Algorithm Design and Engineering for Graph Connectivity Problems

A Dissertation

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

by

Dionysios Kefallinos

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Ioannina
School of Engineering
Ioannina 2024

Dedication

.

To my beloved family.

Acknowledgements

Firstly I would like to thank my family for the support that offered my at any level of my studies and the opportunity I had to follow my dream.

Then I would like to express my deep gratitude to my supervisor Loukas Georgiadis Associate Professor of the Department of Computer Science and Engineering of the University of Ioannina, first for the trust he showed me from the beginning when we started this journey and second and foremost for the perfect guidance, help and patience I received from him through all the years we worked together in order to accomplish this dissertation.

Finally I would like to thank my colleagues especially Anna and Eirini for the beautiful moments and the good cooperation we had.

Table of Contents

List of Figures

List of Tables

List of Algorithms

ABSTRACT

Dionysios Kefallinos, Ph.D., Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, 2024.
Algorithm Design and Engineering for Graph Connectivity Problems.
Advisor: Loukas Georgiadis, Associate Professor.

This dissertation deals with topics related to some notions of graph connectivity in directed graphs and mixed graphs. The edge and vertex connectivity of a graph are important measures of its resilience as a network. There is a great variety of real-life relations that can be modeled by graphs. A few of the most notable examples include road networks, the World Wide Web, social networks, and other communication networks. Such a graph may consist of millions of edges and vertices as for example road or social networks. Therefore, we are interested both in asymptotically fast algorithms and in algorithms that are efficient in practice. In this dissertation, we present efficient implementations and empirical studies of important algorithms for basic connectivity problems in directed graphs. Moreover, we propose efficient methods of speeding up these algorithms for practical instances, which improved their execution times by one to two orders of magnitude. Finally, we provide new linear-time algorithms for related connectivity problems in mixed graphs.

Firstly we provide an experimental study of algorithms for computing the *edge connectivity* λ of a directed graph efficiently in practice. The *edge connectivity* λ of G is the minimum number of edges whose deletion leaves G not strongly connected. A graph G is *strongly connected* if there is a directed path from each vertex to any other vertex. Computing the edge connectivity of a graph is a classical subject in graph theory, and is an important notion in several application areas, such as transportation, communication, production, scheduling, and power engineering. We presented

the first efficient implementations of Gabow's algorithm [1] which runs in $O(\lambda m \log n)$ time and is based on matroid intersection and packing spanning trees, as well as algorithms based on recent "local search" algorithms for minimum-cut which have been proposed by Chechik et al. [2] and Forster et al. [3], of complexity $O(\lambda^2 m \,\text{polylog} n)$. Also, we proposed practical methods for speeding up these algorithms. In the experimental study, the efficiency of the various algorithms was compared on real graphs, taken from various application areas, as well as on synthetic graphs, aiming to highlight the advantages and disadvantages of each technique.

Then we studied the problem of packing arborescences. An s-arborescence of G is a directed spanning tree, i.e., an acyclic subgraph of G where s has in-degree zero and every other vertex has in-degree one. The problem of packing arborescences is to compute a maximum cardinality set of edge-disjoint arborescences of a directed graph. We explored the design space of efficient algorithms for packing arborescences of a directed graph in practice and we conducted a thorough empirical study to highlight the merits and weaknesses of each technique. In particular, we presented efficient implementations of Gabow's arborescence packing algorithm [1] of complexity $O(k^2n^2)$, Tarjan's algorithm [4] of complexity $O(k^2m^2)$, and the algorithm of Tong and Lowler [5] of complexity $O(k^2mn)$. In addition, we proposed a simple but very efficient method that significantly improves the running time of Gabow's algorithm in practice. In our experimental study, the performance of the various algorithms was compared on real-world taken from a wide range of applications, as well as on synthetic graphs.

Finally, we considered some orientation problems in mixed graphs related to the concept of 2-edge strong connectivity. A mixed graph G is a graph that consists of both undirected and directed edges. An orientation of G is formed by orienting all the undirected edges of G, i.e., converting each undirected edge $\{u, v\}$ into a directed edge that is either (u, v) or (v, u). Many mixed or undirected graph orientation problems have been studied in the literature, depending on the properties we wish a particular orientation to have. An orientation R of G is a strong orientation when the resulting graph is strongly connected. More generally, an orientation R of G is a k-strong orientation when the resulting graph is k-edge strongly connected. These concepts have applications in areas such as the design of road and telecommunications networks, and the structural stability of buildings. Our main result was to provide a linear time algorithm for the following problem. Given a mixed graph G, we wish to compute its maximal sets of vertices $C_1, C_2, \ldots, C_k$ with the property that by removing any edge e from G (directed or undirected) there is an orientation R_i of $G \backslash e$ such that all vertices of C_i are strongly connected in R_i. This problem is related to 2-edge strong connectivity, since if G contains only directed edges, then every set C_i corresponds to a 2-edge strong component of G.

Εκτεταμένη Περίληψη

Διονύσιος Κεφαλληνός, Δ.Δ., Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, 2024.
Σχεδίαση και υλοποίηση αλγορίθμων για προβλήματα συνεκτικότητας γραφημάτων.
Επιβλέπων: Λουκάς Γεωργιάδης, Αναπληρωτής Καθηγητής.

Η παρούσα διατριβή ασχολείται με διάφορα προβλήματα που αφορούν τη συνεκτικότητα γραφημάτων. Η συνεκτικότητα ακμών ή κορυφών ενός γραφήματος είναι σημαντικές μετρικές που μας περιγράφουν πόσο ”καλά” συνδεδεμένο είναι το εν λόγω γραφήμα. Διαισθητικά όσο μεγαλύτερη είναι η συνεκτικότητα τόσο πιο πυκνό σε συνδέσεις (ακμές) είναι το γράφημα, κατα συνέπεια είναι πιο δύσκολο να το διασπάσουμε ή να το καταστρέψουμε. Τα γραφήματα μοντελοποιούν δίκτυα, τα οποία ενδεχομένως να αποτελούνται από χιλιάδες ή και εκατομμύρια κόμβους ή ακμές όπως για παράδειγμα τα κοινωνικά δίκτυα, τα οδικά δίκτυα, τα βιολογικά δίκτυα κ.α.. Επομένως χρειαζόμαστε αποδοτικούς αλγορίθμους για να επιλύσουμε προβλήματα που δέχονται ως είσοδο γραφήματα τέτοιου μεγέθους, τόσο ασυμπτωτικά όσο και στη πράξη. Σε αυτή τη διατριβή, παρουσιάζουμε αποδοτικές υλοποιήσεις και εμπειρικές μελέτες σημαντικών αλγορίθμων για βασικά προβλήματα συνεκτικότητας σε κατευθυνόμενα γραφήματα. Επιπλέον, προτείνουμε αποτελεσματικές μεθόδους επιτάχυνσης αυτών των αλγορίθμων, οι οποίες βελτίωσαν τους χρόνους εκτέλεσης κατά μία έως δύο τάξεις μεγέθους. Τέλος, προτείνουμε νέους αλγορίθμους γραμμικού χρόνου για σχετικά προβλήματα συνδεσιμότητας σε μικτά γραφήματα.

Ένα κατευθυνόμενο γράφημα G είναι ισχυρά συνεκτικό αν μεταξύ δύο οποιονδήποτε κόμβων x και y, υπάρχει κατευθυνόμενο μονοπάτι από τον κόμβο x προς τον y και κατευθυνόμενο μονοπάτι από τον κόμβο y προς τον x. Η συνεκτικότητα ακμών $\lambda(G)$ ενός κατευθυνόμενου γραφήματος G ορίζεται ως το ελάχιστο πλήθος ακμών που πρέπει να αφαιρεθούν από το G, έτσι ώστε το γράφημα που προκύ-

πτει να μην είναι ισχυρά συνεκτικό. Το G είναι k-ισχυρά συνεκτικό όταν $\lambda(G) \geq k$. Ο υπολογισμός της συνεκτικότητας ακμών (edge-connectivity) ενός γραφήματος είναι ένα θεμελιώδες πρόβλημα της αλγοριθμικής θεωρίας γραφημάτων, με πληθώρα εφαρμογών σε τομείς όπως στην ανάλυση μεταφορικών, επικοινωνιακών, ηλεκτρικών και κοινωνικών δικτύων, σε συστήματα παραγωγής, και στη χρονοδρομολόγηση εργασιών. Η εργασία μας [6] είχε ως σκοπό τη μελέτη των τεχνικών σχεδίασης αποδοτικών αλγορίθμων για τον υπολογισμό της συνεκτικότητας ακμών σε κατευθυνόμενα γραφήματα, εστιάζοντας τόσο στη θεωρητική όσο και στην πρακτική επίδοση των σχετικών αλγορίθμων. Στο πλαίσιο αυτό, μελετήθηκαν και υλοποιήθηκαν αποδοτικοί αλγόριθμοι που υπολογίζουν τη συνεκτικότητα ακμών σε κατευθυνόμενα γραφήματα και πραγματοποιήθηκε μια ενδελεχής πειραματική ανάλυση των επιδόσεών τους. Συγκεκριμένα, στην εργασία μας [6] παρουσιάστηκε η πρώτη αποδοτική υλοποίηση του αλγορίθμου του Gabow [1], πολυπλοκότητας $O(\lambda m \log n)$ για γράφημα συνεκτικότητας λ με n κόμβους και m ακμές, ο οποίος βασίζεται στη θεωρία μητροειδών (matroid theory) και στον υπολογισμό ανεξάρτητων συνδετικών δένδρων. Επίσης, δόθηκαν οι πρώτες αποδοτικές υλοποιήσεις αλγορίθμων που βασίζονται σε πρόσφατες τεχνικές «τοπικής αναζήτησης» (local search) για την εύρεση ελάχιστων τομών, οι οποίες έχουν προταθεί από τους Chechik et al. [2] και Forster et al. [7], πολυπλοκότητας $O(\lambda^2 m \, \mathrm{polylog} n)$. Επιπροσθέτως, προτάθηκαν πρακτικές μέθοδοι επιτάχυνσης αυτών των αλγορίθμων. Στην πειραματική μελέτη συγκρίθηκε η αποδοτικότητα των διαφόρων αλγορίθμων σε πραγματικά γραφήματα, τα οποία έχουν ληφθεί από ποικίλα πεδία εφαρμογών, καθώς και σε συνθετικά γραφήματα, αποσκοπώντας στο να αναδειχθούν τα πλεονεκτήματα και μειονεκτήματα κάθε τεχνικής. Τα πειραματικά δεδομένα έδειξαν ότι ο αλγόριθμος του Gabow επιτυγχάνει συνολικά την καλύτερη απόδοση στην πράξη. Επιπλέον, παρατηρήθηκε ότι μια μέθοδος επιτάχυνσής που προτάθηκε στην εργασία [6], η οποία βασίζεται στη γρήγορη αρχικοποίηση των συνδετικών δένδρων που χρησιμοποιεί ο αλγόριθμος του Gabow μέσω καθοδικής διερεύνησης, επιτυγχάνει αξιοσημείωτη βελτίωση του χρόνου εκτέλεσης του αλγορίθμου του Gabow. Ένα ακόμα σημαντικό εύρημα της πειραματικής μελέτης της εργασίας [6] ήταν ότι οι αλγόριθμοι τοπικής αναζήτησης αποδίδουν καλά στην πράξη όταν η συνεκτικότητα λ του γραφήματος είναι μικρή, αλλά δεν μπορούν να ανταγωνιστούν τον αλγόριθμο του Gabow για μεγαλύτερες τιμές του λ.

Σε συνέχεια της προηγούμενης εργασίας, μελετήσαμε το πρόβλημα υπολογισμού ενός συνόλου ανεξάρτητων συνδετικών δένδρων μέγιστου πλήθους σε κατευθυνόμενα γραφήματα. Έστω G ένα κατευθυνόμενο γράφημα με αφετηριακό κόμβο s. Ένα συνδετικό δένδρο T του G, με αφετηρία τον κόμβο s, είναι ένα άκυκλο υπογράφημα του G, με την εξής ιδιότητα. Για κάθε κόμβο $v \neq s$, το T περιέχει μια μοναδική διαδρομή από τον s προς τον v. Αυτό συνεπάγεται ότι ο s δεν έχει καμία εισερχόμενη ακμή, και συγχρόνως κάθε κόμβος $v \neq s$ έχει μια μοναδική εισερχόμενη ακμή. Δύο συνδετικά δένδρα είναι ανεξάρτητα (edge-disjoint) όταν δεν έχουν καμιά ακμή κοινή. Σε ένα σύνολο ανεξάρτητων συνδετικών δένδρων $T = \{T_1, T_2, \ldots, T_k\}$, οποιαδήποτε δύο συνδετικά δένδρα T_i, $T_j \in T$ είναι μεταξύ τους ανεξάρτητα. Ένα κλασικό αποτέλεσμα του Edmonds [8] δηλώνει ότι το μέγιστο πλήθος ανεξάρτητων συνδετικών δένδρων ενός κατευθυνόμενου γραφήματος G, με αφετηριακό κόμβο s, ισούται με την ελάχιστη τιμή $k = c_G(s)$ μιας s-τομής του G. Μια s-τομή του G είναι ένα σύνολο ακμών C, τέτοιο ώστε το $G \setminus C$ να μην περιέχει μονοπάτι από τον s προς κάποιο κόμβο v. Αυτή η έννοια σχετίζεται με τη συνεκτικότητα ακμών $\lambda(G)$ ενός ισχυρά συνεκτικού κατευθυνόμενου γραφήματος G, καθώς ισχύει $\lambda(G) = \min\{c_G(s), c_{G^R}(s)\}$, όπου G^R το αντίστροφο γράφημα του G. Στο πλαίσιο της εργασίας μας [9], μελετήθηκαν και υλοποιήθηκαν αποδοτικοί αλγόριθμοι για τον υπολογισμό ενός μέγιστου συνόλου ανεξάρτητων συνδετικών δένδρων ενός κατευθυνόμενου γραφήματος. Οι αλγόριθμοι που υλοποιήθηκαν και συμπεριλήφθηκαν στην πειραματική μελέτη της εργασίας [9] είναι ο αλγόριθμος του Gabow [6] πολυπλοκότητας $O(k^2n^2)$, ο αλγόριθμος του Tarjan [4] πολυπλοκότητας $O(k^2m^2)$, και ο αλγόριθμος των Tong και Lowler [5] πολυπλοκότητας $O(k^2mn)$. Επιπροσθέτως, προτάθηκαν μέθοδοι επιτάχυνσης αυτών των αλγορίθμων στην πράξη. Στην πειραματική μελέτη της εργασίας [9], συγκρίθηκε η αποδοτικότητα των διαφόρων αλγορίθμων σε γραφήματα που προέρχονται από τον πραγματικό κόσμο και από ένα ευρύ φάσμα εφαρμογών, καθώς και σε συνθετικά γραφήματα. Τα πειραματικά αποτελέσματα ανέδειξαν τα πλεονεκτήματα και μειονεκτήματα κάθε τεχνικής. Συγκεκριμένα, παρατηρήθηκε ότι ο αλγόριθμος του Gabow επιτυγχάνει συνολικά την καλύτερη απόδοση, καθώς είναι σημαντικά ταχύτερος από τους άλλους δύο αλγόριθμους, ενώ δεύτερος σε απόδοση έρχεται ο αλγόριθμος του Tarjan. Επίσης, παρατηρήθηκε ότι μια απλή αλλά πολύ αποτελεσματική ευρετική τεχνική που προτάθηκε στην εργασία [9], επιτάχυνε τον χρόνο εκτέλεσης του αλγορίθμου του Gabow κατά δύο τάξεις μεγέθους στις περισσότερες περιπτώσεις.

Τέλος, στην περιοχή της συνεκτικότητας μεικτών γραφημάτων, μελετήθηκαν προβλήματα που σχετίζονται με την έννοια της 2-ισχυρής συνεκτικότητας ακμών (2-edge strong connectivity). Ένα μεικτό γράφημα (mixed graph) G περιέχει τόσο κατευθυνόμενες ακμές όσο και μη κατευθυνόμενες ακμές. Ένας προσανατολισμός (orientation) R του μεικτού γραφήματος G πραγματοποιείται όταν δώσουμε κατευθύνσεις σε όλες τις μη κατευθυνόμενες ακμές. Δηλαδή, μετατρέπουμε κάθε μη κατευθυνόμενη ακμή u, v σε κατευθυνόμενη, η οποία είναι είτε η ακμή (u, v) είτε η ακμή (v, u). Στη βιβλιογραφία έχουν μελετηθεί πολλά προβλήματα προσανατολισμού μεικτών ή μη κατευθυνόμενων γραφημάτων, ανάλογα με τις ιδιότητες που επιθυμούμε να έχει ένας συγκεκριμένος προσανατολισμός. Ένας προσανατολισμός R του G είναι ισχυρός (strong orientation) όταν το γράφημα που προκύπτει είναι ισχυρά συνεκτικό. Πιο γενικά, ένας προσανατολισμός R του G είναι k-ισχυρός (k-strong orientation) όταν το γράφημα που προκύπτει είναι k-ισχυρά συνεκτικό. Αυτές οι έννοιες έχουν εφαρμογές σε τομείς όπως ο σχεδιασμός οδικών και τηλεπικοινωνιακών δικτύων. Σε ένα μεικτό γράφημα G, ένα σύνολο κόμβων $S \subseteq V(G)$ είναι ανθεκτικό (resilient) ως προς την ισχυρή συνεκτικότητα αν έπειτα από οποιαδήποτε αφαίρεση ακμής του γραφήματος, υπάρχει προσανατολισμός των μη κατευθυνόμενων ακμών ώστε το σύνολο S να παραμείνει ισχυρά συνεκτικό. Στο πλαίσιο της διδακτορικής διατριβής, μελετήσαμε το πρόβλημα υπολογισμού των ανθεκτικών συνιστωσών ενός μεικτού γραφήματος G, δηλαδή των μέγιστων συνόλων κόμβων $C_1, C_2, \ldots, C_k$, όπου κάθε C_i είναι ανθεκτικό. Δηλαδή, για κάθε ακμή e του G, υπάρχει προσανατολισμός R_i του G τέτοιος ώστε όλοι οι κόμβοι του C_i να είναι ισχυρά συνεκτικοί. Το πρόβλημα αυτό σχετίζεται με τη 2-ισχυρή συνεκτικότητα ακμών, καθώς αν το G περιέχει μόνο κατευθυνόμενες ακμές, τότε κάθε ανθεκτική συνιστώσα είναι 2-ισχυρά συνεκτική. Το κύριο αποτέλεσμα στην εργασία μας [10] ήταν ένας αλγόριθμος γραμμικού χρόνου για τον υπολογισμό των ανθεκτικών συνιστωσών ενός μεικτού γραφήματος. Ο αλγόριθμος αυτός βασίζεται κυρίως σε δύο βασικές τεχνικές: (1) στην κατασκευή μιας συλλογής $\mathcal{H}$ βοηθητικών γραφημάτων που διατηρεί τις ανθεκτικές συνιστώσες του G, και (2) σε μια αναγωγή στο πρόβλημα υπολογισμού των συνεκτικών συνιστωσών ενός μη κατευθυνόμενου γραφήματος μετά τη διαγραφή συγκεκριμένων ζευγών στοιχείων του γραφήματος που αποτελούνται από ένα κόμβο και μια ακμή. Για την επίλυση του δεύτερου προβλήματος, η εργασία [10] εκμεταλλεύεται τις ιδιότητες των SPQR-δένδρων των δισυνεκτικών συνιστωσών κάθε γραφήματος $H \in \mathcal{H}$.

Chapter 1

Introduction

Graphs are mathematical structures that model many diverse natural or man-made systems, which capture both the structure and the dynamics of the underlying system. Examples include but are not limited to the world-wide web, transportation, communication and social networks, databases, biological systems, circuits, and the control-flow of computer programs. A graph consists of vertices (also called nodes) which are connected by edges (also called links). A distinction is made between undirected graphs, where edges indicate a two-way relationship, in that each edge can be traversed in both directions or directed graphs (digraphs) where each edge has an orientation and can be traversed in one direction. Mixed graphs are the graphs that contain both directed and undirected edges. In this dissertation we study connectivity problems in directed and mixed graphs, from the perspective of algorithm design, analysis and engineering. Specifically, the studied topics are the computation of the edge-connectivity in directed graphs, the computation of a maximum packing of edge-disjoint arborescences in directed graphs, as well as the 2-edge strong connectivity in mixed graphs.

In more detail, we present carefully engineered implementations and empirical studies of important algorithms for basic connectivity problems in directed graphs. Also, we propose efficient methods of speeding up these algorithms for practical instances, which improve their execution times by one to two orders of magnitude. Finally, we provide new linear-time algorithms for related connectivity problems in mixed graphs. Our experimental studies are motivated from the fact that despite the significant advances that have been made in the design and analysis of algorithms in recent years, many such theoretical results have received very little attention from the practical perspective [11]. As a result, the practical potential of the corresponding algorithms and their underlying techniques is far from understood.

1.1 Graph connectivity

Let $G = (V, E)$ be a directed graph **(digraph)**, with m edges and n vertices. Digraph G is *strongly connected* if there is a directed path from each vertex to every other vertex. The *strongly connected components* of G are its maximal strongly connected subgraphs. Two vertices $u, v \in V$ are *strongly connected* if they belong to the same strongly connected component of G. The *size* of a strongly connected component is given by its number of vertices. An edge (resp., a vertex) of G is a *strong bridge* (resp., a *strong articulation point*) if its removal increases the number of strongly connected components. Note that strong bridges (resp., strong articulation points) are 1-edge (resp., 1-vertex) cuts for digraphs. More generally, a set of edges $C \subseteq E$ is a *cut* if $G \setminus C$ is not strongly connected. If $|C|= k$ then we refer to C as a k-cut of G.

A digraph G is k-edge-connected if it has no $(k-1)$-cuts. We say that two vertices v and w are k-edge-connected, denoted by $v \leftrightarrow_k w$, if there are k edge-disjoint directed paths from v to w and k edge-disjoint directed paths from w to v. (Note that a path from v to w and a path from w to v may not be edge-disjoint). By Menger's theorem [12], v and w are k-edge-connected if and only if the removal of any set of at most $k - 1$ edges leaves v and w in the same strongly connected component. We define a *k-edge-connected component* of a digraph $G = (V, E)$ as a maximal subset $U \subseteq V$ such that u and v are k-edge-connected for all $u, v \in U$. The k-edge-connected components of G form a partition of V, since $v \leftrightarrow_k w$ is an equivalence relation [13]. Graph connectivity is a fundamental concept in graph theory, and is defined as the minimum number of elements (vertices or edges) that need to be removed to separate the remaining vertices into two or more connected components. More precisely, we consider the following two measures of connectivity:

- The *vertex-connectivity* of a graph is the minimum number of vertices that have to be removed to leave the graph not strongly connected.
- The *edge-connectivity* of a graph is the minimum number of edges that have to be removed to leave the graph not strongly connected.

Graph connectivity is a classical subject of graph theory and has been studied extensively. Many algorithms for the computation of *edge–connectivity* and *vertex–connectivity* of undirectd and directed graphs have been developed over the years. Many of these algorithms work by solving a number of max–flow problems.

Flows and cuts. Let $G = (V, E)$ be a digraph, where each edge e has a nonnegative capacity $c(e)$. For a pair of vertices s and t, an s-t flow of G is a function f on E such that $0 \leq f(e) \leq c(e)$, and for every vertex $v \neq s, t$ the incoming flow is equal to outgoing flow, i.e., $\sum_{(u,v)\in E} f(u, v) = \sum_{(v,u)\in E} f(v, u)$. The value of the flow is defined as $|f|= \sum_{(s,v)\in A} f(s, v)$.

An s-t cut is a set of edges C such that there is no path from s to t in $G - C$. We refer to a minimum cardinality s-t cut as an s-t *min-cut*. We can compute an s-t min-cut via a well-known reduction to maximum-flow. By Ford and Fulkerson [14], we have that the cardinality of the s-t min-cut is equal to the maximum s-t flow in $G = (V, E)$, where in our case the edges have unit edge capacities ($c(e) = 1$ for all $e \in E$). Let $G(V, E)$ be a directed graph. An s-*cut* is the set of edges directed from S to $V \setminus S$, where S is any vertex set that contains vertex s such that $S \subset V$. Then, the *edge connectivity* λ of a graph G is the minimum cardinality of an s-cut of G or G^R. This observation implies the following straightforward approach to compute the edge connectivity of G: Compute the minimum cardinality s-t cut in G and the minimum cardinality s-t cut in G^R, for all $t \in V - s$. This can be done by executing $2(n-1)$ maximum-flow computations. Assuming that the graph is simple, i.e., without parallel edges, the value $f(s, t)$ of any maximum s-t flow is bounded by n, so we can compute $f(s, t)$ in $O(mn)$ time by running at most n iterations of the Ford-Fulkerson method [14]. This gives an $O(mn^2)$ time bound for computing the edge connectivity.

Even and Tarjan [15] provided an $O(m^{3/2}n)$ bound for computing λ via maximum-flow computations, by proving that Dinic's algorithm [16] computes the maximum-flow in a digraph G with unit edge capacities in $O(m^{3/2})$ time. Even and Tarjan also proved that Dinic's algorithm runs in $O(n^{2/3}m)$ time if G is simple, from which they obtained an $O(n^{5/3}m)$ time bound for computing the edge connectivity of a simple digraph. Schnorr [17] showed how to compute the edge connectivity λ of a digraph with n calls to a maximum-flow algorithm and presented an algorithm that computes λ in $O(\lambda mn)$ time. Mansour and Schieber [18] achieved an $O(\min\{mn, \lambda^2 n^2\})$ time bound by exploiting a relation between minimum cuts and dominating sets. Their algorithm also uses maximum-flow computations, but chooses the source and sink vertices based on the notion of out- and in-dominating sets (defined in Section 2.2).

Currently, the state of the art algorithm for computing the edge-connectivity λ of a digraph is the algorithm of Gabow [1] which runs in $O(\lambda m \log n)$ time. Gabow's algorithm is inspired by matroid intersection and is based on the idea of packing spanning trees.

In this Chapter we review the necessary notation and the basic definitions that is used through this thesis. Moreover we describe some basic tools that are used such as dfs traversal.

1.2 Notation

Let $G = (V, E)$ be a directed graph (digraph) with vertex set $V = V(G)$ and edge set $E = E(G)$. We denote the number of vertices and edges by n and m, respectively. The *reverse digraph* of G, denoted by $G^R = (V, E^R)$, is the digraph that results from G by reversing the direction of all edges. We denote by $V \setminus S$, the graph obtained after deleting a set S of vertices from V. A spanning tree T of an undirected graph G is a connected subgraph of G which includes all of the vertices of G and has no cycle.

Let $G = (V, E)$ be a strongly connected directed graph. Let $S \subseteq V$. The out-degree (resp., in-degree) of S, denoted by $\delta^+(S)$ (resp., $\delta^-(S)$), is the number of edges directed from S to $V \setminus S$ (resp., from $V \setminus S$ to S). For a vertex $v \in V$, $\delta^+(v)$ (resp., $\delta^-(v)$) denotes its out-degree (resp., in-degree). We let $\delta = \min_{v \in V}\{\delta^+(v), \delta^-(v)\}$ denote the minimum degree of the graph. For a subgraph H of G, we let $\delta^+_H(v)$ and $\delta^-_H(v)$, respectively, denote the out-degree and the in-degree of v in H. We let $E^-(v) = \{(u, v) \in E\}$, i.e., the set of edges directed to v. The out-volume (resp., in-volume) of S, denoted by $vol^{+}(S)$ (resp., $vol^{-}(S)$), is the number of edges leaving (resp., entering) the vertices of S, i.e., $vol^{+}(S) = \sum_{v \in S} \delta^{+}(v)$ (resp., $vol^{-}(S) = \sum_{v \in S} \delta^{-}(v)$).

1.3 Basic Tools

1.3.1 Trees

In graph theory, a tree is an undirected graph in which any two vertices are connected by exactly one path, or equivalently a connected acyclic undirected graph. A forest is an undirected graph in which any two vertices are connected by at most one path or equivalently a disjoint union of trees.

A spanning tree T of a graph G is a subgraph that is a tree which includes all of the vertices of G. In general, a graph may have several spanning trees, up to n^{n-2} spanning trees according to Cayley's formula.

A rooted tree is a tree with a designated root vertex. A rooted tree may be directed, called a directed rooted tree, either making all its edges point away from the root in which case it is called an arborescence or out-tree or making all its edges point towards the root in which case it is called an anti-arborescence or in-tree. A rooted forest is a disjoint union of rooted trees.

Throughout the thesis, we assume that the edges of a tree T are directed away from the root. For each directed edge (u, v) in T, we say that u is a parent of v (and we denote it by $t(v)$) and that v is a child of u. Every vertex except the root has a unique parent. If there is a (directed) path from vertex v to vertex w in T, we say that v is an ancestor of w and that w is a descendant of v. Also, for a rooted tree T, we let $T(v)$ denote the subtree of T rooted at v, and we also view $T(v)$ as the set of descendants of v.

1.3.2 Depth-first Search

A depth-first search (in short DFS) is a widely used traversal algorithm that starts from a root vertex and visits all adjacent vertices of the most recently visited vertex through its outgoing edges, When there are no unexplored outgoing edges from the current vertex, the algorithm backtracks to the previously visited vertex. Thus, the

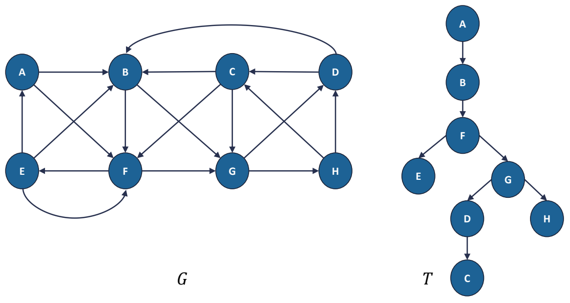

Figure 1.1: A directed Graph G and a tree T constructed by a DFS traversal of G.

algorithm explores the graph as far as possible along each branch (forming at any given moment an active path) before backtracking. A stack data structure can be used to keep track of the vertices discovered so far along the active path, which helps in backtracking of the graph. The edges of the graph that form the active paths during the DFS generate a tree, which is called depth-first search tree of the graph. There is no unique DFS tree for a given graph and starting vertex as the tree depends on the order that the outgoing edges of each vertex are visited. Let T be a DFS tree of a digraph G, starting from a given vertex s. A preorder of T is a total order of the vertices of T such that, for every vertex v, the descendants of v are ordered consecutively, with v first. It can be obtained by a depth-first traversal of T, by ordering the vertices in the order they are first visited by the traversal.

1.3.3 Breadth-first Search

Breadth-first search (BFS) is another common graph traversal algorithm. It starts from a root vertex and explores all the vertices at the current depth prior moving on to the vertices at the next depth level. During the execution of a BFS, a queue data structure can be used to keep track of the child vertices that were encountered but not yet explored.

