it. (We do not maintain this correspondence explicitly in any data structure, as this would be impractically expensive in terms of space (and therefore time). It is only a conceptual convention that we make, and we will use it extensively in the sequel.) Furthermore, since $\mathcal{T}$ is rooted, we let the nodes of $\mathcal{T}$ be partitioned

[2]However, in order to efficiently maintain $\mathcal{T}$ after the insertion of new edges to G, we may have to attach more information to $\mathcal{T}$, so that its total size will be $O(n^2)$. (See Section 8.4.2.)

into levels according to their depth, where the level of the root is 0. Thus we may speak of a subgraph of G at some level of $\mathcal{T}$. Observe that $\mathcal{T}$ contains $3k$ levels, for some $k \geq 0$, and that every maximal 3-edge-connected subgraph of G is contained at some level $3k'$, for $k' \leq k$, as a leaf. It is useful to note that $\mathcal{T}$ satisfies the following.

Property 8.2. *If we shrink a maximal 3-edge-connected subgraph of G into a single vertex, then the decomposition tree of the resulting graph is given again by $\mathcal{T}$.*

Now we will give a high-level description of the changes that $\mathcal{T}$ undergoes when a new edge (x, y) is inserted to G. Theorem 8.1 implies that at most $3n - 3$ insertions of edges may affect the decomposition tree, in any sequence of edge insertions, where n is the total number of vertices inserted. By Property 8.2, we have that a newly inserted edge can affect the decomposition tree only if its endpoints lie in different maximal 3-edge-connected subgraphs of G. Thus we consider the nearest common ancestor N of the maximal 3-edge-connected subgraphs of G that contain x and y. This corresponds to the deepest node in the decomposition tree that contains both x and y, and it should be clear that the insertion of (x, y) affects only the decomposition of N. We distinguish three different cases, depending on whether (1) N is the root or a 3-ecc node, or (2) N is a 1-ecc node, or (3) N is a 2-ecc node. The whole procedure is summarized in Algorithm 57.

Algorithm 57: Update the decomposition tree $\mathcal{T}$ after inserting the edge (x, y) to the graph G

```
1  procedure insert(x, y)
2  begin
3      X ← the leaf of 𝒯 that contains x
4      Y ← the leaf of 𝒯 that contains y
5      N ← the nearest common ancestor of X and Y on 𝒯
6      C_1 ← the child of N that contains x
7      C_2 ← the child of N that contains y
8      if N is the root of 𝒯 or a 3-ecc node then
9          merge C_1 and C_2 into a new 1-ecc node
10     end
11     else if N is a 1-ecc node then
12         T ← the tree of the 2-edge-connected components of N
13         X_1, ..., X_k ← the path on T with endpoints C_1 and C_2
14         foreach i ∈ {1, ..., k − 1} do
15             (x_i, y_i) ← the edge of N that corresponds to (X_i, X_{i+1}) of T
16         end
17         (y_0, x_k) ← (x, y)
18         foreach i ∈ {1, ..., k} do
19             S_i ← the cactus of the 3-edge-connected components of X_i
20             D_(i,1), ..., D_(i,t(i)) ← the cycle-path on S_i with endpoints the 3-ecc of X_i that
                 contains y_{i−1} and the 3-ecc of X_i that contains x_i
21             merge3ecc(D_(i,1), ..., D_(i,t(i)))
22         end
23         merge all X_1, ..., X_k into a new 2-ecc node of 𝒯
24     end
25     else if N is a 2-ecc node then
26         S ← the cactus of the 3-edge-connected components of N
27         D_1, ..., D_k ← the cycle-path on S with endpoints C_1 and C_2
28         if V(S) = {D_1, ..., D_k} then
29             R ← the grandparent of N on 𝒯
30             if N is the only 2-ecc of R then condense the subtree of R into R
31             else condense all the proper descendants of N into a new 3-ecc node
32         end
33         else
34             merge3ecc(D_1, ..., D_k)
35             insert(x, y)
36         end
37     end
38 end
```

Algorithm 58: merge the children $D_1, \ldots, D_k$ of a 2-ecc node X

1. **procedure** `merge3ecc`$(D_1, \ldots, D_k)$
2. **begin**
3.   **for** $i \in \{1, \ldots, k\}$ **do**
4.   **if** *D_i is a leaf of $\mathcal{T}$* **then**
5.   let D_i', D_i'' and D_i''' be a new 1-ecc, 2-ecc and 3-ecc node, respectively, where D_i', D_i'' and D_i''' correspond to the same graph as D_i
6.   *parent*$(D_i''') \leftarrow D_i''$
7.   *parent*$(D_i'') \leftarrow D_i'$
8.   *parent*$(D_i') \leftarrow D_i$
9.   **end**
10.   **end**
11. $S \leftarrow$ the cactus of the 3-edge-connected components of X
12. $\mathcal{E} \leftarrow$ the edge set of $S[D_1 \cup \cdots \cup D_k]$
13.   merge all $D_1, \ldots, D_k$ into a new 3-ecc node
14.   **foreach** *edge e in $\mathcal{E}$* **do**
15.   `insert`(e)
16.   **end**
17. **end**

8.3.3.1 N is the root or a 3-ecc node

In this case (x, y) joins two different connected components C_1 and C_2 of N, and so it becomes a bridge of N that connects them. Thus we only have to merge C_1 and C_2 into a new 1-ecc node (see Fig. 8.3).

8.3.3.2 N is a 1-ecc node

In this case (x, y) joins two different 2-edge-connected components C_1 and C_2 of N. Without loss of generality, assume that $x \in C_1$ and $y \in C_2$. Let T be tree of the 2-edge-connected components of N, and let $P = X_1, \ldots, X_k$ be the path on T with endpoints C_1 and C_2. (Thus we have $X_1 = C_1$ and $X_k = C_2$.) Then the vertices that are contained in $X_1, \ldots, X_k$ become 2-edge-connected, and for every pair of vertices not both of which belong to $X_1 \cup \cdots \cup X_k$ the edge-connectivity remains the same. Furthermore,

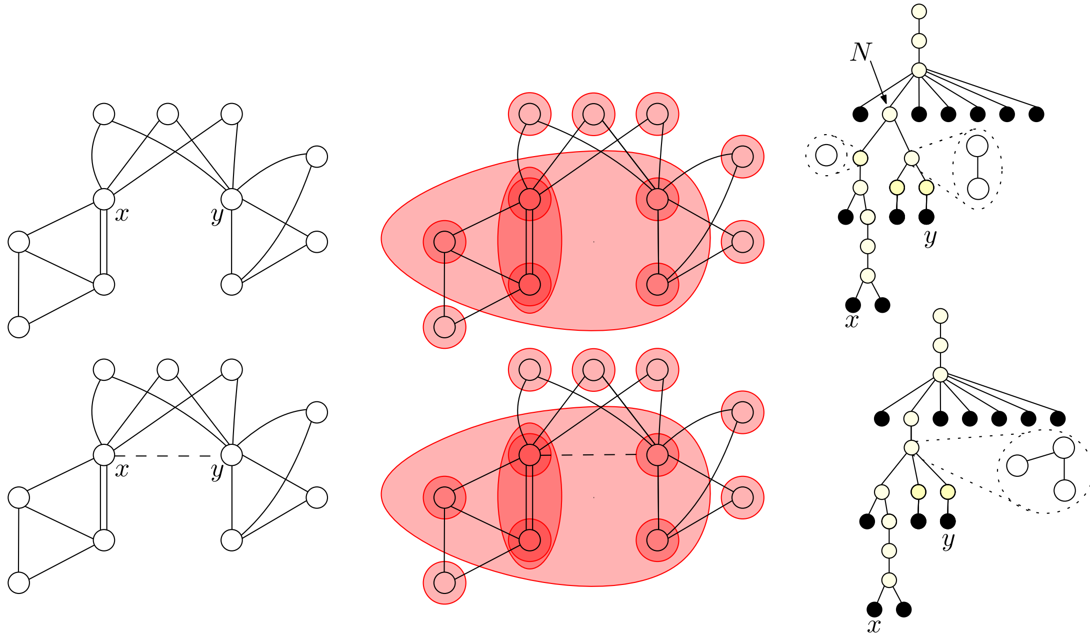


Figure 8.3: Here we show the changes to the decomposition tree on the insertion of the edge (x, y). The nearest common ancestor N of (the maximal 3-edge-connected subgraphs that contain) x and y is a 3-ecc node. Therefore, we need to know the two children of N that contain x and y; then we retrieve the associated trees, join them properly with (an equivalent of) the edge (x, y), and then we merge the two nodes. Finally, we associate with it the new tree that has been formed. In this figure, the middle graphs show the decomposition of the graph into 3-edge-connected components at every level. Also, in the decomposition trees, we can see the trees associated with the nodes we have to merge and the tree associated with the merged node. (The new edge (x, y) is stored inside this tree.) The 3-edge-connected components are highlighted with pink colour. Lower intensity signifies greater depth in the decomposition tree.

for every $i \in \{1, \dots, k-1\}$, let (x_i, y_i) be the bridge of N that corresponds to the edge (X_i, X_{i+1}) of P, and let also $(x, y) = (y_0, x_k)$ (this is for notational convenience). Then, every pair of edges in $\{(x_i, y_i) \mid i \in \{1, \dots, k-1\}\} \cup \{(x, y)\}$ is a 2-edge cut of N. Thus we have to merge all 2-ecc nodes $X_1, \dots, X_k$ into a new 2-ecc node C, and no change takes place on $\mathcal{T}$ outside the subtree of C.

Now we have to consider the changes that possibly take place in the subtree of C. For every $i \in \{1, \dots, k\}$, let S_i be the cactus of the 3-edge-connected components of X_i, and let Q_i be the cycle-path on S_i with endpoints the 3-ecc of X_i that contains y_{i-1} and the 3-ecc of X_i that contains x_i. Let also $D_{(i,1)}, \dots, D_{(i,t(i))}$ be the subgraphs of X_i that correspond to the nodes of Q. Then, the vertices that are contained in $D_{(i,1)}, \dots, D_{(i,t(i))}$

become 3-edge-connected; furthermore, for every pair of vertices of X_i not both of which lie in $D_{(i,1)} \cup \cdots \cup D_{(i,t(i))}$ the edge-connectivity remains the same. Thus we have to merge all $D_{(i,1)}, \ldots, D_{(i,t(i))}$ into a new 3-ecc node D_i; furthermore, no change in the subtree of X_i takes place outside the subtree of D_i. Now we have to consider the graphs $D_{(i,1)}, \ldots, D_{(i,t(i))}$ as the connected components of the new graph D_i. However, in order to maintain the decomposition subtree of D_i on $\mathcal{T}$, we have to take care of two things. First, some graphs $D_{(i,j)}$, for $j \in \{1, \ldots, t(i)\}$, might be leaves (prior to the insertion of (x, y)). This means that they are 3-edge-connected subgraphs, and so we have to include them in the subtree of D_i by expanding their corresponding nodes on $\mathcal{T}$ with the addition of three intermediary nodes. And secondly, there might exist edges in X_i that connect some of $D_{(i,1)}, \ldots, D_{(i,t(i))}$. (Note that these edges correspond naturally to those of S_i that connect the nodes of Q_i.) These edges are now included in D_i, and so we have to consider their effect on the decomposition subtree of D_i. Thus, in order to capture fully the effect on $\mathcal{T}$ of the insertion of (x, y), we have to re-insert those edges to G. Observe that these edges constitute parts of 1- or 2-edge cuts of D_i, and so they will be re-inserted only once in order to perform the insertion of (x, y). However, their re-insertion may force other edges of G, that lie in graphs on deeper levels, to be re-inserted to G. Nevertheless, the nearest common ancestor involved in each such re-insertion will either be a 3-ecc or a 1-ecc node. As a consequence, we note that no new maximal 3-edge-connected subgraphs will be formed after the insertion of (x, y) to G. Figure 8.4 is an example of this case.

We refer to the three step process of (a) merging all $D_{(i,1)}, \ldots, D_{(i,t(i))}$ into a single node D_i, (b) expanding every node $D_{(i,j)}$ that is a leaf, for $j \in \{1, \ldots, t(i)\}$, with the addition of three intermediary nodes, and (c) re-inserting into G the inter-edges between the subgraphs $D_{(i,1)}, \ldots, D_{(i,t(i))}$ of X_i, as *merging* the 3-ecc nodes $D_{(i,1)}, \ldots, D_{(i,t(i))}$ into D_i. This procedure is shown in Algorithm 58. (The variable "parent" in Lines 6, 7 and 8 denotes the parent relation of $\mathcal{T}$.)

8.3.3.3 N is a 2-ecc node

In this case (x, y) joins two different 3-edge-connected components C_1 and C_2 of N. We note that this is the only case in which the formation of new maximal 3-edge-connected subgraphs of G may take place (by merging together smaller maximal 3-edge-connected subgraphs of G). Let S be cactus of the 3-edge-connected components of N, and let $Q = X_1, \ldots, X_k$ be the cycle-path on S with endpoints C_1

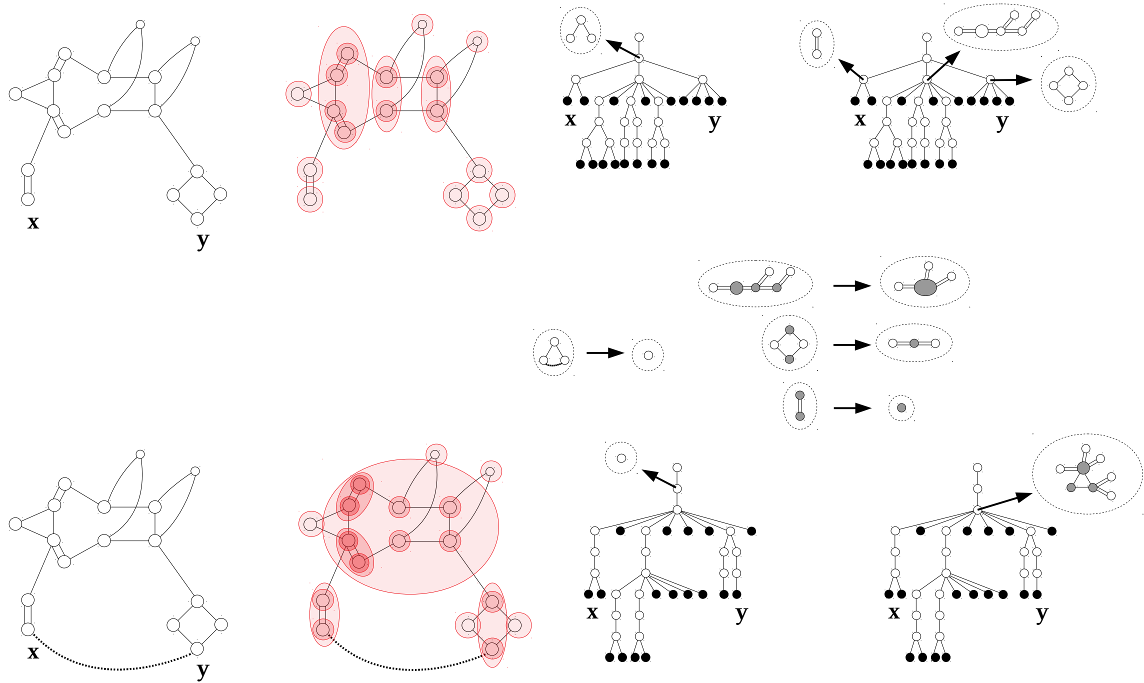


Figure 8.4: Here we insert the edge (x, y) to the graph. This joins two different 2-edge-connected components (at the first decomposition level), which lie in the tree T associated with the nearest common ancestor of **x** and **y**. The path of those 2-eccs on T consists of three nodes, which we have to merge. So first we have to retrieve their associated cactuses (which are shown in this figure), to merge some of their nodes using `merge3ecc`, and then to join them on a new cactus using (essentially) the edges of the cycle. Here we can see the effect of `merge3ecc` on the three cactuses. With gray are shown the nodes that we have to merge. The resulting three cactuses get joined on their gray (compressed) nodes along the edges of the triangle, one of which is (essentially) the new one, and the other two lay on a higher level (inside the tree T of the 2-eccs). Thus, those two edges got pushed down one level. Also notice that the height of the decomposition tree increased.

and C_2. Then the vertices that are contained in $X_1, \dots, X_k$ become 3-edge-connected, and for every pair of vertices not both of which belong to $X_1 \cup \dots \cup X_k$ the edge-connectivity remains the same. Thus, if Q contains all the nodes of S, then N becomes 3-edge-connected. Therefore, if N prior to the insertion of (x, y) was the only 2-edge-connected component of its grandparent R on $\mathcal{T}$, then this means that R becomes 3-edge-connected, and so its subtree on T is condensed into R. Otherwise, all the proper descendants of N are condensed into a new single 3-ecc node of $\mathcal{T}$, which corresponds to the new maximal 3-edge-connected subgraph that has been formed.

Now suppose that Q does not contain all the nodes of S. Then we have to perform

a merging of the 3-ecc nodes $X_1, \ldots, X_k$ into a new node D, and then repeat the insertion of the edge (x, y). Let X and Y be the maximal 3-edge-connected subgraphs of G that contain x and y, respectively. Then the nearest common ancestor of X and Y on $\mathcal{T}$ is a descendant of D, and it can either be a 3-ecc node or a 1-ecc node (in which case we are work as previously), or a 2-ecc node again. In the last case, either the formation of a new maximal 3-edge-connected subgraph of G will take place (and we are done), or we will have to merge again some 3-ecc nodes of $\mathcal{T}$ into a new node and then repeat the insertion of (x, y). Observe that, eventually, this process must terminate, since every time that we have to merge some 3-ecc nodes into a new node, we leave some of their siblings on the same level, and then the computation involves only the subtree of the new node.

Figure 8.5 is a simple example of this case, where the whole graph becomes 3-edge-connected.

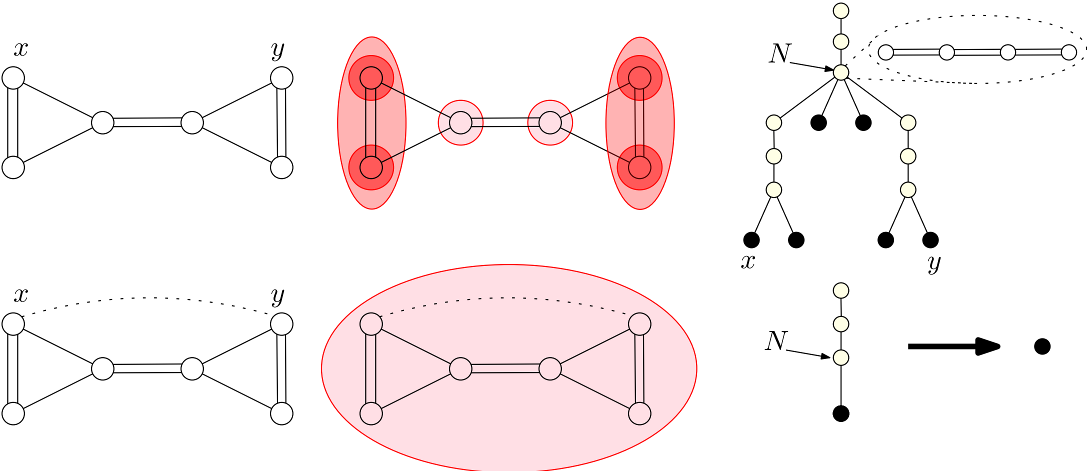


Figure 8.5: This is a simple example where the insertion of a new edge (x, y) makes the whole graph 3-edge-connected. This happens immediately, because the nearest common ancestor N of x and y is a 2-ecc node, where the associated cactus is the cycle-path connecting the nodes that contain x and y. Thus we merge all leaves of N, we discard the whole subtree N, and assign as a child of N the new maximal 3-edge-connected subgraph that has been formed. Notice that the parent of N has only one child, and the parent of the parent of N has again only one child. Thus, we replace the grandparent of N with the child of N, and we discard N, its parent, and its grandparent.

8.4 Maintaining the decomposition tree after insertions

As seen in Algorithms 57 and 58, in order to efficiently update $\mathcal{T}$ after inserting an edge (x, y) to G, we have to provide efficient implementations for the following procedures:

(1) Find the leaves X and Y of $\mathcal{T}$ that contain x and y, respectively.

(2) Find the nearest common ancestor N of X and Y on $\mathcal{T}$.

(3) Find the grandchild of N that contains a particular vertex $v \in N$. (This is needed in Line 20 of Algorithm 57, in order to find the 3-eccs of X_i that contain y_{i-1} and x_i.)

(4) Merge sets of nodes of $\mathcal{T}$ (lying on the same level) into a new single node.

(5) Find the nodes of $\mathcal{T}$ that have to get merged, and the inter-connection edges between the subgraphs that correspond to them.

We will provide two different solutions for this set of procedures. In Section 8.4.1 we rely on the ideas and the data structures of [35, 59, 69], in order to provide an algorithm that uses $O(n)$ space and runs in $O(n^2 \log^2 n + m\alpha(m, n))$ time, for any sequence of m edge and n vertex insertions. In Section 8.4.2 we rely on [59] and [58], in order to provide an algorithm that uses $O(n^2)$ space and runs in $O(n^2\alpha(n, n) + m\alpha(m, n))$ time, for any sequence of m edge and n vertex insertions. We note that the first algorithm is more readily amenable to implementations. The second algorithm, although it is asymptotically more time-efficient, it uses the sophisticated data structures of [58] (summarized and expanded in Section 8.6), and it may be less efficient than the first algorithm in practice.

In both algorithms we use the same solution for (1): an optimal disjoint set union data structure DSU_{3ecs} [64], that operates on $V(G)$, and uses as representatives the leaves of T (which correspond bijectively to the maximal 3-edge-connected subgraphs of G). DSU_{3ecs} supports the operations *find*$_{3ecs}(x)$ and *unite*$_{3ecs}(x, y)$. *find*$_{3ecs}(x)$ returns a pointer to the maximal 3-edge-connected subgraph of G that contains x, and *unite*$_{3ecs}(x, y)$ unites the sets of vertices of the maximal 3-edge-connected subgraphs of G that contain x and y. DSU_{3ecs} uses $O(n)$ space, and it can perform any sequence of m operations *find*$_{3ecs}$ and *unite*$_{3ecs}$ in $O(m\alpha(m, n))$ time in total [64]. (Thus we get this expression in the time-bounds of both algorithms.)

Furthermore, for (5) we rely on data structures associated with the nodes of $\mathcal{T}$, that represent trees or cactuses. In particular, recall that (the graph corresponding to) every 1-ecc node X is a connected component of (the graph corresponding to) its parent, and the children of X correspond to its 2-edge-connected components. Thus, we associate with every 1-ecc node a data structure that represents the tree of its 2-edge-connected components. This data structure is used in order to find the children of X that we have to merge (in Line 13 of Algorithm 57), and the inter-connection edges between (the graphs that correspond to) those children (in Line 15 of Algorithm 57). Similarly, every 2-ecc node X has an associated data structure that represents the cactus of its 3-edge-connected components. This data structure is used in order to find the children of X that we have to merge (in Lines 20 and 27 of Algorithm 57), and the inter-connection edges between (the graphs that correspond to) those children (in Line 12 of Algorithm 58). More details about the operations supported by these data structures will be given in Section 8.4.1.

Finally, we note that the $O(n^2)$ expression in the time-bounds of both algorithms is a bottleneck in their total running time. In fact, Lines 34 and 35 of Algorithm 57 and Line 15 of Algorithm 58 may create a sequence of recursive calls, so that, even for a single edge insertion, we may have to perform $O(n^2)$ re-insertions of already inserted edges. However, since by Theorem 8.1 there are only $O(n)$ edge insertions that can affect the decomposition tree, and every re-insertion (performed internally by the algorithm) can only affect deeper levels than before, and there can be only $O(n)$ levels in the decomposition tree, we have that any sequence of edge insertions to G can initiate at most $O(n^2)$ calls to procedure `insert` of Algorithm 57.

8.4.1 An $O(n^2 \log^2 n + m\alpha(m,n))$-time algorithm for the incremental maintenance of $\mathcal{T}$

An efficient solution for (2) is to use the top-trees data structure of Alstrup et al. [6]. With this data structure we can perform nearest common ancestor queries in dynamic trees with n nodes in amortized $O(\log n)$ time per query. Furthermore, with top-trees we can also answer level ancestor queries within the same time-bounds. Thus we can solve (3) efficiently with a query for the $d+2$ level ancestor of V, where V is the leaf of $\mathcal{T}$ that contains v, and d is the level of N. (We may assume that every node of $\mathcal{T}$ has an attribute for its level on $\mathcal{T}$.)

The merging of nodes in (4) can be performed by redirecting the parents of the children of the nodes with the least number of children to the node with the most number of children (breaking ties arbitrarily), and then discarding these nodes. (We may assume that every node of $\mathcal{T}$ has an attribute for its number of children.) In other words, suppose that we have to merge the nodes $X_1, \dots, X_k$, where X_1 has the greatest number of children among $X_1, \dots, X_k$. Then we redirect the parents of the children of $X_2, \dots, X_k$ to X_1, and then we discard $X_2, \dots, X_k$ (and all the information associated with them). Since the nodes of every level of $\mathcal{T}$ are $O(n)$, this procedure ensures that at most $O(n \log n)$ redirections can take place in total in every level. However, for every such redirection, a deletion and an insertion of an edge of $\mathcal{T}$ must take place, and this takes $O(\log n)$ amortized time using the top-trees. Thus we get an $O(n \log^2 n)$ time-bound for every level of $\mathcal{T}$, and since there are at most $O(n)$ levels in $\mathcal{T}$, we get the $O(n^2 \log^2 n)$ time-bound in total.

Now, for (5), we assume, as above, that every 1-ecc node has an associated data structure that represents the tree of its 2-edge-connected components, and every 2-ecc node has an associated data structure that represents the cactus of its 3-edge-connected components. We use the associated data structures for the trees of the 2-edge-connected components in order to be able to perform efficiently the following. In Line 13 of Algorithm 57, we access the associated data structure T of N, in order to find its children $X_1, \dots, X_k$ that we have to merge in Line 23. Thus we assume that the data structure for trees supports the operation `compressPath(T, x, y)`, which, given a pointer T to a tree, and pointers x and y to nodes of this tree, it finds the simple path P that connects these nodes on the tree, returns pointers to the nodes of P and the edges of P, and then merges all nodes of P into a new node and also returns a pointer to this node. We assume that the nodes and the edges of T have some information associated with them. Specifically, since every edge e of T corresponds essentially to a bridge of N, and this bridge is an edge (x, y) of G, we let e point to the edge (x, y) of G. This information is needed for Lines 15 and 20. Furthermore, since every node v of T corresponds essentially to a child V of N, we let v point to V. This is used precisely in order to find the children $X_1, \dots, X_k$ of N that we have to merge. Conversely, every child of N must point to its corresponding node of T, in order to be able to find the endpoints of the path on T that we have to find with the call to `compressPath`. Finally, since we may have to merge some children of the root or of a 3-ecc node in Line 9 of Algorithm 57, we also have to link the associated (representations of the) trees

of those children, with the addition of an edge that corresponds to the one that has been (re-)inserted to G. Specifically, suppose that an edge (x,y) is (re-)inserted to G, and that the nearest common ancestor N of X and Y is the root or a 3-ecc node, where X and Y are the leaves of $\mathcal{T}$ that contain x and y, respectively. Let C_1 and C_2 the the children of N that contain x and y, respectively. Then, in Line 9, we have to merge C_1 and C_2 into a new 1-ecc node C. Now, the data structure associated with C must represent a tree that is formed by linking the tree represented by the associated data structure to C_1 with the tree represented by the associated data structure to C_2, through the addition of an edge $e=(u,v)$ with endpoints corresponding to the children of C_1 and C_2 that contain x and y, respectively. (These children of C_1 and C_2 can be found with a query for the $d+2$ level ancestors of X and Y, respectively, where d is the level of N.) Furthermore, since e corresponds essentially to (x,y), the associated information to e must be a pointer to (x,y). Thus, the data structure for trees must support the operation $\mathtt{joinTrees}(\mathtt{T}_1,\mathtt{T}_2,(\mathtt{u},\mathtt{v}))$, which, given a pointer T_1 to a (representation of a) tree $\tilde{T}_1$, a pointer T_2 to a (representation of a) tree $\tilde{T}_2$, a pointer u to a node $\tilde{u}$ of $\tilde{T}_1$, a pointer v to a node $\tilde{v}$ of $\tilde{T}_2$, and some extra information to be associated with the edge $(\tilde{u},\tilde{v})$, links the trees $\tilde{T}_1$ and $\tilde{T}_2$ through the addition of the edge $(\tilde{u},\tilde{v})$.

The associated data structures for the cactuses of the 3-edge-connected components of 2-ecc nodes are used to perform the analogous operations that are performed by the data structures for trees. Specifically, in Lines 20 and 27 of Algorithm 57, we access the associated data structure for the 3-edge-connected components of a node X of $\mathcal{T}$, in order to find its children that we have to merge later on (with a call to procedure `merge3ecc`). Thus we assume that the data structure for cactuses supports the operation $\mathtt{compressCyclePath}(\mathtt{S},\mathtt{x},\mathtt{y})$, which, given a pointer S to a cactus, and pointers x and y to nodes of this cactus, it finds the cycle-path Q on S with endpoints these nodes, it returns pointers to the nodes of Q and the edges of S between the nodes of Q, and then merges all nodes of Q into a new node and also returns a pointer to this node. We assume that the nodes and the edges of S have some information associated with them. Specifically, since every edge e of S corresponds essentially to an edge of X, and this edge is an edge (x,y) of G, we let e point to (x,y). This information is needed for Line 12 of Algorithm 58. Furthermore, since every node v of S corresponds essentially to a child V of X, we let v point to V. This is used precisely in order to find the children of X that we have to merge. Conversely, every child of

X must point to its corresponding node of S, in order to be able to find the end-points of the cycle-path on S that we have to find with the call to `compressCyclePath`. Finally, since we may have to merge the children $X_1, \dots, X_k$ of the 1-ecc node N in Line 23 of Algorithm 57, we also have to link the associated (representations of the) cactuses of those children, with the addition of a cycle that corresponds to the one that has been formed by the (re-)insertion of (x, y) to G. Specifically, when we merge $X_1, \dots, X_k$ into a new 2-ecc node X, the associated cactus of X must be that which is formed by joining the cactuses associated with $X_1, \dots, X_k$ with a cycle corresponding to $D_1, \dots, D_k$, where D_i, for $i \in \{1, \dots, k\}$, is the new 3-ecc node that has been formed by merging $D_{(i,1)}, \dots, D_{(i,t(i))}$ through the call `merge3ecc` in Line 21. Furthermore, the edges of the new cactus that correspond to the edges $(D_1, D_2), \dots, (D_{k-1}, D_k), (D_k, D_1)$ of X, must correspond to the edges $(x_1, y_1), \dots, (x_{k-1}, y_{k-1}), (y, x)$. This information is needed in Line 12 of Algorithm 58. Thus, the data structure for cactuses must support the operation $\mathsf{joinCactuses}(\mathsf{S_1}, \dots, \mathsf{S_k}, (\mathsf{d_1}, \mathsf{d_2}), \dots, (\mathsf{d_k}, \mathsf{d_1}))$, which, given pointers $S_1, \dots, S_k$ to (representations of) cactuses $\tilde{S}_1, \dots, \tilde{S}_k$, pointers $d_1, \dots, d_k$ to nodes $\tilde{d}_1, \dots, \tilde{d}_k$ of $\tilde{S}_1, \dots, \tilde{S}_k$, respectively, and some extra information to be associated with the edges $(\tilde{d}_1, \tilde{d}_2), \dots, (\tilde{d}_k, \tilde{d}_1)$, links the cactuses $\tilde{S}_1, \dots, \tilde{S}_k$ through the addition of the cycle $(\tilde{d}_1, \tilde{d}_2), \dots, (\tilde{d}_k, \tilde{d}_1)$.

In Section 8.5 we provide efficient implementations for the associated data structures for trees and cactuses. These implementations use size $O(n)$ for a collection of trees or cactuses with n nodes, and can perform any sequence of operations in $O(n \log n)$ time in total. We use a data structure for trees in every $3k + 1$ level of the tree, and a data structure for cactuses in every $3k + 2$ level of the tree. Since the number of nodes of the trees (of the 2-eccs) or of the cactuses (of the 3-eccs) that correspond to the nodes of every level of $\mathcal{T}$ can be at most n, and there are at most $O(n)$ levels on $\mathcal{T}$, we thus have that the operations in the associated data structures can take time $O(n^2 \log n)$ in total for any sequence of edge insertions to G.

Finally, Algorithm 59 shows how we can handle the insertion of a new vertex v to G. We simply introduce three new nodes C, C', and C'' (an 1-ecc node, a 2-ecc node, and a 3-ecc node, respectively), that correspond to v, and we set *parent*$(C) \leftarrow$ *root*, *parent*$(C') \leftarrow C$, and *parent*$(C'') \leftarrow C'$. Since the leaf of $\mathcal{T}$ that contains v is C'', we also set the representative *find*$_{3ecs}(v) \leftarrow C''$. We also associate a (representation of a) trivial tree to C, and a (representation of a) trivial cactus to C'. The DSU data structure and the data structures in Section 8.5 allow for new node insertions, and the same

time-bounds hold (where n is interpreted as the total number of vertices that will have been inserted to G at the moment we estimate the time that it took to perform all the operations of the algorithm so far).

Algorithm 59: Update the decomposition tree $\mathcal{T}$ after inserting a new vertex v to the graph G

```
1  procedure insert(v)
2  begin
3      let C, C′ and C″ be a new 1-ecc, 2-ecc and 3-ecc node, respectively
4      parent(C) ← root of 𝒯
5      parent(C′) ← C
6      parent(C″) ← C′
7      let the representative of v in DSU_3ecs be C″
8      initialize a new trivial tree T, and associate it with C
9      initialize a new trivial cactus S, and associate it with C′
10 end
```

8.4.2 An $O(n^2\alpha(n,n)+m\alpha(m,n))$-time algorithm for the incremental maintenance of $\mathcal{T}$

Now we will describe a more time-efficient algorithm to handle insertions to the graph. (This algorithm, however, uses $O(n^2)$ space.) First, we can handle the operations in (5) exactly as we did in Section 8.4.1, although here we use the more sophisticated data structures for trees and cactuses that are described in Section 8.6. The implementations given in Section 8.6 use size $O(n)$ for a collection of trees or cactuses with n nodes, and can perform any sequence of operations in $O(n\alpha(n,n))$ time in total. Since we use a data structure for trees in every $3k+1$ level, and a data structure for cactuses in every $3k+2$ level, and there are at most $O(n)$ levels in $\mathcal{T}$, we thus we get the $O(n^2\alpha(n,n))$ expression in the total time-bound .