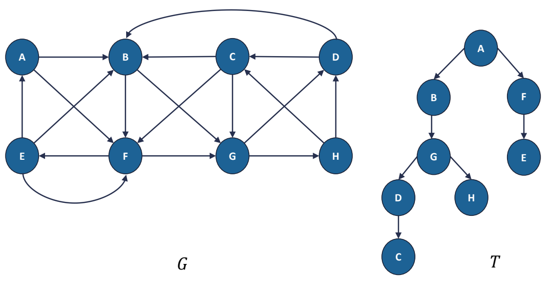

Figure 1.2: A directed Graph G and a tree T of G constructed by a BFS traversal of G.

1.4 Dissertation outline

Dissertation contains 5 chapters. Chapter 2 involves an experimental study of algorithms for computing the *edge-connectivity* of a directed graph in practice. Computing the *edge-connectivity* of a graph is a classical subject in graph theory, and is an important notion in several application areas, such as transportation,communication, production, scheduling, and power engineering. We discuss how to achieve efficient implementations of some important algorithms that have been proposed in the related literature, including the state-of-the-art algorithm of Gabow [1], Furthermore, we introduce a new practical heuristic of the algorithm of Gabow [1] that significantly improves its performance. We present a thorough empirical analysis of these algorithms using real data from application areas, as well as some artificial data.

In Chapter 3, we address the question of how efficiently we can compute a maximum packing of edge-disjoint arborescences in practice, compared to the time required to determine the *edge-connectivity* of a graph. To that end, we explore the design space of efficient algorithms for packing arborescences of a directed graph in practice and we conduct a thorough empirical study to highlight the merits and weaknesses of each technique. In particular, we present an efficient implementation of Gabow's arborescence packing algorithm [1] and provide a simple but efficient heuristic that significantly improves its running time in practice.

Next, in Chapter 4, motivated by recent work in 2-edge strong connectivity in digraphs [13, 19, 20], we introduce the following strong connectivity orientation problem in mixed graphs. Given a mixed graph G, we wish to compute its maximal sets of vertices $C_1, C_2, \dots, C_k$ with the property that for every $i \in \{1, \dots, k\}$, and every edge e of G (directed or undirected), there is an orientation R_i of $G \setminus e$ such that all vertices of C_i are strongly connected in R_i. We refer to these maximal vertex sets as the *edge-resilient strongly orientable blocks* of G. These concepts are motivated by several diverse applications, such as the design of road and telecommunication networks, and the structural stability of buildings.

Lastly in Chapter 5, we discuss possible future extensions and some open related problems that could be studied on the horizon of further research.

Chapter 2

Computing the edge-connectivity of a directed graph

A directed graph G is *strongly connected* if there is a directed path from each vertex to every other vertex. The *edge connectivity* λ of a graph G is the minimum number of edges whose deletion leaves a digraph that is not strongly connected. In this chapter, we provide efficient implementations of algorithms for computing the *edge-connetivity* of a directed graph and we present a thorough experimental analysis of these algorithms. We also suggest a heuristic method that improves the performance of the algorithm of Gabow [1] significantly.

2.1 Techniques and Related work

Let $G = (V, E)$ be a directed graph (digraph), with m edges and n vertices. Digraph G is *strongly connected* if there is a directed path from each vertex to every other vertex. The *edge connectivity* λ of G is the minimum number of edges whose deletion leaves a digraph that is not strongly connected. The *reverse digraph* of G, denoted by $G^R = (V, E^R)$, is the digraph that results from G by reversing the direction of all edges. An *s-cut* is the set of edges directed from S to $V \setminus S$, where S is any vertex set that contains s such that $S \subset V$. Then, λ is the minimum cardinality of an s-cut of G or G^R. This observation implies the following straightforward approach to compute the edge connectivity of G: Compute the minimum cardinality s-t cut in G and the minimum cardinality s-t cut in G^R, for all $t \in V - s$. This can be done by executing $2(n-1)$ maximum-flow computations. Assuming that the graph is simple, i.e., without parallel edges, the value $f(s,t)$ of any maximum s-t flow is bounded by n, so we can compute $f(s,t)$ in $O(mn)$ time by running at most n iterations of the Ford-Fulkerson method [14]. This gives an $O(mn^2)$ time bound for computing the edge connectivity. Even and Tarjan [15] provided an $O(m^{3/2}n)$ bound for computing λ via maximum-flow computations, by proving that Dinic's algorithm [16] computes the maximum-flow in a digraph G with unit edge capacities in $O(m^{3/2})$ time. Even and Tarjan also proved that Dinic's algorithm runs in $O(n^{2/3}m)$ time if G is simple, from which they obtained an $O(n^{5/3}m)$ time bound for computing the edge connectivity of a simple digraph. Schnorr [17] showed how to compute the edge connectivity λ of a digraph with n calls to a maximum-flow algorithm and presented an algorithm that computes λ in $O(\lambda mn)$ time. Mansour and Schieber [18] achieved an $O(\min\{mn, \lambda^2 n^2\})$ time bound by exploiting a relation between minimum cuts and dominating sets. Their algorithm also uses maximum-flow computations, but chooses the source and sink vertices based on the notion of out- and in-dominating sets (defined in Section 2.2).

Currently, the state of the art algorithm for computing the edge-connectivity λ of a digraph is the algorithm of Gabow [1] which runs in $O(\lambda m \log n)$ time. Gabow's algorithm is inspired by matroid intersection and is based on the idea of packing spanning trees. A *spanning tree* of an undirected connected graph G is a connected acyclic spanning subgraph of G. We extend this definition to directed graphs by ignoring the edge orientations. Let s be an arbitrary start vertex of a strongly connected digraph G. An *s-arborescence* of G is a directed spanning tree, i.e., an acyclic subgraph of G where s has in-degree zero and every other vertex has in-degree one. (Thus, any s-arborescence is a spanning tree of G but not vice versa.)

Edmonds [8] showed that the maximum number of edge-disjoint s-arborescences of G equals the minimum cardinality of an s-cut.

Edmonds [21] also proved the following *Matroid Characterization* of s-cuts: The edges of a directed graph can be partitioned into k s-arborescences if and only if they can be partitioned into k spanning trees and every vertex except s has in-degree k. (Referring to this result as the Matroid Characterization of minimum-cut is justified by the fact that the spanning trees with the above property are formed by the intersection of two matroids.)

Another approach towards computing the edge or vertex connectivity of a digraph is based on “local search” algorithms [2, 3, 7, 22] for identifying small cuts of bounded volume. This idea was introduced by Chechik et al. [2], who provided a local search procedure that identifies an s-cut of cardinality at most k, so that in the induced partition $(S, V \setminus S)$, where $s \in S$, the number of edges that originate from vertices in S is bounded by $O(k\Delta)$ for some parameter Δ. Based on this procedure, Chechik et al. provided faster algorithms for computing the maximal k-edge and k-vertex connected subgraphs of a directed graph. Simplified but randomized local search algorithms were presented by Nanongkai et al. [22] and Forster and Yang [7]. These algorithms improved the dependence on k in the running time of the local search procedure from $k^{O(k)}$ to k^2. Forster et al. [3] applied their improved local search procedure to provide a randomized algorithm that computes (with high probability) the vertex connectivity κ of a digraph in $\tilde{O}(\kappa \cdot \min\{\kappa m, \kappa^{1/2} m^{1/2} n + \kappa^2 n\})$ time.[1] A simplified version of their approach gives an $\tilde{O}(\lambda^2 m)$-time randomized algorithm to compute (with high probability) the edge connectivity λ of a digraph.

We note that the edge connectivity of a simple undirected graph can be computed in $\tilde{O}(m)$ time, randomized [23, 24] or deterministic [25, 26]. In particular, the deterministic algorithm of Kawarabayashi and Thorup [26], as well as its improvement by Henzinger et al. [25], apply Gabow’s algorithm on a contracted graph, for which the latter runs in $\tilde{O}(m)$ time.

2.2 Tested algorithms

In this Section we give an overview of the tested algorithms that are included in the experimental analysis for computing the *edge-connectivity* of a directed graph.

[1]The notation $\tilde{O}(\cdot)$ hides poly-logarithmic factors.

2.2.1 Maximum-flow based algorithms

Let $f(s,t)$ denote the value of the maximum s-t flow in a strongly connected directed graph $G=(V,E)$ (with unit edge capacities). The standard approach to compute the edge connectivity of G via maximum flow is to compute the minimum value of $f(s,t)$ and $f(t,s)$ for all $t \in V-s$. Since we assume that G is simple, we have $f(u,v) \leq n-1$ for any $u,v \in V$. Hence, we can compute each maximum flow value by applying at most $n-1$ iterations of the Ford-Fulkerson method, in $O(mn^2)$ total time. Even and Tarjan [15] improved this bound to $O(n^{5/3}m)$, by proving that Dinic's algorithm [16] computes the maximum-flow in a (simple) digraph with unit edge capacities in $O(n^{2/3}m)$ time. Schnorr [17] observed that it suffices to make just n call to a maximum flow algorithm: Let $V=\{v_1,v_2,\ldots,v_n\}$. Then $\lambda=\min\{f(v_1,v_2),f(v_2,v_3),\ldots,f(v_{n-1},v_n),f(v_n,v_1)\}$. He also presented an algorithm that computes λ in $O(\lambda mn)$ time.

Implementation details

In our implementations of the above maximum-flow based algorithms, we apply the following obvious practical improvement. When we compute $f(u,v)$, for some vertices u and v, we terminate the computation early if the current u-v flow exceeds $\min\{\delta, f_{\min}\}$, where $f_{\min}$ is the minimum of the flow values computed so far.

2.2.2 Mansour-Schieber algorithm

Mansour and Schieber [18] also apply a maximum-flow based approach. Their algorithm, however, achieves better time bounds by choosing carefully the source and sink vertices in the maximum flow computations, based on the notion of out- and in-dominating sets. They provide two versions of their algorithm, with running times $O(mn)$ and $O(\lambda^2n^2)$, respectively.

A set $O \subseteq V$ (resp., $I \subseteq V$) is an *outgoing dominating set* (resp., *ingoing dominating set*) if for every vertex $v \in V \setminus O$ (resp., $v \in V \setminus I$) there is a vertex $u \in O$ (resp., $u \in I$) such that $(u,v) \in E$ (resp., $(v,u) \in E$). Mansour and Schieber show that if G contains a cut $(S, V \setminus S)$ of cardinality less than δ^-, then for any outgoing dominating set O there is at least one vertex in $O \cap (V \setminus S)$. Similarly, if G contains a cut $(S, V \setminus S)$ of cardinality less than δ^+, then for any ingoing dominating set I there is at least one vertex in $I \cap S$. Suppose that we construct O and I so that both sets contain a vertex

v. The above result implies that if there is a cut $(S, V \setminus S)$ of cardinality less than δ in G, then there is a vertex $x \in O$ such that $\lambda = f(v, x)$, or a vertex $y \in I$ such that $\lambda = f(y, v)$.

The algorithm computes the edge connectivity in two phases. In the first phase it grows an outgoing dominating set $O = \{v = x_0, x_1, x_2, \ldots\}$ and computes the maximum number of edge-disjoint paths from $\{v = x_0, x_1, \ldots x_{i-1}\}$ to x_i when the next vertex x_i is added to O. Analogously, in the second phase it grows an ingoing dominating set $I = \{v = y_0, y_1, y_2, \ldots\}$ and computes the maximum number of edge-disjoint paths from y_i to $\{v = y_0, y_1, \ldots y_{i-1}\}$ when the next vertex y_i is added to I. Let k be the minimum over all these maxima. Then, $\lambda = \min\{k, \delta\}$. Mansour and Schieber show that this algorithm, that we refer to as Mansour-Schieber I, computes the edge connectivity in $O(mn)$ time.

A variant of the above algorithm, that we refer to as Mansour-Schieber II, works as follows. It tests if G is k-edge connected for exponentially increasing values of k, i.e., $k = 2^1, 2^2, \ldots$. To test if G is k-edge connected for $k \leq \delta$, it applies two phases similar to Mansour-Schieber I. During the first phase it grows an outgoing dominating set $O = \{v = x_0, x_1, x_2, \ldots\}$ and tests if G has k edge-disjoint paths from $\{v = x_0, x_1, \ldots x_{i-1}\}$ to x_i when the next vertex x_i is added to O. Analogously, during the second phase it grows an ingoing dominating set $I = \{v = y_0, y_1, y_2, \ldots\}$ and tests if G has k edge-disjoint paths from y_i to $\{v = y_0, y_1, \ldots y_{i-1}\}$ when the next vertex y_i is added to I. This process identifies a value i such that $2^{i-1} \leq \lambda < 2^i$. To complete the computation of λ, Mansour and Schieber test if G is k-edge connected for $k = \min\{2^i, \delta\}$, with the following modification. When we test if G has k edge-disjoint paths from $\{v = x_0, x_1, \ldots x_{i-1}\}$ to x_i, or k edge-disjoint paths from y_i to $\{v = y_0, y_1, \ldots y_{i-1}\}$, if we find that the maximum number of edge-disjoint paths is $k' < k$, then we record this value. If the minimum k' among these maxima is less than δ then $\lambda = k'$, otherwise $\lambda = \delta$. The running time of this algorithm is $O(\lambda^2 n^2)$.

Implementation details.

We store G using the same data structures as in the implementation of the maximum-flow based algorithms of Section 2.2.1. We also store the outgoing dominating set O explicitly, so that we can list all of its vertices in $O(|O|)$ time. Moreover, we maintain two boolean arrays, *Omark* and *Imark*, such that $v \in O$ if and only if $Omark[v] = 1$, and, similarly, $v \in I$ if and only if $Imark[v] = 1$. To search for a path from O to a vertex $x \in V \setminus O$ in the residual network, we execute a breadth-first search (BFS), where we initialize the BFS queue so that it contains all vertices in O. Similarly, to search for a path from a vertex $y \in V \setminus I$ to I, we execute a BFS from y until we reach a vertex v such that $Imark[v] = 1$.

2.2.3 Gabow's algorithm

Gabow [1] showed how to compute efficiently the edge connectivity of a digraph G by applying the Matroid Characterization of s-cuts. Gabow's algorithm computes a minimum s-cut by constructing a complete k-intersection T of G, i.e., a collection of k edge-disjoint spanning trees $T_1, \dots, T_k$, such that each vertex $v \neq s$ has in-degree k, s has in-degree zero, and k is as large as possible. In the k-th iteration, the algorithm starts with a complete $(k-1)$-intersection and tries to enlarge it so that it becomes a complete k-intersection. To that end, it executes a *round robin* algorithm that maintains a forest T_k and tries to locate "joining" edges that will make T_k a spanning tree of G, while satisfying the invariant that $\delta_T^-(s) = 0$ and $\delta_T^-(v) \leq k$ for all $v \neq s$. In the following, we call a vertex v *deficient* if $\delta_T^-(v) < k$.

In more detail, during the k-th iteration T_k is a forest of rooted trees, referred to as *f-trees*, where each f-tree F_z is rooted at its unique deficient vertex z. For $z \neq s$, $\delta_T^-(z) = k-1$. An edge $e = (x, y)$ is *joining* if x and y are in different f-trees of T_k. The round robin algorithm looks to enlarge T_k by one edge at a time, and simultaneously to increase the in-degree of a deficient vertex $z \neq s$ by one. To that end, it examines an edge e_1 in $E^-(z) \setminus T$. If e_1 is joining then we are done. Otherwise, it adds e_1 in some T_i and looks for a joining edge in the fundamental cycle $C(e, T_i)$. To break the cycle, we can remove from T_i an edge $e_2 \in C(e, T_i)$. Then, we can add $e_2 = (u, v)$ to some other tree of T, or replace e_2 with a edge in $E^-(v) \setminus T$. This pattern continues until a joining edge is found. The sequence of edges that leads to a joining edge is called an *augmenting path*. We define this notion formally below.

An ordered pair of edges e, f is called a *swap* if $f \in C(e, T_i)$ for some $T_i \in T$. To *execute the swap* is to replace f in T_i by e. A *partial augmenting path P from z* is a sequence of distinct edges $e_1, \dots, e_l$, such that:

(i) $e_1 \in E^-(z) \setminus T$.

(ii) For each $i < l$ either

(a) $e_{i+1} \in C(e_i, T_j)$, where T_j contains e_{i+1} but not e_i, or

(b) $e_i, e_{i+1} \in E^-(v)$, for some vertex v, where T contains e_i but not e_{i+1}.

(iii) Executing all swaps of P (i.e., the pairs e_i, e_{i+1} of (ii.a)) gives a new collection of forests.

An *augmenting path P from z* is a partial augmenting path from z whose last edge e_l is joining for z. To *augment T along P* is to execute each swap of P and add e_l to T_k. This increases the in-degree of z by one (so z is no longer deficient), while no other in-degree changes.

Each iteration is organized as a sequence of at most $\lceil \log n \rceil$ rounds. At the start of each round all f-trees are active except F_s. Then, we repeatedly choose an active f-tree F_z and search for an augmenting path from z. If no such path is found, then the algorithm terminates and reports an s-cut of cardinality $k-1$. Otherwise, we have found an augmenting path P from z and we augment T along P. Thus, we enlarge T by one edge and F_z is joined to another f-tree F_w. The resulting f-tree of T_k is rooted at w (since z in longer deficient), and becomes inactive for the rest of the current round. Gabow showed that with an appropriate implementation, each round runs in $O(m)$ time. Furthermore, he showed that by organizing the search for augmenting paths carefully, all augmentations can be executed at the end of a round.

Gabow [1] also presented the following slight variant of the round robin algorithm, which is more suitable for dense graphs. At the start of the r-th round only the f-trees with at most 2^r vertices are made active. Then, the r-th round is executed in $O(m_r + 2^r n)$ time, where m_r is the number of non-joining edges labelled during that round. This results to an overall $O(m \log(n^2/m))$ running time of the round robin algorithm. In our experiments, however, we did not consider this variant as our input graphs, which are derived from real-world applications, are sparse.

Finding an augmenting path.

To keep track of an augmented path, Gabow's algorithm stores with every examined edge e a label $l(e)$ that indicates the previous edge f in a potential augmented path, i.e., $l(e) = f$. The algorithm executes a scanning procedure that searches for an augmented path from z, during which it maintains a queue Q of non-joining edges that need to be examined. When we process the next edge of Q we consider the fundamental cycle $C(e, T_i)$ that e forms in a tree T_i (or forest if $i = k$) of T.

In the following we let variable i denote the index of the current tree T_i that the algorithm considers, and let ℓ be a variable that stores an edge label. Moreover, we let L_i denote the subtree of T_i that consists of z and every vertex on a labelled edge of T_i.

Initially, we set $Q \leftarrow \emptyset$, $i \leftarrow 1$, and $\ell \leftarrow \perp$. Then, we execute the *Label Step* for all edges $g \in E^{-}(z) \setminus T$.

Label Step. Assign $l(g) \leftarrow \ell$. If g is a joining edge then the search is successful and g is the last edge of P. Otherwise, add g to the end of Q.

If no joining edge is found in $E^{-}(z) \setminus T$, then we execute the following steps until the algorithm halts.

Next Edge Step. If Q is empty then the search is unsuccessful. Otherwise, delete the first edge e from Q. If $e \in T_i$ then set $i \leftarrow (i \mod k) + 1$.

Fundamental Cycle Step. If both endpoints of e are in L_i then continue with the *Next Edge Step*. Otherwise let A be the path of unlabelled edges of $C(e, T_i)$, ordered from L_i to an endpoint of e.

Label A Step. For the edges $f \in A$ in order do: Set $\ell \leftarrow e$, $g \leftarrow f$ and execute *Label Step*. Let $f \in E^{-}(v)$. If no edge of $E^{-}(v) \setminus T$ is labelled then set $\ell \leftarrow f$ and execute *Label Step* for each $g \in E^{-}(v) \setminus T$.

Implementation details.

We execute simultaneously an iteration of the algorithm in G and in G^R. That is, during the k-th iteration, we first try to enlarge a given complete $(k-1)$-intersection of G to a complete k-intersection, and then try to do the same in G^R. If either of these attempts fails, then we immediately report that the minimum cardinality of a cut in G is $k-1$, and so $\lambda = k-1$.

Graph and k-intersection representation. We store G in static adjacency lists, using two arrays: *out*, which stores the out-neighbors of each vertex, and *first*, which stores the starting position in the array *out* of the out-neighbors of each vertex. That is, the out-neighbors of a vertex v are stored in positions $\mathit{first}[v]$ to $\mathit{first}[v+1]-1$ of the array *out*. We store G^R in an analogous manner.

We represent the current k-intersection T of G as follows. For each edge $e \in E$, we maintain a status that indicates if e is used or unused in T. Moreover, the status of a used edge e corresponds to the id j of the tree T_j (or forest if $j = k$) of T that contains e. In order to search for an augmenting path efficiently, we maintain the following data structures. We represent each rooted tree T_j (or forest if $j = k$) in the current k-intersection of G with three arrays. For each vertex v in T_j, the first array stores the parent $p_j(v)$ of v in T_j, the second stores the depth of v in T_j, and the third stores a pointer to the corresponding edge of the digraph, which is either $(p_j(v), v)$ or $(v, p_j(v))$.

Additionally, we maintain an array *root* of size n that indicates the root of each vertex in the corresponding f-tree in T_k that contains it. That is, $\mathit{root}[v] = z$ if and only if $v \in F_z$. This allows us to immediately detect if (x, y) is a joining edge, by checking if $\mathit{root}[x] \neq \mathit{root}[y]$. At the beginning of a round, $\mathit{root}[v] = v$ for all vertices v. When we join two f-trees, F_z and F_x, while we augment T along an augmenting path from z, we only set $\mathit{root}[z] = x$. Finally, at the end of a round we update $\mathit{root}[v]$ to the root of the current f-tree containing v, for all vertices v. Also, at the end of each round we update the depth and parent arrays only for the trees T_j, for $j < k$, of the k-intersection that have an edge that participates in some augmenting path. We detect these trees when we execute all augmentations at the end of the round. Note that we delay these updates until the end of each round, since the round robin algorithm may terminate early, i.e., when the search for an augmenting path from some deficient vertex is unsuccessful.

We store the corresponding data structures for the current k-intersection of G^R. In order to decrease the space requirements, we consider a variant of Gabow's algorithm that computes the minimum s-cut in G and the minimum s-cut in G^R separately. That is, we first run $\lambda' + 1$ iterations in G, where λ' is the minimum cardinality of an s-cut in G, and then run $\lambda + 1$ iterations in G^R. This allows us to reuse the same data structures for storing a k-intersection in G and subsequently a k-intersection in G^R. We refer to this variant as the "serial version" of Gabow's algorithm. The running time of this version is $O((\lambda' + \lambda) m \log n)$.

Fast initialization. We also consider the following method to initialize the forest T_k at the beginning of the k-th iteration. Let T be the complete $(k-1)$-intersection of G. Then, we initialize T_k to be a depth-first search (DFS) spanning forest of $G \setminus T$, that is, the subgraph of G induced by the unused edges. We note that each f-tree F_z in T_k has a unique deficient vertex, which is its root z. Hence, this initialization complies with the requirements of the algorithm. Finally, we make all f-trees F_z in T_k, except F_s, active. We begin the DFS that constructs T_k from s, so that if it visits all vertices of G, then we immediately have a complete k-intersection and are able to advance to the next iteration. As the experimental results of Section 3.3 suggest, the above initialization, despite its simplicity, can yield significant speedups in practice.

2.2.4 Local search based algorithms

Let v be any vertex of a digraph $G = (V, E)$. A *k-out set* (resp., *k-in set*) of v is a subset $S \subseteq V$ that contains v such that $\delta^+(S) \leq k$ (resp., $\delta^-(S) \leq k$). Chechik et al. [2] presented a local search algorithm, based on depth-first search, that computes a k-out (or k-in) set of a vertex v and a given volume Δ in $O(k^{O(k)}\Delta)$ time. More precisely, the algorithm of Chechik et al., when executed from a vertex v, provides the following guarantees: If there exists a k-out set of v of volume at most Δ, then the algorithm returns a k-out set of volume $O(k\Delta)$, or concludes that G does not contain any k-out set of v of volume Δ. A faster randomized algorithm was given later by Forster et al. [3] (see also [7, 22]). Their algorithm finds a k-edge-out set of volume $O(k\Delta)$, or concludes that no k-edge-out set of volume Δ exists, in $O(k^2\Delta)$ time, with success probability $p \geq 1/2$. The algorithm allows only false negatives, i.e., with probability $1-p$ it may incorrectly decide that there is no k-edge-out set of volume Δ. Hence, by repeating this local search $O(\log n)$ times, we can ensure that the correct answer is computed with high probability.

Foster et al. [3] showed how to use this local search to test k-edge-connectivity as follows. We search for k-out and k-in sets of exponentially growing volume, starting from $\Delta = k$ and up to $\Delta = m/k$. For a fixed value of Δ we execute a sequence of local searches, initiated from a set R_Δ of $\tilde{O}(m/\Delta)$ vertices. Specifically, for each vertex $v \in R_\Delta$ we execute a local search in G and in G^R, in order to identify a k-out or a k-in set of v of volume $O(k\Delta)$. The vertices in R_Δ consist of the endpoints of m/Δ edges of G, sampled uniformly at random. (Hence, we have a good chance of hitting a vertex in a k-out or a k-in set of volume $O(\Delta)$, if G contains such a set.) As soon as we find such a k-out or k-in set, we can immediately report that $\lambda \leq k$. On the other hand, if the local searches fail to identify a k-out or k-in set, there is still a chance that such a set exists in G but has volume at least m/k. In order to handle this case, we can compute the max-flow between randomly selected sources and sinks, as follows. We sample two edges $e = (u, v)$ and $f = (w, z)$ uniformly at random, and test if the minimum u-z cut has cardinality at least k. This can be done by running k iterations of the Ford-Fulkerson method in $O(mk)$ time. Note that if G contains a k-out or a k-in set S that was not found during the local searches, then (w.h.p.) both S and $V \setminus S$ have volume at least m/k, so we have $O(1/k)$ probability that $u \in S$ and $z \in V \setminus S$. Hence, if we repeat this process $O(k \log n)$ times then we will find S with high probability in $O(k^2 m \log n)$ time.

Finally, to compute λ, we can test k-edge-connectivity for exponentially growing values of k, starting from $k = 1$ and up to $k = \delta$. This way, we find an interval $I = [k, \ell]$ such that $k \leq \lambda \leq \ell = \min\{\delta, 2k\}$, and can compute the exact value of λ by a binary search on I.

Implementation details.

As we mentioned above, for a fixed value of volume Δ, the algorithm performs local searches initiated from a set R_Δ of $\tilde{O}(m/\Delta)$ vertices, where R_Δ consists of the endpoints of m/Δ edges, sampled uniformly from G. Observe that when we execute this process for small values of Δ, we are likely to sample edges whose endpoints were already included in R_Δ. To exclude this possibility, we instead go through all vertices v of G, and start a local search from v in order to identify a k-out set (rep., k-in set) of v with probability $\delta^+(v)/\Delta$ (resp., $\delta^-(v)/\Delta$). This way, we also do not need to store R_Δ explicitly.

In order to detect local edge cuts we apply algorithm 2.1. The algorithm takes as inputs a vertex $x \in V$ and two integers v and k. The basic idea is to detect a set S coantaing vertex x with at most k edges oriented from S to $V - S$ and a volume at most v. To that end, we apply Depth-first Search (DFS) on the starting vertex x and we force early termination according to the desired volume. Specifically, we repeat k iterations of DFS, and if DFS terminates normally at some iteration, i.e., without having to apply the early termination condition, then the set of reachable vertices S have at most k outgoing edges from S to $V - S$ with volume at most v, otherwise, we certify that no such a set exists.

Algorithm 2.1: An algorithm for detecting a local edge cut

```
input  : $x$, $v$, $k$
output : a vertex set that contains $x$ with volume $\leq v$ and $E(V, V-S) \leq k$ or
         failure
1 for $i \leftarrow 1$ to $k$ do
2     Grow a DFS tree $T$ starting from $x$, stopping early at some point to get
        $y \in V(T)$;
3     If the DFS terminates normally, then return $V(T)$;
4     Reverse all edges along the unique path from $x$ to $y$ in the tree $T$ , unless
        this is the last iteration;
5 end
6 return failure;
```

2.3 Empirical Analysis

We implemented our algorithms in C++, using g++ v.7.4.0 with full optimization (flag -O3) to compile the code. The reported running times were measured on a GNU/Linux machine, with Ubuntu (18.04.5 LTS): a Dell PowerEdge R715 server 64-bit NUMA machine with four AMD Opteron 6376 processors and 128GB of RAM memory. Each processor has 8 cores sharing a 16MB L3 cache, and each core has a 2MB private L2 cache and 2300MHz speed. In our experiments we did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the `high_resolution_clock` function of the standard library `chrono`, averaged over ten different runs.

The real-world graphs we used in our experiments are reported in Table 2.1. Table 3.1 gives an overview of the algorithms we consider.

Table 2.1: Real-world graphs sorted by file size of their largest SCC; n is the number of vertices, m the number of edges, and δ_{avg} is the average vertex degree; λ and δ denote, respectively, the edge connectivity and the minimum (in or out) degree.

	Original Graph				Largest SCC						
Dataset	n	m	file size	δ_{avg}	n	m	file size	δ_{avg}	λ	δ	type and source
Rome99	3.3k	8.8k	98k	2.65	3.3k	8.8k	98k	2.64	1	1	road network [27]
p2p-gnutella25	22.6k	54.7k	685k	2.41	5.1k	17.9k	199k	3.48	1	1	peer2peer [28]
Oracle-16k	15.6k	48.2k	582k	3.08	10.4k	29,9k	320k	2.88	1	1	memory profiling [29]
enron	69.2k	276.1k	3.0M	3.98	8.0k	147.3k	1.6M	17.81	1	1	email [30]
web-NotreDame	325k	1.4M	22M	4.6	48.7k	267k	3.4M	5.49	1	1	web graph [28]
soc-Epinions1	75.8k	508k	5.9M	6.71	32.2k	442k	5.1M	13.74	1	1	social network [28]
Amazon0302	262k	1.2M	18M	4.71	241k	1.1M	17M	4.67	1	1	prod. co-purchase [28]
wiki-Talk	2.3M	5.0M	69M	2.1	111k	1.4M	18M	12.93	1	1	social network [28]
web-Stanford	281k	2.3M	34M	8.2	150k	1.5M	22M	10.47	1	1	web graph [28]
Amazon0601	403k	3.4M	49M	8.4	395k	3.3M	48M	8.35	1	1	prod. co-purchase [28]
web-BerkStan	685k	7.6M	113M	11.09	334k	4.5M	66M	13.50	1	1	web graph [28]
Oracle-4M	4.1M	14.6M	246M	3.55	2.8M	8.4M	137M	2.95	1	1	memory profiling [29]
sap-4M	4.1M	12.0M	183M	2.92	4.0M	11.9M	181M	2.91	1	1	memory profiling [29]
Oracle-11M	10.7M	33.9M	576M	3.18	6.4M	15.9M	261M	2.47	1	1	memory profiling [29]
sap-11M	11.1M	36.4M	668M	3.27	11.1M	36.3M	673M	3.26	1	1	memory profiling [29]
LiveJournal	4.8M	68.9M	1.1G	14.23	3.8M	65.3M	1G	17.06	1	1	social network [30]

Table 2.2: An overview of the algorithms considered in our experimental study. The bounds refer to a simple digraph with n vertices and m edges, edge connectivity λ, and λ' minimum cardinality of an s-cut.