To perform the merging of nodes in (4) we use an optimal disjoint set union data structure DSU_i [64], on every level i of $\mathcal{T}$, that operates on the node set of that level and uses as representatives nodes on that level that we consider to be "active". The function of an active node is to represent all the nodes that have been merged with it; initially, all nodes of $\mathcal{T}$ are active, and, throughout, the parent pointer on $\mathcal{T}$ exists only

for the active nodes. DSU_i supports the operations $\textit{find}_i(x)$ and $\textit{unite}_i(x, y)$. $\textit{find}_i(x)$ returns a pointer to the active node that has been merged with x, and $\textit{unite}_i(x, y)$ merges the sets of nodes that are represented by the active nodes x and y, and sets as representative one of x or y (while deactivating the other). Thus, in order to find the parent of an active node X of $\mathcal{T}$ on level i, we use $\textit{find}_{i-1}(\textit{parent}(X))$. DSU_i uses $O(n)$ space, and it can perform any sequence of m operations $\textit{find}_i$ and $\textit{unite}_i$ in $O(m\alpha(m, n))$ time in total [64].

To perform efficiently the operations (2) and (3), we basically maintain, for every inter-connection edge $e = (x, y)$ of G (that is, for every edge e whose endpoints, at the time of its insertion, lie in different maximal 3-edge-connected subgraphs of G), two paths of $\mathcal{T}$ whose nodes contain an endpoint of e, that start from the leaves of $\mathcal{T}$ that contain x and y, and go up until at least their nearest common ancestor. In order to achieve this, we augment the information associated with $\mathcal{T}$ as follows. For every node Z of $\mathcal{T}$ we associate a linked list L_Z, where every element of L_Z corresponds to an endpoint of an inter-connection edge that lies in an ancestor of Z. Thus, the existence of those lists means that we may need as much as $O(n^2)$ space (since the number of inter-connection edges can be at most $O(n)$, and the number of levels of $\mathcal{T}$ is $O(n)$). Every element of L_Z is a pointer that points to an element of L_C, for some child C of Z. Also, every element of L_Z has a pointer to Z (so that we can find in constant time the node in whose associated list this pointer lies). Thus we may say that a pointer of L_Z lies in Z. Finally, for every inter-connection edge $e = (x, y)$, we maintain two pointers e_x and e_y, that point to the elements of a list L_Z, for some node Z that contains both x and y.

Now we work as follows. Let $e = (x, y)$ be an edge that is inserted to G for the first time. (I.e., this is the first time that we call `insert`(e) of Algorithm 57. Notice, of course, that an edge $e' = (x, y)$ may have previously been inserted to G, since we allow for multiple edges. But now we consider e as a new insertion. Furthermore, we can easily determine whether `insert`(e) is the first time we call `insert` on e, by using a flag every time we call `insert` from Line 35 of Algorithm 57 or Line 15 of Algorithm 13, in order to signify that this is a re-insertion.) First we find the leaves X and Y of $\mathcal{T}$ that contain x and y, respectively, by calling $\textit{find}_{3ecs}(x)$ and $\textit{find}_{3ecs}(y)$, respectively. Suppose that $X \neq Y$ (for otherwise there is nothing to do). Then we crawl up the tree, starting from X and Y, by following the parents alternatingly, marking all the nodes that we meet, and keeping them in a stack in order to unmark

them later. When we meet a node N that is already marked, we have that this is the nearest common ancestor of X and Y on $\mathcal{T}$, and we unmark all the nodes that we traversed. Now we start again from X, and we follow the parents until we reach N. For every node Z that we traverse, including N, we append a new pointer $Z_{(e,x)}$ to L_Z. If $Z = X$, then we let $Z_{(e,x)}$ point to *null*. Otherwise, let C be the child of Z that we traversed before reaching Z; then we let $Z_{(e,x)}$ point to $C_{(e,x)}$. We repeat the same process starting from Y. Finally, we let e_x point to $N_{(e,x)}$ and we let e_y point to $N_{(e,y)}$. Observe that this process may need as many as $O(n)$ calls to the parent function, and $O(n)$ pointer manipulation operations. Thus, since the total number of first-time edge insertion in any sequence of edge insertions is $O(n)$, and every call to the parent function involves a call to *find*$_i$, for some i, the total cost of this process in the running time of the algorithm is $O(n^2\alpha(n,n))$.

Now suppose that we re-insert an edge $e = (x, y)$. Then we can find the nearest common ancestor of the leaves X and Y of $\mathcal{T}$ that contain x and y, respectively, by following the path of pointers starting from e_x and e_y, and moving on to deeper levels in parallel, until we reach two pointers that lie on two nodes C and D that have not got merged together, in which case the common parent of C and D is the nearest common ancestor of X and Y on $\mathcal{T}$. Then, since C and D will have to get merged, we let e_x point to $C_{(e,x)}$ and e_y point to $D_{(e,y)}$, in order to avoid traversing the same path of pointers again. Since the total number of edges that can affect the decomposition tree, in any sequence of edge insertions, is $O(n)$, and every time an edge is re-inserted it is moved into deeper levels, observe that the total cost of this process in the running time of the algorithm is $O(n^2\alpha(n,n))$. Now suppose that we want to find the grandchild of N that contains x. (We can follow the analogous procedure in order to find the grandchild of N that contains y.) Then we can use the pointer $N_{(e,x)}$ stored in L_N. This points to a pointer $C_{(e,x)}$ in L_C, for a node C that is merged with a child C' of N. And then $C_{(e,x)}$ points to $D_{(e,x)}$ in L_D, for a node D that is merged with a child D' of C'. Thus we can find D in constant time, and then we can find D' with a call *find*$_i(D)$, where i is the level of D. Again, since there can be at most $O(n^2)$ re-insertions of edges, the total cost of this process in the running time of the algorithm is $O(n^2\alpha(n,n))$.

All the data structures with which $\mathcal{T}$ is equipped (the DSU data structures, and the associated data structures for trees and cactuses), allow for new node insertions, and the same time-bounds hold. Thus, the insertion of new vertices to G is performed

as shown in Algorithm 59.

8.5 Data structures for trees and cactuses

To maintain the decomposition tree of the maximal 3-edge-connected subgraphs, we will need the following two data structures, which we may use as black boxes in the algorithm. They operate on collections of trees and cactuses, respectively. We assume that there is some information associated with every edge of every tree or cactus [3].

The first is a data structure that works on a collection of trees and supports the following operations.

- $\mathtt{compressPath}(\mathtt{T}, \mathtt{x}, \mathtt{y})$. Given a tree T and nodes x and y on T, this operation finds the simple path P that connects x and y on T and compresses it into a new node z (converting T into a smaller tree). Furthermore, it returns pointers to all nodes and (information associated with the) edges on P, as well as to the new node z. Finally, if (u, v) was an edge of T, prior to the call of this operation, with $u \notin P$ and $v \in P$, then (u, z) (in the new tree) must contain the same information that was previously associated with (u, v).

- $\mathtt{joinTrees}(\mathtt{T}_1, \mathtt{T}_2, (\mathtt{x}, \mathtt{y}))$. Given two trees T_1, T_2 and nodes $x \in T_1$ and $y \in T_2$, this operation links the trees T_1 and T_2 by introducing a new edge (x, y).

The second data structure works on a collection of cactuses and supports the following operations.

- $\mathtt{compressCyclePath}(\mathtt{S}, \mathtt{x}, \mathtt{y})$. Given a cactus S and nodes x and y on S, this operation finds the "cycle-path" P that connects x and y on S. It compresses all nodes on P into a new node z by properly squeezing the involved cycles. Furthermore, it returns pointers to all nodes on P, to all (information associated with the) edges that connect pairs of those nodes (if there are any), as well as to the new node z. Finally, if (u, v) was an edge of S, prior to the call of this operation, with $u \notin P$ and $v \in P$, then (u, z) (in the new cactus) must contain the same information that was previously associated with (u, v).

[3] For our algorithm, this information is a pointer to an edge of the original graph (whose maximal 3-edge-connected subgraphs we want to maintain).

- `joinCactuses`$(S_1, \ldots, S_k, (x_1, x_2), \ldots, (x_k, x_1))$. Given a collection of cactuses $S_1, \ldots, S_k$ and nodes $x_i \in S_i$, $i \in \{1, \ldots, k\}$, this operation links the cactuses $S_1, \ldots, S_k$ into a larger cactus by introducing the edges $(x_1, x_2), \ldots, (x_k, x_1)$ (which induce a new cycle).

In Sections 8.5.1 and 8.5.2, we provide implementations for data structures maintaining trees and cactuses where we can perform any sequence of operations in $O(n \log n)$ time, assuming that the total number of nodes is at most n. To achieve this time-bound, we will draw ideas from [35], [59], and [69]. Using these datastructures, we spend an additional $O(n^2 \log n)$ time for the incremental maintenance of the decomposition tree. To see this, recall that we use representations for the trees of the 2-edge-connected components at levels $3k + 1$ of the decomposition tree (where $0 \leq k < n$), and representations for the cactuses of the 3-edge-connected components at levels $3k + 2$ (where $0 \leq k < n$). Then, every operation `compressPath`, `joinTrees`, `compressCyclePath` and `joinCactuses` is performed on structures on the same level. Since there are $O(n)$ levels on which these operations may be performed, and the total number of nodes at any level is $O(n)$, we get the $O(n^2 \log n)$ time bound.

In Section 8.6, we use the more sophisticated implementations for those data structures provided by [58], in order to get a $O(n\alpha(n, n))$-time bound for performing any sequence of operations on trees or cactuses (on the same level), which gives $O(n^2\alpha(n, n))$ in the total time bound of the algorithm for maintaining the decomposition tree.

8.5.1 An implementation for trees

(a) Representation

An idea is to represent the trees as rooted trees. For every node x we let $p(x)$ denote its parent (if x is the root, we leave $p(x)$ undefined). Also, every node x has a pointer $x.edge$ to the edge $(x, p(x))$ that connects x with its parent. We use a DSU data structure to perform mergings of nodes. The DSU data structure supports the operations $find(x)$ and $unite(x, y)$. Thus, in order to access the parent of node x we actually use $find(p(x))$ (although we will not write this explicitly in what follows).

(b) Operations

We execute *compressPath*(T, x, y) as follows. We start from x and y, and we alternatingly climb up the tree following the parents, marking all nodes that we traverse, until we meet an already marked node z. (This ensures that we traverse at most $2d-1$ nodes, where d is the number of nodes of the simple path connecting x and y.) Then there are two possibilities: either (1) $z = x$ or $z = y$, or (2) z is neither x nor y. In the first case, assuming w.l.o.g. that $z = y$, the path that we have to compress consists of the nodes $x, p(x), p(p(x)), \ldots, y$. In the second case, the path consists of two parts: $x, p(x), \ldots, z$ and $y, p(y), \ldots, z$. In case (1) we work as follows. We start from x; we retrieve the edge $(x, p(x))$ (using the pointer stored in x), we call *unite*$(x, p(x))$, setting as representative of the new set $p(x)$, and we repeat, until we reach $z = y$. In case (2) we work as in case (1) twice, starting from both x and y, until we reach z each time.

The *joinTrees*$(T_1, T_2, (x, y))$ operation works as follows. First, we determine which of T_1, T_2 is smaller (w.r.t. its number of nodes). This can be done easily if we maintain for every tree an attribute *size* indicating the number of its nodes. Assume w.l.o.g. that the smallest tree is T_1 (or that its size is equal to that of T_2). Then we reroot T_1 at x, and we set $p(x) \leftarrow y$, $x.\mathit{edge} \leftarrow (x, y)$, and $T_2.\mathit{size} \leftarrow T_2.\mathit{size} + T_1.\mathit{size}$. The rerooting of T_1 at x is performed as follows. First we find the path P connecting x with the root r of T_1, by following the parents. Then we process the nodes of P in reverse order, starting from r until we reach x (which we do not process), and we simply hand over the information of every child to its parent. To be precise, let v be a node that we process, and let u be the node on P that has $p(u) = v$. Then we set $p(v) \leftarrow u$ and $v.\mathit{edge} \leftarrow u.\mathit{edge}$.

(c) Analysis

For convenience in the time analysis, we may use a simple implementation for the DSU data structure which performs all unions in $O(n \log n)$ time and every *find* in $O(1)$ worst-case time. Now, the total time it takes to perform any sequence of *compressPath* operations is $O(n \log n)$. This is because, every time *compressPath*(T, x, y) is performed, it takes $O(|P|)$ time to find the simple path P that connects x and y, using $O(|P|)$ calls to the *find* operation. Then we merge the nodes of P (thereby deleting them essentially), so they will not be traversed again. Thus, the total time to find all paths, in all *compressPath* calls, is $O(n)$. Therefore, the time bound for all *compressPath* operations is dominated by that for performing the mergings (using the *union* operation), and thus it is $O(n \log n)$.

When we perform a *joinTrees* operation on two trees, we may have to reroot the smallest tree T, which means that we may have to traverse all of its nodes. This takes $O(|T|)$ time and uses $O(|T|)$ calls to *find*. However, a node can only be accessed during a rerooting (that is, it may lie on the smallest tree) at most $O(\log n)$ times (because the trees are subsequently joined to produce a bigger tree whose size is at least twice that of the smallest). Thus, we need at most $O(n \log n)$ steps to perform all rerootings, using at most $O(n \log n)$ calls to the *find* operation. Thus, the total time, for any sequence of *compressPath* and *joinTrees* operations, is at most $O(n \log n)$.

8.5.2 An implementation for cactuses

(a) Representation

In order to efficiently determine the cycle-path connecting two nodes of a cactus, we represent the cactus as a rooted tree with "real nodes" and "cycle nodes". Every real node corresponds to a cactus node, and every cycle node corresponds to a cycle. Thus, for every cactus S, there is a *cactus tree* T on the set of real and cycle nodes corresponding to S, which is formed by adding edges corresponding to the incidence relation of the real nodes to the cycle nodes. Specifically, there is an edge (x, C) in T, where x is a real node and C is a cycle node, if and only if the cactus node corresponding to x lies on the cactus cycle corresponding to C. We root this tree on an arbitrary real node. Thus, the parent of a real node (different from the root) is a cycle node, and the parent of a cycle node is a real node. The parent of a (real or cycle) node v is denoted as $p(v)$. We use a DSU data structure to perform mergings of real nodes. This data structure supports *find*(x) and *unite*(x, y), Thus, in order to access the parent of a cycle node C, we use *find*$(p(C))$.

In order to retrieve the information associated with the cactus edges and to efficiently perform the "cycle-squeezing" operation, we use doubly-linked circular lists to represent the cycles of the cactuses. Specifically, for every cycle C of a cactus, there is a doubly-linked circular list L consisting of as many nodes as there are in C. The nodes of L correspond to the nodes of C and are ordered accordingly. Every node x of L has pointers x.*left* and x.*right* to its neighboring nodes, and pointers x.*leftEdge* and x.*rightEdge* to the respective cactus edges.

Finally, we have the following correspondence between the nodes of the cactus trees and the nodes of the circular lists. Let x be a real node of a cactus tree with

parent C. Then C corresponds to a cactus cycle and therefore to a circular list L. Then we have a pointer for x to its corresponding node on L, and vice versa. Furthermore, $p(C)$ also corresponds to a node on L, but we do not keep a pointer from $p(C)$ to this node (because it cannot be uniquely determined from $p(C)$, as this can be the parent of many cycle nodes). Instead, we have a pointer from C to the node of L corresponding to $p(C)$. For notational convenience, we may use the same name to denote a real node and its corresponding circular list node. Also, for a cycle node C, we may denote the circular list node pointed to by C as $p(C)$.

(b) Operations

`compressCyclePath`

To perform `compressCyclePath`$(\mathtt{S}, \mathtt{x}, \mathtt{y})$ we work as follows. First, we determine the cycle-path P connecting x and y. To do this, we work on the cactus tree associated with S; we start from x and y, and we alternatingly climb up the tree following the parents, marking all the nodes that we traverse, until we meet an already marked node z. Then there are two possibilities: either (1) $z = x$ or $z = y$, or (2) z is neither x nor y. In the first case, assuming w.l.o.g. that $z = x$, P is given by the real nodes on the ascended path starting from x. In the second case, P is given by two parts: by the real nodes on the path starting from x and ending in z, and by the real nodes on the path starting from y and ending in z. (Note that z can be either a cycle node or a real node.) In case (1), assuming w.l.o.g. that $z = x$, let $u_1, C_1, \dots, C_k, u_{k+1}$, where $u_1 = x$ and $u_{k+1} = y$, be the cycle-path connecting x and y, including the cycles nodes that are connected with every two consecutive real nodes. Then we only have to perform the operations `squeezeCycle`(u_i, u_{i+1}, C_i), for every $i \in \{1, \dots, k\}$, successively. In case (2), we work as in case (1) for every one of the two parts that form P. Furthermore, in the case that z is a cycle node, we also have to perform `squeezeCycle`(u, v, z), where u and v are the second-last nodes of the two parts that form P. In particular, let $u_1, C_1, \dots, u_k, z$ and $v_1, C'_1, \dots, v_l, z$, where $u_1 = x$ and $v_1 = y$, be the two paths that we ascended, starting from x and y, respectively. Then, in addition to all operations `squeezeCycle`(u_i, u_{i+1}, C_i), for every $i \in \{1, \dots, k-1\}$, and `squeezeCycle`(v_i, v_{i+1}, C'_i), for every $i \in \{1, \dots, l-1\}$, we also have to perform `squeezeCycle`(u_k, v_l, z).

Now let us describe in detail the operation `squeezeCycle`(u, v, C), where u and v are real nodes connected with a cycle node C. (Intuitively, we want this operation to

"squeeze" the cycle C by merging its two nodes u and v.) We distinguish two cases, depending on whether (i) u and v are related as ancestor and descendant, or (ii) u and v are siblings on the rooted cactus tree (the later occurs in case (2) above, when z is a cycle node).