Algorithm	Abbr.	Technique	Complexity	Ref.
Ford-Fulkerson	FF	Compute v_1-v_2, v_2-v_3, ...v_{n-1}-v_n, v_n-v_1 maximum flows	$O(mn^2)$	[14]
Dinic	Dinic		$O(n^{5/3}m)$	[16, 15]
Mansour-Schieber I	MS	Compute out/in-dominating sets O/I and O-v and v-I maximum flows	$O(mn)$	[18]
Mansour-Schieber II	MS II		$O(\lambda^2 n^2)$	[18]
Gabow	G	Compute a λ-intersection in G and a λ-intersection in G^R simultaneously	$O(\lambda m \log n)$	
Gabow serial version	G-S	Compute a λ'-intersection in G and then a λ-intersection in G^R	$O((\lambda' + \lambda)m \log n)$	[1]
Local Search	LS	Random sampling and local search for exponentially increasing volume	$\tilde{O}(\lambda^2 m)$	[3]

2.3.1 Strongly connected graphs

In our first experiment, we use as input the largest SCC of the real-world graphs shown in Table 2.1. Interestingly, each such component contains a vertex of in-degree or out-degree one, so we have $\lambda = \delta = 1$. So, in this case the algorithms just verify that the graphs have a 1-edge cut. The results of this experiment are given in Table 2.3, where we terminate the execution of an algorithm if it exceeds 2 hours. See also Figure 2.1 for the corresponding plot. From the results, we observe that MS has overall the worst performance, followed by FF and Dinic. The remaining algorithms, MS II, G, G-S (and their improved versions), as well as LS are orders of magnitude faster. Algorithms G and G-S have similar performance, which is due to the fact that λ' (the minimum cardinality of an s-cut in G) is usually equal to λ in all these instances. The same observation holds for their improved versions G-DFS and G-S-DFS. Comparing G with G-DFS and G-S with G-S-DFS, we observe that our improved initialization of the current forest T_k incurs a speedup larger than 5 on average. Finally, we note that LS also performs very well on these instances, and it is even faster than G and G-S.

Table 2.3: Running times in seconds of the algorithms for SCCs of the real-world graphs of Table 2.1. The execution of an algorithm was terminated if it exceeded 2 hours.

Graph	FF	Dinic	MS	MS II	G-S	G-S-DFS	G	G-DFS	LS
rome99	0.43	0.08	1.42	0.01	0.02	0.01	0.03	0.01	0.01
p2p-gnutella25	3.89	3.25	4.32	0.01	0.01	0.01	0.02	0.01	0.01
oracle-16k	6.54	6.69	7.20	0.01	0.01	0.01	0.02	0.01	0.01
enron	14.55	12.62	24.64	0.02	0.04	0.01	0.05	0.01	0.01
web-NotreDame	148.32	63.32	241.68	0.04	0.09	0.02	0.11	0.02	0.07
soc-Epinions1	487.58	547.36	590.47	0.14	0.14	0.03	0.09	0.04	0.08
Amazon0302	>2h	>2h	>2h	0.62	1.11	0.24	0.98	0.15	0.18
wiki-Talk	3,443.95	6,178.35	>2h	0.56	0.20	0.12	0.24	0.08	0.37
web-Stanford	>2h	>2h	>2h	0.57	2.71	0.19	2.50	0.17	0.22
amazon0601	>2h	>2h	>2h	1.41	1.45	0.35	1.34	0.34	0.61
web-BerkStan	>2h	>2h	>2h	0.80	1.71	0.21	1.90	0.21	0.49
oracle-4M	>2h	>2h	>2h	3.08	5.58	0.80	4.58	0.80	2.12
sap-4M	>2h	>2h	>2h	3.70	6.41	1.01	6.90	0.91	2.04
oracle-11M	>2h	>2h	>2h	4.50	13.96	1.43	13.46	1.27	3.35
sap-11M	>2h	>2h	>2h	10.38	52.59	6.76	41.73	5.64	6.94
LiveJournal	>2h	>2h	>2h	42.22	20.79	6.57	20.17	6.26	11.76

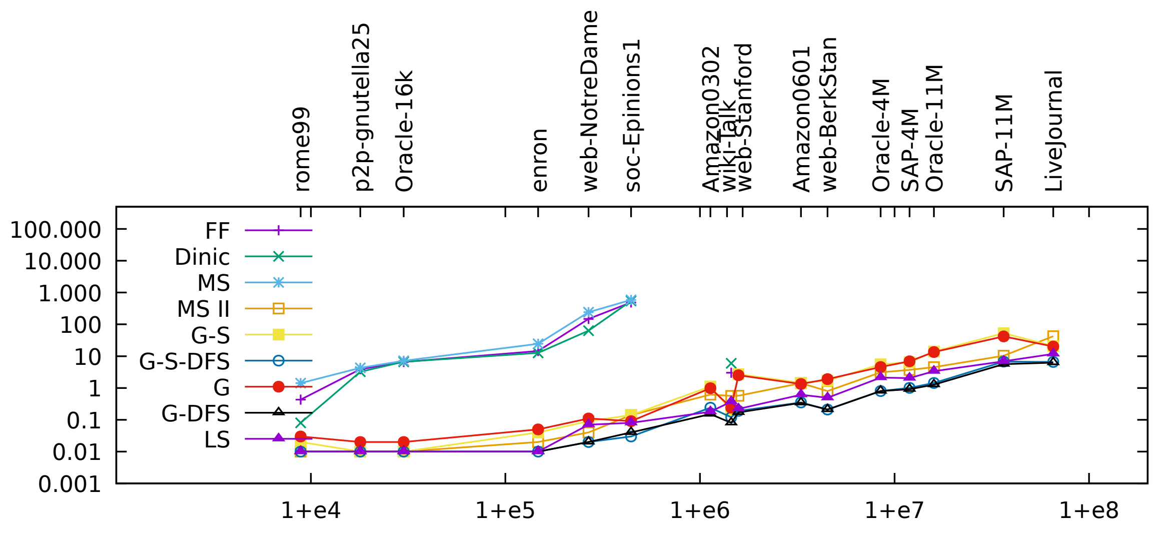

Figure 2.1: Plot of the running times in seconds of the algorithms for SCCs of the real-world graphs of Table 2.1.

2.3.2 2-edge connected graphs

Next, we consider 2-edge connected subgraphs of some real-word graphs, whose characteristics are reported in Table 2.4. Here, all input graphs have edge-connectivity $\lambda = \delta = 2$. The running times of the algorithms are reported in Table 2.5, and are plotted in Figure 2.2. We observe that in this experiment, unlike the $\lambda = 1$ case, Dinic outperforms FF in all instances, and in several cases it even outperforms MS. Again, MS II has superior performance compared to MS, but still it is not competitive with G, G-S (and their improved versions), and LS on several instances. We also observe that G-DFS and G-S-DFS achieve larger speedups compared to G and G-S, respectively, than in the $\lambda = 1$ case. Indeed, on average, G-S-DFS is more than 22 times faster than G-S, while G-DFS is more than 26 times faster than G. The local search based algorithm LS also performs well on these 2-edge connected instances, but it is still more than 6 times slower than G and more than 26 times slower than G-DFS on average.

Table 2.4: Characteristics of 2-edge connected subgraphs of some real-word graphs from Table 2.1; n is the number of vertices, m the number of edges; λ and δ denote, respectively, the edge connectivity and the minimum (in or out) degree.

Graph	n	m	filesize	δ	λ
rome99_2ECC-1	2249	6467	64K	2	2
web-NotreDame_2ECC-2	1409	6856	66K	2	2
web-NotreDame_2ECC-1	1462	7279	70K	2	2
web-BerkStan_2ECC-1	1106	8206	73K	2	2
twitter-higgs-retweet_2ECC-1	1099	9290	81K	2	2
web-NotreDame_2ECC-3	1416	13226	124K	2	2
web-BerkStan_2ECC-3	4927	28142	292K	2	2
enron_2ECC-1	4441	123527	1.3M	2	2
web-Stanford_2ECC-1	5179	129897	1.3M	2	2
web-Stanford_2ECC-2	10893	162295	1.8M	2	2
Amazon-302_2ECC-1	55414	241663	3.0M	2	2
web-BerkStan_2ECC-2	12795	347465	3.6M	2	2
soc-Epinions1_2ECC-1	17117	395183	4.0M	2	2
web-BerkStan_2ECC-4	29145	439148	5.2M	2	2
web-Google_2ECC-1	77480	840829	11M	2	2
WikiTalk_2ECC-1	49430	1254898	14M	2	2
Amazon-601_2ECC-1	276049	2461072	34M	2	2

2.3.3 Augmented graphs

In our final experiment, we consider how the running time of each algorithm is affected as the edge connectivity value λ increases. To that end, we augment the real-word graphs as follows. Let G be an input strongly connected digraph. For a given parameter k, we create an augmented instance G_k of G by executing the following procedure. We go through the vertices of G and, for each vertex v, we add $k - \delta^-(v)$ edges directed to v if $\delta^-(v) < k$. Then, we make a second pass over the vertices and, for each vertex v, we add $k - \delta^+(v)$ edges directed away from v if $\delta^+(v) < k$. Notice that the resulting graph has minimum degree $\delta \geq k$.

Table 2.6 reports the characteristics of the augmented graphs produced by the above procedure. In Table 2.7 we report the running times of each algorithm with

Table 2.5: Running times in seconds of the algorithms for the 2-edge connected subgraphs of Table 2.4. The execution of an algorithm was terminated if it exceeded 2 hours.

Graph	FF	Dinic	MS	MS II	G-S	G-S-DFS	G	G-DFS	LS
rome99_2ECC-1	1.76	0.078	0.86	1.06	0.13	0.01	0.11	0.01	0.07
web-NotreDame_2ECC-2	0.8	0.079	0.33	0.01	0.02	0.01	0.03	0.01	0.06
web-NotreDame_2ECC-1	1	0.097	0.35	0.01	0.01	0.01	0.03	0.01	0.07
web-BerkStan_2ECC-1	0.72	0.12	0.30	0.01	0.03	0.01	0.02	0.01	0.03
twitter-higgs-retweet_2ECC-1	0.88	0.62	0.27	0.01	0.03	0.01	0.01	0.01	0.05
web-NotreDame_2ECC-3	0.85	0.25	0.56	0.06	0.05	0.01	0.08	0.01	0.07
web-BerkStan_2ECC-3	14.64	0.74	3.20	1.89	0.28	0.01	0.28	0.01	0.19
enron_2ECC-1	35.5	8.27	6.64	0.02	0.16	0.01	0.14	0.01	1.00
web-Stanford_2ECC-1	38.97	9.2	8.29	0.05	0.09	0.02	0.17	0.01	0.79
web-Stanford_2ECC-2	124.05	10.19	59.45	3.50	0.55	0.03	0.76	0.04	1.44
Amazon-302_2ECC-1	2332.06	138.93	979.38	538.58	4.98	0.28	3.54	0.23	4.46
web-BerkStan_2ECC-2	591.72	120.35	72.22	0.15	0.60	0.04	0.59	0.03	2.33
soc-Epinions1_2ECC-1	486.4	375.77	246.36	14.14	0.23	0.11	0.22	0.08	6.14
web-BerkStan_2ECC-4	1213.97	81.12	221.27	45.33	0.78	0.16	1.05	0.14	4.96
web-Google_2ECC-1	11126.71	188.02	2542.86	216.45	11.51	0.43	8.66	0.47	11.60
WikiTalk_2ECC-1	4390.89	3917.49	2892.46	5.71	0.50	0.14	0.43	0.15	22.06
Amazon-601_2ECC-1	>2h	>2h	>2h	5,868.82	44.14	2.41	36.57	2.92	71.36

an augmented graph G_k as input, for $k \in \{2, 4, 8, 16\}$. The running times are plotted in Figure 2.3, where the graphs are ordered by their number of edges, and also in Figure 2.4, where the graphs are grouped by type.

Now, we observe that Dinic starts to outperform FF as λ increases, while MS is faster than Dinic on most instances. Unlike our previous experiments, MS II runs much slower than MS, and its performance degrades more sharply for larger values of λ. The same phenomenon is apparent for LS, which is no longer competitive with Gabow's algorithm as λ increases. We note that this behavior is expected for MS II and LS, as the running time of both algorithms has a quadratic dependence on λ. Finally, we observe that the four versions of Gabow's algorithm, G, G-S, G-DFS and G-S-DFS, are very robust. Again, our improved initialization of the current forest T_k in each iteration of Gabow's algorithm provides significant speedups: more than

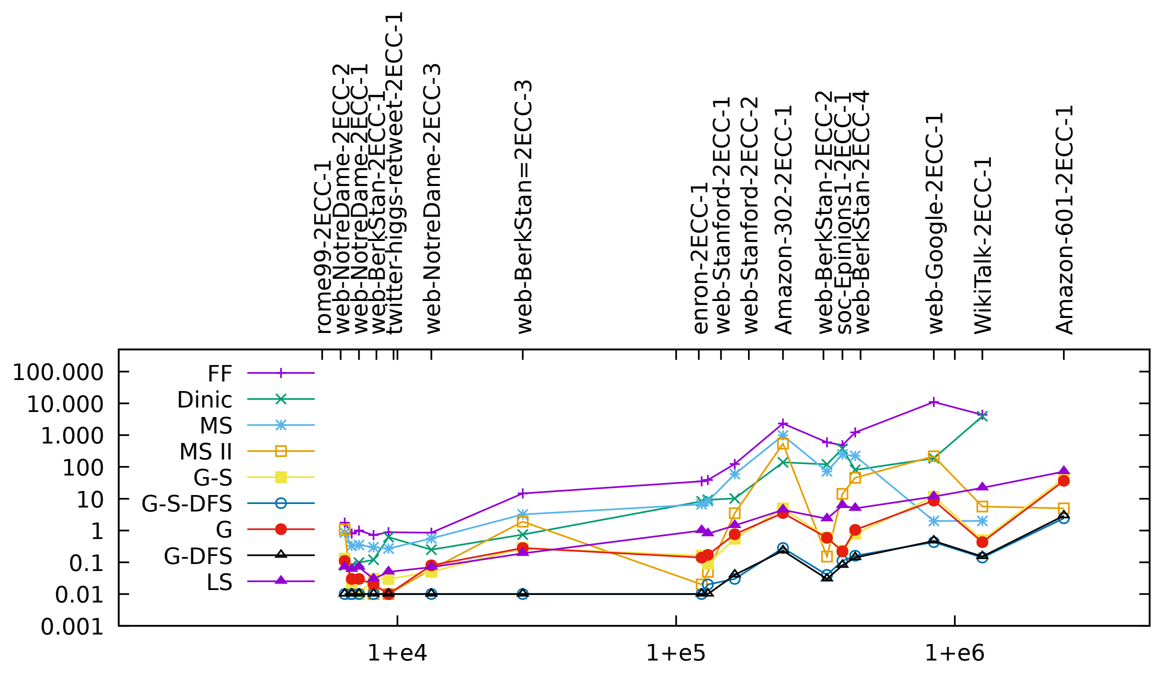

Figure 2.2: Plot of the running times in seconds of the algorithms for the 2-edge connected subgraphs of Table 2.4.

a factor of 10 on average for G-S-DFS, and more than a factor of 9 on average for G-DFS.

Table 2.6: Characteristics of augmented graphs, resulting from real-world graphs of Table 2.1 after inserting some edges; n is the number of vertices, m the number of edges; λ and δ denote, respectively, the edge connectivity and the minimum (in or out) degree.

Graph	n	m	filesize	δ	λ
rome99-EC2	3352	9869	92k	2	2
rome99-EC4	3352	15468	144k	4	4
rome99-EC8	3352	31952	296k	8	8
rome99-EC16	3352	65542	605k	16	16
p2p-Gnutella25-EC2	5152	19765	184k	2	2
p2p-Gnutella25-EC4	5152	27565	257k	4	4
p2p-Gnutella25-EC8	5152	50076	466k	8	8
p2p-Gnutella25-EC16	5152	100669	591k	16	16
enron-EC2	8271	151651	1.5M	2	1
enron-EC4	8271	162999	1.6M	4	3
enron-EC8	8271	190618	1.9M	8	7
enron-EC16	8271	253345	2.5M	16	15
oracle-16k-EC2	10405	39756	356k	2	2
oracle-16k-EC4	10405	62589	576k	4	4
oracle-16k-EC8	10405	112138	1.1M	8	8
oracle-16k-EC16	10405	215195	2.0M	16	16

Table 2.7: Running times in seconds of the algorithms for the augmented graphs of Table 2.6.

Graph	FF	Dinic	MS	MS II	G-S	G-S-DFS	G	G-DFS	LS
rome99-EC2	1.20	0.73	2.76	2.58	0.06	0.01	0.07	0.01	0.16
rome99-EC4	3.14	6.01	4.64	18.43	0.04	0.01	0.04	0.01	1.11
rome99-EC8	12.48	14.35	9.66	41.88	0.12	0.01	0.15	0.01	8.23
rome99-EC16	44.35	27.38	16.43	145.04	0.41	0.02	0.37	0.02	63.05
p2p-Gnutella25-EC2	6.48	9.26	7.05	0.36	0.03	0.01	0.04	0.01	0.23
p2p-Gnutella25-EC4	14.51	19.29	10.38	14.94	0.10	0.01	0.10	0.01	1.88
p2p-Gnutella25-EC8	36.72	38.48	22.64	80.31	0.28	0.01	0.27	0.02	14.43
p2p-Gnutella25-EC16	113.80	70.49	39.48	355.46	0.69	0.05	0.69	0.07	114.45
enron-EC2	25.65	21.25	30.95	0.68	0.04	0.03	0.06	0.02	0.02
enron-EC4	67.24	95.11	43.17	3.53	0.26	0.03	0.22	0.03	5.33
enron-EC8	170.04	174.73	75.76	40.62	0.71	0.05	0.62	0.08	36.98
enron-EC16	468.09	330.42	136.44	294.01	1.78	0.16	1.74	0.20	237.62
oracle-16k-EC2	22.38	36.26	32.17	1.02	0.07	0.03	0.06	0.02	0.44
oracle-16k-EC4	69.31	102.28	42.34	50.01	0.22	0.04	0.19	0.04	5.48
oracle-16k-EC8	181.06	190.01	74.96	304.84	0.69	0.05	0.53	0.05	41.23
oracle-16k-EC16	536.85	440.09	179.11	1732.77	2.14	0.16	1.58	0.16	337.99

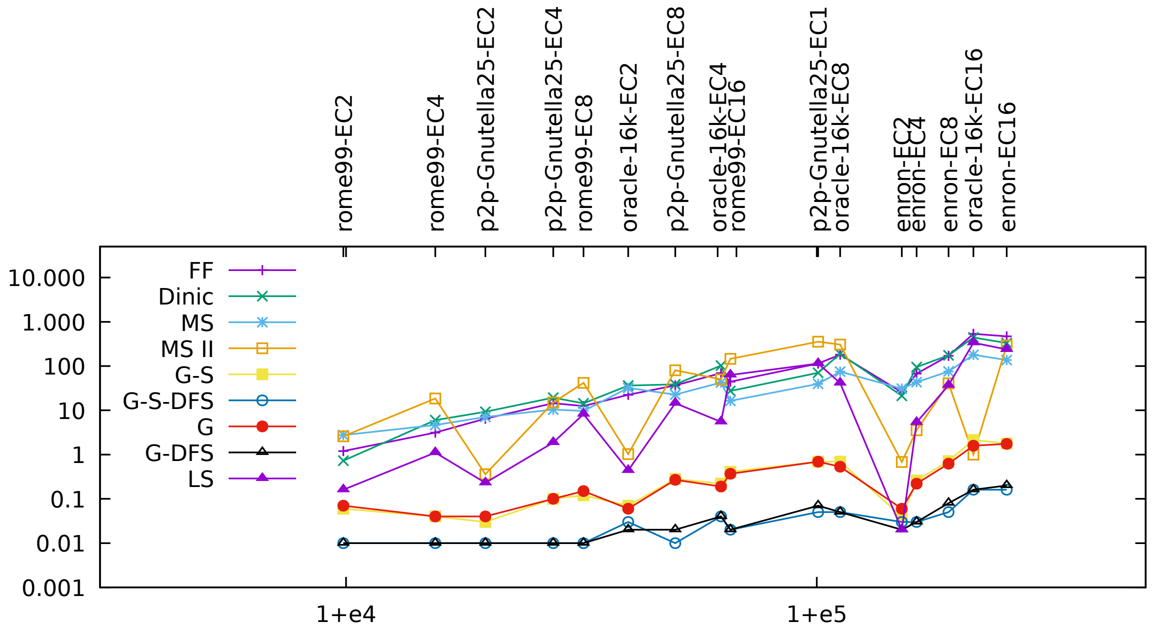

Figure 2.3: Plot of the running times in seconds of the algorithms for the augmented graphs of Table 2.6.

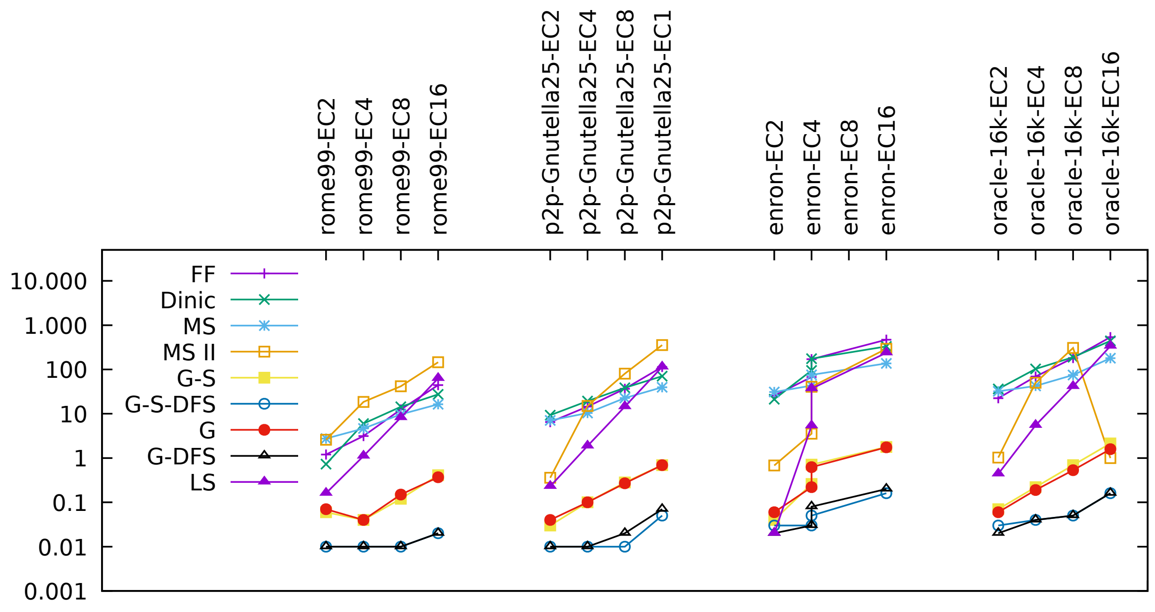

Figure 2.4: A different plot of the running times in seconds of the algorithms for the augmented graphs of Table 2.6: here we aim at showing how the running time is affected by increasing the edge connectivity; graphs are shown in an ordinal way, and their position on the x-axis is not representative of their number of edges, as in Figure 2.3.

2.3.4 Faster implementations of local searched based algorithm

In this section we revisit the implementation of the local searched based algorithm due to the work of [31]. The authors provided two heuristic methods named Local1+ and Local2+ that improve the performance of the local searched based algorithm. Since the main overhead of the algorithm is the subroutine that detects local cuts, the authors presented conditions that allows the subroutine to terminate earlier, based on the sum of degrees of the visited vertices instead of the number of accessed edges.

We implemented and tested the proposed implementations of Local1+ and Local2+ [31] on augmented graphs and as we can see in Table 2.8 Local1+ and Local2+ improve the performane of local search based algorithm significantly, however Gabow's algorithm clearly outperforms Local1+ and Local2+ and the dominance of the algorithm of Gabow is becoming obvious as the edge conectivity increases.

Table 2.8: Running times in seconds among various implementations of Local search based algorithm and implemenations of Gabow's algorithm. LS1+ denotes Local1+. Similarly, LS2+ denotes Local2+.

Graph	G-S	G-S-DFS	G	G-DFS	LS	LS1+	LS2+
rome99-EC2	0.06	0.01	0.07	0.01	0.16	0.14	0.03
rome99-EC4	0.04	0.01	0.04	0.01	1.11	0.47	0.14
rome99-EC8	0.12	0.01	0.15	0.01	8.23	1.71	0.81
rome99-EC16	0.41	0.02	0.37	0.02	63.05	5.85	6.84
p2p-Gnutella25-EC2	0.03	0.01	0.04	0.01	0.23	0.14	0.05
p2p-Gnutella25-EC4	0.10	0.01	0.10	0.01	1.88	0.68	0.27
p2p-Gnutella25-EC8	0.28	0.01	0.27	0.02	14.43	3.15	1.61
p2p-Gnutella25-EC16	0.69	0.05	0.69	0.07	114.45	14.15	11.64
enron-EC2	0.04	0.03	0.06	0.02	0.02	00.00	0.00
enron-EC4	0.26	0.03	0.22	0.03	5.33	2.18	0.78
enron-EC8	0.71	0.05	0.62	0.08	36.98	6.98	2.26
enron-EC16	1.78	0.16	1.74	0.20	237.62	32.52	13.13
oracle-16k-EC2	0.07	0.03	0.06	0.02	0.44	0.80	0.11
oracle-16k-EC4	0.22	0.04	0.19	0.04	5.48	2.58	0.63
oracle-16k-EC8	0.69	0.05	0.53	0.05	41.23	7.84	3.70
oracle-16k-EC16	2.14	0.16	1.58	0.16	337.99	32.16	30.57

2.4 Concluding Remarks

In this work we provided a comprehensive experimental study of algorithms that compute the edge connectivity of a directed graph. We also presented efficient implementations of Gabow's matroid intersection and packing spanning trees algorithm [1], and applied a simple heuristic that improved its practical performance significantly. Furthermore, we gave an implementation of a randomized algorithm for computing the edge connectivity λ, based on local search [2, 3, 7, 22, 31]. Our experimental results showed that this algorithm is competitive with Gabow's algorithm when λ is small, but its performance degrades rapidly as λ increases. Finally our code is publicly available at https://github.com/rogatienne/ALENEX21EdgeConnectivity.

Chapter 3

Packing arborescences

In this chapter we explore the design space of efficient algorithms that compute a maximum packing of edge-disjoint arborescences. In particular, we present an efficient implementation of Gabow's arborescence packing algorithm [1], and we provide a simple but efficient heuristic that significantly improves its running time in practice. Then, we conduct a thorough empirical study to highlight the merits and weaknesses of each technique.

3.1 Background and Definitions

A *spanning tree* of an undirected connected graph G is a connected acyclic spanning subgraph of G. We extend this definition to directed graphs by ignoring the edge orientations. Let T be a *spanning tree* of a directed graph G rooted at s; T is an *s-arborescence* of G if s has in-degree zero and every other vertex has in-degree one. (Thus, any s-arborescence is a spanning tree of G but not vice versa.) The *arborescence packing problem* for vertex s is to construct the greatest possible number of edge-disjoint

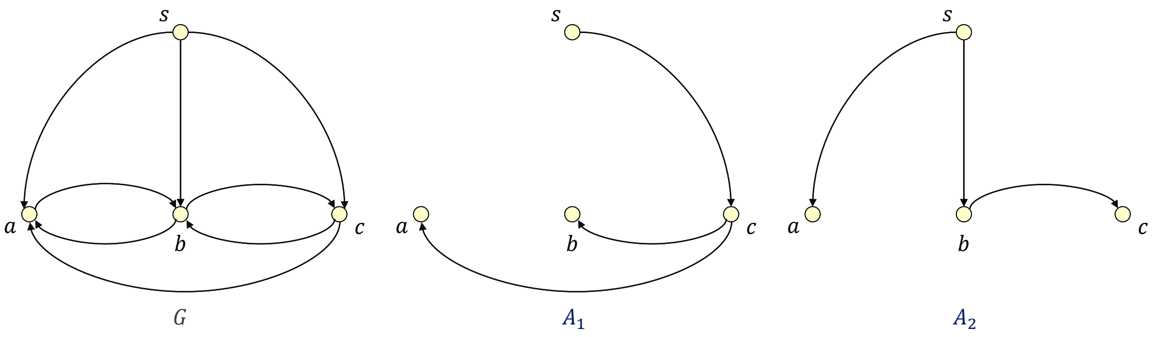

Figure 3.1: A directed graph G with start vertex s. Subgraphs A_1 and A_2 are two edge-disjoint s-arborescences of G ($\{A_1\ ,\ A_2\}$ is a maximum set of edge-disjoint s-arborescences of G).

s-arborescences. See Figure 3.1. These concepts are useful in applications such as modeling broadcasting and evacuation [32].

Here we consider the arborescence packing problem from a practical perspective. Our starting point is the following fundamental theorem of Edmonds:

Theorem 3.1. *(Edmonds [8]) The maximum number of edge-disjoint s-arborescences of G equals the minimum cardinality of an s-cut.*

Edmonds gave an algorithmic proof, but the algorithm is complicated and seems to require exponential time in the worst-case [1]. Later, Lovasz [33] gave an elegant proof of Edmonds' theorem. Tarjan [4] presented an $O(k^2m^2)$-time algorithm to compute a maximum packing of edge-disjoint s-arborescences, where $k = c_G(s)$. Schiloach [34], presented later an $O(k^2mn)$-time algorithm. The same bound was achieved by Tong and Lowler [5], who also claimed that Schiloach's algorithm is flawed. See also [35, 36] for recent interesting generalizations of Edmonds' theorem.

Currently, the best bound for the arborescence packing problem is $O(mk\log n + nk^4\log^2 n)$, which was achieved by Bhalgat et al. [37] using the concept of edge splitting [38, 39]. Let G be a digraph with start vertex s. Define the *s-v edge-connectivity* $c_G(s,v)$, for any vertex $v \neq s$, as the cardinality of the minimum s-v cut. We say that a vertex v is *eligible* if $indegree(v) \geq outdegree(v)$. For such a vertex v, we can assume that $indegree(v) = outdegree(v)$ since we can add multiple edges from v to s without affecting the s-w edge-connectivity $c_G(s,w)$ of any $w \neq s$. Splitting off two edges (x,y) and (y,z) means deleting these two edges, and adding a new edge (x,z). This operation can be done so that the s-v edge-connectivity $c_G(s,v)$ is preserved for any $v \neq y$. Splitting off of an eligible vertex v means to split off pairs of edges incident on v so that each pair consists of an edge entering v and an edge leaving v, without affecting the connectivity of the remaining graph until v has no outgoing edge. Now, v can be removed from the graph without affecting the connectivity of the remaining graph [37]. Bhalgat et al. describe a procedure that removes any specified set S of eligible vertices while maintaining the s-v edge-connectivity of all $v \notin S$. Their algorithm extends the edge-connectivity algorithm of Gabow so that it can compute a splitting of all vertices in S. Then, it recursively computes a maximum arborescence packing of the resulting graph, which can then be used to recover a maximum arborescence packing of the original graph by putting back the vertices in S.

3.2 Tested algorithms

Here, We provide an overview of all the tested algorithm that we consider in our experimental analysis.

Let A be a partial s-arborescence of G. We say that A is *good* if $c_{G-A}(s) \geq k-1$. All the algorithms we consider try to enlarge a partial s-arborescence A of G by adding one edge at a time. We note that we can combine any packing arborescences algorithm with Gabow's edge connectivity algorithm as follows. First, we run Gabow's edge connectivity algorithm on G, and compute a complete k-intersection T of G in $O(km \log n^2/m)$ time. Then, we keep in G only the edges of $E(T)$ and run the packing arborescences algorithm on the reduced graph. (This is valid by the Matroid Characterization of s-cuts.) If the packing algorithm runs in $O(f(m,n))$ time on the original graph, then the combined algorithm runs in $O(km \log n^2/m + f(nk,n))$ total time.

3.2.1 The algorithm of Tarjan

Tarjan [4] computes a maximum packing of arborescences $\mathcal{A} = \{A_1, \ldots, A_k\}$ by executing k iterations of the following procedure. During the j-th iteration (for $j = 1, 2, \ldots, k$), it computes an arborescence A_j of G such that A_j is pairwise edge-disjoint with the arborescences in $\mathcal{A}^{(j-1)} = \{A_1, \ldots, A_{j-1}\}$, and $G^{(j)} = G \setminus \mathcal{A}^{(j)}$ has $c_{G^{(j)}}(s) = k - j$. (I.e., A_j is good for $G^{(j-1)}$.) To compute A_j we work as follows. We initialize a vertex set $S = \{s\}$, the set of edges $E(A_j) = \emptyset$ and mark all edges of $G^{(j-1)}$ as usable. Then, we perform the following step until $S = V(G)$. We find a usable edge $e = (u, v)$ such that $u \in S$ and $v \in V(G) \setminus S$, mark e as unusable and compute $c_{G'}(S)$, where $G' = G^{(j-1)} \setminus (A_j \cup \{e\})$. If $c_{G'}(S) \geq k - j$ then we add v to S, and add e to A_j.

To test if $c_{G'}(S) \geq k-j$, it suffices to determine if we can send at least $k-j$ units of flow from S to v (where we assign unit edge capacities). This can be done in $O(km)$ time by executing at most $k-j$ iterations of the Ford-Fulkerson method [14]. Since we perform k iterations, and in each iteration we test at most m edges, the total running time is $O(k^2m^2)$. This bound is reduced to $O(km \log n^2/m + k^4n^2) = O(k^4n^2)$ if we first compute a complete k-intersection of G by Gabow's edge connectivity algorithm, and then run Tarjan's algorithm on the reduced graph that contains only the edges of $E(T)$.

Implementation details In our experiments, we noticed that the order in which we examine the usable edges may affect the running time of the algorithm significantly. Hence, we considered two versions of the algorithm: In the first version, we maintain the vertices of S in a stack, and examine the usable edges $e = (u, v)$ starting from the most recently added vertices $u \in S$. In the second version, we maintain S in a FIFO queue, and examine the usable edges $e = (u, v)$ starting from the least recently added vertices $u \in S$. As in turns out, in our experiments the stack version performed significantly better on most instances. (See Appendix 3.3.)

3.2.2 The algorithm of Tong and Lawler

The algorithm of Tong and Lawler [5] applies a divide and conquer approach. Similarly to Tarjan's algorithm, it grows a partial arborescence A of G by trying to add one edge at a time. Initially $V(A) = \{s\}$ and $E(A) = \emptyset$. A candidate edge $e = (u, v)$, such that $u \in V(A)$ and $v \in V(G) \setminus V(A)$, is selected according to the following rule: $u \neq s$, unless there is no other candidate edge. Then, we compute $c_{G'}(s)$, where $G' = G - (A \cup \{e\})$, and consider the following two cases. (a) If $c_{G'}(s) \geq k-1$ then the examination of e is *successful*. In this case, we add v to $V(A)$, and add e to $E(A)$. If A is now an arborescence, that is $V(A) = V(G)$, then we set $G = G - A$ and recursively compute $k-1$ edge-disjoint arborescences in G. (b) Otherwise, we have $c_{G'}(S) = k-2$, and the examination of e is *unsuccessful*. In this case, the algorithm of Tong and Lawler applies divide and conquer as follows. Let $(S, V(G) \setminus S)$ be a minimum s-cut of G, where $s \in S$. It is easy to observe that $e \in E(S, V(G) \setminus S)$, hence $v \in V(G) \setminus S$, and moreover, that there must be an edge $e' \in E(A)$ such that $e' \in E(S, V(G) \setminus S)$. Next, we split G into two auxiliary graphs G_1 and G_2, with corresponding partial arborescences A_1 and A_2 as follows. To construct G_1, we contract the vertices of S into s and delete all edges directed to s. Then, A_1 consists of the edges of A that were not deleted and is a partial arborescence of G_1 (but it may not be a full arborescence yet). Similarly, we construct G_2 by contracting the vertices of $V(G) \setminus S$ into v and delete self-loops. To form a corresponding partial arborescence A_2 of A in G_2, we delete all the edges of A that are directed from S to $V(G) \setminus S$ except for the first edge (x, y) on a path from S to $V(G) \setminus S$ in A. That is, x is reachable from s through a path that contains only vertices in $S \cap V(A)$. So, A_2 consists of the edges of A that were not deleted and is a partial arborescence of G_2 (but it may not be a full arborescence yet). We recursively compute k edge-disjoint arborescences $A_1^1, A_2^1, \ldots, A_k^1$ of G_1 and $A_1^2, A_2^2, \ldots, A_k^2$ of G_2, where $E(A_1^1) \subseteq E(A_1)$ and $E(A_1^2) \subseteq E(A_2)$. Then, we combine these arborescences to form k edge-disjoint arborescences of G. This combination is easy to perform because each arborescence of G_1 is edge-disjoint from exactly $k-1$ arborescences of G_2 and vice versa. Hence, to form the desired arborescences of G, we combine each pair of non-disjoint arborescences.

As in Tarjan's algorithm, we can test if an edge e can be included in the partial arborescence A in $O(km)$ time by executing k iterations of the Ford-Fulkerson method. Since at most kn edges are added in the packing $\mathcal{A}$, the total time spent for successful edge additions is $O(k^2mn)$. On the other hand, since we can split G at most n times, there are at most n unsuccessful edge examinations. Thus, the total time spent for unsuccessful edge examinations is $O(kmn)$, which results to a total running time of $O(k^2mn)$.

Tong and Lawler also observed that the running time of their algorithm can be improved to $O(kmn + k^3n^2)$ after some preprocessing. The preprocessing phase computes a flow of value k from s to each other vertex $t \neq s$. After we have computed an s-t flow, we delete the edges entering t with zero flow. By repeating this process for all vertices $t \neq s$, after $O(kmn)$ time we are left with a subgraph H of G with $O(kn)$ edges and $c_H(s) = k$. Thus, we can compute k edge-disjoint arborescences of H in $O(k^3n^2)$ time. If we use Gabow's edge connectivity algorithm to compute a complete

k-intersection of G instead of H, then we obtain an $O(km \log n^2/m + k^3n^2) = O(k^3n^2)$ time bound.

Implementation details As in our implementation of Tarjan's algorithm, we examine candidate edges (to be included in the partial arborescence A) using a stack or a FIFO queue. As with Tarjan's algorithm, the running time of the Tong-Lawler algorithm depends on the order in which we examine candidate edges. In our experiments the stack version of the Tong-Lawler algorithm outperformed the queue version, but not consistently. (See Appendix 3.3.) When we split G, we create two new graph instances for G_1 and G_2, and assign new vertex ids so that they are in the ranges $[1, |V(G_1)|]$ and $[1, |V(G_2)|]$ respectively. To restore the original vertex ids, we maintain mappings $h_i : V(G_i) \mapsto V(G)$, $i = 1, 2$. Moreover, for each edge in G_1 and G_2 that has exactly one endpoint in a contracted part of the graph (i.e., for each edge (x, y) such that $x \in S$ and $y \in V(G) \setminus S$, or vice versa), we associate it with the corresponding original edge of G. This information suffices to combine the arborescences in G_1 and G_2 and form the arborescences of G. Thus, after a split, we no longer need to keep the initial graph in memory.

3.2.3 The algorithm of Gabow

Gabow [1] presented an $O(k^2n^2)$-time algorithm to compute a maximum arborescence packing. First, it computes a complete k-intersection T of the input digraph for s. We let G be the subgraph with edges $E(T)$. Then, it repeats the following procedure, until $k = 0$:

1. Compute a complete $(k-1)$-intersection T of G.
2. Find a good s-arborescence A of G, using the algorithm described below in Section 3.2.3.
3. Decrease k by one and repeat the procedure on $G - A$.

Similarly to the algorithms of Tarjan, and of Tong and Lawler, in Step 2 Gabow's algorithm tries to enlarge a partial arborescence by adding one edge at a time. Unlike the these other algorithms, however, Gabow's algorithm does not perform flow computations, but relies on the framework of his edge-connectivity algorithm. Hence, we first provide an overview of how Gabow computes the value $k = c_s(G)$ of a minimum s-cut, together with a complete k-intersection T of G, in $O(km \log n^2/m)$ time.

Computing a complete k-intersection T of G

Recall that a complete k-intersection T of G is a collection of k edge-disjoint spanning trees $T_1, \ldots, T_k$, such that each vertex $v \neq s$ has in-degree k and s has in-degree zero. Gabow's algorithm computes T in k iterations, where in the k'-th iteration ($k' = 1, \ldots, k$), it begins with a complete $(k'-1)$-intersection and tries to enlarge it so that it becomes a complete k'-intersection. To that end, it executes a *round robin* algorithm that maintains a forest $T_{k'}$ and tries to locate "joining" edges that will make $T_{k'}$ a spanning tree of G, while satisfying the invariant that $\delta_T^-(s) = 0$ and $\delta_T^-(v) \leq k'$ for all $v \neq s$. In the following, we call a vertex v *deficient* if $\delta_T^-(v) < k'$.

In more detail, during the k'-th iteration $T_{k'}$ is a forest of rooted trees, referred to as *f-trees*, where each f-tree F_z is rooted at its unique deficient vertex z. For $z \neq s$, $\delta_T^-(z) = k'-1$. An edge $e = (x, y)$ is *joining* if x and y are in different f-trees of $T_{k'}$. The round robin algorithm looks to enlarge $T_{k'}$ by one edge at a time, and simultaneously to increase the in-degree of a deficient vertex $z \neq s$ by one. To that end, it examines an edge e_1 in $E^-(z) \setminus E(T)$. If e_1 is joining then we are done. Otherwise, it adds e_1 in some T_i and looks for a joining edge in the fundamental cycle $C(e, T_i)$. To break the cycle, we can remove from T_i an edge $e_2 \in C(e, T_i)$. Then, we can add $e_2 = (u, v)$ to some other tree of T, or replace e_2 with a edge in $E^-(v) \setminus E(T)$. This pattern continues until a joining edge is found. The sequence of edges that leads to a joining edge is called an *augmenting path*. We define this notion formally below.

An ordered pair of edges e, f is called a *swap* if $f \in C(e, T_i)$ for some $T_i \in T$. To *execute the swap* is to replace f in T_i by e. A *partial augmenting path P from z* is a sequence of distinct edges $e_1, \ldots, e_l$, such that:

1. $e_1 \in E^-(z) \setminus E(T)$.
2. For each $i < l$ either
   (a) $e_{i+1} \in C(e_i, T_j)$, where T_j contains e_{i+1} but not e_i, or
   (b) $e_i, e_{i+1} \in E^-(v)$, for some vertex v, where T contains e_i but not e_{i+1}.
3. Executing all swaps of P (i.e., the pairs e_i, e_{i+1} of (2a)) gives a new collection of forests.

An *augmenting path P from z* is a partial augmenting path from z whose last edge e_l is joining for z. To *augment T along P* is to execute each swap of P and add e_l to $T_{k'}$. This increases the in-degree of z by one (so z is no longer deficient), while no other in-degree changes.

Each iteration is organized as a sequence of at most $\lceil \log n \rceil$ rounds. At the start of each round all f-trees are active except F_s. Then, we repeatedly choose an active f-tree F_z and search for an augmenting path from z. If no such path is found, then the algorithm terminates and reports an s-cut of cardinality $k' - 1$. Otherwise, we have found an augmenting path P from z and we augment T along P. Thus, we enlarge T by one edge and F_z is joined to another f-tree F_w. The resulting f-tree of $T_{k'}$ is rooted at w (since z is no longer deficient), and becomes inactive for the rest of the current round. Gabow showed that with an appropriate implementation, each round runs in $O(m)$ time. Furthermore, he showed that by organizing the search for augmenting paths carefully, all augmentations can be executed at the end of a round.

Computing a good s-arborescence

We now give an overview of how Gabow computes a good s-arborescence A of G in Step 2 of his arborescence packing algorithm. The algorithm maintains the following subgraphs of G: a partial s-arborescence A of G, a working graph $H = G - A$, and a complete $(k-1)$-intersection T for s on G. It uses the following key concept. An *enlarging path* consists of an edge $e \in E^+(A)$, and if $e \in T$, an augmenting path P for the $(k-1)$-intersection $T - e$ on $H - e$. Gabow shows that if $V(A) \neq V(G)$, then there is always an enlarging path.

The algorithm marks the edges that are known to belong in any complete $(k-1)$-intersection contained in H. It also maintains a set X of vertices such that each $u \in X$ has all edges of $E^+(u) \cap E^+(V(A))$ marked. The algorithm is divided into "periods", where each period enlarges either A or X. Initially, A contains only the start vertex s, X is empty, and all edges are unmarked. Then, we repeatedly apply the following procedure that locates an enlarging path, until it halts:

Period Step. If A is an arborescence, that is $V(A) = V(G)$ then halt. Otherwise, choose a vertex $u \in V(A) \setminus X$ and execute the *Edge Step*.

Edge Step. If all edges in $E^+(u) \cap E^+(V(A))$ are marked then add u to X and continue with the next *Period Step*. Otherwise, choose an unmarked edge $e \in E^+(u) \cap$

$E^+(V(A))$. If $e \notin E(T)$, then add e to A and continue with the next *Period Step*.

Search Step. At this point e belongs in T. Search for an augmenting path for the $(k-1)$-intersection $T - e$ in $H - e$. If the search is successful, then use the augmenting path to enlarge A and continue with the next *Period Step*. If the search is unsuccessful then mark e and go to *Edge Step*.

The correctness of this procedure is based on the following facts. Suppose $e \in E^+(u) \cap E^+(V(A))$ has no enlarging path, i.e., the *Search Step* was unsuccessful. Let L be the set of edges labelled in the search, and let e' be any edge in $E^+(u)$. Then, e belongs to any complete $(k-1)$-intersection contained in H, and no edge of L is in an enlarging path for e', with respect to the current A and T. This implies that for any $v \in X$, no edge of $E^+(v)$ has an enlarging path.

To make the search for enlarging paths fast, each unsuccessful search in a period contracts the edges in L that become labelled during an unsuccessful search for an edge $e \in E^+(u)$. The contraction is valid since, for each tree $T_i \in T$, $i = 1, 2 \ldots, k-1$, the edges in $L \cap (T_i - e)$ form a tree. Contracting $V(L)$ into a single vertex v results in a graph H' that has a complete $(k-1)$-intersection that contains all edges in $E^-_{H'}(v)$. Then, for any edges $e' \in E^+_{H'}(v)$, graphs H and H' have the same enlarging paths for e', and unsuccessful searches in $H - e'$ and $H - e$ label the same edges not in L.

To implement the above procedure efficiently, Gabow's algorithm performs the contractions implicitly. To that end, it maintains a partition of $V(G)$ into disjoint sets $S_1, \ldots, S_l$, such that each set S_j induces a tree in each $T_i \in T$. Each vertex is labelled with the name of the set that contains it and also, for each $i = 1, \ldots, k-1$, each set S_j is labelled with its root vertex in T_i. The *Search Step* for an edge $e = (u, w) \in T_i$ removes e from T and H, and searches for an augmenting path P from w. During this search, when a vertex v is reached, if $v \in S_j$ then the search continues from the root of S_j in T_i. If the search is successful, then we augment along P. Otherwise, when the search is unsuccessful, we add e back to H and T, and update the vertex partition $\{S_j\}$ by merging together all sets S_j that contain an end of an edge that was labelled during the search.

Gabow shows that this algorithm constructs an s-arborescence A in $O(kn^2)$ time; there are at most kn searches (at most one per edge), and at most $2n$ periods. The latter follows from the fact that a period enlarges A or X, and each set can be enlarged less than n times. Moreover, each period can be implemented to run in $O(kn)$ time.

Since we compute k arborescences, the total running time is $O(k^2n^2)$.

Practical speedup In order to speedup Gabow's algorithm in practice, we implemented the following simple heuristic. Let G be the current graph. We compute an s-arborescence A of G, e.g., by executing a depth-first search (DFS) from s, and test if A is good. To do that, it suffices to test if $c_{G-T}(s) = k - 1$. If this is the case, then we can keep A in the packing. Otherwise, we simply discard A, and compute a good s-arborescence of G using Gabow's algorithm. In either case, after we have computed a good s-arborescence A of G, we decrease k by one and repeat the procedure on $G - A$.

Despite its simplicity, the above modification provides significant speedups in practice, as the experimental results of Section 3.3 suggest. Moreover, we can immediately observe that the overall $O(k^2n^2)$ running time still holds for this variant of Gabow's algorithm.

3.3 Empirical Analysis

We implemented our algorithms in `C++`, using `g++ 7.5.0` with full optimization (flag -O4) to compile the code. The reported running times were measured on a GNU/Linux machine, with Ubuntu (18.04.6 LTS): a Dell Precision Tower 7820 server 64-bit NUMA machine with an Intel(R) Xeon(R) Gold 5220R processor and 192GB of RAM memory. The processor has 24.75MB of cache memory and 18 cores. In our experiments we did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the `high_resolution_clock` class of the standard library `chrono`, averaged over ten different runs.

Table 3.1 gives an overview of the algorithms we consider in our experimental study. We did not include the algorithm of Bhalgat et al. because some important details are omitted from the extended abstract of [37].[1]

We base our implementation of Gabow's arborescence packing algorithm on efficient implementations of Gabow's edge connectivity algorithm presented in [6]. Also, for the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), we report the running time of their stack-based implementations. We compare the stack-based against the

[1]We are unaware of a full version of [37].

Table 3.1: An overview of the algorithms considered in our experimental study. The bounds refer to a digraph G with minimum s-cut value $k = c_s(G)$, n vertices and m edges; $m \leq kn$ if G contains only the edges of a complete k-intersection.

Algorithm	Technique	Complexity	Ref.
Tarjan (Tar)	Test if a usable edge can be added via max-flow	$O(k^2m^2)$	[4]
	Run on a complete k-intersection	$O(k^4n^2)$	
Tong-Lawler (TL)	Graph splitting via min-cut	$O(k^2mn)$	[5]
	Run on a complete k-intersection	$O(k^3n^2)$	
Gabow (Gab)	Compute complete k'-intersections ($k' \leq k$) and enlarging paths	$O(k^2n^2)$	[1]

queue-based implementations in Appendix 3.3. For the experimental evaluation, we considered two types of graphs: (i) real-world directed graphs, augmented with additional edges in order to increase their edge-connectivity, and (ii) k-cores of undirected graphs.

Augmented graphs In our first experiment, we consider how the running time of each algorithm is affected as the minimum s-cut value $k = c_s(G)$ increases. To that end, we augment some real-word graphs as follows. Let G be an input strongly connected digraph. For a given parameter β, we create an augmented instance G_β of G by executing the following procedure. We go through the vertices of G and, for each vertex v, we add $\beta - \delta^-(v)$ edges directed to v if $\delta^-(v) < \beta$, where each added edge originates from a randomly chosen vertex. Then, we make a second pass over the vertices and, for each vertex v, we add $\beta - \delta^+(v)$ edges directed away from v if $\delta^+(v) < \beta$, where each added edge is directed to a randomly chosen vertex. Notice that the resulting graph has minimum degree $\delta \geq \beta$.

Table 3.2 reports the characteristics of the augmented graphs G_β produced by the above procedure for $\beta \in \{2, 4, 8, 16\}$. Here, we also give the number of edges (m') in a complete k intersection T of G (with respect to the start vertex s). In Table 3.3 we report the corresponding running times of each algorithm. The execution of an algorithm was terminated if it exceeded one hour. We also report the running times of two versions of Gabow's algorithm that computes a complete k intersection T of G: the standard version (Gab-EC), and a version that uses DFS to do a fast initialization of the forest $T_{k'}$ at the beginning of the k'-th iteration (Gab-EC-DFS). Both implementations are taken from [6]. For the algorithms of Tarjan (Tar), and of Tong and Lawler (TL), we report both the running time when the input is the original graph G (above) and a complete k intersection of G (below). In the latter case, the algorithms receive a complete k intersection T of G as input, and we do not account for the time required to compute T.

k-cores A k-core of an undirected graph G is a maximal subgraph of G such that $\delta(v) \geq k$ for all $v \in V(H)$. This concept is useful in the analysis of social networks [42] as well as in several other applications [43]. In this experiment, we use the k-core decomposition algorithm of the SNAP software library and tools [44], and use subgraphs of this decomposition as inputs, for various values of k. We transform each such undirected graph to a directed graph by orienting each edge in both directions. Table 3.4 reports the characteristics of the resulting graphs. In Table 3.5 we report the corresponding running times of each algorithm. Again, we terminated the execution of an algorithm if that exceeded one hour.

Stack-based vs queue-based implementations Tables 3.6 and 3.4 compare the stack-based against the queue-based implementation of the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), for augmented and k-core graphs, respectively.

Table 3.2: Characteristics of augmented graphs, resulting from some real-world graphs after inserting some edges; n is the number of vertices, m the number of edges; δ denotes the minimum vertex (in or out) degree, and $c_s(G)$ denotes the cardinality of the minimum s-cut; m' is the number of edges in a complete k intersection of G (for the start vertex s).

Graph	n	m	δ	$c_s(G)$	m'	type and source
enron-EC2	8271	151651	2	2	8270	email network [28]
enron-EC4	8271	162999	4	4	24810	
enron-EC8	8271	190618	8	8	57890	
enron-EC16	8271	253345	16	16	124050	
p2p-Gnutella25-EC2	5152	19765	2	2	10302	peer2peer network [28]
p2p-Gnutella25-EC4	5152	27565	4	4	20604	
p2p-Gnutella25-EC8	5152	50076	8	8	41208	
p2p-Gnutella25-EC16	5152	100669	16	16	82416	
rome99-EC2	3352	9869	2	2	6702	road network [40]
rome99-EC4	3352	15468	4	4	13404	
rome99-EC8	3352	31952	8	8	26808	
rome99-EC16	3352	65542	16	16	53616	
s38584-EC2	16310	42128	2	2	32618	VLSI circuit [41]
s38584-EC4	16310	80250	4	4	65236	
s38584-EC8	16310	160297	8	8	130472	
s38584-EC16	16310	323963	16	16	260944	
web-Stanford-EC2	150475	2334929	2	2	300948	web graph [28]
web-Stanford-EC4	150475	3307506	4	4	601896	
web-Stanford-EC8	150475	2379878	8	8	1203792	
web-Stanford-EC16	150475	3643794	16	16	2407584	

Table 3.3: Running times in seconds of the algorithms for the augmented graphs of Table 3.2. For the algorithms of Tarjan (Tar), and of Tong and Lawler (TL), we report the running time when the input is the original graph G (above) and a complete k intersection of G (below).

Graph	Gab-EC	Gab-EC-DFS	Tar	TL	Gab	Gab-DFS
enron-EC2	0.01	0.01	0.01	1.09	0.15	0.01
			0.01	0.04		
enron-EC4	0.02	0.01	36.34	118.17	9.81	0.05
			7.43	7.59		
enron-EC8	0.06	0.01	304.29	614.18	38.42	0.28
			95.58	131.91		
enron-EC16	0.17	0.02	1840.76	2782.78	130.58	1.50
			951.43	1319.77		
p2p-Gnutella25-EC2	0.01	0.01	1.19	3.90	1.39	0.02
			0.79	1.12		
p2p-Gnutella25-EC4	0.02	0.01	8.46	23.76	6.57	0.04
			7.05	14.34		
p2p-Gnutella25-EC8	0.04	0.01	55.21	124.36	21.63	0.17
			53.41	91.05		
p2p-Gnutella25-EC16	0.09	0.01	403.92	760.74	61.32	0.80
			450.80	629.79		
rome99-EC2	0.01	0.01	0.45	0.37	0.36	0.34
			0.35	0.24		
rome99-EC4	0.02	0.01	3.20	7.26	2.63	0.02
			3.19	5.56		
rome99-EC8	0.03	0.01	21.67	50.86	8.65	0.10
			21.62	37.24		
rome99-EC16	0.06	0.01	160.01	312.30	25.32	0.49
			156.29	241.95		
s38584-EC2	0.03	0.01	14.12	37.02	14.73	12.76
			8.88	23.45		
s38584-EC4	0.06	0.01	131.86	262.07	75.42	0.14
			117.79	98.85		
s38584-EC8	0.14	0.02	1012.85	2007.45	245.01	0.68
			878.79	1258.60		
s38584-EC16	0.34	0.04	>1h	>1h	717.20	3.29
			>1h	>1h		
web-Stanford-EC2	0.60	0.29	>1h	>1h	520.22	1.39
			2307.26	2350.42		
web-Stanford-EC4	1.45	0.69	>1h	>1h	>1h	5.40
			>1h	>1h		
web-Stanford-EC8	3.26	1.03	>1h	>1h	>1h	14.32
			>1h	>1h		
web-Stanford-EC16	7.73	3.28	>1h	>1h	>1h	70.98
			>1h	>1h		

Table 3.4: Characteristics of k-core graphs, extracted from real-world graphs in [28]; n is the number of vertices, m the number of edges; δ denotes the minimum vertex degree, and $c_s(G)$ denotes the cardinality of the minimum s-cut (which equal the edge-connectivity since the graphs are undirected); m' is the number of edges in a complete k intersection.

Graph	n	m	δ	$c_s(G)$	m'	type and source
facebook_combined-core02	3964	176318	2	2	7926	social circles from facebook
facebook_combined-core04	3754	175332	4	4	11258	
facebook_combined-core25	1366	118810	25	25	6824	
facebook_combined-core50	616	75246	50	30	19064	
Email-Enron-core09	5088	206472	9	9	45783	email network
Email-Enron-core10	4513	196594	10	10	45120	
Email-Enron-core16	2873	157506	16	16	45952	
Email-Enron-core18	2561	147332	18	18	46080	
CA-AstroPh-core18	5049	244004	18	18	90864	collaboration network
CA-AstroPh-core25	3202	175520	4	4	12804	
CA-AstroPh-core29	2441	139070	29	2	4880	
CA-AstroPh-core32	1926	112830	32	32	61600	
Gowalla_edges-core11	22742	851196	11	6	136443	social network
Gowalla_edges-core12	19938	791666	12	5	99684	
Gowalla_edges-core15	13833	639244	15	4	55328	
Gowalla_edges-core20	8161	456014	20	8	65280	

Table 3.5: Running times in seconds of the algorithms for the k-core graphs of Table 3.4. For the algorithms of Tarjan (Tar), and of Tong and Lawler (TL), we report the running time when the input is the original graph G (above) and a complete k intersection of G (below).

Graph	Gab-EC	Gab-EC-DFS	Tar	TL	Gab	Gab-DFS
facebook_combined-core02	0.01	0.01	2.47	8.71	2.00	0.01
			0.22	0.43		
facebook_combined-core04	0.01	0.01	6.88	20.32	2.88	0.02
			0.29	0.86		
facebook_combined-core25	0.01	0.01	3.97	6.82	0.95	0.02
			0.29	0.53		
facebook_combined-core50	0.04	0.01	43.41	47.76	4.59	0.55
			29.38	22.59		
Email-Enron-core09	0.03	0.01	109.84	156.29	30.06	0.17
			44.41	55.41		
Email-Enron-core10	0.03	0.01	107.00	101.79	28.60	0.17
			47.09	54.95		
Email-Enron-core16	0.04	0.01	136.26	170.74	20.65	0.35
			106.58	74.56		
Email-Enron-core18	0.04	0.01	151.68	177.93	19.80	0.42
			119.37	77.12		
CA-AstroPh-core18	0.25	0.01	894.66	1029.08	63.18	1.86
			716.73	412.42		
CA-AstroPh-core25	0.02	0.01	6.74	16.29	2.03	0.04
			1.83	1.16		
CA-AstroPh-core29	0.01	0.01	0.72	2.68	0.53	0.01
			0.08	0.21		
CA-AstroPh-core32	0.15	0.01	556.99	545.93	28.09	1.61
			537.45	270.58		
Gowalla_edges-core11	0.10	0.06	2086.57	2559.07	404.29	0.49
			567.03	394.37		
Gowalla_edges-core12	0.07	0.04	915.08	1094.83	219.33	0.31
			80.10	166.56		
Gowalla_edges-core15	0.04	0.02	197.84	321.91	71.65	0.15
			28.64	49.67		
Gowalla_edges-core20	0.05	0.02	327.64	412.93	68.57	0.32
			118.93	109.56		

Table 3.6: Running times in seconds of the stack-based and queue-based implementations of the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), for the augmented graphs of Table 3.2. We report the running time when the input is the original graph G (above) and a complete k intersection of G (below).

Graph	Tar		TL	
	stack	queue	stack	queue
enron-EC2	0.01	0.01	1.09	2.41
	0.01	0.01	0.04	0.48
enron-EC4	36.34	111.44	118.17	131.24
	7.23	8.18	7.59	6.73
enron-EC8	304.29	591.30	614.18	786.56
	95.58	136.25	131.91	118.14
enron-EC16	1840.76	2662.05	2782.78	3340.18
	951.43	1298.16	1319.77	1343.09
p2p-Gnutella25-EC2	1.19	3.66	3.90	4.41
	0.79	1.13	1.12	1.59
p2p-Gnutella25-EC4	8.46	22.00	23.76	23.61
	7.05	14.67	14.34	14.73
p2p-Gnutella25-EC8	55.21	123.56	124.36	119.05
	53.41	99.95	91.005	88.94
p2p-Gnutella25-EC16	403.92	744.87	760.74	768.10
	450.80	662.15	629.79	620.41
rome99-EC2	0.45	0.40	0.37	0.45
	0.35	0.32	0.24	0.21
rome99-EC4	3.20	8.12	7.26	6.98
	3.19	6.08	5.56	3.88
rome99-EC8	21.67	49.48	50.86	47.44
	21.62	37.11	37.24	35.71
rome99-EC16	160.01	306.58	312.30	320.36
	156.29	239.31	241.95	243.20
s38584-EC2	14.12	39.62	37.02	44.71
	8.88	25.96	23.45	26.93
s38584-EC4	131.86	286.66	262.07	282.07
	117.79	102.47	98.85	114.64
s38584-EC8	1012.85	2040.86	2007.45	1867.42
	878.22	1541.89	1258.60	1265.24
s38584-EC16	>1h	>1h	>1h	>1h
	>1h	>1h	>1h	>1h
web-Stanford-EC2	>1h	>1h	>1h	>1h
	2307.26	3100.36	2350.42	>1h
web-Stanford-EC4	>1h	>1h	>1h	>1h
	>1h	>1h	>1h	>1h

Table 3.7: Running times in seconds of the stack-based and queue-based implementations of the algorithms of Tarjan (Tar) and of Tong-Lawler (TL), for the k-core graphs of Table 3.4. We report the running time when the input is the original graph G (above) and a complete k intersection of G (below). The execution of an algorithm was terminated if it exceeded 1 hour.