(i) *u and v are related as ancestor and descendant*

We may assume w.l.o.g. that v is an ancestor of u. Thus we have $p(u) = C$ and $p(C) = v$. We distinguish three cases, depending on whether u has no siblings, or u is the leftmost or rightmost child of C (to be defined shortly), or neither of the previous two cases holds. First, if u has no siblings, then we simply return the two edges that connect u and v, we discard the cycle C, and we merge u and v, setting as representative of the union v. To return the two edges we use the pointer from u to the node in the circular list corresponding to C (let us call it u again). Then the two edges are precisely $u.\mathit{leftEdge}$ and $u.\mathit{rightEdge}$. Now let us consider the case that u is the leftmost or rightmost child of C. This is determined by checking whether $u.\mathit{left}$ or $u.\mathit{right}$, respectively, is the node pointed to by C (which essentially corresponds to v). So let us assume that $u.\mathit{left} = v$ (the case that $u.\mathit{right} = v$ is treated analogously). The first thing to do is to return the edge that connects u and v. This is simply $u.\mathit{leftEdge}$. Then we discard the node u from the circular list and we update it accordingly. That is, we set $u.\mathit{right}.\mathit{left} \leftarrow v$, $v.\mathit{right} \leftarrow u.\mathit{right}$, and $v.\mathit{rightEdge} \leftarrow u.\mathit{rightEdge}$. Finally, we merge u and v (the real nodes on the cactus tree), setting as representative of the union v.

Now we consider the third case. That is, we have that $u.\mathit{left} \neq v$ and $u.\mathit{right} \neq v$. (Notice that, in this case, there is no cactus edge connecting u and v.) Now the first thing to do is to determine the smallest segment of the circular list L corresponding to C that contains u and v. To achieve this efficiently, we start from u and v, and we alternatingly follow the *left* pointers, marking all the nodes that we traverse, until we meet an already marked node z. Then we have that either $z = v$ or $z = u$. Let us assume that $z = v$ (the other case is treated analogously). Now we update the cactus tree as follows. Let P be the internal path on L connecting u and v following the *left* pointers starting from u (that is, $P = u.\mathit{left}, u.\mathit{left}.\mathit{left}, \dots, v.\mathit{right}$). We introduce a new cycle node C', and we let C' be the parent of all the real nodes that correspond to nodes of P. The parent of C' is set to be v, and we introduce a new circular list node $\tilde{v}$, corresponding to v on the cycle C'. Thus, there is a pointer from C' to $\tilde{v}$. Now

we detach P from L and we attach it appropriately to the circular list corresponding to C'. This is done by setting $v.\mathit{right}.\mathit{left} \leftarrow \tilde{v}$, $\tilde{v}.\mathit{right} \leftarrow v.\mathit{right}$, $u.\mathit{left}.\mathit{right} \leftarrow \tilde{v}$, and $\tilde{v}.\mathit{left} \leftarrow u.\mathit{left}$. To maintain the cactus edges, we also set $\tilde{v}.\mathit{rightEdge} \leftarrow v.\mathit{rightEdge}$ and $\tilde{v}.\mathit{leftEdge} \leftarrow u.\mathit{leftEdge}$. In the list L we simply discard u and we update L accordingly. That is, we set $u.\mathit{right}.\mathit{left} \leftarrow v$ and $v.\mathit{right} \leftarrow u.\mathit{right}$. Again, to maintain the cactus edge, we also set $v.\mathit{rightEdge} \leftarrow u.\mathit{rightEdge}$. Finally, we merge u and v (the real nodes on the cactus tree), setting as representative of the union v.

(ii) u and v are related as siblings

Let us assume first that (the circular list nodes corresponding to) u and v are neighbors. This means that either $u.\mathit{left} = v$ or $u.\mathit{right} = v$. Suppose that $u.\mathit{left} = v$ (the other case is treated similarly). First we return $u.\mathit{leftEdge}$: the edge that connects u and v. Then we discard u from the circular list and we update it accordingly. That is, we set $v.\mathit{right} \leftarrow u.\mathit{right}$, $u.\mathit{right}.\mathit{left} \leftarrow v$, and $v.\mathit{rightEdge} \leftarrow u.\mathit{rightEdge}$. Finally, we merge u and v (on the cactus tree), setting as representative of the union v.

Now suppose that u and v are not neighbors. This means that $u.\mathit{left} \neq v$ and $u.\mathit{right} \neq v$. (Notice that, in this case, there is no cactus edge connecting u and v.) Let C be the common parent of u and v, and let $w = p(C)$. Let also L be the circular list corresponding to C, and denote as w the node of L pointed to by C. Now the first thing to do is to determine the smallest part of L that contains u and v. To achieve this efficiently, we start from u and v, and we alternatingly follow the *left* pointers, marking all the nodes that we traverse, until we meet an already marked node z. Then we have that either $z = v$ or $z = u$. Let us assume that $z = v$ (the other case is treated analogously). Let P be the internal path on L connecting u and v following the *left* pointers starting from u (that is, $P = u.\mathit{left}, u.\mathit{left}.\mathit{left}, \dots, v.\mathit{right}$). Here we have that either $w \notin P$ or $w \in P$. If $w \notin P$, then we work precisely as in the second paragraph of case (ii) above. So let us assume that $w \in P$. Then we introduce a new cycle node C', we let C' be the parent of all the real nodes that correspond to nodes of $P \setminus w$, and we let C' point to w. Then we introduce a new circular list node $\tilde{v}$ (corresponding to the node that will be formed by merging u and v), we let C point to $\tilde{v}$, and we attach the segment $L \setminus (P \cup \{u, v\})$ of L to $\tilde{v}$. Again, this is done precisely as in the second paragraph of case (ii) above. Furthermore, we discard the circular list node u, and we properly update the list corresponding to C'. That is, we set $v.\mathit{right} \leftarrow u.\mathit{right}$, $u.\mathit{right}.\mathit{left} \leftarrow v$, and $v.\mathit{rightEdge} \leftarrow u.\mathit{rightEdge}$. Finally, we let the

parent of C be v, and we merge u and v (the real nodes on the cactus tree), setting as representative of the union v.

`joinCactuses`

To perform $\texttt{joinCactuses}(S_1, \dots, S_k, (x_1, x_2), \dots, (x_k, x_1))$, we first have to find the largest (w.r.t. its number of nodes) cactus among $S_1, \dots, S_k$. This can be done easily if we keep for every cactus an attribute *size*, signifying its number of nodes. Now let us assume, w.l.o.g., that one of the cactuses with the greatest size is S_k. Then we have to do two things: to reroot the cactuses $S_1, \dots, S_{k-1}$ on their nodes that are to be connected on a new cycle, and then to form the cycle $(x_1, x_2), \dots, (x_k, x_1)$.

Let us describe how to perform a rerooting of a cactus S at a node $x \in S$. We assume that x is not the root of the cactus tree of S (for otherwise there is nothing to do). First we find the path on the cactus tree that connects x with the root by following the parents. This has the form $u_1, C_1, \dots, C_{t-1}, u_t$, where $u_1, \dots, u_t$ are real nodes with $u_1 = x$, and $C_1, \dots, C_{t-1}$ are cycle nodes. Then we process this path in reverse order, and for every triple (u, C, v), where u, v are real nodes and C is a cycle node, we do the following. (Due to reverse processing, notice that we have $p(v) = C$ and $p(C) = u$.) Let L be the circular list corresponding to C. Then we simply change the pointer of C, so that it points to the node of L corresponding to v, and we reverse the parent relation between the nodes of (u, C, v). (Thus, we set $p(u) \leftarrow C$ and $p(C) \leftarrow v$.) Finally, we update the status of x so that it is recognized as the root.

Thus we may assume that the cactus tree of every S_i, $i \in \{1, \dots, k-1\}$, is rooted at x_i. Now we introduce a new cycle node C, and we set the parent of every x_i, $i \in \{1, \dots, k-1\}$, to be C, and the parent of C to be x_k. Then we introduce new circular list nodes $\tilde{x}_1, \dots, \tilde{x}_k$, pointers between x_i and $\tilde{x}_i$ (in both directions), for every $i \in \{1, \dots, k-1\}$, and we let C point to $\tilde{x}_k$. All nodes $\tilde{x}_i$ are linked in a circular structure according to their ordering on the new cycle. That is, we set $\tilde{x}_i.\mathit{left} \leftarrow \tilde{x}_{i-1}$, for $i \in \{2, \dots, k\}$, $\tilde{x}_1.\mathit{left} \leftarrow \tilde{x}_k$, $\tilde{x}_i.\mathit{right} \leftarrow \tilde{x}_{i+1}$, for $i \in \{1, \dots, k-1\}$, and $\tilde{x}_k.\mathit{right} \leftarrow \tilde{x}_1$. Furthermore, we fix the pointers to the newly introduced edges. That is, we set $\tilde{x}_i.\mathit{leftEdge} \leftarrow (x_{i-1}, x_i)$, for $i \in \{2, \dots, k\}$, $\tilde{x}_1.\mathit{leftEdge} \leftarrow (x_k, x_1)$, $\tilde{x}_i.\mathit{rightEdge} \leftarrow (x_i, x_{i+1})$, for $i \in \{1, \dots, k-1\}$, and $\tilde{x}_k.\mathit{rightEdge} \leftarrow (x_k, x_1)$. Finally, we set $S_k.\mathit{size} \leftarrow S_1.\mathit{size} + \cdots + S_{k-1}.\mathit{size}$, and the description of $\texttt{joinCactuses}(S_1, \dots, S_k, (x_1, x_2), \dots, (x_k, x_1))$ is complete.

(c) Analysis

First, if we use a simple implementation for the DSU data structure which performs all unions in $O(n \log n)$ time and every *find* in $O(1)$ worst-case time, we can argue as in the analysis for the data structure for trees, that the total time it takes to find and merge all cycle-path during any sequence of `compressCyclePath` operations is $O(n \log n)$. Thus we only have to consider the total time it takes to perform all `squeezeCycle` operations. Recall that `squeezeCycle`(u, v, C) has to find the smallest segment P of the circular list corresponding to C that contains u and v (all other operations of `squeezeCycle`(u, v, C) take $O(1)$ time in total). This is performed in $O(|P|)$ time, and then P forms a new circular list on its own (plus at most one more node). We say that the two new cycles into which C was squeezed are *formed by* C. (Thus, we may observe that after any repeated application of `squeezeCycle` operations on C and on cycles formed by C, we have that all cycles formed by C form a cactus.) We overload our terminology by saying that a node of a cycle formed by C is also a node of C. Now we may argue as follows. Fix a cycle C of size k and a node $x \in C$. Then we observe that in any sequence of `squeezeCycle` operations on C or on cycles formed by C, x can be part of the smallest segment (on a circular list) explored by `squeezeCycle` at most $O(\log k)$ times. Thus, the total time to perform any sequence of `squeezeCycle` operations on C or on cycles formed by C is $O(k \log k)$. Now observe that the cycles are introduced into the collection of cactuses by the operation *joinCactuses*; and that once a cycle has been introduced, it can only get squeezed to form smaller cactuses. Thus, let $k_1, \dots, k_t$ be the sizes of all cycles that have been introduced in the data structure by the operation *joinCactuses*. Then we have that the total time to perform all `squeezeCycle` operations is $O(k_1 \log k_1 + \cdots + k_t \log k_t) = O(k_1 \log n + \cdots + k_t \log n) = O((k_1 + \cdots + k_t) \log n) = O(n \log n)$, since $k_i = O(n)$, for every $i \in \{1, \dots, t\}$, and $k_1 + \cdots + k_t = O(n)$.

When we perform a *joinCactuses* operation on some cactuses, we may have to reroot the smallest ones, which means that we may have to access all of their nodes. However, we can argue as in the analysis of the data structure for trees, in order to show that the total time-bound for all rerootings is $O(n \log n)$. All the other operations performed in *joinCactuses* take time analogous to the number of the cactuses involved. Since the total size of all cycles that we can introduce (while maintaining a collection of cactuses on n nodes) is $O(n)$, this shows that the total time for any sequence of *joinCactuses* operations is $O(n \log n)$.

8.6 Improved data structures for trees and cactuses

Here we provide data structures and algorithms for the operations `compressPath`, `joinTrees`, `compressCyclePath` and `joinCactuses`, on collections of trees and cactuses with at most n nodes, with time bounds better than $O(n \log n)$. To be precise, any sequence of m operations on collections of trees or cactuses with at most n nodes can be performed in $O((n+m)\alpha(m,n))$ time in total. The analysis for this time bound is essentially contained in [59] and [58]; our own additions in the algorithms of [59] and [58] involve only a worst-case $O(1)$ calls to some DSU operations and a worst-case $O(1)$ pointer manipulations, for every operation performed. Also, the arguments that establish correctness are sufficiently contained in the description of the algorithms.

Now, to achieve this time bound, we will basically use the data structures provided by La Poutré et al. [59], [58], with some minor additions to suit our purposes. These data structures rely on the so-called "fractionally rooted trees" (FRT), introduced in [58]. In the next section we give a brief overview of the operations supported by FRTs. Then we describe the implementations for collections of trees and cactuses. We only give as many details of the data structures and algorithms of [59] and [58] as are needed in order to show where our own additions fit in and establish our results.

8.6.1 Fractionally rooted trees

The FRT data structure operates on a forest F equipped with a partition of its edges such that the classes corresponding to this partition induce subtrees of F. Before we describe the operations supported by FRT, we introduce some terminology. We say that a node x of F belongs to an edge class if it is incident to at least one edge of that class. Also, we say that an edge class intersects a path P if P contains at least one edge of that class. Now let x, y be two nodes of a tree of F, and let P be the simple path that connects them. We call a node z on P a *boundary node* of P if it is either one of x, y or it is incident to two edges of P which belong to different classes. A *boundary edge set* for a boundary node z on P is a set of (0, 1 or 2) edges incident to z, one from each edge class to which z belongs and which intersects P. (We do not demand that the edges in a boundary edge set for z lie on P; however, one of their endpoints must be z.) A *boundary list* for P is a list consisting of the boundary nodes of P, where each boundary node has a sublist that contains a boundary edge set for it on P. Now let L be a list of nodes where each node has a sublist of edges.

We say that an edge class *occurs* in L, or that L *contains* an edge class, if there is an edge of that class in some sublist of L. The edge classes *of* L are precisely those that occur in it. A *joining list* J is a list of nodes with sublists of edges such that the union of the classes occurring in J induces some subtree in F. (We note that this is always the case for a boundary list of a path.) In addition, the nodes in J must be the nodes belonging to at least two edge classes occurring in J, and the sublist for each node contains an edge for each class in J to which this node belongs. (Thus we have that a boundary list of a path is a joining list if we remove the endpoints of the path from the list.)

Now the operations supported by FRT are the following.

- `link`(x, y). Let x, y be two nodes lying on different trees of F. Then join the two trees by introducing a new edge (x, y) in the FRT data structure.
- `boundary`(x, y). Let x, y be two nodes lying on the same tree of F. Then return a boundary list for the simple path with endpoints x and y.
- `joinclasses`(J). Let J be a joining list. Then join all the edge classes of which an edge occurs in the list.

We say that a call `boundary`(x, y) is essential if there are at least two different edge classes that intersect the simple path that connects x and y. (Equivalently, a call `boundary`(x, y) is essential if the boundary list that it provides contains more than two nodes.)

By [58], we have the following result.

Theorem 8.2. *(Theorem* 9.2 *in [58]) There is an implementation of fractionally rooted trees with the following guarantee. Suppose that we start with an empty forest, and we perform* n *insertions of nodes and* m *operations* `link`, `boundary`, *and* `joinclasses`, *where every essential call of* `boundary` *is immediately followed by* `joinclasses` *on a joining list that contains all the edge classes that occur in the list provided by* `boundary`*. Then all these operations can be performed in total* $O((n + m)\alpha(m, n))$ *time.*

8.6.2 An implementation for trees

(a) Representation

For the operations `compressPath` and `joinTrees` on collections of trees, we essentially use the solution of La Poutré for the incremental maintenance of the 2-edge-connected components in general graphs with asymptotically optimal time complexity [58].

Thus we represent the collection of trees $\mathcal{C}$ as a forest F, which is implemented as an FRT data structure. Every tree $T \in \mathcal{C}$ corresponds to a tree T_{FRT} of F. The edges of F are partitioned into edge classes, where every edge class induces a subtree of F. Some edges of F are called *bridges* and belong to singleton classes which are called *quasi* classes. All other edge classes are called *real*. There is an one-to-one correspondence between the edges of a tree $T \in \mathcal{C}$ and the bridges of T_{FRT}. Thus every edge of T_{FRT} that is a bridge contains a pointer to its corresponding edge on T. Every node x of T_{FRT} belongs to at most one real class. For every node x that belongs to a real class, we maintain an edge x_{assoc} of that class. By shrinking the subtrees that are induced by the real classes into single nodes, we get a natural isomorphism between T and T_{FRT}. (The idea here is that the subtrees induced by the real classes correspond to maximal sets of nodes that got merged due to the operation `compressPath`.) Thus, every node $x \in T$ corresponds to a subset S of T_{FRT}, and we associate x with a node x_{FRT} of S; conversely, every node in S is associated with x. To implement the later, we use a DSU data structure DSU_F on the nodes of F, where the representatives of the sets are nodes of the trees of $\mathcal{C}$. DSU_F supports the operations $find_F$ and $unite_F$. Thus, in order to find which node of $\mathcal{C}$ corresponds to a node u of F, we use $find_F(u)$. DSU_F (and every other DSU data structure in the sequel) is implemented using rooted trees, with path compression and union by size, thus achieving the asymptotically optimal time complexity [64].

We can summarize the above properties as follows.