Graph	Tar		TL	
	stack	queue	stack	queue
facebook_combined-core02	2.47	8.04	8.71	6.63
	0.22	0.37	0.43	0.36
facebook_combined-core04	6.88	18.83	20.32	11.51
	0.29	0.83	0.86	0.95
facebook_combined-core25	3.97	6.32	6.82	3.97
	0.29	0.51	0.53	0.31
facebook_combined-core50	43.41	44.34	47.76	8.54
	29.38	21.88	22.59	3.11
Email-Enron-core09	109.84	146.24	156.29	221.64
	44.41	55.64	55.41	47.98
Email-Enron-core10	107.00	94.40	101.79	116.07
	47.09	53.62	54.95	41.25
Email-Enron-core16	136.26	161.54	170.74	226.42
	106.58	90.43	74.56	94.21
Email-Enron-core18	151.68	168.80	177.93	240.51
	119.37	90.58	77.12	104.76
CA-AstroPh-core18	894.66	991.31	1029.08	619.59
	716.73	99.58	412.42	213.50
CA-AstroPh-core25	6.74	15.42	16.29	19.77
	1.83	1.66	1.16	0.75
CA-AstroPh-core29	0.72	2.50	2.68	3.02
	0.08	0.19	0.21	0.10
CA-AstroPh-core32	556.99	526.15	545.93	991.77
	537.45	427.66	270.58	349.73
Gowalla_edges-core11	2086.57	2468.99	2559.07	>1h
	567.03	455.23	394.37	403.34
Gowalla_edges-core12	915.08	1425.66	1094.83	2482.81
	80.10	187.11	166.56	169.45
Gowalla_edges-core15	197.84	498.99	321.91	780.18
	28.64	53.81	49.67	47.25
Gowalla_edges-core20	327.64	552.16	412.93	847.61
	118.93	122.30	109.56	105.48

3.4 Concluding Remarks

From the results, we observe that TL has overall the worst performance, even compared to Tar despite the inferior upper bound of the latter. Indeed, on average Tar runs twice as fast compared to TL. This is due to the overhead incurred in TL for splitting a graph G into two auxiliary graphs G_1 and G_2, and manipulating mappings from the vertex ids of G_1 and G_2 to those in G. Furthermore, we observe that TL runs consistently faster on the complete k intersection T of G compared to the original graph G. While this is expected since T has fewer edges, on the other hand we note that it may be easier to find good candidate edges to augment a partial arborescence if the graph contains some additional edges. The same observation holds for Tar as well, but here we see that in one instance (p2p-Gnutella25-EC16) the algorithm runs faster on G rather than on T. Moreover, we note that the executions of TL and Tar on T outperform Gab in some instances.

Next, we turn to the algorithms of Gabow. First, we verify that the edge-connectivity algorithms Gab-EC and Gab-EC-DFS are very effective. Regarding the arborescence packing algorithms, we first note that Tar and TL perform close to Gab when the edge-connectivity $c_G(s)$ is small (EC2 instances), but quickly become uncompetitive when the edge-connectivity increases. Overall, in our experiment, Gab was 50% faster than Tar on average. Finally, we note that Gab-DFS is faster than Gab by two orders of magnitude on most instances. This is due to the fact that our simple heuristic very often manages to construct a good arborescence by a simple DFS traversal.

Here too, we observe that Tar outperforms TL on most instances, but unlike the augmented graphs, their difference is marginal. Both TL and Tar run consistently faster on the complete k intersection T of G compared to the original graph G. Again, the executions of TL and Tar on T outperform Gab in some instances, but overall Gab is 50% faster. Also, our heuristic was very effective in this experiment as well, since Gab-DFS ran faster than Gab by two orders of magnitude. The code of the implemented algorithms is publicly available at https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/sakiskef/PackingArborescencesAlgorithm snapshot=820587054832b664c8f3cdf3d4a55d2e5fbaa084.

Chapter 4

Orientation of mixed graphs and related problems

A mixed graph G is a graph that consists of both undirected and directed edges. An orientation of G is formed by orienting all the undirected edges of G, i.e., converting each undirected edge $\{u, v\}$ into a directed edge that is either (u, v) or (v, u). The problem of finding an orientation of a mixed graph that makes it strongly connected is well understood and can be solved in linear time. Here we introduce the following orientation problem in mixed graphs. Given a mixed graph G, we wish to compute its maximal sets of vertices $C_1, C_2, \ldots, C_k$ with the property that by removing an edge e from G (directed or undirected), there is an orientation R_i of $G \setminus e$ such that all vertices in C_i are strongly connected in R_i. We discuss properties of those sets, and we show how to solve this problem in linear time by reducing it to the computation of the 2-edge twinless strongly connected components of a directed graph.

4.1 Theoretical framework and Related Work

A mixed graph G contains both undirected edges and directed edges. We denote an edge with endpoints u and v by $\{u, v\}$ if it is undirected, and by (u, v) if it is directed from u to v. An *orientation* R of G is formed by orienting all the undirected edges of G, i.e., converting each undirected edge $\{u, v\}$ into a directed edge that is either (u, v) or (v, u). Several (undirected or mixed) graph orientation problems have been studied in the literature, depending on the properties that we wish an orientation R of G to have. See, e.g., [45, 46, 47, 48]. An orientation R of G such that R is strongly connected is called a *strong orientation of* G. More generally, an orientation R of G such that R is k-edge strongly connected is called a *k-edge strong orientation of* G. Motivated by recent work in 2-edge strong connectivity in digraphs [13, 19, 20], we introduce the following strong connectivity orientation problem in mixed graphs. Given a mixed graph G, we wish to compute its maximal sets of vertices $C_1, C_2, \ldots, C_k$ with the property that for every $i \in \{1, \ldots, k\}$, and every edge e of G (directed or undirected), there is an orientation R of $G \setminus e$ such that all vertices of C_i are strongly connected in R. We refer to these maximal vertex sets as the *edge-resilient strongly orientable blocks* of G. See Figure 4.1. Note that when G contains only directed edges, then this definition coincides with the usual notion of 2-edge strong connectivity, i.e., each C_i is a 2-edge strongly connected component of G. We show how to solve this problem in linear time, by providing a linear-time algorithm for computing the 2-edge twinless strongly connected components [49], that we define next. Moreover, as a consequence of our algorithm, it follows that $\{C_1, \ldots, C_k\}$ is a partition of V.

We recall some concepts in directed graphs. A digraph $G = (V, E)$ is *strongly connected* if there is a directed path from each vertex to every other vertex. The *strongly connected components* (SCCs) of G are its maximal strongly connected subgraphs. Two vertices $u, v \in V$ are *strongly connected* if they belong to the same strongly connected component of G. We refer to a pair of antiparallel edges, (x, y) and (y, x), of G as *twin edges*. A digraph $G = (V, E)$ is *twinless strongly connected* if it contains a strongly connected spanning subgraph (V, E') without any pair of twin edges. The *twinless strongly connected components* (TSCCs) of G are its maximal twinless strongly connected subgraphs. Two vertices $u, v \in V$ are *twinless strongly connected* if they belong to the same twinless strongly connected component of G. Equivalently, u and v are twinless strongly connected if G contains a path from u to v and a path from v to u so that

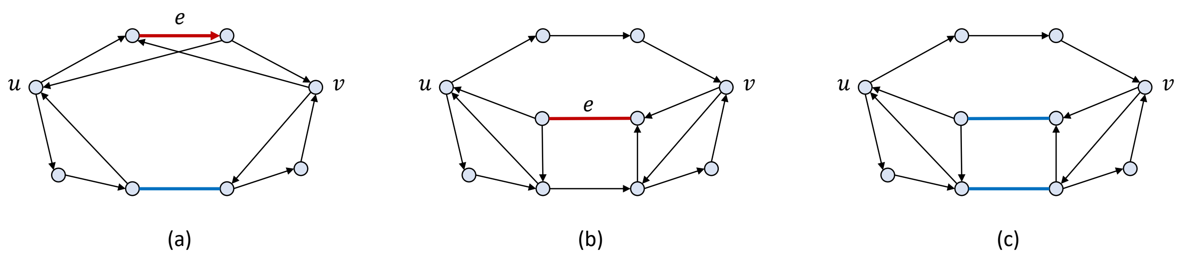

Figure 4.1: Examples illustrating the notion of edge-resilient strongly orientable blocks of a mixed graph G; undirected edges are shown in blue color and the specified edge e is shown in red color. (a)-(b) Vertices u and v are not in the same edge-resilient strongly orientable block of G. After the deletion of edge e (which is directed in (a) and undirected in (b)), there is no orientation of $G \setminus e$ such that u and v are strongly connected. (c) Here u and v are in the same edge-resilient strongly orientable block of G, since for any edge e, there is an orientation of $G \setminus e$ such that u and v are strongly connected.

the union of these two paths does not contain any pair of twin edges. Raghavan [50] provided a characterization of twinless strongly connected digraphs, and, based on this characterization, presented a linear-time algorithm for computing the TSCCs of a digraph.

An edge (resp., a vertex) of a digraph G is a *strong bridge* (resp., a *strong articulation point*) if its removal increases the number of strongly connected components. A strongly connected digraph G is 2-*edge strongly connected* if it has no strong bridges, and it is 2-*vertex strongly connected* if it has at least three vertices and no strong articulation points. Two vertices $u, v \in V$ are said to be 2-*edge strongly connected* (resp., 2-*vertex strongly connected*) if there are two edge-disjoint (resp., two internally vertex-disjoint) directed paths from u to v and two edge-disjoint (resp., two internally vertex-disjoint) directed paths from v to u (note that a path from u to v and a path from v to u need not be edge- or vertex-disjoint). Equivalently, by Menger's theorem [12] we have that u and v are 2-edge strongly connected if they remain in the same SCC after the deletion of any edge. A 2-*edge strongly connected component* (resp., 2-*vertex strongly connected component*) of a digraph $G = (V, E)$ is defined as a maximal subset $C \subseteq V$ such that every two vertices $u, v \in C$ are 2-edge strongly connected (resp., 2-vertex strongly connected). Also, note that the subgraph induced by C is not necessarily

2-edge strongly connected (resp., 2-vertex strongly connected).

The above notions extend naturally to the case of twinless strong connectivity. An edge $e \in E$ is a *twinless strong bridge* of G if the deletion of e increases the number of TSCCs of G. Similarly, a vertex $v \in V$ is a *twinless strong articulation point* of G if the deletion of v increases the number of TSCCs of G. A linear-time algorithm for detecting all twinless strong bridges can be derived by combining the linear-time algorithm of Italiano et al. [51] for computing all the strong bridges of a digraph, and a linear-time algorithm for computing all the edges that belong to a cut-pair in a 2-edge-connected undirected graph [52]. Georgiadis and Kosinas [52] showed that the computation of twinless strong articulation points reduces to the following problem in undirected graphs, which is also of independent interest: Given a 2-vertex-connected (biconnected) undirected graph H, find all vertices v that belong to a vertex-edge cut-pair, i.e., for which there exists an edge e such that $H \setminus \{v, e\}$ is not connected. Then, [52] presented a linear-time algorithm that not only finds all such vertices v, but also computes the number of vertex-edge cut-pairs of v (i.e., the number of edges e such that $H \setminus \{v, e\}$ is not connected). Alternatively, it is possible to compute the vertices that form a vertex-edge cut-pair by exploiting the structure of the triconnected components of H, represented by an SPQR tree [53, 54] of H.

A *2-edge twinless strongly connected component (2eTSCC) of G* is a maximal subset of vertices C such that any two vertices $u, v \in C$ are in the same TSCC of $G \setminus e$, for any edge e. Two vertices u and v are *2-edge twinless strongly connected* if they belong to the same 2eTSCC. See Figure 4.2. Jaberi [49] studied some properties of the 2-edge twinless strongly connected components and presented an $O(mn)$-time algorithm for a digraph with m edges and n vertices. We provide a linear-time algorithm that is based on two notions: (i) a collection of auxiliary graphs $\mathcal{H}$ that preserve the 2-edge twinless strongly connected components of G and, for any $H \in \mathcal{H}$, the SCCs of H after the deletion of any edge have a very simple structure, and (ii) a reduction to the problem of computing the connected components of an undirected graph after the deletion of certain vertex-edge cuts.

The notions of twinless strong connectivity and mixed graph orientations are indeed related and are motivated by several diverse applications, such as the design of road and telecommunication networks, the structural stability of buildings [55, 56, 57, 50], and the analysis of biological networks [58]. Given a mixed graph G, it is natural to ask whether it has an orientation so that the resulting directed graph is

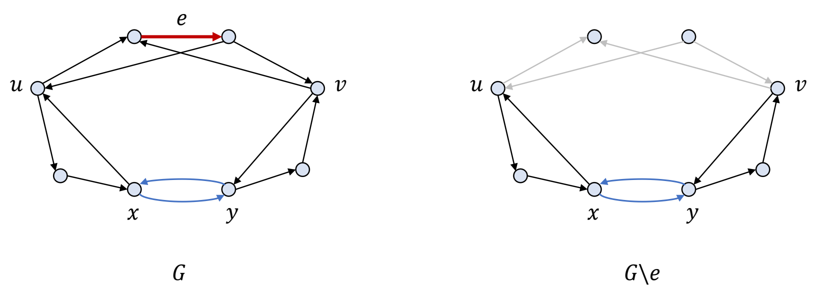

Figure 4.2: Vertices u and v of the strongly connected digraph G are 2-edge strongly connected but not 2-edge twinless strongly connected. The deletion of the edge e leaves u and v in a strongly connected subgraph that must contain both twin edges (x, y) and (y, x).

strongly connected [57, 59] or, more generally, k-edge strongly connected [60, 61, 62]. Raghavan [50] noted that testing whether a digraph G is twinless strongly connected is equivalent to testing whether a mixed graph has a strong orientation. We can also observe that the computation of the twinless strongly connected components is equivalent to the following generalization of strong orientations of a mixed graph: Given a mixed graph G, we wish to compute its maximal sets of vertices $C_1, C_2, \ldots, C_k$ with the property that for every $i \in \{1, \ldots, k\}$ there is an orientation R of G such that all vertices of C_i are strongly connected in R. We call those sets the *strongly orientable blocks* of G. Similarly, we show that the computation of the edge-resilient strongly orientable blocks reduces to the computation of the 2-edge twinless strongly connected components.

The computation of edge-resilient strongly orientable blocks is related to 2-edge strong orientations of mixed graphs in the following sense. A mixed graph G has a 2-edge strong orientation only if it consists of a single edge-resilient strongly orientable block. While finding a strong orientation of a mixed graph is well understood and can be solved in linear time [57, 59], computing a k-edge strong orientation for $k > 1$ seems much harder. Frank [60] gave a polynomial-time algorithm for this problem based on the concept of submodular flows. Faster algorithms were later presented by Gabow [61], and by Iwata and Kobayashi [62]. More efficient algorithms exist for computing a k-edge strong orientation of an undirected graph [63, 64, 65]. In particular, the algorithm of Bhalgat and Hariharan [63] runs in $\widetilde{O}(nk^4 + m)$ time[1],

[1]The notation $\widetilde{O}(\cdot)$ hides poly-logarithmic factors.

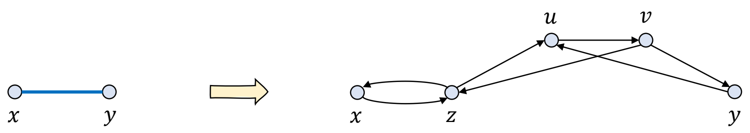

Figure 4.3: Replacing an undirected edge $\{x, y\}$ with a gadget.

for an undirected graph with n vertices and m edges.

4.2 Reduction to the computation of the 2-edge twinless strongly connected components

Let G be a mixed graph. By *splitting* a directed edge (x, y) of a graph G, we mean that we remove (x, y) from G, and we introduce a new auxiliary vertex z and two edges $(x, z), (z, y)$. By *replacing with a gadget* an undirected edge $\{x, y\}$ of a graph G, we mean that we remove $\{x, y\}$ from G, and we introduce three new auxiliary vertices z, u, v and the edges $(x, z), (z, x), (z, u), (u, v), (v, y), (y, u), (v, z)$. See Figure 4.3. Note that the definition of gadget replacement is not symmetric for x, y, but this does not affect the correctness of our approach. (We can assume an arbitrary order of the vertices of G, so when we perform the gadget replacements of the undirected edges, the role of each vertex in the gadgets is predetermined.) We refer to a non-auxiliary vertex as an *ordinary vertex*. Also, we call (u, v) the *critical edge* of the gadget. The idea of using this gadget is twofold. Firstly, by removing the critical edge, we simulate the operation of removing $\{x, y\}$ from the original graph. Secondly, if we remove an edge that does not belong to the gadget, then the only paths from x to y and from y to x inside the gadget must use the pair of twin edges (x, z) and (z, x). These properties are useful in order to establish a correspondence between orientations and twinless strong connectivity for our applications.

Now we can reduce the computation of the edge-resilient strongly orientable blocks of a mixed graph to the computation of the 2eTSCC of a digraph. As a warm-up, we show how to compute the strongly orientable blocks of a mixed graph via a reduction to the computation of the TSCCs of a digraph. This is achieved by Algorithm 4.1, whose correctness follows easily from known results. As a consequence of this method for computing $C_1, \ldots, C_k$, we can see that $\{C_1, \ldots, C_k\}$ is a partition of V. Furthermore,

$C_1, \ldots, C_k$ satisfy the stronger property, that there is an orientation R of G such that, for every $i \in \{1, \ldots, k\}$, $R[C_i]$ is strongly connected.

Algorithm 4.1: A linear-time algorithm for computing the strongly orientable blocks of a mixed graph G

```
1 split every directed edge of $G$;
2 replace every undirected edge $\{x, y\}$ of $G$ with a pair of twin edges
    $(x, y), (y, x)$;
3 compute the TSCCs $T_1, \ldots, T_k$ of the resulting graph;
4 **return** the sets of ordinary vertices of $T_1, \ldots, T_k$;
```

Proposition 4.1. *Algorithm 4.1 is correct.*

Proof. Let G be the input graph, let G' be the graph derived from G after we have performed steps 1 and 2, and let C be one of the sets returned by the algorithm. First we will show that there is an orientation R of G such that all vertices of C are strongly connected in R. Then we will show that $C \cup \{s\}$ does not have this property for any $s \in V(G) \setminus C$.

Since C is a twinless strongly connected component of G', by [50] we know that there is a subgraph G_0 of G' that contains no pair of twin edges and is such that C is strongly connected in G_0. Let $\{x, y\}$ be an undirected edge of G, and let $(x, y), (y, x)$ be the pair of twin edges of G' that replaced $\{x, y\}$ in Step 2. Assume w.l.o.g. that G_0 contains (x, y). Then we orient $\{x, y\}$ in G as (x, y). We do this for every undirected edge of G, and let R be the resulting orientation. Now it is not difficult to see that all vertices of C are strongly connected in R, and so C is contained within a strongly orientable block of G.

To establish the maximality of C, let s be a vertex in $V(G) \setminus C$. This means that s is not twinless strongly connected with the vertices of C in G'. Then we have that either (1) s and (the vertices of) C are not strongly connected, or (2) there is a pair of twin edges $(x, y), (y, x)$ of G', such that for every path P from s to (a vertex of) C in G' and every path Q from (a vertex of) C to s in G', we have $(x, y) \in P$ and $(y, x) \in Q$. In case (1) we have that s is not strongly connected with C in G, even if we allow every undirected edge to be replaced by a pair of twin edges. In case (2), let $\{x, y\}$ be the undirected edge of G that was replaced in G' with the pair of twin edges $(x, y), (y, x)$. (Notice that, due to Step 1, all pairs of twin edges in G' are due to Step 2.)Then we have that, even if we replace every undirected edge of G - except $\{x, y\}$ - with a pair of twin edges, then we still have to replace $\{x, y\}$ with the pair of twin edges $(x, y), (y, x)$ in order to have s strongly connected with C. In any case, we have that there is no orientation R of G such that s is strongly connected with C in R.

Notice that this argument also shows that there is no orientation of G such that a vertex $t \in C$ becomes strongly connected with a vertex $s \in V(G) \setminus C$. Thus, the sets returned by Algorithm 4.1 are all the strongly orientable blocks of G. □

Now Algorithm 4.2 shows how we can compute all the edge-resilient strongly orientable blocks $C_1, \ldots, C_k$ of a mixed graph. As a consequence of this method for computing $C_1, \ldots, C_k$, we can see that the edge-resilient strongly orientable blocks partition the vertex set V.

Algorithm 4.2: A linear-time algorithm for computing the edge-resilient strongly orientable blocks of a mixed graph G

```
1 split every directed edge of $G$
2 replace every undirected edge of $G$ with a gadget
3 compute the 2eTSCCs $T_1, \ldots, T_k$ of the resulting graph
4 return the sets of ordinary vertices of $T_1, \ldots, T_k$
```

Proposition 4.2. *Algorithm 4.2 is correct.*

Proof. Let G be the input graph, let G' be the graph derived from G after we have performed steps 1 and 2, and let C be one of the sets returned by the algorithm. First, we will show that, for every edge e, there is an orientation R of $G \setminus e$ such that all vertices of C are strongly connected in R. Then we will show that $C \cup \{s\}$ does not have this property for any $s \in V(G) \setminus C$.

Now let e be an edge of G. Suppose first that e is a directed edge of the form (x, y). Let $(x, z), (z, y)$ be the two edges of G' into e was split. Then we have that all vertices of C are twinless strongly connected in $G' \setminus (x, z)$. By [50], this implies that there is a subgraph G_0 of $G' \setminus (x, z)$ that contains no pair of twin edges and is such that all vertices of C are strongly connected in it. Let $\{x', y'\}$ be an undirected edge of G, and assume w.l.o.g. that $(x', z'), (z', x')$ is the pair of twin edges in the gadget of G' that replaced $\{x', y'\}$. Assume w.l.o.g. that G_0 contains (x', z'). Then we orient

$\{x', y'\}$ in G as (x', y'). We do this for every undirected edge of G, and let R be the resulting orientation of $G \setminus (x, y)$. Then it is not difficult to see that all vertices of C are strongly connected in R. Now suppose that e is an undirected edge of the form $\{x, y\}$. Let (u, v) be the critical edge of the gadget of G' that replaced the undirected edge $\{x, y\}$ of G. Then we have that all vertices of C are twinless strongly connected in $G' \setminus (u, v)$. Now let G_0 and R be defined similarly as above. Then it is not difficult to see that all vertices of C are strongly connected in R.

To establish the maximality of C, let s be a vertex in $V(G) \setminus C$. This means that s is not 2-edge twinless strongly connected with the vertices of C in G', and so there is an edge e of G' such that s is not in the same twinless strongly connected component of $G' \setminus e$ that contains C. Assume first that e is either (x, z) or (z, y), where $(x, z), (z, y)$ is the pair of edges of G' into which a directed edge (x, y) of G was split.Then we have that either (1) s and (the vertices of) C are not strongly connected, or (2) there is a pair of twin edges $(x', z'), (z', x')$ of G', such that for every path P from s to (a vertex of) C in $G' \setminus e$ and every path Q from (a vertex of) C to s in $G' \setminus e$, we have $(x', z') \in P$ and $(z', x') \in Q$. In case (1) we have that s is not strongly connected with C in $G \setminus (x, y)$, even if we allow every undirected edge to be replaced by a pair of twin edges. In case (2), let $\{x', y'\}$ be the undirected edge of G that was replaced with the gadget of G' that contains the pair of edges $(x', z'), (z', x')$. Then we have that, even if we replace every undirected edge of G - except $\{x', y'\}$ - with a pair of twin edges, then we still have to replace $\{x', y'\}$ with the pair of twin edges $(x', y'), (y', x')$ in order to have s strongly connected with C in $G \setminus (x, y)$. In any case, we have that there is no orientation R of $G \setminus (x, y)$ such that s is strongly connected with C in R. Now, if e is an edge of a gadget that replaced an undirected edge $\{x, y\}$ of G, then with a similar argument we can show that there is no orientation R of $G \setminus \{x, y\}$ such that s is strongly connected with C in R.

Notice that this argument also shows that if t is a vertex in C and s is a vertex in $V(G) \setminus C$, then there is an edge e (directed or undirected) such that there is no orientation of $G \setminus e$ that makes s and t strongly connected. Thus, the sets returned by Algorithm 4.2 are all the edge-resilient strongly orientable blocks of G. $\square$

Our goal in the following sections is to provide a linear-time implementation of Algorithm 4.2. To achieve this, it suffices to provide a linear-tine algorithm for computing the 2-edge twinless strongly connected components of a digraph.

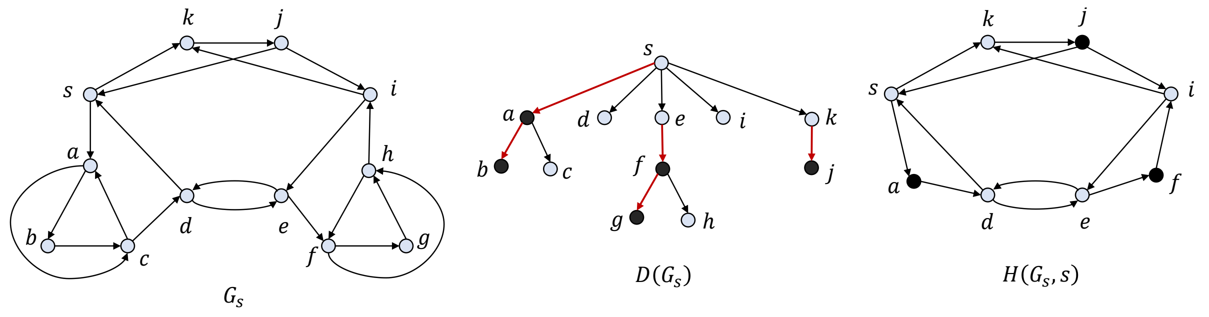

Figure 4.4: A flow graph G_s with start vertex s, its dominator tree $D(G_s)$, and the auxiliary graph $H(G_s, s)$. The bridges of G_s are colored red in $D(G_s)$. Marked vertices and auxiliary vertices are colored black in $D(G_s)$ and $H(G_s, s)$, respectively.

4.3 Connectivity-preserving auxiliary graphs

In this section, we describe how to construct a set of auxiliary graphs that preserve the 2-edge twinless strongly connected components of a twinless strongly connected digraph, and moreover have the property that their strongly connected components after the deletion of any edge have a very simple structure. We base our construction on the auxiliary graphs defined in [13] for computing the 2-edge strongly connected components of a digraph, and perform additional operations in order to achieve the desired properties. We note that a similar construction was given in [66] to derive auxiliary graphs (referred to as 2-connectivity-light graphs) that enable the fast computation of the 3-edge strongly connected components of a digraph. Still, we cannot apply directly the construction of [66], since we also need to maintain twinless strong connectivity.

4.3.1 Flow graphs and dominator trees

A *flow graph* is a directed graph with a distinguished *start vertex* s such that every vertex is reachable from s. For a digraph G, we use the notation G_s in order to emphasize the fact that we consider G as a flow graph with source s. Let $G = (V, E)$ be a strongly connected graph. We will let s be a fixed but arbitrary start vertex of G. Since G is strongly connected, all vertices are reachable from s and reach s, so we can refer to the flow graphs G_s and G_s^R.

Let G_s be a flow graph with start vertex s. A vertex u is a *dominator* of a vertex v (u *dominates* v) if every path from s to v in G_s contains u; u is a *proper dominator* of v if u

dominates v and $u \neq v$. The dominator relation is reflexive and transitive. Its transitive reduction is a rooted tree, the *dominator tree* $D(G_s)$: u dominates v if and only if u is an ancestor of v in $D(G_s)$. See Figure 4.4. For every vertex $x \neq s$ of G_s, $d(x)$ is the immediate dominator of x in G_s (i.e., the parent of x in $D(G_s)$). For every vertex r of G_s, we let $D(r)$ denote the subtree of $D(G_s)$ rooted at r. Lengauer and Tarjan [67] presented an algorithm for computing dominators in $O(m\alpha(m,n))$ time for a flow graph with n vertices and m edges, where α is a functional inverse of Ackermann's function [68]. Subsequently, several linear-time algorithms were discovered [69, 70, 71, 72].

An edge (u,v) is a *bridge* of a flow graph G_s if all paths from s to v include (u,v).[2] The following properties were proved in [51].

Property 4.3. ([51]) *Let s be an arbitrary start vertex of G. An edge $e = (u,v)$ is a strong bridge of G if and only if it is a bridge of G_s, in which case $u = d(v)$, or a bridge of G_s^R, in which case $v = d^R(u)$, or both.*

Let G_s be a strongly connected digraph. For every bridge (x,y) of G_s, we say that y is a *marked* vertex. (Notice that s cannot be marked.) Property 4.3 implies that the bridges of G_s induce a decomposition of $D(G_s)$ into rooted subtrees. More precisely, for every bridge (x,y) of G_s, we remove the edge (x,y) from $D(G_s)$. (By Property 4.3, this is indeed an edge of $D(G_s)$.) Thus we have partitioned $D(G_s)$ into subtrees. Every tree T in this decomposition inherits the parent relation from $D(G_s)$, and thus it is rooted at a vertex r. We denote T as $T(r)$ to emphasize the fact that the root of T is r. Observe that the root r of a tree $T(r)$ is either a marked vertex or s. Conversely, for every vertex r that is either marked or s, there is a tree $T(r)$.

In our constructions, we will use the next lemma, which follows from well-known properties of the dominator tree. (See, e.g., [19, 13].)

Lemma 4.1. *Let G_s be a strongly connected digraph, and let r be a marked vertex of G_s. Then we have the following.*

1. *For every vertex x in $D(r)$, there is a path from r to x in G_s that uses only vertices from $D(r)$.*
2. *For every vertex x in $D(s) \setminus D(r)$, there is a path from s to x in G_s that uses only vertices from $D(s) \setminus D(r)$.*

[2]Throughout the paper, to avoid confusion we use consistently the term *bridge* to refer to a bridge of a flow graph and the term *strong bridge* to refer to a strong bridge in the original graph.

3. *For every vertex x in $D(s) \setminus D(r)$, there is a path from x to $d(r)$ in G_s that uses only vertices from $D(s) \setminus D(r)$.*

4. *If (x, y) is an edge of G_s such that $x \in D(s) \setminus D(r)$ and $y \in D(r)$, then $(x, y) = (d(r), r)$.*

4.3.2 Construction of auxiliary graphs

Now let G_s be a strongly connected digraph, and let r be either a marked vertex of G_s or s. We define the *auxiliary* graph $H(G_s, r)$ as follows. In G_s we shrink every $D(z)$, where z is a marked vertex such that $d(z) \in T(r)$ into z. Also, if $r \neq s$, we shrink $D(s) \setminus D(r)$ into $d(r)$. During those shrinkings, we maintain all edges, except for self-loops. Also, in [13] multiple edges are converted into single edges. Here, multiple edges are converted into double edges, in order to avoid introducing new strong bridges in the auxiliary graphs. The resulting graph is $H(G_s, r)$. We consider $H(G_s, r)$ as a flow graph with start vertex r. Notice that it consists of the subgraph of G_s induced by $T(r)$, plus some extra vertices and edges. To be specific, the vertex set of $H(G_s, r)$ consists of the vertices of $T(r)$, plus all marked vertices z of G_s such that $d(z) \in T(r)$, plus $d(r)$ if $r \neq s$. The vertices of $T(r)$ are called *ordinary* in $H(G_s, r)$. The vertices of $H(G_s, r) \setminus T(r)$ are called *auxiliary* in $H(G_s, r)$. In particular, if $r \neq s$, $d(r)$ is called the *critical* vertex of $H(G_s, r)$, and $(d(r), r)$ is called the *critical* edge of $H(G_s, r)$. (Thus, $H(G_s, s)$ is the only auxiliary graph of G_s that has no critical vertex and no critical edge.)

Now we will give a more precise description of the edges of $H(G_s, r)$. First, let $z_1, \dots, z_k$ be the collection of all marked vertices of G_s such that $d(z_i) \in T(r)$, for $i \in \{1, \dots, k\}$. Then we can partition the vertex set of G_s into $T(r)$, $D(z_1), \dots, D(z_k)$, and $D(s) \setminus D(r)$, if $r \neq s$. Now, due to Lemma 4.1, an edge (x, y) of G_s satisfies one of:

(i) $x \in T(r)$ and $y \in T(r)$

(ii) $x \in T(r)$ and $y \in D(s) \setminus D(r)$ (assuming that $r \neq s$)

(iii) $x \in D(z)$ and $y \in D(z)$, for some $z \in \{z_1, \dots, z_k\}$

(iv) $x \in D(z)$ and $y \in T(r)$, for some $z \in \{z_1, \dots, z_k\}$

(v) $x \in D(z)$ and $y \in D(s) \setminus D(r)$, for some $z \in \{z_1, \dots, z_k\}$ (assuming that $r \neq s$)

(vi) $(x, y) = (d(z), z)$, for some $z \in \{z_1, \ldots, z_k\}$

(vii) $x \in D(s) \setminus D(r)$ and $y \in D(s) \setminus D(r)$ (assuming that $r \neq s$)

(viii) $(x, y) = (d(r), r)$ (assuming that $r \neq s$)

Each of those cases provides the *corresponding* edge of $H(G_s, r)$ as follows. $(i) \mapsto (x, y)$, $(ii) \mapsto (x, d(r))$, $(iii) \mapsto$ *none*. $(iv) \mapsto (z, y)$. $(v) \mapsto (z, d(r))$. $(vi) \mapsto (d(z), z)$. $(vii) \mapsto$ *none*. $(viii) \mapsto (d(r), r)$. For every edge $\tilde{e}$ of $H(G_s, r)$ we say that an edge e of G_s *corresponds* to $\tilde{e}$ if $\tilde{e}$ is the edge corresponding to e. (Notice that the correspondences (i), (vi) and $(viii)$ between edges of G_s and edges of $H(G_s, r)$ are bijective.) Using this correspondence, for every path P in G_s whose endpoints are vertices in $H(G_s, r)$, there is a *corresponding* path P_H in $H(G_s, r)$ that has the same endpoints. The following Lemma summarizes the properties of the corresponding path and establishes a kind of inverse to it.

Lemma 4.2. *Let G_s be a strongly connected digraph, and let r be either a marked vertex of G_s or s. Let P be a path in G_s whose endpoints lie in $H(G_s, r)$. Then there is a corresponding path P_H in $H(G_s, r)$ with the same endpoints. In particular, every occurrence of an edge $(x, y) \in P$, such that either* (1) *x and y are ordinary vertices in $H(G_s, r)$, or* (2) *or $(x, y) = (d(z), z)$, where z is a marked vertex of G_s with $d(z) \in T(r)$, or* (3) *$(x, y) = (d(r), r)$ (if $r \neq s$), is maintained in P_H. Conversely, for every path $\tilde{P}$ in $H(G_s, r)$, there is a path P in G_s such that $P_H = \tilde{P}$.*

Proof. To construct P_H we simply apply the above correspondence, which maps edges of G_s to edges of $H(G_s, r)$, to every edge of P. (Notice that some edges of P may be nullified.) It is a simple matter to verify that P_H is indeed a path, and it also satisfies the claimed property. Now, given a path $\tilde{P}$, we will construct a path P in G_s such that $P_H = \tilde{P}$. This is done essentially by maintaining some of the edges of $\tilde{P}$ and by expanding some of its edges to paths of G_s. In particular, every edge (x, y) of $\tilde{P}$ satisfying (1), (2), or (3), is maintained in P. Now, suppose that $\tilde{P}$ contains $(d(z), z)$ such that z is a marked vertex of G_s with $d(z) \in T(r)$. Then, the successor of $(d(z), z)$ in $\tilde{P}$, if it exists, is either an edge of the form (z, y), where $y \in T(r)$, or $(z, d(r))$ (assuming that $r \neq s$). In the first case, let (x, y) be an edge of G_s that corresponds to (z, y) in $H(G_s, r)$. Then we have that x is a descendant of z in $D(G_s)$, and so, by Lemma 4.1, there is a path in G_s from z to x that uses only vertices from $D(z)$. Thus we replace (z, y) in $\tilde{P}$ with precisely this path. In the second case, let (x, y) be an edge of G_s that corresponds to $(z, d(r))$ in $H(G_s, r)$. Then we have $x \in D(z)$ and $y \in D(s) \setminus D(r)$. Now, by Lemma 4.1, there is a path P_1 from z to x that uses only vertices from $D(z)$. Furthermore, by the same Lemma, there is a path P_2 from y to $d(r)$ that uses only vertices from $D(s) \setminus D(r)$. Thus we replace $(z, d(r))$ in $\tilde{P}$ with $P_1 + (x, y) + P_2$. Now suppose that $r \neq s$ and $\tilde{P}$ contains $(d(r), r)$. Then, the immediate predecessor of $(d(r), r)$ in $\tilde{P}$, if it exists, is either an edge of the form $(x, d(r))$, where $x \in T(r)$, or $(z, d(r))$, where z in a marked vertex of G_s with $d(z) \in T(r)$. Now we can treat those cases as before, using Lemma 4.1 in order to expand the corresponding edges of $(x, d(r))$ or $(z, d(r))$ in G_s into paths of G_s. $\square$

An immediate consequence of Lemma 4.2 is the following.

Corollary 4.1. *Let G_s be a strongly connected digraph, and let r be either a marked vertex of G_s or s. Then $H(G_s, r)$ is strongly connected.*

Furthermore, [13] provided the following result.

Theorem 4.4. ([13]) *Let G_s be a strongly connected digraph, and let $r_1, \ldots, r_k$ be the marked vertices of G_s.*

(i) For any two vertices x and y of G_s, $x \overset{G_s}{\leftrightarrow}_{2e} y$ if and only if there is a vertex r (a marked vertex of G_s or s), such that x and y are both ordinary vertices of $H(G_s, r)$ and $x \overset{H(G_s,r)}{\leftrightarrow}_{2e} y$.

(ii) The collection $H(G_s, s)$, $H(G_s, r_1), \ldots, H(G_s, r_k)$ of all the auxiliary graphs of G_s can be computed in linear time.

We provide the analogous result for 2-edge twinless strong connectivity.

Proposition 4.5. *Let x, y be two vertices of a strongly connected digraph G_s. Then $x \overset{G_s}{\leftrightarrow}_{2et} y$ if and only if there is a vertex r (a marked vertex of G_s or s), such that x and y are both ordinary vertices of $H(G_s, r)$ and $x \overset{H(G_s,r)}{\leftrightarrow}_{2et} y$.*

Proof. $(\Rightarrow)$ Suppose that $x \leftrightarrow_{2et} y$ in G_s. Then x and y are not separated by any bridge of G_s, and so they both belong to the same graph $H(G_s, r)$, for some r (a marked vertex of G_s or s), as ordinary vertices. Now let $e = (u, v)$ be an edge of $H(G_s, r)$. We will prove that $x \leftrightarrow_t y$ in $H(G_s, r) \setminus e$. We distinguish five different cases for e.

(1) u and v are both ordinary vertices of $H(G_s, r)$.

(2) v is an auxiliary vertex of $H(G_s, r)$, but not $d(r)$ (if $r \neq s$).

(3) u is an auxiliary vertex of $H(G_s, r)$, but not $d(r)$ (if $r \neq s$).

(4) $(u, v) = (d(r), r)$ (if $r \neq s$).

(5) $v = d(r)$ (if $r \neq s$).

We will now show how to handle cases (1) to (5).

(1) Let u and v be ordinary vertices of $H(G_s, r)$. Then $e = (u, v)$ is an edge of G_s. Since $x \leftrightarrow_{2et} y$ in G_s, there exist two paths P and Q in $G_s \setminus e$, such that P starts from x and ends in y, Q starts from y and ends in x, and for every edge $(z, w) \in P$ we have that $(w, z) \notin Q$. Now consider the induced paths P_H and Q_H of P and Q, respectively, in $H(G_s, r)$. These have the property that P_H starts from x and ends in y, and Q_H starts from y and ends in x. Now we will show that there are subpaths P'_H and Q'_H of P_H and Q_H, respectively, that have the same endpoints as P_H and Q_H, respectively, and the property that for every $(z, w) \in P'_H$ we have that $(w, z) \notin Q'_H$. We will construct P'_H and Q'_H by eliminating some segments of P_H and Q_H, respectively, that form loops. So let (z, w) be an edge of P_H with the property that $(w, z) \in Q_H$. Then we have that z and w cannot both be ordinary vertices of $H(G_s, r)$ (for otherwise we would have that $(z, w) \in P$ and $(w, z) \in Q$). Thus, let us take first the case that $(z, w) = (d(h), h)$, where h is an auxiliary vertex of $H(G_s, r)$. Then we have that $(h, d(h)) \in Q_H$. But, in order for Q_H to use the edge $(h, d(h))$, it must first pass from the bridge $(d(h), h)$. Thus we can simply discard the segment $(d(h), h), (h, d(h))$ from Q_H, as this contributes nothing for Q_H in order to reach y from x. Now let us take the case that z is an auxiliary vertex of $H(G_s, r)$, but $(z, w) \neq (d(r), r)$ (if $r \neq s$). If $w = d(z)$ we work as before. But this supposition exhausts this case since (w, z) is an edge of $H(G_s, r)$, and so by Lemma 4.1 we have that w must necessarily be $d(z)$. Now let us take the case that $(z, w) = (d(r), r)$ (assuming that $r \neq s$). Then we have that $(r, d(r)) \in Q_H$. But then, in order for Q_H to end at an ordinary vertex, it must immediately afterwards use the bridge $(d(r), r)$. Thus we can eliminate the segment $(r, d(r)), (d(r), r)$ from Q_H. Finally, the case $(z, w) = (r, d(r))$ is treated similarly as $(z, w) = (d(r), r)$. This shows that $x \leftrightarrow_t y$ in $H(G_s, r) \setminus e$.

In cases (2) and (4), the challenge in proving $x \leftrightarrow_t y$ in $H(G_s, r) \setminus e$ is the same as that in (1), since e happens to be an edge of G_s. Thus, cases (1), (2) and (4) have essentially the same proof. So we are left to consider cases (3) and (5).

(3) Let's assume that u is an auxiliary vertex of $H(G_s, r)$ with $u \neq d(r)$ (if $r \neq s$). Then, since $x \leftrightarrow_{2et} y$ in G_s, there exist two paths P and Q in $G_s \setminus (d(u), u)$, such that P starts from x and ends in y, Q starts from y and ends in x, and for every edge $(z, w) \in P$ we have that $(w, z) \notin Q$. Now consider the induced paths P_H and Q_H of P and Q, respectively, in $H(G_s, r)$. By Lemma 4.2, these paths do not pass from the bridge $(d(u), u)$. Therefore, they also cannot use the edge (u, v). Now we can reason as in the proof for case (1), to show that there are subpaths P'_H and Q'_H of P_H and G_H, respectively, with the same endpoints, respectively, and the property that for every $(z, w) \in P'_H$ we have that $(w, z) \notin Q'_H$. This shows that $x \leftrightarrow_t y$ in $H(G_s, r) \setminus (x, y)$.

(5) Let's assume that $r \neq s$ and $v = d(r)$. Then, since $x \leftrightarrow_{2et} y$ in G_s, there exist two paths P and Q in $G_s \setminus (d(r), r)$, such that P starts from x and ends in y, Q starts from y and ends in x, and for every edge $(z, w) \in P$ we have that $(w, z) \notin Q$. Now consider the induced paths P_H and Q_H of P and Q, respectively, in $H(G_s, r)$. By Lemma 4.2, these paths do not pass from the bridge $(d(r), r)$. Therefore, they also cannot use the edge (v, u). Now we can reason as in the proof for case (1), to show that there are subpaths P'_H and Q'_H of P_H and G_H, respectively, with the same endpoints, respectively, and the property that for every $(z, w) \in P'_H$ we have that $(w, z) \notin Q'_H$. This shows that $x \leftrightarrow_t y$ in $H(G_s, r) \setminus (x, y)$.

Now, since cases (1) to (5) exhaust all possibilities for e, we infer that $x \leftrightarrow_t y$ in $H(G_s, r) \setminus e$. Due to the generality of e, we conclude that $x \leftrightarrow_{2et} y$ in $H(G_s, r)$.

($\Leftarrow$) Suppose that $x \not\leftrightarrow_{2et} y$ in G_s. If there is no r such that x and y are both in $H(G_s, r)$ as ordinary vertices, then we are done. So let us assume that there is an r such that x and y are both in $H(G_s, r)$ as ordinary vertices. By [13], we have that if $x \not\leftrightarrow_{2e} y$ in G_s, then also $x \not\leftrightarrow_{2e} y$ in $H(G_s, r)$, and therefore $x \not\leftrightarrow_{2et} y$ in $H(G_s, r)$.Now let's assume that $x \leftrightarrow_{2e} y$ in G_s. Then $x \not\leftrightarrow_{2et} y$ in G_s implies that there is an edge e in G_s such that, for every two paths P and Q in $G_s \setminus e$ such that P starts from x and ends in y, and Q starts from y and ends in x, we have that there exists an edge $(u, v) \in P$ such that $(v, u) \in Q$. We will strengthen this by showing that there exists an edge $(u, v) \in P$ such that $(v, u) \in Q$ and u, v are ordinary vertices of $H(G_s, r)$. We will achieve this by showing that, if either u or v is not an ordinary vertex of $H(G_s, r)$, then we can replace either the part of P that contains (u, v) with one that doesn't contain it, or the part of Q that contains (v, u) with one that doesn't contain it. (So that, finally, since the resulting paths must have the property that they share a pair of antiparallel edges, this pair cannot but join ordinary vertices of $H(G_s, r)$.) We will distinguish three cases, depending on the location of u in $D(G_s)$. (1) Let's assume that u is a descendant in $D(G_s)$ of an auxiliary vertex z of $H(G_s,r)$, with $z \neq d(r)$ (if $r \neq s$). Let's assume first that $(u,v) = (z,d(z))$, where z is an auxiliary vertex in $H(G_s,r)$. Since P starts from x (an ordinary vertex in $H(G_s,r)$), we have that P must cross the bridge $(d(z),z)$ in order to reach z. But, since P eventually returns again to $d(z)$ through $(u,v) = (z,d(z))$, it is unnecessary to traverse this part from $d(z)$ to $d(z)$ (that uses $(u,v) = (z,d(z))$), and so we can skip it. Now let's assume that $v \neq d(z)$. Then, by Lemma 4.1, we have that v must also be a descendant in $D(G_s)$ of z. Now, in order for P to use the edge (u,v), it must first pass from the bridge $(d(z),z)$ (since it starts from x, an ordinary vertex of $H(G_s,r)$). Furthermore, in order for Q to use the edge (v,u), it must also pass from the bridge $(d(z),z)$. But this means that v is reachable from z without using (u,v), and so we can replace the part of P from z to (u,v), with that of Q from z to v. (2) Now let's assume that u is not a descendant of r in $D(G_s)$ (of course, this presupposes that $r \neq s$). Suppose, first, that $(u,v) = (d(r),r)$. Then, since Q uses the edge $(v,u) = (r,d(r))$, but then has to return to $H(G_s,r)$, it has no other option but to pass eventually from the bridge $(d(r),r)$, after visiting vertices which are not descendants of r in $D(G_s)$. Thus we can simply skip this part of Q that starts from r and returns again to r (and uses $(v,u) = (r,d(r))$). Now let's assume that $(u,v) \neq (d(r),r)$. Then, by Lemma 4.1 we have that v is also not a descendant of r in $D(G_s)$. Since P reaches v, in order to end at y (inside $H(G_s,r)$) it has to pass from the bridge $(d(r),r)$. The same is true for Q, since it ends at x (inside $H(G_s,r)$). But then Q can avoid using (v,u), and we replace the part of Q that starts from v and ends in $d(r)$ with that of P that starts from v and ends in $d(r)$. (3) Finally, we are left to assume that u is an ordinary vertex of $H(G_s,r)$. If v is also an ordinary vertex of $H(G_s,r)$, we are done. Otherwise, we can revert back to the previous two cases, interchanging the roles of u and v.

Thus we have shown that there is an edge e in G_s such that, for every two paths P and Q in $G_s \setminus e$ such that P starts from x and ends in y, and Q starts from y and ends in x, we have that there exists an edge $(u,v) \in P$ such that $(v,u) \in Q$, and u,v are both ordinary vertices of $H(G_s,r)$ $(*)$. Now we distinguish three cases, depending on the location of the endpoints of e relative to $D(G_s)$. (1) Let us assume that both endpoints of e are ordinary vertices in $H(G_s,r)$. Now let's assume for the sake of contradiction that there is a path P' in $H(G_s,r) \setminus e$ from x to y, and a path Q' in $H(G_s,r) \setminus e$ from y to x, such that for every edge $(z,w) \in P'$ we have that $(w,z) \notin Q'$

$(**)$. By Lemma 4.2, there is a path P in G_s from x to y such that $P_H = P'$, and a path Q in G_s from y to x such that $Q_H = Q'$. Then, by Lemma 4.2, both P and Q avoid e. But also by the same Lemma, we have that an edge $(u, v) \in H(G_s, r)$ with both endpoints ordinary in $H(G_s, r)$ is used by P (resp. by Q) if and only if it is used by P' (resp. Q'). But then the property $(**)$ of P' and Q' violates the property $(*)$ of P and Q. This shows that $x \not\leftrightarrow_t y$ in $H(G_s, r) \setminus e$ in this case. (2) Now let's assume that both endpoints of e are descendants of r in $D(G_s)$, but also that one endpoint of e is a descendant in $D(G_s)$ of an auxiliary vertex z in $H(G_s, r)$, where $z \neq d(r)$ (if $r \neq s$). Then, by removing $(d(z), z)$ from G_s, we have that no ordinary vertex from $H(G_s, r)$ can reach the edge e. Therefore, $(*)$ implies that for every path P from x to y in $G_s \setminus (d(z), z)$ and every path Q from y to x in $G_s \setminus (d(z), z)$, we have that there exists an edge $(u, v) \in P$ such that $(v, u) \in Q$, and u, v are both ordinary vertices of $H(G_s, r)$. Now let's assume for the sake of contradiction that there is a path P' in $H(G_s, r) \setminus \{(d(z), z)\}$ from x to y, and a path Q' in $H(G_s, r) \setminus \{(d(z), z)\}$ from y to x, such that for every edge $(z, w) \in P'$ we have that $(w, z) \notin Q'$ $(**)$ Then, by Lemma 4.2, both P and Q avoid $(d(z), z)$. But also by the same Lemma, we have that an edge $(u, v) \in H(G_s, r)$ with both endpoints ordinary in $H(G_s, r)$ is used by P (resp. by Q) if and only if it is used by P' (resp. Q'). But then the property $(**)$ of P' and Q' violates the property $(*)$ of P and Q. This shows that $x \not\leftrightarrow_t y$ in $H(G_s, r) \setminus e$ in this case. (3) There remains to consider the case that one endpoint of e is not a descendant of r in $D(G_s)$ (assuming, of course, that $r \neq s$). Then, by removing $(d(r), r)$ from G_s, we have that no path that starts from an ordinary vertex of $H(G_s, r)$ and uses the edge e can return to $H(G_s, r)$. Therefore, $(*)$ implies that for every path P from x to y in $G_s \setminus (d(r), r)$ and every path Q from y to x in $G_s \setminus (d(r), r)$, we have that there exists an edge $(u, v) \in P$ such that $(v, u) \in Q$, and u, v are both ordinary vertices of $H(G_s, r)$. Now let's assume for the sake of contradiction that there is a path P' in $H(G_s, r) \setminus \{(d(r), r)\}$ from x to y, and a path Q' in $H(G_s, r) \setminus \{(d(r), r)\}$ from y to x, such that for every edge $(z, w) \in P'$ we have that $(w, z) \notin Q'$ $(**)$ Then, by Lemma 4.2, both P and Q avoid $(d(r), r)$. But also by the same Lemma, we have that an edge $(u, v) \in H(G_s, r)$ with both endpoints ordinary in $H(G_s, r)$ is used by P (resp. by Q) if and only if it is used by P' (resp. Q'). But then the property $(**)$ of P' and Q' violates the property $(*)$ of P and Q. This shows that $x \not\leftrightarrow_t y$ in $H(G_s, r) \setminus e$ in this case. Any of those cases implies that $x \not\leftrightarrow_{2e} y$ in $H(G_s, r)$, and therefore $x \not\leftrightarrow_{2et} y$ in $H(G_s, r)$. □

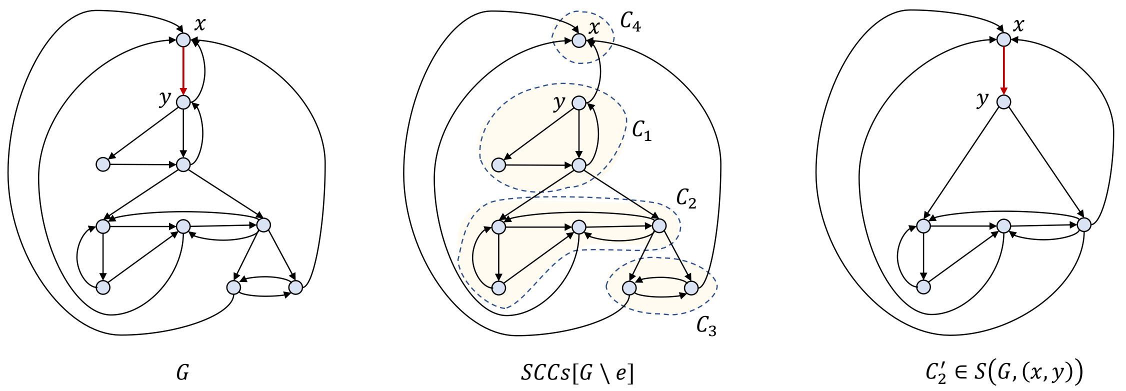

Figure 4.5: A strongly connected digraph G with a strong bridge $e = (x, y)$ shown red. The deletion of e splits G into four strongly connected components C_1, C_2, C_3 and C_4 (numbered in topological order). C'_2 is the digraph in $S(G, e)$ that corresponds to C_2 after attaching x and y to it, and all the edges due to the S-operation.

Now let G be a strongly connected digraph and let (x, y) be a strong bridge of G. We will define the S-*operation* on G and (x, y), which produces a set of digraphs as follows. Let $C_1, \dots, C_k$ be the strongly connected components of $G \setminus (x, y)$. Now let $C \in \{C_1, \dots, C_k\}$. We will construct a graph C' as follows. First, notice that either $x \notin C$ and $y \in C$, or $y \notin C$ and $x \in C$, or $\{x, y\} \cap C = \emptyset$. Then we set $V(C') = V(C) \cup \{x\}$, or $V(C') = V(C) \cup \{y\}$, or $V(C') = V(C) \cup \{x, y\}$, respectively. Every edge of G with both endpoints in C is included in C'. Furthermore, for every edge (u, v) of G such that $u \in C$ and $v \notin C$, we add the edge (u, x) to C'. Also, for every edge (u, v) of G such that $u \notin C$ and $v \in C$, we add the edge (y, v) to C'. Finally, we also add the edge (x, y) to C'. Now we define $S(G, (x, y)) := \{C'_1, \dots, C'_k\}$. See Figure 4.5.

Intuitively, we can describe the idea of the S-operation on G and (x, y) as follows. Let $\mathcal{D}$ be the directed acyclic graph (DAG) of the strongly connected components of $G \setminus (x, y)$. Then we have that $\mathcal{D}$ has a single source that contains y, and a single sink that contains x. Now, for every node C of $\mathcal{D}$, we merge all nodes of $\mathcal{D} \setminus \{C\}$ that are reachable from C into x, and all nodes of $\mathcal{D} \setminus \{C\}$ that reach C into y. (We ignore all the other nodes of $\mathcal{D}$.) We also reconnect this graph by adding the edge (x, y). We perform this operation for every node of $\mathcal{D}$, and thus we get the collection of graphs $S(G, (x, y))$.

The purpose of this operation is to maintain the connectivity information within every strongly connected component of $G \setminus (x, y)$. In particular, we have the following correspondence between paths of G and paths in the graphs of $S(G,(x,y))$. Let u,v be two vertices of the same strongly connected component C of $G\setminus(x,y)$ and let P be a path in G from u to v. Then we construct the *compressed* path $\tilde{P}$ of P as follows. First, every edge of P with both endpoints in C is maintained in $\tilde{P}$. If P lies entirely within C, we are done. Otherwise, let $(z_1,w_1),\dots,(z_t,w_t)$ be a maximal segment of P consisting of edges that have at least one of their endpoints outside of C. (Thus we have $z_1\in C$ and $w_t\in C$.) Notice that this segment is unique because if a path leaves C it has to use the edge (x,y) in order to return to C (and P can use (x,y) only once). Now, if $(z_1,w_1)=(x,y)$, we replace this segment in $\tilde{P}$ with $(x,y),(y,w_t)$. Similarly, if $(z_t,w_t)=(x,y)$, we replace this segment with $(z_1,x),(x,y)$. Otherwise, we replace this segment with $(z_1,x),(x,y),(y,w_t)$. Then, it should be clear that $\tilde{P}$ is a path in C' from u to v. The construction of the compressed path, together with a kind of converse to it, is summarized in the following.

Lemma 4.3. *Let G be a strongly connected digraph and let (x,y) be a strong bridge of G. Let C be a strongly connected component of $G\setminus(x,y)$, and let C' be the corresponding graph in $S(G,(x,y))$. Let also u and v be two vertices in C. Then, for every path P in G from u to v, there is a (compressed) path $\tilde{P}$ in C' from u to v with the following three properties.* (1) *Every edge $e\in P$ that lies in C is maintained in P. Also, every occurrence of (x,y) in P is maintained in $\tilde{P}$.* (2) *If $x\notin C$ and (z,w) is an edge of P such that $z\in C$ and $w\notin C$, then $\tilde{P}$ contains (z,x).* (3) *If $y\notin C$ and (z,w) is an edge of P such that $z\notin C$ and $w\in C$, then $\tilde{P}$ contains (y,w). Conversely, for every path P' in C' from u to v, there is a path P in G from u to v such that $\tilde{P}=P'$.*

Proof. The first part was discussed above. For the converse, we construct the path P by replacing some edges of P' with paths of G (if needed) as follows. First, every edge of P' that lies in C is maintained in P. Furthermore, every occurrence of (x,y) in P' is also maintained in P. Now, if $x\notin C$ and there is an edge $(z,x)\in P'$, then we trace a path from z to x in the DAG $\mathcal{D}$ of the strongly connected components of $G\setminus(x,y)$ (possibly using internal paths from the intermediate components in $\mathcal{D}$), and we let it replace (z,x) in P. Similarly, if $y\notin C$ and there is an edge $(y,z)\in P'$, then we trace a path from y to z in $\mathcal{D}$ (possibly using internal paths from the intermediate components in $\mathcal{D}$), and we let it replace (y,z) in P. It should be clear that P so constructed is indeed a path in G from u to v such that $\tilde{P}=P'$. □

Lemma 4.3 implies the following.

Corollary 4.2. *Let G be a strongly connected digraph and let e be a strong bridge of G. Then every graph of $S(G,e)$ is strongly connected.*

The following two lemmas are useful strengthenings of Lemma 4.3.

Lemma 4.4. *Let G be a strongly connected digraph and let (x,y) be a strong bridge of G. Let C be a strongly connected component of $G \setminus (x,y)$, and let C' be the corresponding digraph in $S(G,(x,y))$. Let P' be a path in C' with both endpoints in C. Then there is a path P in G with $\tilde{P} = P'$ such that for every pair of twin edges $(z,w) \in P$ and $(w,z) \in P$ we have that both (z,w) and (w,z) are in C'.*

Proof. By Lemma 4.3 we have that there is a path P in G such that $\tilde{P} = P'$. In particular, the endpoints of P lie in C. Now let (z,w) be an edge such that $(z,w) \in P$, $(w,z) \in P$, and $(z,w) \notin C'$. (If no such edge exists, then we are done.) We will show that we can eliminate all those instances from P, and maintain the property that $\tilde{P} = P'$. First, let us suppose, for the sake of contradiction, that $(x,y) \in \{(z,w),(w,z)\}$. Since $(z,w) \notin C'$, we have $(z,w) \neq (x,y)$. Therefore, $(w,z) = (x,y)$ and $(z,w) = (y,x)$. Since $(z,w) = (y,x) \notin C'$, we have that neither x nor y is in C. Thus, since P starts from C, in order to reach y it must pass from (x,y). Then, since it uses (y,x), in order to reach C it must again use (x,y). But this contradicts the fact that P may use an edge only once. This shows that $(x,y) \notin \{(z,w),(w,z)\}$.

Now, since $(x,y) \notin \{(z,w),(w,z)\}$, the existence of the edges (z,w) and (w,z) in G implies that z and w belong to the same strongly connected component D of $G \setminus (x,y)$. Notice that D must be different from C, because otherwise, we would have $(z,w) \in C$. Thus, due to the fact that P may use (x,y) only once, we have that D is accessed by P only once (because we need to pass from (x,y) in order to either reach D from C, or reach C from D). Now, if P first uses (z,w) and afterwards (w,z), we have a circuit that starts with (z,w) and ends with (w,z) that can be eliminated from P. Similarly, if P first uses (w,z) and then (z,w), we have a circuit that starts with (w,z) and ends with (z,w) that can be eliminated from P. In both cases, $\tilde{P} = P'$ is maintained.