Property 8.3. *Let x, y be two nodes of a tree $T \in \mathcal{C}$, and let P be the simple path on T that connects x and y. Let also $\tilde{P}$ be the simple path on T_{FRT} that connects x_{FRT} and y_{FRT}. Then there is an one-to-one correspondence between the edges of P and the edges of $\tilde{P}$ that are bridges. This correspondence is compatible with that between the nodes of T and the nodes of T_{FRT}. In other words, for every bridge (u, v) on $\tilde{P}$, there is an edge $(find_F(u), find_F(v))$ on P.*

(b) Operations

`compressPath`

Let x and y be two distinct nodes of a tree $T \in \mathcal{C}$, and let P be the simple path on T that connects x and y. To perform *compressPath*(T, x, y) we work as follows. First we call *boundary*(x_{FRT}, y_{FRT}) to obtain a boundary list L for the simple path $\tilde{P}$ that connects x_{FRT} and y_{FRT} on T_{FRT}. According to Property 8.3, in order to retrieve the edges of P, it is sufficient to identify the edges of $\tilde{P}$ that are bridges. Since bridges belong to singleton (quasi) classes, we have that every bridge of $\tilde{P}$ must lie in the sublist of some node in L. Thus we can retrieve the bridges of $\tilde{P}$ by scanning the sublists of L. (Incidentally, we note here that the efficiency of the operation *boundary* lies in the fact that it only computes boundary nodes and corresponding edge classes, without always traversing the entire path.) Furthermore, using again Property 8.3, we can retrieve the nodes of P using the operation find_F of DSU_F on the endpoints of every bridge of $\tilde{P}$.

Now, for every node u in L that is incident to a real class, we append u_{assoc} to the sublist of u. This is to ensure that the real class to which u belongs will get joined to all classes that occur in L. (Because there is a possibility that the edge classes to which u belongs and that intersect $\tilde{P}$ are quasi classes, and so L does not contain any edge from the real edge class to which u belongs.) If either x_{FRT} or y_{FRT} still has only one edge in its sublist, then it is removed from L. (This is to ensure that L is a joining list.) If L is non-empty, then we call *joinclasses*(L). Otherwise, we have that x_{FRT} and y_{FRT} are connected with a bridge (x_{FRT}, y_{FRT}) on T_{FRT}. Then we just change the status of (x_{FRT}, y_{FRT}) so that it is no longer marked as a bridge, and we convert the quasi class that contains (x_{FRT}, y_{FRT}) to a real class. We let the associated edge of every node on $\tilde{P}$ be any edge on $\tilde{P}$. (For this purpose, we may keep in a variable e_{temp} the edge that was in the boundary edge set of x_{FRT} after the call *boundary*(x_{FRT}, y_{FRT}); so now we set $u_{\mathit{assoc}} \leftarrow e_{\mathit{temp}}$ for every node u on $\tilde{P}$.) We introduce a new node z on T that replaces the entire path P, and we let z_{FRT} be any node on $\tilde{P}$. Finally, we unite all nodes on $\tilde{P}$ using DSU_F, and we let z be the representative of the resulting set. This completes the description of *compressPath*, which is summarized in Algorithm 60.

`joinTrees`

Let T_1 and T_2 be two distinct trees of $\mathcal{C}$, and let $x \in T_1$ and $y \in T_2$. Then *joinTrees*$(T_1, T_2, (x, y))$ is performed by simply calling *link*(x_{FRT}, y_{FRT}), where (x_{FRT}, y_{FRT}) is marked as a bridge and the corresponding singleton edge class is marked as a quasi class. Finally we let (x_{FRT}, y_{FRT}) point to (x, y). (See Algorithm 61.)

Algorithm 60: *compressPath*(T, x, y)

```
1  P ← ∅ // the set of nodes of the simple path on T that connects x and y
2  ℰ ← ∅ // the set of edges of T to be returned
3  L ← boundary(x_FRT, y_FRT)
4  let e_temp be the edge in the sublist of x_FRT in L
5  foreach node u in L do
6      foreach edge e in the sublist of u do
7          if e is a bridge then
8              ẽ ← edge of T pointed to by e
9              ℰ ← ℰ ∪ {ẽ}
               // let e = (z_1, z_2)
10             x_1 ← find_F(z_1)
11             x_2 ← find_F(z_2)
12             P ← P ∪ {x_1, x_2}
13         end
14     end
15     if u_assoc ≠ ∅ then append u_assoc to the sublist of u
16     if the sublist of u contains only one edge then remove u from L
17 end
18 if L ≠ ∅ then joinclasses(L)
19 else
20     unmark (x_FRT, y_FRT) as a bridge
21     mark {(x_FRT, y_FRT)} as a real class
22 end
23 foreach node u in L ∪ {x_FRT, y_FRT} do set u_assoc ← e_temp
24 let z be a new node on T that replaces the path P
25 unite all nodes in L ∪ {x_FRT, y_FRT} using DSU_F; let z be the representative
26 set z_FRT ← x_FRT
27 return {P, ℰ, z}
```

Algorithm 61: *joinTrees*$(T_1, T_2, (x, y))$

1 `link`(x_{FRT}, y_{FRT})
2 mark (x_{FRT}, y_{FRT}) as a bridge
3 mark $\{(x_{FRT}, y_{FRT})\}$ as a quasi class
4 make (x_{FRT}, y_{FRT}) point to (x, y)

8.6.3 An implementation for cactuses

(a) Representation

Following [58], we represent the cactuses with a data structure that generalizes the concept of the tree of cycles. Recall that the tree of cycles of a cactus is the graph that represents the incidence relation of the nodes of the cactus to its cycles [35], [59]. By the definition of the cactus, we have that this graph is a tree. Now we generalize the tree of cycles as follows.

- We partition the cactus S into subcactuses. Based on this partition, we have the graph T that represents the incidence relation of the nodes of S to the subcactuses. (Again, we have that T is a tree.)
- We allow T to be extended with extra nodes, that serve as copies of the nodes of S. However, we demand that the resulting graph is also a tree. All nodes of T that serve as copies of the same node of S are said to belong to the same *cluster*. (Thus, the set of clusters is a partition of the set of nodes of T that correspond to nodes of S.)

We call this type of representation of a cactus S a *tree of cactuses*. See Figure 8.6 for an example of this kind of representation. (We note that, contrary to the tree of cycles, a tree-of-cactuses representation of a cactus is not unique.) We distinguish two types of nodes in a tree of cactuses: those that correspond to the nodes of S, and they are called *real nodes*, and those that correspond to the subcactuses of S, and they are called *cactus nodes*. Thus we have a correspondence between the nodes of S and the real nodes of a tree-of-cactuses representation T of S. Furthermore, we may speak of the *nodes* of a cactus node C, and by that we mean the nodes of the subcactus of S that corresponds to C.

In our implementation, every node of S has a pointer to a unique corresponding node on T; whenever no confusion arises, we will denote these two nodes with the

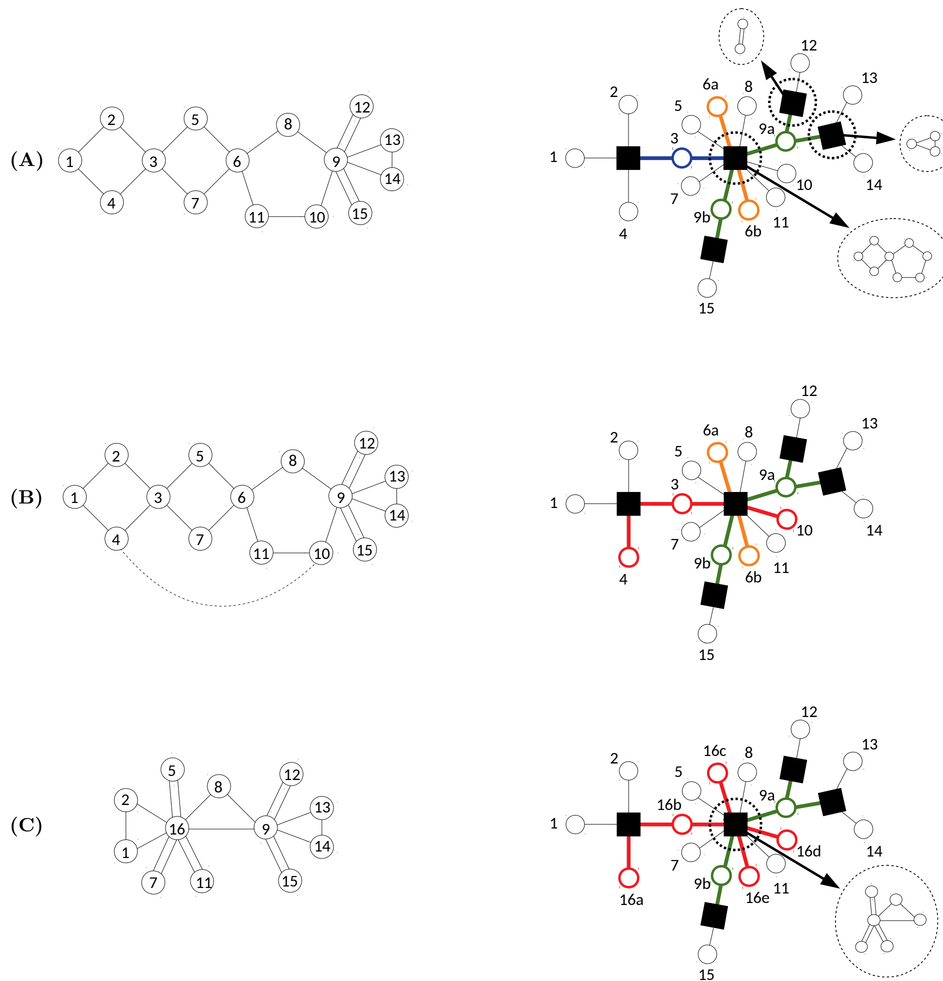


Figure 8.6: An example of a tree-of-cactuses representation of a cactus. The square nodes are the cactus nodes. In figures A and C we can see the corresponding subcactuses of some cactus nodes. The edges that belong to the same edge class are painted with the same colour. The edges in black colour constitute singleton edge classes. We can see that some real nodes correspond to the same nodes of the cactus (and thus they belong to the same cluster). For example, real nodes $6a$ and $6b$, in figure A, correspond to node 6 of the cactus. In figure B we can see an intermediate step of the call `compressCyclePath`$(4, 10)$: this is where we have to determine the path on the tree that connects 4 and 10. In figure C we can see the result of this call. The nodes 4, 3, 6 and 10 were merged into a new node, which is called 16. Observe that, in order to determine that 6 was also part of the cycle path connecting 4 and 10, we were guided by the associated data structure of an intermediary cactus node. Notice also that the operation `compressCyclePath` does not affect the structure of the tree of cactuses, but only its edge classes, the names of its real nodes, and the associated data structures of the involved cactus nodes.

same symbol. Conversely, every node of T has a pointer to its corresponding node on S. Specifically, to maintain the later correspondence we use a DSU data structure DSU_{cls} on the real nodes of T whose sets coincide with the clusters. This data structure supports the operations $find_{cls}$ and $unite_{cls}$. Every cluster has a representative real node z in it, which can be found with a call $find_{cls}(x)$ on any node x in this cluster. Then we only need to maintain the pointer of z to its corresponding node on S. Finally, every edge (x, C), where x is a real node and C is a cactus node, has pointers to its endpoints x and C.

Now let T be a tree of cactuses of a cactus S. We equip the set of edges of T with the following equivalence relation. If (x, C_1) and (x, C_2) are two edges of T, where x is a real node and C_1, C_2 are two cactus nodes, then (x, C_1) and (x, C_2) are equivalent. Furthermore, if (x, C) and (x', C') are two edges of T, where x and x' belong to the same cluster and C, C' are cactus nodes, then (x, C) and (x', C') are equivalent. It should be clear that this is indeed an equivalence relation, and that its equivalence classes induce subtrees of T. Thus, on a set of trees of cactuses (representing a set of cactuses), each one equipped with its respective equivalence relation, we can perform the operations of an FRT data structure. Furthermore, it should also be clear that every node x (in a non-trivial tree of cactuses) belongs to a unique edge class, and we associate to x an edge x_{assoc} of that class.

Before we describe how we can use the FRT operations to perform `compressCyclePath` and `joinCactuses`, let us give an overview of what is involved in using a tree-of-cactuses representation T of a cactus S in order to determine cycle paths on S. So let x, y be two nodes of S. Let also Q be the cycle path on S connecting x and y, and let $P = z_1, C_1, \dots, C_{k-1}, z_k$ be the simple path on T with endpoints x and y, where $z_1 = x$, $z_k = y$, and $C_1, \dots, C_{k-1}$ are the intermediary cactus nodes. Then we have that Q consists of (the nodes of S that correspond to) all the real nodes on P, plus (the nodes of S that correspond to) some nodes $w_1, \dots, w_t$ that are incident to the cactuses $C_1, \dots, C_{k-1}$. The nodes $w_1, \dots, w_t$ are precisely those that appear on the cycle paths of the subcactuses of S corresponding to $C_1, \dots, C_{k-1}$ that form part of Q. Thus, in order to determine $w_1, \dots, w_t$, we are guided by some data structures associated with $C_1, \dots, C_{k-1}$ that represent cactuses. (See also Figure 8.6.) For this purpose we use the data structures of [59], that represent the cactuses as rooted trees of cycles equipped with *circular split-find* structures. (These representations are used to solve the problem of the incremental maintenance of the 3-edge-connected

components of 2-edge-connected graphs in asymptotically optimal time. We will not give a full exposition of this representation here, but later on we will describe how to augment it in order to facilitate the retrieval of the cactus edges.) Furthermore, in order to retrieve the edges of S that connect any pair of consecutive nodes on Q, we use precisely the data structures associated with $C_1, \dots, C_{k-1}$, as the information concerning the cactus edges is stored in the intermediary cactuses.

Now we recall that a circular split-find data structure operates on a collection of circular lists, where every list is related with a representative element, and supports the operations *find* and *split*. *find*(x) on a list node x returns the representative of the list containing x. *split*(x, y) on two distinct nodes x and y of the same list creates two new circular lists, consisting of the nodes from x (resp. y) and up to - but excluding - y (resp. x), following the same direction in both cases (e.g., the left one), and relates each new list with a representative element. [56] provides an implementation for the circular split-find problem, where we can perform m operations *split* and *find* on a collection of circular lists on n element in $O(n + m\alpha(m, n))$ time in total.

In the tree-of-cycles representation of cactuses used by [59], the circular lists correspond to the cycles of the cactus. Specifically, let C be a cycle of a cactus. Then, for every node x of C, there is an element *repr*(x, C) that corresponds to the edge (x, C) of the tree of cycles. Now, the circular list corresponding to C consists precisely of the elements *repr*(x, C), for every node x of C, ordered in the same way in which the nodes of C occur on C.

(We note that, in order to retrieve x or C from an element *repr*(x, C), we use a *find* operation on a DSU data structure that operates on the collection of all *repr* elements. In fact, we use two different DSU data structures: one for retrieving the node x from an element *repr*(x, C), and one for retrieving C from *repr*(x, C). However, we will not mention explicitly those DSU data structures, because with an optimal implementation [64] they do not affect the asymptotical time bounds.)

Now we have to augment the information on the circular lists, in order to be able to return the cactus edges that connect consecutive nodes on cycle paths on a cactus. To achieve this, we simply store four pointers *left*, *right*, *leftEdge* and *rightEdge* on every *repr* element. Specifically, let C be a cycle of the cactus, and suppose that we have a specific orientation of its nodes (that is, a way to determine, given two consecutive nodes x and y on C, which one is on the left and which one is on the right). Now let x and y be two consecutive nodes of C, where x is on the left of y. Then we have

$repr(y, C).left = repr(x, C)$, $repr(x, C).right = repr(y, C)$, $repr(y, C).leftEdge = (x, y)$ and $repr(x, C).rightEdge = (x, y)$.

Finally, for every edge (x, C) on a tree of cactuses, where x is a real node and C is a cactus node, we have a pointer from (x, C) to the node in the associated structure of C that corresponds to x, and reversely. (To be more precise, in order to maintain the reverse correspondence we use a DSU data structure that operates internally on the nodes of the tree of cycles representing C. This is because the operations on this data structure merge nodes into larger sets from time to time; and thus we maintain representatives of those sets, and pointers from those representatives to their corresponding nodes on the tree of cactuses. However, we will not mention explicitly the calls to this DSU data structure in what follows, since they do not affect the asymptotical time complexity overall.)

(b) Operations

`compressCyclePath`

To perform `compressCyclePath(S, x, y)` (with $x \neq y$), we first have to determine the cycle path connecting x and y on S. Let us use the same symbols, x and y, to denote the real nodes on the tree-of-cactuses representation T of S that correspond to x and y, respectively. As we noted in subsection (a), we have to determine two things: first, the simple path $P = z_1, C_1, \dots, C_{k-1}, z_k$ on T that connects x and y, where $z_1 = x$, $z_k = y$, and $C_1, \dots, C_{k-1}$ are the intermediary cactus nodes, and second, the real nodes $w_1, \dots, w_t$ that are incident to some of the cactuses $C_1, \dots, C_{k-1}$ and that have to get merged too with the other nodes. The property that characterizes the nodes $w_1, \dots, w_t$ can be explained as follows. For every edge (z, C) on P, where z is a real node and C is a cactus node, we let $\tilde{z}$ denote the node in the associated tree of cycles of C that is pointed to by (z, C) (and corresponds essentially to z, and to any other node in the cluster of z). Now suppose that, for some $i \in \{1, \dots, k-1\}$, (z_i, C_i) and (C_i, z_{i+1}) belong to different edge classes. Then $\tilde{z}_i$ and $\tilde{z}_{i+1}$ are different nodes of (the associated structure to) C, and so we have to find and merge the cycle path on C with endpoints $\tilde{z}_i$ and $\tilde{z}_{i+1}$ (and also return the cactus edges that connect consecutive nodes on this path). The nodes on this cycle path are a subset of $w_1, \dots, w_t$; and with this procedure, applied to all $i \in \{1, \dots, k-1\}$, we get precisely all $w_1, \dots, w_t$.