Thus, after all those eliminations, we have that, if (z,w) and (w,z) are edges of P, then both of them must be in C'. □

Lemma 4.5. *Let G be a strongly connected digraph and let (x,y) be a strong bridge of G. Let C be a strongly connected component of $G \setminus (x,y)$, and let C' be the corresponding digraph in $S(G,(x,y))$. Let P' and Q' be two paths in C' whose endpoints lie in C. Then* *there are paths P and Q in G with $\tilde{P} = P'$ and $\tilde{Q} = Q'$, such that for every pair of edges $(z, w) \in P$ and $(w, z) \in Q$ we have that both (z, w) and (w, z) are in C'.*

Proof. By Lemma 4.4 we have that there is a path P in G with $\tilde{P} = P'$ such that for every pair of twin edges $(z, w) \in P$ and $(w, z) \in P$ we have that both (z, w) and (w, z) are in C'. Similarly, by Lemma 4.4 we have that there is a path Q in G with $\tilde{Q} = Q'$ such that for every pair of twin edges $(z, w) \in Q$ and $(w, z) \in Q$ we have that both (z, w) and (w, z) are in C'.

Now let us suppose that there is an edge $(z, w) \in P$, such that $(w, z) \in Q$, and at least one of (z, w) and (w, z) is not in C'. (If no such edge exists, then we are done.) Let us suppose, for the sake of contradiction, that $(x, y) \in \{(z, w), (w, z)\}$. We may assume w.l.o.g. that $(z, w) = (x, y)$. Now, since at least one of (z, w) and (w, z) is not in C', we have that $(w, z) = (y, x) \notin C'$. This implies that $y \notin C$. Thus, since Q starts from C and reaches y, it must use the edge (x, y). But then, since $(x, y) \in Q$ and $(y, x) \in Q$ and $(y, x) \notin C'$, we have a contradiction to the property of Q provided by Lemma 4.4. This shows that $(x, y) \notin \{(z, w), (w, z)\}$.

Since $(x, y) \notin \{(z, w), (w, z)\}$ and $(z, w), (w, z)$ are edges of the graph, we have that z and w belong to the same strongly connected component D of $G \setminus (x, y)$, and therefore both (z, w) and (w, z) lie entirely within D. Notice that $D \neq C$, because otherwise we would have that both (z, w) and (w, z) are edges of C. Then we have that either (1) C reaches D in $G \setminus (x, y)$, or (2) D reaches C in $G \setminus (x, y)$, or (3) neither C reaches D in $G \setminus (x, y)$, nor D reaches C in $G \setminus (x, y)$. Notice that case (3) is impossible due to the existence of P (and Q): because in this case, since P starts from C, in order to reach D it must use (x, y), and then, in order to return to C, it must again use (x, y). Thus, it is sufficient to consider cases (1) and (2). (Notice that (1) implies that $x \notin C$, and (2) implies that $y \notin C$.)

Notice that since both P and Q go outside of C at some point, in order to return back to C they have the following structure: by the time they exit C, they use a path to reach x (possibly the trivial path, if $x \in C$), then they use (x, y), and finally they use another path to return to C (possibly the trivial path, if $y \in C$). Thus, the parts of P and Q from C to x and from y to C can be substituted by any other path in G (that has the same endpoints), and the property $\tilde{P} = P'$ and $\tilde{Q} = Q'$ is maintained. In the following, we will exploit this property.

Let u and v be the first and the last vertex, respectively, of the subpath of P inside D, and let u' and v' be the first and the last vertex, respectively, of the subpath of Q

inside D. Suppose first that (1) is true. Let $P[v, x]$ be the subpath of P from the last occurrence of v to the first (and only) occurrence of x, and let $Q[v', x]$ be the subpath of Q from the last occurrence of v' to the first (and only) occurrence of x. Since D is strongly connected, there is a path S from u to v inside D, and we may assume that all vertices of S are distinct. Then, we replace the part of P inside D with S. (Observe that $\tilde{P} = P'$ is maintained.) Since D is strongly connected, there is also at least one path S' from u' to a vertex p in S, that uses each vertex only once, and intersects with S only at p. Then we consider the path S'' from u' to v in D that is formed by first using S' from to u' to p, and then the subpath of S from p to v. We let S'' replace the part of Q inside D, and we substitute $Q[v', x]$ with $P[v, x]$. (Observe that Q is still a path, and satisfies $\tilde{Q} = Q'$.) Now, we have that there is no pair of twin edges $(z', w') \in P$ and $(w', z') \in Q$ such that both (z', w') and (w', z') belong to D. Furthermore, since we replaced the part of Q from D to x with the subpath of P from D to x, we have that there is no pair of twin edges $(z', w') \in P$ and $(w', z') \in Q$ such that (z', w') and (w', z') are on this path, due to Lemma 4.4.

Now let us suppose that (2) is true. (The argument here is analogous to the previous one.) Let $P[y, u]$ be the subpath of P from the first (and only) occurrence of y to the first occurrence of u, and let $Q[y, u']$ be the subpath of Q from the first (and only) occurrence of y to the first occurrence of u'. Since D is strongly connected, there is a path S from u to v inside D, and we may assume that all vertices of S are distinct. Then, we replace the part of P inside D with S. (Observe that $\tilde{P} = P'$ is maintained.) Since D is strongly connected, there is also at least one path S' in D from a vertex $p \in S$ to v', that uses each vertex only once and intersects with S only in p. Then we consider the path S'' from u to v' in D that is formed by first using the subpath of S from to u to p, and then the path S' from p to v'. We let S'' replace the part of Q inside D, and we substitute $Q[y, u']$ with $P[y, u]$. (Observe that Q is still a path, and satisfies $\tilde{Q} = Q'$.) Now, we have that there is no pair of twin edges $(z', w') \in P$ and $(w', z') \in Q$ such that both (z', w') and (w', z') belong to D. Furthermore, since we replaced the part of Q from y to D with the subpath of P from y to D, we have that there is no pair of twin edges $(z', w') \in P$ and $(w', z') \in Q$ such that (z', w') and (w', z') are on this path, due to Lemma 4.4.

Thus, we have basically shown that we can eliminate all instances where there are edges $(z, w) \in P$ and $(w, z) \in Q$ such that at least one of (z, w) and (w, z) is not in C', while still maintaining $\tilde{P} = P'$ and $\tilde{Q} = Q'$, by substituting parts of Q with parts of P. This shows that for every pair of edges $(z,w) \in P$ and $(w,z) \in Q$ we have that both (z,w) and (w,z) are in C'. $\square$

Now we can show that the S-operation maintains the relation of 2-edge twinless strong connectivity.

Proposition 4.6. *Let G be a strongly connected digraph and let (x,y) be a strong bridge of G. Then, for any two vertices $u, v \in G$, we have $u \overset{G}{\leftrightarrow}_{2et} v$ if and only if u and v belong to the same graph C of $S(G,(x,y))$ and $u \overset{C}{\leftrightarrow}_{2et} v$.*

Proof. ($\Rightarrow$) Let $u \leftrightarrow_{2et} v$ in G. This implies that u and v remain strongly connected after the removal of any edge of G. In particular, u and v must belong to the same strongly connected component C of $G \setminus (x,y)$, and thus u and v belong to the same graph C' of $S(G,(x,y))$. Now let e be an edge of C'. Suppose first that both endpoints of e lie in C. Then, $u \leftrightarrow_{2et} v$ in G implies that there are paths P and Q in $G \setminus e$ such that P goes from u to v, Q goes from v to u, and there is no edge (z,w) such that $(z,w) \in P$ and $(w,z) \in Q$. Now consider the compressed paths $\tilde{P}$ and $\tilde{Q}$, of P and Q, respectively, in C'. Then we have that $\tilde{P}$ is a path from u to v, $\tilde{Q}$ is a path from v to u, and neither $\tilde{P}$ nor $\tilde{Q}$ contains e. Furthermore, for every $(z,w) \in \tilde{P}$ that lies in C, we have that $(w,z) \notin \tilde{Q}$. Now take an edge $(z,w) \in \tilde{P}$ such that at least one of z and w lies outside of C. There are three cases to consider: either $x \in C$ and $y \notin C$, or $x \notin C$ and $y \in C$, or $C \cap \{x,y\} = \emptyset$. Let us consider first the case that $x \in C$ and $y \notin C$. Then we must have that either $(z,w) = (x,y)$, or $z = y$ and $w \in C$. Suppose that $(z,w) = (x,y)$ and $(y,x) \in \tilde{Q}$. But since $y \notin C$, in order for $\tilde{Q}$ to reach y it must necessarily pass from the edge (x,y). Thus the segment $(x,y),(y,x)$ from $\tilde{Q}$ can be eliminated. Now, if $z = y$ and $w \in C$, then we can either have $w = x$ or $w \neq x$. If $w = x$, then we can reason for $\tilde{P}$ as we did before for $\tilde{Q}$, in order to see that the segment $(w,z),(z,w)$ can be eliminated from $\tilde{P}$. Otherwise, we simply have that (w,y) cannot be an edge of C', and therefore (w,z) is not contained in $\tilde{Q}$. Thus we have demonstrated that there are subpaths of $\tilde{P}$ and $\tilde{Q}$, that have the same endpoints, respectively, and do not share any pair of antiparallel edges. We can use the same argument for the other two cases (i.e., for the cases $x \notin C$ and $y \in C$, and $C \cap \{x,y\} = \emptyset$).

Now suppose that at least one of the endpoints of e lies outside of C. Then, since $u \leftrightarrow_{2et} v$ in G, we have that there are paths P and Q in $G \setminus (x,y)$ such that P goes from u to v, Q goes from v to u, and there is no edge (z,w) such that $(z,w) \in P$ and

$(w,z) \in Q$. Since P and Q avoid the edge (x,y), it is impossible that they use edges with at least one endpoint outside of C (for otherwise they cannot return to C using edges of the DAG of the strongly connected components of $G \setminus (x,y)$). Thus P and Q lie within C, and so they avoid e. This concludes the proof that $u \leftrightarrow_{2et} v$ in C'.
($\Leftarrow$) Suppose that $u \not\leftrightarrow_{2et} v$ in G. If u and v are in two different strongly connected components of $G \setminus (x,y)$, then we are done. Otherwise, suppose that u and v belong to the same strongly connected component C of $G \setminus (x,y)$, and let C' be the corresponding graph in $S(G,(x,y))$. Suppose first that $u \not\leftrightarrow_{2e} v$ in G. Then we may assume, without loss of generality, that there is an edge e such that u cannot reach v in $G \setminus e$. Obviously, we cannot have $e = (x,y)$. Furthermore, it cannot be the case that at least one of the endpoints of e lies outside of C (precisely because (x,y) cannot separate u and v). Thus, e lies within C. Now let us assume, for the sake of contradiction, that u can reach v in $C' \setminus e$. So let P' be a path in $C' \setminus e$ from u to v. Then, by Lemma 4.3, there is a path P in G from u to v such that $\tilde{P} = P'$. Since, $e \in C$ but $e \notin P'$, by the same lemma we have that P avoids e, a contradiction. Thus, v is not reachable from u in $C' \setminus e$, and so $u \not\leftrightarrow_{2et} v$ in C'.

So let us finally assume that $u \leftrightarrow_{2e} v$ in G. Since $u \not\leftrightarrow_{2et} v$ in G, this means that there is an edge $e \in G$ such that for every path P in $G \setminus e$ from u to v, and every path Q in $G \setminus e$ from v to u, there is a pair of twin edges $(z,w) \in P$ and $(w,z) \in Q$ $(*)$. Suppose first that e does not lie in C. Now let P' be a path in $C' \setminus (x,y)$ from u to v, and let Q' be a path in $C' \setminus (x,y)$ from v to u. Then, by Lemma 4.3 there are paths P and Q in G such that $\tilde{P} = P'$ and $\tilde{Q} = Q'$. Thus, P is a path from u to v, and Q is a path from v to u. Since $(x,y) \notin P'$ and $(x,y) \notin Q'$, Lemma 4.3 implies that $(x,y) \notin P$ and $(x,y) \notin Q$. Thus, since P and Q have both of their endpoints in C and avoid (x,y), they cannot but lie entirely within C. Thus, $P = \tilde{P} = P'$ and $Q = \tilde{Q} = Q'$, and both P and Q avoid e. Then, by $(*)$, there is a pair of twin edges $(z,w) \in P'$ and $(w,z) \in Q'$. This shows that $u \not\leftrightarrow_{t} v$ in $C' \setminus (x,y)$.

Now suppose that e lies in C. Let P' be a path in $C' \setminus e$ from u to v, and let Q' be a path in $C' \setminus e$ from v to u. Then, by Lemma 4.5 there are paths P and Q in G with $\tilde{P} = P'$ and $\tilde{Q} = Q'$, such that for every pair of twin edges $(z,w) \in P$ and $(w,z) \in Q$ we have that both (z,w) and (w,z) are in C'. Since $\tilde{P} = P'$ and $e \notin P'$, Lemma 4.3 implies that P is a path from u to v that avoids e. Similarly, since $\tilde{Q} = Q'$ and $e \notin Q'$, Lemma 4.3 implies that Q is a path from v to u that avoids e. Thus, by $(*)$ we have that there is a pair of twin edges $(z,w) \in P$ and $(w,z) \in Q$. Then, we have that both

(z, w) and (w, z) are in C'. Thus, since $\tilde{P} = P'$ and $\tilde{Q} = Q'$, by Lemma 4.3 we have that $(z, w) \in P'$ and $(w, z) \in Q'$. This shows that $u \not\leftrightarrow_t v$ in $C' \setminus e$. In any case, we conclude that $u \not\leftrightarrow_{2et} v$ in C'. $\square$

We can combine Propositions 4.5 and 4.6 in order to derive some auxiliary graphs that maintain the relation of 2-edge twinless strong connectivity of the original graph. Then we can exploit the properties of those graphs in order to provide a linear-time algorithm for computing the 2-edge twinless strongly connected components. First, we introduce some notation. Let G_s be a strongly connected digraph, and let r be either a marked vertex of G_s or s. Then we denote $H(G_s, r)$ as H_r. Furthermore, if r' is either a marked vertex of H_r^R or r, we denote $H(H_r^R, r')$ as $H_{rr'}$. In the following, when we refer to the immediate dominator $d(x)$ of a vertex x, or to the subtree $T(x)$ of a dominator tree decomposition, we specify the corresponding underlying graph (e.g., we say that $d(x)$ is the immediate dominator of x in $H_{rr'}$).

Now let x be a vertex in $H_{rr'}$. Then x is also a vertex in H_r. We distinguish four cases, depending on whether x is ordinary or auxiliary in $H_{rr'}$ and H_r. If x is ordinary in $H_{rr'}$ and ordinary in H_r, it is called an oo-vertex. If x is ordinary in $H_{rr'}$ and auxiliary in H_r, it is called an oa-vertex. If x is auxiliary in $H_{rr'}$ and ordinary in H_r, it is called ao-vertex. And if x is auxiliary in both $H_{rr'}$ and H_r, it is called an aa-vertex.

Now we have the following.

Corollary 4.3. *Let G_s be a strongly connected digraph, and let x, y be two vertices of G_s. Then $x \leftrightarrow_{2et} y$ in G_s if and only if x and y are both oo-vertices in A and $x \leftrightarrow_{2et} y$ in A, where A is either* (1) *H_{ss}, or* (2) *a graph in $S(H_{sr}, (d(r), r))$, or* (3) *H_{rr}, or* (4) *a graph in $S(H_{rr'}, (d(r'), r'))$ (where r and r' are marked vertices, $d(r)$ in* (2) *denotes the immediate dominator of r in H_s^R, and $d(r')$ in* (4) *denotes the immediate dominator of r' in H_r^R).*

Proof. This is an immediate consequence of Propositions 4.5 and 4.6. $\square$

We note that the graphs $H_{rr'}$ were used in [13] to compute the 2-edge strongly connected components in linear time. Specifically, it was shown in [13] that the 2-edge strongly connected components of G_s are given by the oo-vertices of: (1) the graph H_{ss}, (2) the strongly connected components of all graphs $H_{sr} \setminus (d(r), r)$, where r is a marked vertex of G_s, and $(d(r), r)$ is the critical edge of H_{sr}, (3) the graphs H_{rr}, for all marked vertices r of G_s, and (4) the strongly connected components of all graphs

$H_{rr'} \setminus (d(r'), r')$, where r is a marked vertex of G_s, r' is a marked vertex of H_r^R, and $(d(r'), r')$ is the critical edge of $H_{rr'}$. Since we can compute all those auxiliary graphs in linear time (Theorem 4.4), this implies that we can compute all 2-edge strongly connected components in linear time.

The following series of lemmas provide useful information concerning the reachability of vertices in the auxiliary graphs when we remove an edge. All of them are basically a consequence of Lemma 4.2, which establishes a two-way correspondence between paths of G_s and paths of $H(G_s, r)$.

Lemma 4.6. *Let G_s be a strongly connected digraph and let r be either a marked vertex of G_s or s. Let (x, y) be an edge of H_r such that y is an ordinary vertex of H_r. Then r can reach every vertex of H_r in $H_r \setminus (x, y)$.*

Proof. Suppose first that x is an ordinary vertex of H_r. Then (x, y) is also an edge of G_s, which is not a bridge of G_s. Now let u be a vertex of H_r, but not $d(r)$ (if $r \neq s$). Then there is a path P from s to u that avoids (x, y). If $r \neq s$, then, by Lemma 4.1, P must necessarily pass from the bridge $(d(r), r)$, and every time it moves out of $D(r)$ it must later on pass again through $(d(r), r)$, in order to reach eventually u. Thus P contains a subpath Q from r to u that visits only vertices of $D(r)$. Now consider the corresponding path Q_H in H_r. By Lemma 4.2, this is precisely a path from r to u in H_r that avoids (x, y). Now let $r \neq s$. Since H_r is strongly connected (by Corollary 4.1), we have that r can reach $d(r)$. Since y is ordinary, (x, y) is not an incoming edge to $d(r)$ in H_r. And since r can reach every vertex of $H_r \setminus \{d(r)\}$ in $H_r \setminus (x, y)$, we conclude that $d(r)$ is also reachable from r in $H_r \setminus (x, y)$.

Now suppose that x is an auxiliary vertex of H_r. If $x = d(r)$ (assuming that $r \neq s$), then we must have that $y = r$. Since H_r is strongly connected (by Corollary 4.1), we have that r can reach every vertex of H_r. Thus, even by removing the incoming edge $(d(r), r)$ of r, we still have that r can reach every vertex of $H_r \setminus (d(r), r)$. So let us assume that $x \neq d(r)$ (if $r \neq s$). Then we have that x is a marked vertex of G_s such that $d(x) \in T(r)$. Now let u be a vertex of H_r but not $d(r)$ (if $r \neq s$). If $u \neq x$, then u is not a descendant of x in $D(G_r)$. Therefore, as a consequence of Lemma 4.1, there is a path P in G_s from s to u that avoids $(d(x), x)$. Arguing as above, we have that there is a subpath Q of P from r to u. Now consider the corresponding path Q_H of Q in H_r. Then, by Lemma 4.2, Q_H avoids $(d(x), x)$, and therefore (x, y). Now, if $u = x$, then there is definitely a path from r to u in H_r (by Corollary 4.1). But then we can also avoid using (x, y) in order to reach u from r in H_r. Finally, we can argue as above, in order to conclude that $d(r)$ is also reachable from r in $H_r \setminus (x, y)$. This completes the proof. □

Lemma 4.7. *Let G_s be a strongly connected digraph and let r be either a marked vertex of G_s or s. Let $(d(z), z)$ be an edge of H_r such that z is a marked vertex of G_s with $d(z) \in T(r)$. Then r can reach every vertex of H_r in $H_r \setminus (x, y)$, except z, which is unreachable from r in $H_r \setminus (x, y)$, and possibly $d(r)$ (if $r \neq s$).*

Proof. We can argue as in the proof of Lemma 4.6 in order to show that, for every vertex u in H_r that is not z or $d(r)$ (if $r \neq s$), there is a path from r to u in H_r that avoids $(d(z), z)$. Now let's assume for the sake of contradiction that there is a path $\tilde{P}$ in H_r from r to z that avoids $(d(z), z)$. Then, by Lemma 4.2 there is path P in G_s from r to z that corresponds to $\tilde{P}$ and it also avoids $(d(z), z)$. Now, since $d(z)$ is not a dominator of r in G_s, there is a path Q from s to r in G_s that avoids $d(z)$. But now the concatenation of Q with P is a path in G_s from s to z that avoids $(d(z), z)$, contradicting the fact that this edge is a bridge in G_s. □

Lemma 4.8. *Let G_s be a strongly connected digraph and let x be a marked vertex of G_s such that $d(x) \in T(s)$ in G_s. Let (x_0, y) be an edge of G_s such that x_0 is a descendant of x in $D(G_s)$ and y is not a descendant of x in $D(G_s)$. Let u be a vertex of H_s. If u reaches s in $G_s \setminus (x_0, y)$, then u reaches s in $H_s \setminus (x, y)$.*

Proof. Let u be a vertex of H_s and let P be a path from u to s in $G_s \setminus (x_0, y)$. Consider the corresponding path P_H of P in H_s. P_H has the same endpoints as P. Thus, if P_H avoids the edge (x, y) of H_s, then we are done. Otherwise, this means that P uses an edge (x_1, y) of G_s, distinct from (x_0, y) (although we may have $x_1 = x_0$, since we allow multiple edges), such that x_1 is a descendant of x in $D(G_s)$. But then we must have that (x, y) is a double edge of H_s, and therefore $H_s \setminus (x, y)$ is strongly connected (since H_s is strongly connected, by Corollary 4.1). Thus u can reach s in $H_s \setminus (x, y)$. □

Lemma 4.9. *Let G_s be a strongly connected digraph and let r be either a marked vertex of G_s or s. Let z by a marked vertex of G_s such that $d(z) \in T(r)$. Let $\{(z, y_1), \ldots, (z, y_k)\}$ be the set of all edges that stem from z in H_r. Then r can reach every vertex of H_r in $H_r \setminus \{(z, y_1), \ldots, (z, y_k)\}$, except possibly $d(r)$ (if $r \neq s$).*

Proof. We can argue as in the proof of Lemma 4.6 in order to show that, for every vertex u in H_r that is not z or $d(r)$ (if $r \neq s$), there is a path from r to u in H_r that avoids $(d(z), z)$, and therefore all of the edges $\{(z, y_1), \dots, (z, y_k)\}$. Furthermore, there is certainly a path in H_r from r to z (by Corollary 4.1), and so we can definitely reach z from r in $H_r \setminus \{(z, y_1), \dots, (z, y_k)\}$. □

Lemma 4.10. *Let G_s be a strongly connected digraph and let r be either a marked vertex of G_s or s. Let u and v be two vertices of H_r, and let (x, y) be an edge of G_s such that either (1) both x and y are ordinary vertices of H_r, or (2) y is a marked vertex of G_s and $x = d(y)$, or (3) $(x, y) = (d(r), r)$ (assuming that $r \neq s$). Then u can reach v in $G_s \setminus (x, y)$ if and only if u can reach v in $H_r \setminus (x, y)$.*

Proof. $(\Rightarrow)$ Let P be a path in $G_s \setminus (x, y)$ that starts from u and ends in v. Now consider the corresponding path P_H of P in H_r. Then, since either one of (1), (2), or (3) holds for (x, y), by Lemma 4.2 we have that (x, y) is not contained in P_H. Thus u can reach v in $H_r \setminus (x, y)$.
$(\Leftarrow)$ Let us assume that u cannot reach v in $G_s \setminus (x, y)$. Now let's assume for the sake of contradiction that there is path $\tilde{P}$ in $H_r \setminus (x, y)$ that starts from u and ends in v. By Lemma 4.2, there is a path P in G_s such that $P_H = \tilde{P}$. Since either one of (1), (2), or (3) holds for (x, y), by the same Lemma we have that (x, y) is not contained in P. But this means that u can reach v in $G_s \setminus (x, y)$ - a contradiction. □

Corollary 4.4. *Let G_s be a strongly connected digraph, and let x, y be two vertices of G_s. Then $x \overset{G_s}{\leftrightarrow}_{2et} y$ if and only if x and y are both ordinary vertices in H and $x \overset{H}{\leftrightarrow}_{2et} y$, where H is either (1) H_{ss}, or (2) H_{rr}, or (3) a graph in $S(H_{sr}, (d(r), r))$, or (4) a graph in $S(H_{rr'}, (d(r'), r'))$ (where r, r', $d(r)$ and $d(r')$ are as in the statement of Corollary 4.3).*

Proof. This is an immediate consequence of Propositions 4.5 and 4.6. □

Now we can describe the structure of the strongly connected components of the graphs that appear in Corollary 4.4 when we remove a strong bridge from them.

Lemma 4.11. *Let G_s be a strongly connected digraph. Let (x, y) be a strong bridge of H_{ss}. Then the strongly connected components of $H_{ss} \setminus (x, y)$ are given by one of the following:*

(i) *$\{x\}$ and $H_{ss} \setminus \{x\}$, where x is auxiliary in H_s and $y = d(x)$ in G_s (see case (1.3) below)*

(ii) *$\{x\}$ and $H_{ss} \setminus \{x\}$, where x is auxiliary in H_{ss} and (x, y) is the only outgoing edge of x in H_{ss} (see case (3.3) below)*

(iii) $\{y\}$ *and* $H_{ss} \setminus \{y\}$*, where* y *is auxiliary in* H_{ss} *and* $x = d(y)$ *in* H_s^R *(see cases* (2.1) *and* (2.2) *below)*

(iv) $\{x\}$*,* $\{y\}$ *and* $H_{ss} \setminus \{x, y\}$*, where* x *is auxiliary in* H_s*,* y *is auxiliary in* H_{ss}*,* $x = d(y)$ *in* H_s^R *and* $y = d(x)$ *in* G_s *(see case* (2.3) *below)*

Proof. Let (x, y) be an edge of H_{ss}. We start by distinguishing four different cases, depending on whether x and y are ordinary or auxiliary vertices of H_{ss}.

(1) Let x and y be both ordinary vertices of H_{ss}. Here we distinguish four different cases, depending on whether x and y are ordinary or auxiliary in H_s. Notice that, in any case, (y, x) is an edge of H_s. (1.1) Let x and y be both oo-vertices. Then, by Lemma 4.6, s can reach all vertices of H_s in $H_s \setminus (y, x)$. Thus, with the removal of (x, y) from H_s^R, every vertex of H_s^R can reach s. Thus, Lemma 4.10 implies that, even by removing (x, y) from H_{ss}, every vertex from H_{ss} can reach s. Now, since x and y are both ordinary vertices of H_{ss}, by Lemma 4.6 we have that s can reach every vertex in $H_{ss} \setminus (x, y)$. We conclude that $H_{ss} \setminus (x, y)$ is strongly connected. (1.2) Let x be oo and y be oa. Then we use precisely the same argument as in (1.1) to conclude that $H_{ss} \setminus (x, y)$ is strongly connected. (1.3) Let x be oa and y be oo. Then, since there is no critical vertex in H_s, x must be a marked vertex of G_s such that $d(x) \in T(s)$. Since (y, x) is an edge of H_s, we must have that $y = d(x)$ in G_s. Thus, by Lemma 4.7, s can reach all vertices of H_s in $H_s \setminus (y, x)$, except x. This implies that s is reachable from all vertices of H_s^R in $H_s^R \setminus (x, y)$, except from x. By Lemma 4.10, this implies that s is reachable from all vertices of H_{ss} in $H_{ss} \setminus (x, y)$, except from x. Furthermore, Lemma 4.10 also implies that all vertices of H_{ss} are reachable from s in $H_{ss} \setminus (x, y)$. Thus, the strongly connected components of $H_{ss} \setminus (x, y)$ are $\{x\}$ and $H_{ss} \setminus \{x\}$. (1.4) Let x and y be both oa. This implies that (y, x) is an edge of H_s both of whose endpoints are auxiliary. But this is impossible to occur since there is no critical vertex in H_s.

(2) Let x be an ordinary vertex of H_{ss} and let y be an auxiliary vertex of H_{ss}. Since H_{ss} does not contain a critical vertex, this means that y is a marked vertex of H_s^R and $x = d(y)$ in H_s^R. Thus, (y, x) is also an edge of H_s. Now we distinguish four different cases, depending on whether x and y are ordinary or auxiliary in H_s. (2.1) Let x be oo and y be ao. Then, by Lemma 4.6, s can reach all vertices of H_s in $H_s \setminus (y, x)$. This implies that s is reachable from all vertices of H_s^R in $H_s^R \setminus (x, y)$. By Lemma 4.10, this implies that s is reachable from all vertices of H_{ss} in $H_{ss} \setminus (x, y)$. Now, since $x = d(y)$ in H_s^R, by Lemma 4.7 we have that s can reach all vertices of

H_{ss} in $H_{ss} \setminus (x, y)$, except y. Thus, the strongly connected components of $H_{ss} \setminus (x, y)$ are $\{y\}$ and $H_{ss} \setminus \{y\}$. (2.2) Let x be oo and y be aa. Then we use precisely the same argument as in (2.1) in order to show that the strongly connected components of $H_{ss} \setminus (x, y)$ are $\{y\}$ and $H_{ss} \setminus \{y\}$. (2.3) Let x be oa and y be ao. This means that y is an ordinary vertex of H_s and x is an auxiliary vertex of H_s. Thus, x is a marked vertex of G_s, and $y = d(x)$ in G_s. By Lemma 4.7, we have that s can reach all vertices of H_s in $H_s \setminus (y, x)$, except x. This implies that s is reachable from all vertices of H_s^R in $H_s^R \setminus (x, y)$, except from x. By Lemma 4.10, this implies that s is reachable from all vertices of H_{ss} in $H_{ss} \setminus (x, y)$, except from x. Now, since $x = d(y)$ in H_s^R, Lemma 4.7 implies that s can reach all vertices of H_{ss} in $H_{ss} \setminus (x, y)$, except y. We conclude that the strongly connected components of $H_{ss} \setminus (x, y)$ are $\{x\}$, $\{y\}$, and $H_{ss} \setminus \{x, y\}$. (2.4). Let x be oa nd y be aa. This implies that (y, x) is an edge of H_s both of whose endpoints are auxiliary. But this is impossible to occur since there is no critical vertex in H_s.