Now we notice that it is sufficient to determine only the subset of cactuses $C_1, \dots, C_{k-1}$ that consists of all C_i, $i \in \{1, \dots, k-1\}$, such that (z_i, C_i) and (C_i, z_{i+1})

belong to different edge classes. (This is because, if (z_i, C_i) and (C_i, z_{i+1}) belong to the same edge class, then z_i and z_{i+1} correspond to the same node of S, and therefore no real nodes incident to C_i will be involved, and no operations on the associated data structure of C_i will have to be performed.) We can determine those cactuses with a call *boundary*(x, y) on T, by discarding the nodes x and y from the resulting boundary list L. (Observe that the intermediary real nodes of P will not appear in L, since all the edges that they are incident to belong to the same edge class.) Let $D_1, \dots, D_l$ be the nodes that appear in L. Then, for every $i \in \{1, \dots, l\}$, there is an $a(i)$ such that $D_i = C_{a(i)}$. Now, for every $i \in \{1, \dots, l\}$, D_i contains a sublist in L consisting of two edges (z'_{i_1}, D_i) and (z'_{i_2}, D_i), such that (z'_{i_1}, D_i) and $(z_{a(i)}, C_{a(i)})$ (resp. (z'_{i_2}, D_i) and $(C_{a(i)}, z_{a(i)+1})$) belong to the same edge class. This implies that z'_{i_1} corresponds to the same node as $z_{a(i)}$, and z'_{i_2} corresponds to the same node as $z_{a(i)+1}$. Thus, the node (in the associated data structure) of D_i pointed to by (z'_{i_1}, D_i) (resp. (z'_{i_2}, D_i)) is precisely the node that is pointed to by $(z_{a(i)}, C_{a(i)})$ (resp. $(C_{a(i)}, z_{a(i)+1})$), and so we can get $\tilde{z}_{a(i)}$ (resp. $\tilde{z}_{a(i)+1}$) from this pointer.

Now, for every $i \in \{1, \dots, l\}$, we have to do the following things: (1) determine the cycle path on D_i whose nodes we have to merge, (2) find the cactus edges that connect consecutive nodes on this path, (3) find the corresponding nodes on the tree of cactuses T, and (4) merge the nodes on this path and properly update the associated data structure of D_i. Let z_1 and z_2 be nodes on D_i that are the endpoints of the cycle path we have to determine. (We recall that z_1 and z_2 are given by the pointers stored in (z'_{i_1}, D_i) and (z'_{i_2}, D_i).) We will not provide the details on how to determine this path (i.e., how to perform (1)), as these can be found in [59]. (In particular, we use the procedure *TreePath*$_3$ in [59].) To perform (2), we use the pointers that we introduced in the *repr* elements of the circular split-find data structures. To be more precise, once we have determined the cycle path that connects z_1 and z_2, we then have to perform a *split* operation on every cycle involved in this path. (For the full details of what is involved in this step, see the procedure *AdjustCycles* in [59].) So let C be a cycle that we have to split on nodes u_1 and u_2. (We have that u_1 and u_2 are nodes on the cycle path connecting z_1 and z_2.) Then we check whether *repr*(u_1, C).*left* = *repr*(u_2, C) or *repr*(u_1, C).*right* = *repr*(u_2, C), or both. If either case holds, we have to return *repr*(u_1, C).*leftEdge* or *repr*(u_1, C).*rightEdge*, or both, respectively. For (3) we simply use the pointer of every node on the cycle path to the corresponding edge of T. (In particular, notice that for z_1 and z_2 we will get edges that belong to the same edge

classes as (z'_{i_1}, D_i) and (z'_{i_2}, D_i), respectively. Every other edge (w, D_i) that we get, belongs to a different edge class, and provides one of the extra nodes $w_1, \dots, w_t$ that we have to merge (and which cannot be derived simply from the call *boundary*).) Finally, we will only provide the details to (4) that have to do with the additional information that we have attached to the associated data structure of C; for the rest, we refer again to [59]. We first have to ensure that the pointers of the *repr* elements of the circular lists are updated correctly after the splittings. So let C be a cycle that we split on nodes u_1 and u_2. Then *repr*(u_1, C) and *repr*(u_2, C) are assigned to different circular lists; let us call them L_1 and L_2, respectively. Suppose that L_1 contains more that one element (for otherwise there is nothing to do for *repr*(u_1, C)). Then, one of *repr*(u_1, C)*.left*, *repr*(u_1, C)*.right* has been assigned to L_1, and the other one has been assigned to L_2. Assume w.l.o.g. that *repr*(u_1, C)*.left* has been assigned to L_1 (i.e., the same list that *repr*(u_1, C) has been assigned to). Then we must set *repr*(u_1, C)*.right* $\leftarrow$ *repr*(u_2, C)*.right* and *repr*(u_1, C)*.rightEdge* $\leftarrow$ *repr*(u_2, C)*.rightEdge*. The other case for *repr*(u_1, C) and the analogous cases for *repr*(u_2, C) are treated similarly. To conclude (4), we note that the correspondence between the nodes of (the associated data structure to) D_i and the edges of the tree of cactuses T is maintained, because, although the nodes of the cycle path on D_i that connects z_1 and z_2 got merged, this was done using a DSU data structure internal to the associated data structure to D_i. Thus we only have to make sure that, in order to access the node in an associated tree of cycles that corresponds to an edge of the tree of cactuses, we first perform a *find* on the node pointed to by this edge, using the internal DSU data structure. Algorithm 62 shows the operations that are performed in the associated data structure of D_i.

Thus, from the internal operations on the associated structure of every cactus node D_i, $i \in \{1, \dots, l\}$, we get a collection of edges $(w_{i,1}, D_i), \dots, (w_{i,k_i}, D_i)$ that we include in the edge sublist of D_i in L. Furthermore, from the edges $\{(w_{i,j}, D_i) \mid i \in \{1, \dots, l\}, j \in \{1, \dots, k_i\}\}$ we get the nodes of the cycle path on S that connects x and y, by using the pointer from every $(w_{i,j}, D_i)$ to $w_{i,j}$, and then the pointer from $\mathit{find}_{\mathit{cls}}(w_{i,j})$ to the corresponding node of S.

To conclude `compressCyclePath`$(\mathsf{S}, \mathsf{x}, \mathsf{y})$ we call *joinclasses*(L), we return a pointer to a new node z that takes the place of all the nodes on the cycle path on S connecting x and y, we merge all $w_{i,j}$, $i \in \{1, \dots, l\}$, $j \in \{1, \dots, k_i\}$ into a larger cluster (using the DSU_{cls} data structure), and we let z point to the representative of this cluster, and reversely. The operation `compressCyclePath`$(\mathsf{S}, \mathsf{x}, \mathsf{y})$ is summarized in Algorithm 63.

Algorithm 62: `updateCactus`(D, z_1, z_2)

```
// compress the cycle path on D that connects z1 and z2 using the
   associated data structure; return the set of cactus edges that connect
   consecutive nodes on this path, and a list of edges on the tree of
   cactuses that contains D that correspond to the nodes of this path
```

1 $\mathcal{E} \leftarrow \emptyset$ `// the set of cactus edges to be returned`
2 *edgelist* $\leftarrow \emptyset$ `// the list of corresponding edges to be returned`
3 find the cycle path $P = \{u_1, \ldots, u_k\}$ of D that connects z_1 and z_2 `// for this step we refer to [59]`
4 **foreach** $i \in \{1, \ldots, k-1\}$ **do**
5   let C be the cycle that contains u_i and u_{i+1}
6   **if** *repr*(u_i, C).*left* = *repr*(u_{i+1}, C) **then**
7         $\mathcal{E} \leftarrow \mathcal{E} \cup$ *repr*(u_i, C).*leftEdge*
8   **end**
9   **if** *repr*(u_i, C).*right* = *repr*(u_{i+1}, C) **then**
10         $\mathcal{E} \leftarrow \mathcal{E} \cup$ *repr*(u_i, C).*rightEdge*
11   **end**
12 **end**
13 **foreach** $u \in \{u_1, \ldots, u_k\}$ **do**
14   get the edge $(\tilde{u}, D)$ that is pointed to by u
15   *edgelist* $\leftarrow$ *edgelist* $\cup \{(\tilde{u}, D)\}$
16 **end**
17 merge the nodes on P and properly update the data structure `// again, for this step we refer to [59]`
18 fix the pointers *left*, *right*, *leftEdge* and *rightEdge* of the *repr* elements, as described in the text
19 **return** $\{\mathcal{E}, \textit{edgelist}\}$

`joinCactuses`

To perform `joinCactuses`$(\mathsf{S_1}, \ldots, \mathsf{S_k}, (\mathsf{x_1}, \mathsf{x_2}), \ldots, (\mathsf{x_k}, \mathsf{x_1}))$, we have to link the nodes $x_1, \ldots, x_k$ in a new cycle. Thus we introduce a new cactus node C (that will be made to correspond to the new cycle) and the edges $(x_1, C), \ldots, (x_k, C)$, by perform-

Algorithm 63: compressCyclePath(S, x, y)

1 $P \leftarrow \emptyset$ // the set of nodes of the cycle path on S that connects x and y
2 $\mathcal{E} \leftarrow \emptyset$ // the set of edges of S to be returned
3 $L \leftarrow$ boundary(x, y)
4 remove x and y from L
5 **foreach** *node* C *in* L **do**
6   let (z_1, C) and (z_2, C) be the two edges in the sublist of C in L
7   let $\tilde{z}_1$ be the node pointed to by (z_1, C) in the associated data structure of C
8   let $\tilde{z}_2$ be the node pointed to by (z_2, C) in the associated data structure of C
9     $\{\mathcal{E}_0, \mathit{edgelist}\} \leftarrow$ updateCactus$(C, \tilde{z}_1, \tilde{z}_2)$
10     $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{E}_0$
11   **foreach** *edge* (w, C) *in edgelist* **do**
12   get w using the pointer from (w, C)
13   get the node u of S that corresponds to w, using the pointer of $\mathit{find}_{cls}(w)$
14         $P \leftarrow P \cup \{u\}$
15   **end**
16   append *edgelist* to the sublist of C in L
17 **end**
18 joinclasses(L)
19 let z be a new node on S substituting the cycle path P
20 merge all the real nodes that appear as endpoints of the edges in the sublists of L using DSU_{cls}; let $\tilde{z}$ be the representative
21 make z point to $\tilde{z}$ and $\tilde{z}$ to z
22 **return** $\{P, \mathcal{E}, z\}$

ing $\mathit{link}(x_1, C), \ldots, \mathit{link}(x_k, C)$. This links all the trees of cactuses corresponding to $S_1, \ldots, S_k$ to C, and produces a larger tree of cactuses. For every $x \in \{x_1, \ldots, x_k\}$ that has an associated edge x_{assoc}, we put (x, C) in the edge class of x_{assoc} by a call of *joinclasses*. If an $x \in \{x_1, \ldots, x_k\}$ does not have an associated edge, then (x, C) constitutes a new edge class of its own, and the associated edge of x is set to be

$x_{assoc} \leftarrow (x, C)$.

Then we have to construct the associated data structure to the cactus node C. This must be a rooted tree of cycles which contains only one cycle $\tilde{C}$ with nodes corresponding to $x_1, \dots, x_k$, in this order. Let $\tilde{x}_1, \dots, \tilde{x}_k$ be the nodes of $\tilde{C}$ that correspond to $x_1, \dots, x_k$, respectively. We root the tree arbitrarily to any one of $\tilde{x}_1, \dots, \tilde{x}_k$. Then we construct the circular list corresponding to $\tilde{C}$. The nodes of this list are the elements *repr*$(\tilde{x}_1, \tilde{C}), \dots,$ *repr*$(\tilde{x}_k, \tilde{C})$, in this order. The pointers *left* and *right* on the *repr* elements are easy to fix. Also, we set *repr*$(\tilde{x}_i, \tilde{C})$.*leftEdge* $\leftarrow (x_{i-1}, x_i)$, for every $i \in \{2, \dots, k\}$, *repr*$(\tilde{x}_1, \tilde{C})$.*leftEdge* $\leftarrow (x_k, x_1)$, *repr*$(\tilde{x}_i, \tilde{C})$.*rightEdge* $\leftarrow (x_i, x_{i+1})$, for every $i \in \{1, \dots, k-1\}$, and *repr*$(\tilde{x}_k, \tilde{C})$.*rightEdge* $\leftarrow (x_k, x_1)$. (This is to be able to retrieve the real cactus edges.) The construction of the associated data structure to C is shown in Algorithm 65.

Finally, in order to establish the correspondence between the tree of cycles associated with C and the tree of cactuses which contains C, for every $i \in \{1, \dots, k\}$ we have a pointer from (x_i, C) to $\tilde{x}_i$, and reversely. (This is to be able to find the node within the cactus C that corresponds to x_i, and conversely.) The procedure *joinCactuses* is summarized in Algorithm 64.

8.7 Sparse certificates for the maximal k-edge-connected subgraphs

In this section we discuss constructions of (almost) sparse subgraphs that have the same maximal k-edge-connected subgraphs as the original graph. Following the terminology of [1], we define a *k-certificate* of a graph to be a spanning subgraph that has the same maximal k-edge-connected subgraphs as the original graph.[4]

Using results from Benczúr and Karger [8], we show that (1) in linear time we can construct a k-certificate of $O(kn \log n)$ size, and (2) in $O(m \log^2 n)$ time we can construct a k-certificate of $O(kn)$ size. Thus, by combining (1) and (2), we can construct a k-certificate of $O(kn)$ size in $O(m + kn \log^3 n)$ time. This is a result analogous to [51], which provides certificates for the k-edge-connected components. In fact, we use

[4]This definition can be extended, in order to include objects such as a graph G' from which "we can derive easily" the maximal k-edge-connected subgraphs of the original graph once we have computed those of G'. However, the definition we provided here is enough for our purposes.

Algorithm 64: `joinCactuses`($S_1, \ldots, S_k, (x_1, x_2), \ldots, (x_k, x_1)$)

1 introduce a new cactus node C
2 **foreach** $x \in \{x_1, \ldots, x_k\}$ **do** `link`(x, C)
3 **foreach** $x \in \{x_1, \ldots, x_k\}$ **do** let $\{(x, C)\}$ constitute a new edge class
4 **foreach** $x \in \{x_1, \ldots, x_k\}$ **do**
5   **if** $x_{assoc} \neq \emptyset$ **then**
6   let L be a singleton list consisting of x
7   let the sublist of x be $\{x_{assoc}, (x, C)\}$
8   `joinclasses`(L)
9   **end**
10   **else** $x_{assoc} \leftarrow (x, C)$
11 **end**
12 perform `initialize_cycle`$(C, x_1, \ldots, x_k, (x_1, x_2), \ldots, (x_k, x_1))$, and collect the corresponding nodes $\{\tilde{x}_1, \ldots, \tilde{x}_k\}$
13 **foreach** $i \in \{1, \ldots, k\}$ **do**
14   make (x_i, C) point to $\tilde{x}_i$ and $\tilde{x}_i$ point to (x_i, C)
15 **end**

Algorithm 65: `initialize_cycle`($C, x_1, \ldots, x_k, (x_1, x_2), \ldots, (x_k, x_1)$)

1 create nodes $\tilde{x}_1, \ldots, \tilde{x}_k$ and $\tilde{C}$
2 create a tree of cycles consisting of the edges $(\tilde{x}_1, \tilde{C}), \ldots, (\tilde{x}_k, \tilde{C})$
3 root the tree at $\tilde{x}_1$
4 create the elements $repr(\tilde{x}_1, \tilde{C}), \ldots, repr(\tilde{x}_k, \tilde{C})$
5 initialize a circular split-find data structure on the elements $repr(\tilde{x}_1, \tilde{C}), \ldots, repr(\tilde{x}_k, \tilde{C})$ (in this order), and associated it with $\tilde{C}$ `// here we refer to [59]`
6 fix the pointers *left*, *right*, *leftEdge* and *rightEdge* of the *repr* elements as described in the text
7 **return** $\{\tilde{x}_1, \ldots, \tilde{x}_k\}$

the following concept from [51] (defined formally in [52]): a *forest decomposition* with t forests of a graph G is a collection of forests $F_1, \ldots, F_t$, such that F_1 is a spanning forest of G, and F_i is a spanning forest of $G \setminus (F_1 \cup \cdots \cup F_{i-1})$, for $i \in \{2, \ldots, t\}$. We

also note that (1) is used by Saranurak and Yuan [61] in their $O(m + n^{1+o(1)})$-time algorithm for computing the k-edge-connected subgraphs of an undirected graph, for any $k = \log^{o(1)} n$.

Let G be a graph. An edge of G whose endpoints lie in different *maximal* k-edge-connected subgraphs of G is called a *k-interconnection edge* of G. Algorithm 66 describes an algorithm for computing a k-certificate of G.

Algorithm 66: Compute a certificate for the maximal k-edge-connected subgraphs of G

---

1 let E' be a set of edges of G that contains all its k-interconnection edges
2 compute a forest decomposition $\mathcal{F}$ of $G \setminus E'$ with k forests
3 **return** $\mathcal{F} \cup E'$

Lemma 8.2. *Algorithm 66 outputs a certificate for the maximal k-edge-connected subgraphs of G.*

Proof. If we remove all the k-interconnection edges from G, then the connected components of the resulting graph coincide with the maximal k-edge-connected subgraphs of G. Let S be a maximal k-edge-connected subgraph of G and let E' be a set of edges of G that contains all its k-interconnection edges. Then $[S, \bar{S}] = \emptyset$ in $G \setminus E'$. Now let H be the set of all edges of E' that are contained in S. Then the set $\mathcal{F}_S$ of all the edges of $\mathcal{F}$ that are contained in $S \setminus H$ is a forest decomposition of $S \setminus H$ with k forests[5] (precisely because S, and therefore $S \setminus H$, is disconnected from the rest of the graph in $G \setminus E'$). Thus, we know from the sparsification paper of Eppstein et al. [27] that $\mathcal{F}_S$ (considered as a graph) is a *strong certificate*[6] for the k-edge-connectivity of $S \setminus H$. Thus, $\mathcal{F}_S \cup H$ is a strong certificate for the k-edge-connectivity of $(S \setminus H) \cup H = S$. But S is k-edge-connected, and so $\mathcal{F} \cup E' \supseteq \mathcal{F}_S \cup H$ contains enough edges of S to maintain it as a k-edge-connected subgraph of $\mathcal{F} \cup E'$. Since $\mathcal{F} \cup E'$ is a subgraph of G, we thus have that its maximal k-edge-connected subgraphs coincide with those of G. ∎

Corollary 8.1. *Let $\mathcal{A}$ be an algorithm that, given a graph G with m edges and n vertices, computes in $T(m, n)$ time a subset E' of $E(G)$ with $S(m, n)$ size that contains all its*

[5] more precisely: it either coincides with $S \setminus H$, or it has k forests

[6] For any graph property $\mathcal{P}$, and graph G, a strong certificate for G is a graph G' on the same vertex set such that, for any graph H, $G \cup H$ has property $\mathcal{P}$ if and only if $G' \cup H$ has the property.

k-interconnection edges. Then we can construct a k-certificate of G with $O(kn + S(m,n))$ size in $O(m + n + T(m,n))$ time.

Proof. First we apply algorithm $\mathcal{A}$ in order to compute a subset E' of $E(G)$ with size $S(m,n)$ that contains all its k-interconnection edges. This takes $T(m,n)$ time. Then we apply Algorithm 66: we remove E' from G, we compute a forest decomposition $\mathcal{F}$ of $G \setminus E'$ with k forests, and we return $\mathcal{F} \cup E'$. Using [51], the computation of $\mathcal{F}$ takes time $O(m+n)$. The output has size $O(kn + S(m,n))$. Correctness is guaranteed by Lemma 8.2. □

In the following discussion we refer to Section 8 of [8]. (Note that in [8] the k-interconnection edges are called k-weak edges, and the maximal k-edge-connected subgraphs are called k-strong components.) In [8], they prove that a forest decomposition with at least $4k \log n$ forests contains all the k-interconnection edges of a graph. Since this decomposition can be computed in linear time using the MA-ordering algorithm of Nagamochi and Ibaraki [51], by Corollary 8.1 we get result (1): we can construct a k-certificate of size $O(kn \log n)$ in linear time.

Alternatively, [8] defines the *strength* of an edge e, denoted by k_e, as the largest k' such that e lies entirely within a maximal k'-edge-connected subgraph. Then the k-interconnection edges are precisely those whose strength is less than k. [8] provides an $O(m \log^2 n)$-time algorithm that assigns a value $\tilde{k}_e$ to every edge e, such that $\tilde{k}_e \leq k_e$ and $\sum_{e \in E} 1/\tilde{k}_e = O(n)$. Now we consider the set E' of all edges e that have $\tilde{k}_e < k$. Observe that E' contains all the k-interconnection edges, since $\tilde{k}_e \leq k_e$ for every edge e. Then we have that $O(n) = \sum_{e \in E} 1/\tilde{k}_e \geq \sum_{e \in E'} 1/\tilde{k}_e > \sum_{e \in E'} 1/k = |E'|/k$. Thus, E' has size $O(kn)$. Therefore, by Corollary 8.1 we get immediately our result (2): we can construct a k-certificate of size $O(kn)$ in $O(m \log^2 n)$ time.

8.8 Computing the maximal k-edge-connected subgraphs

Let G be an undirected multigraph with m edges and n vertices, and let $k > 2$ be a fixed integer. Our k-certificates developed in Section 8.7 can be immediately applied to compute the maximal k-edge-connected subgraphs of G. In particular, we first observe that we can apply the local search routine of Chechik et al. [16] (Lemma 5.1) into the framework of Forster et al. [29] (Lemma 7.2), so that we get a deterministic

$O(km\log^2 n + (k^{O(k)} + k^{O(1)}\log n)n\sqrt{n})$-time algorithm for computing the maximal k-edge-connected subgraphs of an undirected graph. By the result of the previous section, we can compute in $O(m + kn\log^3 n)$ time a k-certificate of $O(kn)$ size. More precisely, the certificate has $O(\min\{m, kn\})$ size, since, by construction, it is a subgraph of the original graph. Now we can apply the deterministic variant of the Forster et al. algorithm on this certificate, and thus we get the following.

Theorem 8.3. *There is an $O(m + (k^{O(k)} + k^{O(1)}\log n)n\sqrt{n})$-time algorithm for computing the maximal k-edge-connected subgraphs of an undirected graph.*

Note that for constant $k \geq 3$, the time-bound provided by Theorem 8.3 is $O(m + n\sqrt{n}\log n)$, which improves the randomized bound of $O(m\log^2 n + n\sqrt{n}\log n)$ given by Forster et al. [29]. Similarly, by applying the $O(km\log^2 n + k^3 n\sqrt{n}\log n)$-time algorithm of Forster et al. [29] on our k-certificate we obtain our next result.

Theorem 8.4. *There is a randomized Las Vegas algorithm for computing the maximal k-edge-connected subgraphs of an undirected graph that has $O(m + k^3 n^{3/2}\log n)$ expected running time.*

We next describe another algorithm to compute the maximal k-edge-connected subgraphs of an undirected graph G. We can do that by repeatedly removing any k'-edge cut from G, for $k' < k$, until there are no more k'-edge cuts in the graph for $k' < k$. Then the connected components of the resulting graph coincide with the maximal k-edge-connected subgraphs of G. The best known deterministic algorithm for computing a k'-edge cut of an undirected multigraph, for $k' < k$, or concluding that the graph is k-edge-connected, is given by Gabow [31], and it runs in $O(m + k^2 n\log(n/k))$ time. Thus the total running time of this algorithm would be bounded by $O(mn + k^2 n^2\log(n/k))$.

To improve this bound, we reduce the time that it takes to successively find a min-cut with k'-edges (for $k' < k$). To speed this up, we repeatedly compute a min-cut of a graph and store enough information to facilitate the search for further cuts. To accomplish this task, we rely on Thorup's fully dynamic min-cut algorithm [66]. This algorithm supports edge insertions and deletions in $\widetilde{O}(\sqrt{n})$ time per update, and it maintains a min-cut of size up to $k-1$, for any fixed k (polylogarithmic on the number of vertices). It also demands an additional $O(m+n)$ time to initialize the underlying data structure on a sparse certificate of k-edge-connectivity for G [51]. (Thus, this

algorithm maintains dynamically both a sparse certificate for G, using sparsification by Eppstein et al. [27], and a min-cut of size up to $k-1$ of this certificate.)

Now we can use the fully dynamic min-cut algorithm of [66] on each connected component C of G as follows. As long as there is a k'-edge cut $[S,\bar{S}]$ of C, for some $k' < k$, we remove the edges of $[S,\bar{S}]$ from C, and then we select two arbitrary vertices $x \in S$ and $y \in \bar{S}$, and we reconnect the two resulting components S and $\bar{S}$ by adding k multiple edges between x and y. Thus, the reconnection of the two components S and $\bar{S}$ with k multiple edges ensures that all k'-edge cuts, for $k' < k$, are maintained in each component after the removal of $[S,\bar{S}]$, and that these are all the k'-edge cuts that may appear now in C. Eventually all components of G will become k-edge-connected. In the meantime, we collect all the edge-cuts that we find, in every connected component of G, and in the end we remove all of them from G. It should be clear that the connected components of the resulting graph coincide with the maximal k-edge-connected subgraphs of G. Observe that every search for a k'-edge cut, for $k' < k$, on some connected component of G, is immediately followed by k' deletions and k insertions of edges. Furthermore, the total number of these cuts is $O(n)$. Thus, the total running time of this algorithm is $\widetilde{O}(m+k^{O(1)}n\sqrt{n})$. To provide a more refined analysis, we give a lower bound of the dependency of the complexity of this algorithm on k and on the number of log factors involved. To do that, we note that Thorup's algorithm uses a greedy tree packing of the sparse certificate using $k^7 \log^4 n$ trees, and it maintains it using Frederickson's dynamic minimum spanning tree algorithm [30]. Thus, for every k'-edge cut that we find, the cost of the deletions and insertions that follow is at least $\Omega(k^8\sqrt{n}\log^4 n)$, and the total complexity of the algorithm is at least $\Omega(m + k^8 n\sqrt{n}\log^4 n)$. This yields the following.

Theorem 8.5. *Let $k = \log^{O(1)} n$. Then there is an $O(m + k^{O(1)}n\sqrt{n}\log^{O(1)} n)$-time algorithm for computing the maximal k-edge-connected subgraphs of an undirected graph.*

We remark here that the $O(m + n^{1+o(1)})$ time bound of Saranurak and Yuan [61] requires that $k = \log^{o(1)} n$, while the bounds in Theorems 8.3 and 8.4 hold for any k, and the bound of Theorem 8.5 holds for any $k = \log^{O(1)} n$. As another remark, we note that for constant k the bound provided by Theorem 8.5 is slightly worse than the bound of Theorem 8.3, since there are more log factors involved. However, the time-bound in Theorem 8.5 has polynomial dependency on k (whereas that of Theorem 8.3 has exponential dependency on k). Also, we believe that the algorithm

of Theorem 8.5 can still be relevant, as any future improvement in the time bounds for a fully dynamic min-cut algorithm implies improved time bounds for computing the maximal k-edge-connected subgraphs.

We end this section by listing in Table 8.1 the previously known best time bounds for computing the maximal k-edge-connected subgraphs. The new results obtained in this section are described in Table 8.2.

Table 8.1: Previous best time bounds for computing the maximal k-edge-connected subgraphs.

Algorithm	Time	Type	Note
Chechik et al. [16]	$O(k^{O(k)}(m + n\log n)\sqrt{n})$	Det.	$k \in \mathbb{N}$
Henzinger et al. [42]	$O(n^2 \log n)$	Det.	$k = O(1)$
Forster et al. [29]	$O(k^3 n\sqrt{n}\log n + km\log^2 n)$	Las Vegas Rand.	$k \in \mathbb{N}$
Saranurak and Yuan [61]	$O(m + n^{1+o(1)})$	Det.	$k = \log^{o(1)} n$

Table 8.2: Improved time bounds for computing the maximal k-edge-connected subgraphs obtained in this section.

Algorithm	Time	Type	Note
This chapter (Theorem 8.3)	$O(m + (k^{O(k)} + k^{O(1)}\log n)n\sqrt{n})$	Det.	$k \in \mathbb{N}$
This chapter (Theorem 8.4)	$O(m + k^3 n\sqrt{n}\log n)$	Las Vegas Rand.	$k \in \mathbb{N}$
This chapter (Theorem 8.5)	$O(m + k^{O(1)}n\sqrt{n}\log^{O(1)} n)$	Det.	$k = \log^{O(1)} n$

8.9 A fully dynamic algorithm for maximal k-edge-connectivity

In this section we describe our fully dynamic algorithm for maintaining information about the maximal k-edge-connected subgraphs of a dynamic graph.In more detail, we wish to maintain an undirected graph $G = (V, E)$ throughout an intermixed sequence of the following operations:

insert(x, y): Add edge (x, y) to G;

delete(x, y): Remove edge (x, y) from G (the operation assumes that (x, y) is in G);

max-k*-edge*(x, y): Return *true* if vertices x and y are in the same maximal k-edge-connected subgraph of G, and *false* otherwise.

Before stating our bounds, let us review some simple minded approaches for the problem. In Section 8.7 we showed that the maximal k-edge-connected subgraphs can be computed in time $O(m + n\sqrt{n} \log n)$ (for constant k). Note that if we recompute from scratch the maximal k-edge-connected subgraphs after each update, *max-*k*-edge* queries can be answered in constant time. This yields a simple algorithm that implements *insert* and *delete* operations in time $O(m + n\sqrt{n} \log n)$ and *max-*k*-edge* queries in constant time. On the other side, one could simply do no extra work during *insert* and *delete* operations, but then answering a *max-*k*-edge* query would require recomputing the maximal k-edge-connected subgraphs from scratch, yielding constant time per update and $\widetilde{O}(m + n\sqrt{n}\,)$ time per query. We next show how to implement *insert* and *delete* operations in better $\widetilde{O}(n\sqrt{n}\,)$ time, while still keeping the running time for *max-*k*-edge* queries constant. To achieve our improved bounds, we exploit the sparsification technique of Eppstein et al. [27]. We start with the following definition.

Definition 8.1. Let $k \geq 3$ be a fixed integer. Given an undirected graph $G = (V, E)$ with m edges and n vertices, a *sparse certificate of maximal k-edge-connectivity* for G is a graph G' defined on the same vertex set as G such that the following holds:

(i) G' has $O(n)$ edges;

(ii) For any graph H, any two vertices are in the same maximal k-edge-connected subgraph of $G' \cup H$ if and only if they are in the same maximal k-edge-connected subgraph in $G \cup H$.

The following lemma provides a sufficient condition for inferring that a k-certificate is a sparse certificate of maximal k-edge-connectivity.

Lemma 8.3. *Let G be an undirected graph, and let C be a k-certificate of G with $O(n)$ edges that contains all the k-interconnection edges of G. Then C is a sparse certificate of maximal k-edge-connectivity for G.*

Proof. Condition (i) of Definition 8.1 is satisfied. It remains to establish condition (ii). Since C is a k-certificate of G, it is a subgraph of G that has the same maximal k-edge-connected subgraphs as G. Let Q be the quotient graph that is formed by shrinking every maximal k-edge-connected subgraph of C into a single vertex. Then, since C contains all the k-interconnection edges of G, by Property 8.1 we have that Q has the same decomposition tree into maximal k-edge-connected subgraphs as G. Thus, the changes that this tree undergoes after inserting all the vertices and edges of H into G, are the same as if inserting them to C. Thus, $C \cup H$ has the same maximal k-edge-connected subgraphs as $G \cup H$. This shows that condition (ii) is satisfied too. Therefore C is a sparse certificate of maximal k-edge-connectivity for G. □

Corollary 8.2. *Let $G = (V, E)$ be an undirected graph with m edges and n vertices. A sparse certificate of maximal k-edge-connectivity for G can be computed in time $O(m \log^2 n)$.*

Proof. In Section 8.7 we show that we can construct a k-certificate C of G with $O(n)$ edges in time $O(m \log^2 n)$. Furthermore, this k-certificate has the property that it contains all the k-interconnection edges of G. Thus, Lemma 8.3 implies that C is a sparse certificate of maximal k-edge-connectivity for G. □

We are now ready to apply the sparsification framework of Eppstein et al. [27]:

Theorem 8.6 ([27])**.** *Let $k \geq 3$ be a fixed integer, let $f(n, m)$ be the time required to compute a sparse certificate of maximal k-edge-connectivity, and let $g(n, m)$ the time required to compute the maximal k-edge-connected subgraphs. Then we can build a fully dynamic data structure that can handle* insert *and* delete *operations in time $O(f(n, O(n)) + g(n, O(n)))$ and* max-k-edge *queries in constant time.*

From Corollary 8.2 we have $f(m, n) = O(m \log^2 n)$. From [16] we have $g(m, n) = O(m\sqrt{n})$ for $k \in \{3, 4\}$, and $g(m, n) = O((m + n \log n)\sqrt{n})$ for fixed $k > 4$. (The improved time-bounds for $k = 4$ are derived by using either of the algorithms of [36, 50] for computing 3-edge cuts in linear time.) By fitting those bounds into Theorem 8.6 we obtain:

Theorem 8.7. *Let $G = (V, E)$ be an undirected graph with n vertices. Then we can build a fully dynamic data structure that can handle* insert *and* delete *operations in time $O(n\sqrt{n})$, for $k \in \{3, 4\}$, and in time $O(n\sqrt{n} \log n)$, for fixed $k > 4$, so that it can answer* max-k-edge *queries in constant time.*

Also, for $k = \log^{o(1)} n$, by [61] we have $g(m,n) = O(m + n^{1+o(1)})$, so now Theorem 8.6 gives:

Theorem 8.8. *Let $G = (V, E)$ be an undirected graph with n vertices, and let $k = \log^{o(1)} n$. Then we can build a fully dynamic data structure that can handle* insert *and* delete *operations in time $O(n \log^2 n + n^{1+o(1)})$, so that it can answer* max-k-edge *queries in constant time.*

8.10 Conclusions

We presented two algorithms for maintaining a decomposition tree structure of the maximal 3-edge-connected subgraphs of a graph. The first algorithm uses $O(n)$ space and can handle any sequence of m edge insertions and n vertex insertions in total time $O(n^2 \log^2 n + m\alpha(m,n))$. The second algorithm uses $O(n^2)$ space and can handle any sequence of m edge insertions and n vertex insertions in total time $O(n^2\alpha(n,n) + m\alpha(m,n))$.

We note that one can use this data structure to efficiently answer interspersed queries concerning the maximal 3-edge-connected subgraphs, such as:

- Given vertices x and y, report whether x and y belong to the same maximal 3-edge-connected subgraph (using two calls to a DSU-*find* operation)
- Find the maximal 3-edge-connected subgraph that contains x, in time analogous to its size (plus a call to a DSU-*find* operation)
- Report the size (the number of vertices or edges) of the maximal 3-edge-connected subgraph that contains x in constant time (plus a call to a DSU-*find* operation)
- Return all maximal 3-edge-connected subgraphs in time analogous to their size
- Return the number of all maximal 3-edge-connected subgraphs in constant time

The methods used in this algorithm extend previous work in maintaining the 2- and 3-edge-connected components [35, 59, 69], and may prove useful in solving other similar problems. For instance, it seems possible that we can add more levels to the decomposition tree, and rely on previous work for maintaining the 4- and 5-edge-connected components, in order to maintain the maximal 4- and 5-edge-connected

subgraphs (by properly adjusting the data structures and algorithms in [23, 25]). Furthermore, it seems that any improvement in maintaining the k-edge-connected components, for any constant k, would imply (under some conditions) that this solution can be plugged in to our framework for maintaining the decomposition tree, in order to maintain the maximal k-edge-connected subgraphs (with time-bounds analogous to those provided in this chapter). It seems possible that this framework could also be useful for maintaining the maximal k-vertex-connected subgraphs, by relying on efficient algorithms for maintaining the k-vertex-connected components (such as those described in [7, 57], for the case $k = 3$).

We also showed that we can construct (almost) sparse certificates for them maximal k-edge-connected subgraphs. Our result implies that the difficulty in designing efficient algorithms for computing the maximal k-edge-connected subgraphs lies essentially in sparse graphs, and we can use it in order to speed up the running time of already known algorithms. In particular, by using the algorithm by Chechik et at. [16], we get an $O(m + k^{O(k)} n\sqrt{n} \log n)$-time algorithm for computing the maximal k-edge-connected subgraphs in undirected graphs.

We believe that it is an interesting question whether a k-certificate of $O(n)$ size can be computed in linear time. This would be trivial if we had a linear-time algorithm for computing the maximal k-edge-connected subgraphs of a graph, but it is still an open problem whether this can be done for $k \geq 3$. Thus, we have to perform the construction of the certificates without explicitly computing the maximal k-edge-connected subgraphs, and this seems to be a challenging task.

Bibliography

[1] Anders Aamand, Adam Karczmarz, Jakub Lacki, Nikos Parotsidis, Peter M. R. Rasmussen, and Mikkel Thorup. Optimal decremental connectivity in non-sparse graphs. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, July 10-14, 2023, Paderborn, Germany*, volume 261 of *LIPIcs*, pages 6:1–6:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ICALP.2023.6, doi:10.4230/LIPICS.ICALP.2023.6.

[2] Amir Abboud, Robert Krauthgamer, Jason Li, Debmalya Panigrahi, Thatchaphol Saranurak, and Ohad Trabelsi. Breaking the cubic barrier for all-pairs max-flow: Gomory-hu tree in nearly quadratic time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 884–895. IEEE, 2022. doi:10.1109/FOCS54457.2022.00088.

[3] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. APMF $<$ APSP? gomory-hu tree for unweighted graphs in almost-quadratic time. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1135–1146. IEEE, 2021. doi:10.1109/FOCS52979.2021.00112.

[4] Amir Abboud, Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. All-pairs max-flow is no harder than single-pair max-flow: Gomory-hu trees in almost-linear time. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 2204–2212. IEEE, 2023. doi:10.1109/FOCS57990.2023.00137.

[5] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction.

In Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi, editors, *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 909–918. ACM, 2013. doi:10.1145/2505515.2505751.

[6] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005. doi:10.1145/1103963.1103966.

[7] Giuseppe Di Battista and Roberto Tamassia. On-line maintenance of triconnected components with spqr-trees. *Algorithmica*, 15(4):302–318, 1996. doi:10.1007/BF01961541.

[8] András A. Benczúr and David R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM J. Comput.*, 44(2):290–319, 2015. doi:10.1137/070705970.

[9] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

[10] Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004. doi:10.1016/j.tcs.2003.05.002.

[11] Endre Boros, Konrad Borys, Vladimir Gurvich, and Gábor Rudolf. Generating 3-vertex connected spanning subgraphs. *Discret. Math.*, 308(24):6285–6297, 2008. URL: https://doi.org/10.1016/j.disc.2007.11.067, doi:10.1016/J.DISC.2007.11.067.

[12] Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert Endre Tarjan, and Jeffery R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM J. Comput.*, 38(4):1533–1573, 2008. doi:10.1137/070693217.

[13] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *Proceedings of the 27th ACM Symposium on Computational Geometry,*, pages 1–10, 2011. doi:10.1145/1998196.1998198.

[14] Lijun Chang and Zhiyi Wang. A near-optimal approach to edge connectivity-based hierarchical graph decomposition. *Proc. VLDB Endow.*, 15(6):1146–1158, 2022. URL: https://www.vldb.org/pvldb/vol15/p1146-chang.pdf, doi:10.14778/3514061.3514063.

[15] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. Efficiently computing k-edge connected components via graph decomposition. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 205–216. ACM, 2013. doi:10.1145/2463676.2465323.

[16] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Veronika Loitzenbauer, and Nikos Parotsidis. Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1900–1918. SIAM, 2017. doi:10.1137/1.9781611974782.124.

[17] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 612–623. IEEE, 2022. doi:10.1109/FOCS54457.2022.00064.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[19] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008. URL: https://www.worldcat.org/oclc/227584184.

[20] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[21] Yefim Dinitz. The 3-edge-components and a structural description of all 3-edge-cuts in a graph. In Ernst W. Mayr, editor, *Graph-Theoretic Concepts in Computer*

Science, 18th International Workshop, WG '92, Wiesbaden-Naurod, Germany, June 19-20, 1992, Proceedings, volume 657 of *Lecture Notes in Computer Science*, pages 145–157. Springer, 1992. doi:10.1007/3-540-56402-0_44.

[22] Yefim A. Dinic, Alexander V. Karzanov, and Michael V Lomonosov. On the structure of the system of minimum edge-cuts in a graph. *Studies in Discrete Optimization*, pages 290–306, 1976.

[23] Yefim Dinitz and Ronit Nossenson. Incremental maintenance of the 5-edge-connectivity classes of a graph. In Magnús M. Halldórsson, editor, *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings*, volume 1851 of *Lecture Notes in Computer Science*, pages 272–285. Springer, 2000. doi:10.1007/3-540-44985-X_25.

[24] Yefim Dinitz and Zeev Nutov. A 2-level cactus model for the system of minimum and minimum+1 edge-cuts in a graph and its incremental maintenance. In Frank Thomson Leighton and Allan Borodin, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 509–518. ACM, 1995. doi:10.1145/225058.225268.

[25] Yefim Dinitz and Jeffery R. Westbrook. Maintaining the classes of 4-edge-connectivity in a graph on-line. *Algorithmica*, 20(3):242–276, 1998. doi:10.1007/PL00009195.

[26] Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. *SIAM J. Comput.*, 49(6):1363–1396, 2020. doi:10.1137/17M1146610.

[27] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997. doi:10.1145/265910.265914.

[28] Lisa Fleischer. Building chain and cactus representations of all minimum cuts from hao-orlin in the same asymptotic run time. *J. Algorithms*, 33(1):51–72, 1999. URL: https://doi.org/10.1006/jagm.1999.1039, doi:10.1006/JAGM.1999.1039.

[29] Sebastian Forster, Danupon Nanongkai, Liu Yang, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Computing and testing small connectivity in near-linear time and queries via fast local cut algorithms. In Shuchi Chawla,

editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2046–2065. SIAM, 2020. doi:10.1137/1.9781611975994.126.

[30] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985. doi:10.1137/0214055.

[31] Harold N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Syst. Sci.*, 50(2):259–273, 1995. URL: https://doi.org/10.1006/jcss.1995.1022, doi:10.1006/JCSS.1995.1022.

[32] Harold N. Gabow. The minset-poset approach to representations of graph connectivity. *ACM Trans. Algorithms*, 12(2):24:1–24:73, 2016. doi:10.1145/2764909.

[33] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.

[34] Zvi Galil and Giuseppe F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, 1991. doi:10.1145/122413.122416.

[35] Zvi Galil and Giuseppe F. Italiano. Maintaining the 3-edge-connected components of a graph on-line. *SIAM J. Comput.*, 22(1):11–28, 1993. doi:10.1137/0222002.

[36] Loukas Georgiadis, Giuseppe F. Italiano, and Evangelos Kosinas. Computing the 4-edge-connected components of a graph in linear time. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 47:1–47:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: https://doi.org/10.4230/LIPIcs.ESA.2021.47, doi:10.4230/LIPICS.ESA.2021.47.

[37] Loukas Georgiadis, Evangelos Kipouridis, Charis Papadopoulos, and Nikos Parotsidis. Faster computation of 3-edge-connected components in digraphs. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January*

22-25, 2023, pages 2489–2531. SIAM, 2023. URL: https://doi.org/10.1137/1.9781611977554.ch96, doi:10.1137/1.9781611977554.CH96.

[38] Loukas Georgiadis and Evangelos Kosinas. Linear-time algorithms for computing twinless strong articulation points and related problems. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation, ISAAC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 181 of *LIPIcs*, pages 38:1–38:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. URL: https://doi.org/10.4230/LIPIcs.ISAAC.2020.38, doi:10.4230/LIPICS.ISAAC.2020.38.

[39] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961. arXiv:https://doi.org/10.1137/0109047, doi:10.1137/0109047.

[40] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

[41] Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. Efficient algorithms for computing all low *s-t* edge connectivities and related problems. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 127–136. SIAM, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283398.

[42] Monika Henzinger, Sebastian Krinninger, and Veronika Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, volume 9134 of *Lecture Notes in Computer Science*, pages 713–724. Springer, 2015. doi:10.1007/978-3-662-47672-7_58.

[43] Monika Rauch Henzinger and David P. Williamson. On the number of small cuts in a graph. *Inf. Process. Lett.*, 59(1):41–44, 1996. doi:10.1016/0020-0190(96)00079-8.

[44] Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Lacki, Eva Rotenberg, and Piotr Sankowski. Contracting a planar graph efficiently. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, volume 87 of *LIPIcs*, pages 50:1–50:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. URL: https://doi.org/10.4230/LIPIcs.ESA.2017.50, doi:10.4230/LIPICS.ESA.2017.50.

[45] John E. Hopcroft and Robert Endre Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973. doi:10.1137/0202012.

[46] Wenyu Jin and Xiaorui Sun. Fully dynamic s-t edge connectivity in subpolynomial time (extended abstract). In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 861–872. IEEE, 2021. doi:10.1109/FOCS52979.2021.00088.

[47] Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. A nearly optimal all-pairs min-cuts algorithm in simple graphs. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1124–1134. IEEE, 2021. doi:10.1109/FOCS52979.2021.00111.

[48] Yuan Li, Guoren Wang, Yuhai Zhao, Feida Zhu, and Yubao Wu. Towards k-vertex connected component discovery from large networks. *World Wide Web*, 23(2):799–830, 2020. URL: https://doi.org/10.1007/s11280-019-00725-6, doi:10.1007/S11280-019-00725-6.

[49] Yaowei Long and Thatchaphol Saranurak. Near-optimal deterministic vertex-failure connectivity oracles. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 1002–1010, 2022. doi:10.1109/FOCS54457.2022.00098.

[50] Wojciech Nadara, Mateusz Radecki, Marcin Smulewicz, and Marek Sokolowski. Determining 4-edge-connected components in linear time. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 71:1–71:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: https://doi.org/10.4230/LIPIcs.ESA.2021.71, doi:10.4230/LIPICS.ESA.2021.71.

[51] Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992. doi:10.1007/BF01758778.

[52] Hiroshi Nagamochi and Toshihide Ibaraki. *Algorithmic Aspects of Graph Connectivity*, volume 123 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2008. doi:10.1017/CBO9780511721649.

[53] Chaitanya Nalam and Thatchaphol Saranurak. Maximal k-edge-connected subgraphs in weighted graphs via local random contraction. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 183–211. SIAM, 2023. URL: https://doi.org/10.1137/1.9781611977554.ch8, doi:10.1137/1.9781611977554.CH8.

[54] Seth Pettie, Thatchaphol Saranurak, and Longhui Yin. Optimal vertex connectivity oracles. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 151–161. ACM, 2022. doi:10.1145/3519935.3519945.

[55] Michal Pilipczuk, Nicole Schirrmacher, Sebastian Siebertz, Szymon Torunczyk, and Alexandre Vigny. Algorithms and data structures for first-order logic with connectivity under vertex failures. In *49th International Colloquium on Automata, Languages, and Programming, ICALP*, volume 229 of *LIPIcs*, pages 102:1–102:18, 2022. doi:10.4230/LIPIcs.ICALP.2022.102.

[56] Johannes A. La Poutré. Dynamic Graph Algorithms and Data Structures. *Ph.D. thesis, Utrecht University, The Netherlands*, September 1991.

[57] Johannes A. La Poutré. Maintenance of triconnected components of graphs (extended abstract). In Werner Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 354–365. Springer, 1992. doi:10.1007/3-540-55719-9_87.

[58] Johannes A. La Poutré. Maintenance of 2- and 3-edge-connected components of graphs II. *SIAM J. Comput.*, 29(5):1521–1549, 2000. doi:10.1137/S0097539793257770.

[59] Johannes A. La Poutré, Jan van Leeuwen, and Mark H. Overmars. Maintenance of 2- and 3-edge- connected components of graphs I. *Discret. Math.*, 114(1-3):329–359, 1993. doi:10.1016/0012-365X(93)90376-5.

[60] Mihai Puatracscu and Mikkel Thorup. Planning for fast connectivity updates. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings*, pages 263–271. IEEE Computer Society, 2007. doi:10.1109/FOCS.2007.54.

[61] Thatchaphol Saranurak and Wuwei Yuan. Maximal k-edge-connected subgraphs in almost-linear time for small k. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms, ESA 2023, September 4-6, 2023, Amsterdam, The Netherlands*, volume 274 of *LIPIcs*, pages 92:1–92:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ESA.2023.92, doi:10.4230/LIPICS.ESA.2023.92.

[62] Heli Sun, Jianbin Huang, Yang Bai, Zhongmeng Zhao, Xiaolin Jia, Fang He, and Yang Li. Efficient k-edge connected component detection through an early merging and splitting strategy. *Knowl. Based Syst.*, 111:63–72, 2016. URL: https://doi.org/10.1016/j.knosys.2016.08.006, doi:10.1016/J.KNOSYS.2016.08.006.

[63] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

[64] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. doi:10.1145/321879.321884.

[65] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.*, 18(2):110–127, 1979. doi:10.1016/0022-0000(79)90042-4.

[66] Mikkel Thorup. Fully-dynamic min-cut. *Comb.*, 27(1):91–127, 2007. URL: https://doi.org/10.1007/s00493-007-0045-2, doi:10.1007/S00493-007-0045-2.

[67] Yung H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *J. Discrete Algorithms*, 7(1):130–146, 2009. URL: https://doi.org/10.1016/j.jda.2008.04.003, doi:10.1016/J.JDA.2008.04.003.

[68] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Ling Chen. Enumerating k-vertex connected components in large graphs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 52–63. IEEE, 2019. doi:10.1109/ICDE.2019.00014.

[69] Jeffery R. Westbrook and Robert Endre Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(5&6):433–464, 1992. doi:10.1007/BF01758773.

[70] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. I/O efficient ECC graph decomposition via graph reduction. *VLDB J.*, 26(2):275–300, 2017. URL: https://doi.org/10.1007/s00778-016-0451-4, doi:10.1007/S00778-016-0451-4.

Author's Publications

1. Computing the 5-Edge-Connected Components in Linear Time. Evangelos Kosinas. In ACM-SIAM Symposium on Discrete Algorithms (SODA), 2024.

2. 2-Fault-Tolerant Strong Connectivity Oracles. Loukas Georgiadis, Evangelos Kosinas, and Daniel Tsokaktsis. In SIAM Symposium on Algorithm Engineering and Experiments (ALENEX), 2024.

3. Connectivity Queries under Vertex Failures: Not Optimal, but Practical. Evangelos Kosinas. In European Symposium on Algorithms (ESA), 2023.

4. On 2-Strong Connectivity Orientations of Mixed Graphs and Related Problems. Loukas Georgiadis, Dionysios Kefallinos, and Evangelos Kosinas. In International Workshop on Combinatorial Algorithms (IWOCA), 2023.

5. Computing the 4-Edge-Connected Components of a Graph: An Experimental Study. Loukas Georgiadis, Giuseppe F. Italiano, and Evangelos Kosinas. In European Symposium on Algorithms (ESA), 2022.

6. Computing the 4-Edge-Connected Components of a Graph in Linear Time. Loukas Georgiadis, Giuseppe F. Italiano, and Evangelos Kosinas. In European Symposium on Algorithms (ESA), 2021.

7. Computing Vertex-Edge Cut-Pairs and 2-Edge Cuts in Practice. Loukas Georgiadis, Konstantinos Giannis, Giuseppe F. Italiano, and Evangelos Kosinas. In International Symposium on Experimental Algorithms (SEA), 2021.

8. Linear-Time Algorithms for Computing Twinless Strong Articulation Points and Related Problems. Loukas Georgiadis and Evangelos Kosinas. In International Symposium on Algorithms and Computation (ISAAC), 2020.

Short Biography

In 2017 I graduated with a B.Sc. from the Dept. of Mathematics of the University of Ioannina. In 2021 I received an M.Sc. from the Dept. of Computer Science and Engineering of the University of Ioannina. I am very interested in the design of efficient algorithms and data structures for various graph-related problems.