(3) Let x be an auxiliary vertex of H_{ss} and let y be an ordinary vertex of H_{ss}. Since H_{ss} has no critical vertex, this means that x is a marked vertex of H_s^R with $d(x) \in T(s)$ in H_s^R. Now we distinguish four different cases, depending on whether x and y are ordinary or auxiliary in H_s. In any case, by Lemma 4.6 we have that s can reach every vertex of H_{ss} in $H_{ss} \setminus (x, y)$. (3.1) Let x be ao and y be oo. Let (x_0, y) be an edge of H_s^R that corresponds to (x, y) in H_{ss}. Then (y, x_0) is an edge of H_s. If x_0 is an ordinary vertex of H_s, then by Lemma 4.6 we have that s reaches all vertices of H_s in $H_s \setminus (y, x_0)$. This implies that every vertex of H_s^R can reach s in $H_s^R \setminus (x_0, y)$. Then Lemma 4.8 implies that x can reach s in $H_{ss} \setminus (x, y)$. Thus, every vertex of H_{ss} can reach s in $H_{ss} \setminus (x, y)$, and so we have that $H_{ss} \setminus (x, y)$ is strongly connected. Similarly, if x_0 is an auxiliary vertex of H_s, then, by Lemma 4.7, we have that s reaches all vertices of H_s in $H_s \setminus (y, x_0)$, except x_0, which is unreachable from s in $H_s \setminus (y, x_0)$. This implies that every vertex of H_s^R can reach s in $H_s^R \setminus (x_0, y)$, except x_0, which cannot reach s in $H_s^R \setminus (x_0, y)$. Since x is an ordinary vertex of H_s, we have that $x \neq x_0$, and so x can reach s in $H_s^R \setminus (x_0, y)$. Now Lemma 4.8 implies that x can reach s in $H_{ss} \setminus (x, y)$. Thus, every vertex of H_{ss} can reach s in $H_{ss} \setminus (x, y)$, and so we have that $H_{ss} \setminus (x, y)$ is strongly connected. (3.2) Let x be ao and y be oa. Now let $\{(x_1, y), \dots, (x_k, y)\}$ be the set of all edges of H_s^R that correspond to (x, y). Then $\{(y, x_1), \dots, (y, x_k)\}$ is a set of edges of H_s that stem from y. Thus, by Lemma 4.9 we have that s can reach all vertices of H_s in $H_s \setminus \{(y, x_1), \dots, (y, x_k)\}$. This implies that s can be reached by all vertices of H_s^R in $H_s^R \setminus \{(x_1, y), \dots, (x_k, y)\}$. In particular,

there is a path P from x to s in $H_s^R \setminus \{(x_1,y),\dots,(x_k,y)\}$. Then, by Lemma 4.2, the corresponding path P_H of P in H_{ss} goes from x to s and avoids the edge (x,y) of H_{ss}. This means that s is reachable from x in $H_{ss} \setminus (x,y)$, and therefore every vertex of H_{ss} reaches s in $H_{ss} \setminus (x,y)$. Thus we have established that $H_{ss} \setminus (x,y)$ is strongly connected. (3.3) Let x be aa and y be oo. Let (x_0,y) be an edge of H_s^R that corresponds to (x,y) in H_{ss}. Then (y,x_0) is an edge of H_s. Now, if x_0 is ordinary in H_s or it is auxiliary in H_s but $x_0 \neq x$, then we can argue as in (3.1) in order to establish that $H_{ss} \setminus (x,y)$ is strongly connected. Otherwise, suppose that $x_0 = x$. Then, since x is auxiliary in H_s and y is ordinary in H_s (and H_s contains no critical vertex), we have that $y = d(x)$ in G_s. Now, by Lemma 4.7 we have that s reaches all vertices of H_s in $H_s \setminus (y,x)$, except x, which is unreachable from s in $H_s \setminus (y,x)$. This implies that every vertex of H_s^R can reach s in $H_s^R \setminus (x,y)$, except x, which cannot reach s in $H_s^R \setminus (x,y)$. Now Lemma 4.8 implies that every vertex of H_{ss} can reach s in $H_{ss} \setminus (x,y)$, except possibly x. Thus, we have that either $H_{ss} \setminus (x,y)$ is strongly connected, or its strongly connected components are $\{x\}$ and $H_{ss} \setminus \{x\}$. Furthermore, since (y,x) is the only incoming edge to x in H_s, we have that (x,y) is the only outgoing edge from x in H_s^R. Therefore, since y is ordinary in H_{ss}, we have that x has no descendants in $D(H_s^R)$. This means that (x,y) is the only outgoing edge of x in H_{ss}. (3.4) We do not have to consider the case that x is aa and y is oa, because this implies that (y,x) is an edge between two auxiliary vertices in H_s, which is impossible to occur.

(4) It is impossible that x and y are both auxiliary vertices of H_{ss}, since H_{ss} does not contain a critical vertex. $\square$

Lemma 4.12. *Let G_s be a strongly connected digraph, and let r be a marked vertex of G_s. Let (x,y) be a strong bridge of H_{rr}. Then the strongly connected components of $H_{rr} \setminus (x,y)$ are given by one of the following:*

(i) $\{x\}$ and $H_{rr} \setminus \{x\}$, where x is auxiliary in H_r and $y = d(x)$ in G_s (see case (1.3) below)

(ii) $\{x\}$, $\{d(r)\}$ and $H_{rr} \setminus \{x, d(r)\}$, where x is oa in H_{rr}, y is oo in H_{rr}, $y = d(x)$ in G_s, and $(d(r),x)$ is the only kind of outgoing edge of $d(r)$ in H_{rr} (see case (1.3) below)

(iii) $\{x\}$ and $H_{rr} \setminus \{x\}$, where x is auxiliary in H_{rr} and (x,y) is the only outgoing edge of x in H_{rr} (see cases (3.3) and (3.4) below)

(iv) $\{y\}$ *and* $H_{rr} \setminus \{y\}$*, where* y *is auxiliary in* H_{rr} *and* $x = d(y)$ *in* H_r^R *(see the beginning of* (2)*, and cases* (2.1) *and* (2.2) *below)*

(v) $\{x\}$*,* $\{y\}$ *and* $H_{rr} \setminus \{x, y\}$*, where* x *is oa in* H_{rr}*,* y *is ao in* H_{rr}*,* $x = d(y)$ *in* H_r^R *and* $y = d(x)$ *in* G_s *(see case* (2.3) *below)*

(vi) $\{x\}$*,* $\{y\}$*,* $\{d(r)\}$ *and* $H_{rr} \setminus \{x, y, d(r)\}$*, where* x *is oa in* H_{rr}*,* y *is ao in* H_{rr}*,* $x = d(y)$ *in* H_r^R*,* $y = d(x)$ *in* G_s*, and* $(d(r), x)$ *is the only kind of outgoing edge of* $d(r)$ *in* H_{rr} *(see case* (2.3) *below)*

Proof. First notice that $d(r)$ is an auxiliary vertex of H_r and has only one outgoing edge $(d(r), r)$. Thus, vertex $d(r)$ in H_r^R has only one incoming edge $(r, d(r))$, and so this must be a bridge of H_r^R. This means that $d(r)$ is an aa-vertex of H_{rr}. Now let (x, y) be an edge of H_{rr}. We distinguish four different cases, depending on whether x and y are ordinary or auxiliary vertices of H_{rr}.

(1) Let x and y be both ordinary vertices of H_{rr}. Here we distinguish four different cases, depending on whether x and y are ordinary or auxiliary in H_r. Notice that, in any case, (y, x) is an edge of H_r. (1.1) Let x and y be both oo in H_{rr}. By Lemma 4.6 we have that r can reach all vertices of H_r in $H_r \setminus (y, x)$. Thus we infer that r is reachable from all vertices of H_r^R in $H_r^R \setminus (x, y)$, and so Lemma 4.10 implies that r is reachable from all vertices of H_{rr} in $H_{rr} \setminus (x, y)$. Now, since, x and y are both ordinary vertices of H_{rr}, by Lemma 4.6 we have that r reaches all vertices of $H_{rr} \setminus (x, y)$. We conclude that $H_{rr} \setminus (x, y)$ is strongly connected. (1.2) Let x be oo and y be oa. Then we can argue as in (1.1) in order to conclude that $H_{rr} \setminus (x, y)$ is strongly connected. (1.3) Let x be oa and y be oo. By Lemma 4.7 we have that r can reach all vertices of H_r in $H_r \setminus (y, x)$, except x, which is unreachable from r in $H_r \setminus (y, x)$, and possibly $d(r)$. In any case, since $d(r)$ has only one outgoing edge $(d(r), r)$ in H_r, we also have that $d(r)$ cannot reach x in $H_r \setminus (y, x)$. This implies that r is reachable from all vertices of H_r^R in $H_r^R \setminus (x, y)$, except from x, and possibly $d(r)$; and that, in any case, x cannot reach $d(r)$ in $H_r^R \setminus (x, y)$. Now Lemma 4.10 implies that r is reachable from all vertices of H_{rr} in $H_{rr} \setminus (x, y)$, except from x, and possibly $d(r)$, and that, in any case, x cannot reach $d(r)$ in H_{rr}. We conclude that there are two possibilities: the strongly connected components of $H_{rr} \setminus (x, y)$ are either $\{x\}$ and $H_{rr} \setminus \{x\}$, or $\{x\}$, $\{d(r)\}$ and $H_{rr} \setminus \{x, d(r)\}$. (1.4) Let x and y be both oa. This implies that (y, x) is an edge of H_s both of whose endpoints are auxiliary. But this is impossible to occur since none of those vertices is $d(r)$ (which is aa in H_{rr}).

(2) Let x be ordinary in H_{rr} and let y be auxiliary in H_{rr}. If $y = d(r)$, then x must necessarily be r, since $d(r)$ has only one incoming edge $(r, d(r))$ in H_r^R. So let us consider first the case that $(x, y) = (r, d(r))$, which is the only case in which $d(r)$ can appear as an endpoint of (x, y) if y is auxiliary in H_{rr}. By Corollary 4.1, we have that H_r^R is strongly connected. Thus, even by removing the edge $(r, d(r))$, we still have that r is reachable from every vertex of H_r^R. By Lemma 4.10, this implies that every vertex of H_{rr} is reachable from r in $H_{rr} \setminus (r, d(r))$. Now, by Lemma 4.7 we have that r reaches every vertex of $H_{rr} \setminus (r, d(r))$, except $d(r)$, which is unreachable from r in $H_{rr} \setminus (r, d(r))$. Thus we conclude that the strongly connected components of $H_{rr} \setminus (r, d(r))$ are $\{d(r)\}$ and $H_{rr} \setminus \{d(r)\}$. Now, due to the preceding argument, in what follows we can assume that none of x and y are $d(r)$. This means that y is a marked vertex of H_r^R and $x = d(y)$ in H_r^R. Thus, (y, x) is an edge of H_r. Now we distinguish four possibilities, depending on whether x and y are ordinary or auxiliary in H_r. (2.1) Let x be oo and y be ao. Since x and y are ordinary vertices of H_r, and (y, x) is an edge of H_r, we can argue as in (1.1) in order to show that r is reachable from every vertex of H_{rr} in $H_{rr} \setminus (x, y)$. Now, by Lemma 4.7 we have that r reaches every vertex of H_{rr} in $H_{rr} \setminus (x, y)$, except y, which is unreachable from r in $H_{rr} \setminus (x, y)$. We conclude that the strongly connected components of $H_{rr} \setminus (x, y)$ are $\{y\}$ and $H_{rr} \setminus \{y\}$. (2.2) Let x be oo and y be aa. Then we can argue as in (2.1) in order to show that the strongly connected components of $H_{rr} \setminus (x, y)$ are $\{y\}$ and $H_{rr} \setminus \{y\}$. (2.3) Let x be oa and y be ao. This means that $y = d(x)$ in H_r, and so, by Lemma 4.7, we have that r can reach every vertex of H_r in $H_r \setminus (y, x)$, except x, which is unreachable from r in $H_r \setminus (y, x)$, and possibly $d(r)$. In any case, we have that $d(r)$ cannot reach x in $H_r \setminus (y, x)$, since there is only one outgoing edge $(d(r), r)$ of $d(r)$ in H_r. Thus we have that r is reachable from every vertex of H_r^R in $H_r^R \setminus (x, y)$, except from x, which cannot reach r in H_r^R, and possibly $d(r)$. In any case, we have that $d(r)$ is not reachable from x in $H_r^R \setminus (x, y)$. By Lemma 4.10, this implies that r is reachable from every vertex of H_{rr} in $H_{rr} \setminus (x, y)$, except from x, which cannot reach r in H_r^R, and possibly $d(r)$. In any case, we have that $d(r)$ is not reachable from x in $H_{rr} \setminus (x, y)$. Now, by Lemma 4.7 we have that r can reach every vertex of $H_{rr} \setminus (x, y)$, except y, which is unreachable from r in $H_{rr} \setminus (x, y)$. Furthermore, no vertex of H_{rr} can reach y in $H_{rr} \setminus (x, y)$, since (x, y) is the only incoming edge of y in H_{rr}. We conclude that the strongly connected components of $H_{rr} \setminus (x, y)$ are either $\{x\}$, $\{y\}$ and $H_{rr} \setminus \{x, y\}$, or $\{x\}$, $\{y\}$, $\{d(r)\}$ and $H_{rr} \setminus \{x, y, d(r)\}$. (2.4) Let x be oa and let y be aa. This means that both x and y are

auxiliary vertices of H_r, and so one of them must be $d(r)$. But we have forbidden this case to occur since it was treated in the beginning of (2).

(3) Let x be an auxiliary vertex of H_{rr} and let y be an ordinary vertex of H_{rr}. Since H_{rr} has no critical vertex, this means that x is a marked vertex of H_r^R with $d(x) \in T(r)$ in H_r^R. Now we distinguish four different cases, depending on whether x and y are ordinary or auxiliary in H_r. In any case, by Lemma 4.6 we have that r can reach every vertex of H_{rr} in $H_{rr} \setminus (x, y)$. (3.1) Let x be ao and y be oo. Let (x_0, y) be an edge of H_r^R that corresponds to (x, y) in H_{rr}. Then (y, x_0) is an edge of H_r. If x_0 is an ordinary vertex of H_r, then by Lemma 4.6 we have that r reaches all vertices of H_r in $H_r \setminus (y, x_0)$. This implies that every vertex of H_r^R can reach r in $H_r^R \setminus (x_0, y)$. Then Lemma 4.8 implies that x can reach r in $H_{rr} \setminus (x, y)$. Thus, every vertex of H_{rr} can reach r in $H_{rr} \setminus (x, y)$, and so we have that $H_{rr} \setminus (x, y)$ is strongly connected. Similarly, if x_0 is an auxiliary vertex of H_r, then, by Lemma 4.7, we have that r reaches all vertices of H_r in $H_r \setminus (y, x_0)$, except x_0, which is unreachable from r in $H_r \setminus (y, x_0)$, and possibly $d(r)$. This implies that every vertex of H_r^R can reach r in $H_r^R \setminus (x_0, y)$, except x_0, which cannot reach r in $H_r^R \setminus (x_0, y)$, and possibly $d(r)$. Since x is an ordinary vertex of H_r, we have that $x \neq x_0$, and so x can reach r in $H_r^R \setminus (x_0, y)$. Now Lemma 4.8 implies that x can reach r in $H_{rr} \setminus (x, y)$. Thus, since H_{rr} is strongly connected, we have that every vertex of H_{rr} can reach r in $H_{rr} \setminus (x, y)$, and so we have that $H_{rr} \setminus (x, y)$ is strongly connected. (3.2) Let x be ao and y be oa. Let $\{(x_1, y), \dots, (x_k, y)\}$ be the collection of all edges of H_r^R that correspond to (x, y) in H_{rr}. Then $\{(y, x_1), \dots, (y, x_k)\}$ is a set of outgoing edges from y in H_r. Thus, Lemma 4.9 implies that r can reach all vertices of H_r in $H_r \setminus \{(y, x_1), \dots, (y, x_k)\}$, except possibly $d(r)$. This implies that r can be reached from all vertices of H_r^R in $H_r^R \setminus \{(x_1, y), \dots, (x_k, y)\}$, except possibly from $d(r)$. In particular, there is a path P from x to r in H_r^R that avoids all edges in $\{(x_1, y), \dots, (x_k, y)\}$. Then, by Lemma 4.2 we have that the corresponding path P_H in H_{rr} avoids (x, y). This means that r is reachable from x in $H_{rr} \setminus (x, y)$. Since H_{rr} is strongly connected (by Corollary 4.1), we know that all vertices of H_{rr} can reach r. Thus they can reach r by avoiding the edge (x, y). We conclude that $H_{rr} \setminus (x, y)$ is strongly connected. (3.3) Let x be aa and y be oo. First, let us assume that $x = d(r)$ (in G_s). Since H_{rr} is strongly connected and $(r, d(r))$ is the only incoming edge to $d(r)$ in H_{rr}, we have that every vertex from H_{rr} can reach r in $H_{rr} \setminus (x, y)$, except possibly $d(r)$. Thus, we have that $H_{rr} \setminus (x, y)$ is strongly connected if and only if $d(r)$ has at least two outgoing edges in H_{rr}. Otherwise, (x, y) is the only outgoing edge from $d(r)$ in H_{rr}, and then the strongly connected components of $H_{rr} \setminus (x, y)$ are $\{x\}$ and $H_{rr} \setminus \{x\}$. Now let us assume that $x \neq d(r)$. Notice that x is a marked vertex of H_r^R. Let (x_0, y) be an edge of H_r^R that corresponds to (x, y) in H_{rr}. Then (y, x_0) is an edge of H_r. Now, if x_0 is ordinary in H_r or it is auxiliary in H_r but $x_0 \neq x$, then we can argue as in (3.1) in order to establish that $H_{rr} \setminus (x, y)$ is strongly connected. Otherwise, suppose that $x_0 = x$. Then, since x is auxiliary in H_r and y is ordinary in H_r (and $x \neq d(r)$), we have that $y = d(x)$ in G_s. Now, by Lemma 4.7 we have that r reaches all vertices of H_r in $H_r \setminus (y, x)$, except x, which is unreachable from r in $H_r \setminus (y, x)$, and possibly $d(r)$. This implies that every vertex of H_r^R can reach r in $H_r^R \setminus (x, y)$, except x, which cannot reach r in $H_r^R \setminus (x, y)$, and possibly $d(r)$. Now Lemma 4.8 implies that every vertex of H_{rr} can reach r in $H_{rr} \setminus (x, y)$, except possibly x and $d(r)$. But since H_{rr} is strongly connected (by Corollary 4.1) and there is no edge $(d(r), x)$ in H_{rr} (since x and $d(r)$ are both auxiliary, but not critical, vertices of H_{rr}), we also have that $d(r)$ reaches r in $H_{rr} \setminus (x, y)$. Thus we conclude that either $H_{rr} \setminus (x, y)$ is strongly connected, or its strongly connected components are $\{x\}$ and $H_{rr} \setminus \{x\}$. (3.4) Let x be aa and y be oa. If $x \neq d(r)$, then we can argue as in (3.2) in order to conclude that $H_{rr} \setminus (x, y)$ is strongly connected. Otherwise, let $x = d(r)$. Then, since H_{rr} is strongly connected (by Corollary 4.1), we have that r is reachable from every vertex of H_{rr}. Since $d(r)$ has only one incoming edge $(r, d(r))$ in H_{rr}, this implies that every vertex of $H_{rr} \setminus \{d(r)\}$ can reach r by avoiding $d(r)$. Thus, if $d(r)$ has more than one outgoing edge in H_{rr}, we have that $H_{rr} \setminus (x, y)$ is strongly connected. Otherwise, the strongly connected components of $H_{rr} \setminus (x, y)$ are $\{d(r)\}$ and $H_{rr} \setminus \{d(r)\}$.

(4) It is impossible that both x and y are auxiliary vertices of H_{rr}, since H_{rr} does not contain a critical vertex. □

We can simplify the result of Lemma 4.12 by essentially eliminating cases (ii) and (vi). This is done by observing that, in those cases, the only outgoing edges from $d(r)$ have the form $(d(r), x)$, where x is an oa vertex in H_{rr}. To achieve the simplification, we remove $d(r)$ from H_{rr}, and we add a new edge (r, x). This is in order to replace the path $(r, d(r)), (d(r), x)$ of H_{rr} with (r, x). We call the resulting graph $\widetilde{H}_{rr}$. The following two lemmas demonstrate that $\widetilde{H}_{rr}$ maintains all the information that we need for our purposes.

Lemma 4.13. *For any two oo vertices u, v of $\widetilde{H}_{rr}$ we have $u \overset{\widetilde{H}_{rr}}{\leftrightarrow}_{2et} v$ if and only if $u \overset{H_{rr}}{\leftrightarrow}_{2et} v$.*

Proof. First, observe that x has only one outgoing edge in H_{rr}. This is because x is an auxiliary vertex of H_r, and thus (since $x \neq d(r)$) it has only one incoming edge in H_r. This means that x has only one outgoing edge in H_r^R, and thus it has only one outgoing edge in H_{rr}, since it is ordinary in H_{rr}.

Now let u, v be two *oo* vertices of $\widetilde{H}_{rr}$. Suppose first that $u \overset{H_{rr}}{\leftrightarrow}_{2et} v$. Let e be an edge of $\widetilde{H}_{rr}$ different from (r, x). Then e is also an edge of H_{rr}, and u, v are twinless strongly connected in $H_{rr} \setminus e$. This means that there is a path P in $H_{rr} \setminus e$ from u to v, and a path Q in $H_{rr} \setminus e$ from v to u, such that there is no edge $(z, w) \in P$ with $(w, z) \in Q$. Now, if P and Q do not pass from $d(r)$, then they are paths in $\widetilde{H}_{rr}$, and this implies that u, v are twinless strongly connected in $\widetilde{H}_{rr} \setminus e$. Otherwise, at least one of P, Q has $(r, d(r)), (d(r), x)$ as a subpath. Then we replace every occurrence of this subpath with (r, x). If the outgoing edge of x in $\widetilde{H}_{rr}$ is not (x, r), then we still have that P and Q do not share a pair of antiparallel edges, and so u, v are twinless strongly connected in $\widetilde{H}_{rr}$. Otherwise, since P and Q must end in an *oo* vertex, for every occurrence of $d(r)$ in them they must pass from the path $(r, d(r)), (d(r), x), (x, r)$. Then we can simply discard every occurrence of this triangle. This shows that u, v are twinless strongly connected in $\widetilde{H}_{rr} \setminus e$. Now let $e = (r, x)$. Since $u \overset{H_{rr}}{\leftrightarrow}_{2et} v$, we have that u, v are twinless strongly connected in $H_{rr} \setminus (r, d(r))$. This means that there is a path P in $H_{rr} \setminus (r, d(r))$ from u to v, and a path Q in $H_{rr} \setminus (r, d(r))$ from v to u, such that there is no edge $(z, w) \in P$ with $(w, z) \in Q$. Since $(r, d(r))$ is the only incoming edge to $d(r)$ in H_{rr}, this means that P, Q do not pass from $d(r)$. Also, P, Q do not use the edge (r, x). This means that u, v are twinless strongly connected in $\widetilde{H}_{rr} \setminus (r, x)$. Thus we have $u \overset{\widetilde{H}_{rr}}{\leftrightarrow}_{2et} v$.

Conversely, suppose that u, v are not 2-edge twinless strongly connected in H_{rr}. Then there is an edge $e \in H_{rr}$ and a pair of antiparallel edges $(z, w), (w, z) \in H_{rr}$, such that for every path P in $H_{rr} \setminus e$ from u to v and every path Q in $H_{rr} \setminus e$ from v to u, we have $(z, w) \in P$ and $(w, z) \in Q$. Suppose first that e is not one of $(r, d(r)), (d(r), x)$. Now let us assume, for the sake of contradiction, that there is a path P in $\widetilde{H}_{rr} \setminus e$ from u to v that does not contain (z, w). Then we can construct a path P' in H_{rr} by replacing every occurrence of (r, x) in P with the segment $(r, d(r)), (d(r), x)$. Observe that (z, w) is neither $(r, d(r))$ nor $(d(r), x)$, because (w, z) is also an edge of H_{rr} (and there cannot be an edge $(d(r), r)$, because $(d(r), x)$ is the only outcoming edge of $d(r)$; and also there cannot be an edge $(x, d(r))$, because $(r, d(r))$ is the only incoming edge to $d(r)$ in H_{rr}, and $x \neq r$ because x is auxiliary in H_r but r is ordinary in H_r). Thus

P' is a path in $H_{rr} \setminus e$ from u to v that does not use (z, w) - a contradiction. Similarly, we can show that there is no path Q in $\widetilde{H}_{rr} \setminus e$ from v to u that does not use (w, z). This means that u, v are not twinless strongly connected in $\widetilde{H}_{rr} \setminus e$. Now suppose that e is either $(r, d(r))$ or $(d(r), x)$. Then we can see that u, v are not twinless strongly connected in $\widetilde{H}_{rr} \setminus (r, x)$, because every path in $\widetilde{H}_{rr} \setminus (r, x)$ from u to v, or from v to u, is also a path in $H_{rr} \setminus e$. □

Lemma 4.14. *Let e be an edge of $\widetilde{H}_{rr}$. Then e is a strong bridge of $\widetilde{H}_{rr}$ if and only if it is a strong bridge of H_{rr}. The strongly connected components of $\widetilde{H}_{rr} \setminus e$ are given essentially by Lemma 4.12 (where we replace "H_{rr}" with "$\widetilde{H}_{rr}$").*

Proof. First observe that (r, x) is not a strong bridge of $\widetilde{H}_{rr}$, because x is an ordinary vertex of H_{rr}, and so, by removing any edge from H_{rr}, r can still reach x in H_{rr}. In particular, by removing $(r, d(r))$ or $(d(r), x)$ from H_{rr}, r can still reach x. From this, we can conclude that (r, x) is not a strong bridge of $\widetilde{H}_{rr}$.

Now let e be an edge of $\widetilde{H}_{rr}$, but not (r, x). Then e is also an edge of H_{rr}. Let u, v be two vertices of $\widetilde{H}_{rr}$. We will show that u can reach v in $\widetilde{H}_{rr} \setminus e$ if and only if u can reach v in $H_{rr} \setminus e$. Let P be a path in $\widetilde{H}_{rr} \setminus e$ from u to v. We can construct a path in $H_{rr} \setminus e$ by replacing every occurrence of (r, x) in P with the segment $(r, d(r)), (d(r), x)$. This shows that u can reach v in $H_{rr} \setminus e$. Conversely, let P be a path in $H_{rr} \setminus e$ from u to v. If P passes from $d(r)$, then it must necessarily use the segment $(r, d(r)), (d(r), x))$ (because $(r, d(r))$ is the only incoming edge to $d(r)$ in H_{rr}, and $(d(r), x)$ is the only king of outgoing edge from $d(r)$ in H_{rr}). Then we can construct a path in $\widetilde{H}_{rr} \setminus r$ by replacing every occurrence of the segment $(r, d(r)), (d(r), x))$ with (r, x). This shows that u can reach v in $\widetilde{H}_{rr} \setminus e$. Thus we have that the strongly connected components of $\widetilde{H}_{rr} \setminus e$ coincide with those of $H_{rr} \setminus e$ (by excluding $d(r)$ from the strongly connected component of $H_{rr} \setminus e$ that contains $d(r)$). □

Before we move on to describe the structure of the strongly connected components of the remaining graphs, we will need the following two lemmas.

Lemma 4.15. *Let G_s be a strongly connected digraph and let r be a marked vertex of G_s. Let (x, y) be an edge of H_r such that either both x and y are ordinary in H_r, or y is auxiliary in H_r and $x = d(y)$ in G_s. Let u be a vertex of H_r. If u reaches s in $G_s \setminus (x, y)$, then u reaches $d(r)$ in $H_r \setminus (x, y)$.*

Proof. Let P be a path from u to s in $G_s \setminus (x, y)$. Since r is a marked vertex of G_s, by Lemma 4.1 we have that there is a path Q from s to $d(r)$ that avoids every vertex of $D(r)$. In particular, Q does not use the edge (x, y). Let $P' = P + Q$ be the concatenation of P and Q in G_s. This is a path from u to $d(r)$ in $G_s \setminus (x, y)$. Now consider the corresponding path P_H of P' in H_r. By Lemma 4.2, this is a path from u to $d(r)$ in H_r that avoids (x, y). $\square$

Lemma 4.16. *Let G_s be a strongly connected digraph and let r be a marked vertex of G_s. Let x be a marked vertex of G_s such that $d(x) \in T(r)$. Let also (x_0, y) be an edge of G_s such that x_0 is a descendant of x in $D(G_s)$, y is not a descendant of x in $D(G_s)$, and $y \in T(r)$. Let u be a vertex of H_r. If u reaches s in $G_s \setminus (x_0, y)$, then u reaches $d(r)$ in $H_r \setminus (x, y)$.*

Proof. Let u be a vertex of H_r and let P be a path from u to s in $G_s \setminus (x_0, y)$. Since r is a marked vertex of G_s, by Lemma 4.1 we have that there is a path Q from s to $d(r)$ that avoids every vertex of $D(r)$. In particular, Q does not use the edge (x_0, y). Let $P' = P + Q$ be the concatenation of P and Q in G_s. This is a path from u to $d(r)$ in $G_s \setminus (x, y)$. Now consider the corresponding path P_H of P' in H_r. P_H has the same endpoints as P'. Thus, if P_H avoids the edge (x, y) of H_r, then we are done. Otherwise, this means that P' uses an edge (x_1, y) of G_s, distinct from (x_0, y) (although we may have $x_1 = x_0$, since we allow multiple edges), such that x_1 is a descendant of x in $D(G_s)$. But then we must have that (x, y) is a double edge of H_r, and therefore $H_r \setminus (x, y)$ is strongly connected (since H_r is strongly connected, by Corollary 4.1). Thus u can reach $d(r)$ in $H_r \setminus (x, y)$. $\square$

Lemma 4.17. *Let G_s be a strongly connected digraph, and let r be a marked vertex of H_s^R. Let (x, y) be a strong bridge of H_{sr} where none of x and y is $d(r)$ in H_s^R. Then the strongly connected components of $H_{sr} \setminus (x, y)$ are given by one of the following:*

(i) $\{x\}$ *and* $H_{sr} \setminus \{x\}$, *where x is auxiliary in H_s and $y = d(x)$ in G_s (see case* (1.3) *below)*

(ii) $\{x\}$ *and* $H_{sr} \setminus \{x\}$, *where x is auxiliary in H_{sr} and (x, y) is the only outgoing edge of x in H_{sr} (see case* (3.3) *below)*

(iii) $\{y\}$ *and* $H_{sr} \setminus \{y\}$, *where y is auxiliary in H_{sr} and $x = d(y)$ in H_s^R (see cases* (2.1) *and* (2.2) *below)*

(iv) $\{x\}$, $\{y\}$ *and* $H_{sr} \setminus \{x, y\}$, *where x is auxiliary in H_s, y is auxiliary in H_{sr}, $x = d(y)$ in H_s^R and $y = d(x)$ in G_s (see case* (2.3) *below)*

Proof. Let (x, y) be an edge of H_{sr}, where none of x and y is $d(r)$ in H_s^R. We distinguish four different cases, depending on whether x and y are ordinary or auxiliary vertices of H_{sr}.

(1) Let both x and y be ordinary vertices of H_{sr}. This implies that (y, x) is an edge in H_s. Here we distinguish four cases, depending on whether x and y are ordinary or auxiliary in H_s. (1.1) Let x be oo and y be oo. Then, by Lemma 4.6 we have that s can reach all vertices of H_s in $H_s \setminus (y, x)$. This implies that s can be reached by all vertices of H_s^R in $H_s^R \setminus (x, y)$. Then Lemma 4.15 implies that all vertices of H_{sr} can reach $d(r)$ in $H_{sr} \setminus (x, y)$. Conversely, as an implication of Lemma 4.6, we have that $d(r)$ can reach every vertex of H_{sr} in $H_{sr} \setminus (x, y)$. We conclude that $H_{sr} \setminus (x, y)$ is strongly connected. (1.2) Let x be oo and y be oa. Then we can argue as in (1.1) in order to conclude that $H_{sr} \setminus (x, y)$ is strongly connected. (1.3) Let x be oa and y be oo. Since H_s contains no critical vertex, we have that x is a marked vertex of G_s and $y = d(x)$ in G_s. Then, by Lemma 4.7 we have that s can reach all vertices of H_s in $H_s \setminus (y, x)$, except x, which is unreachable from s in $H_s \setminus (y, x)$. This implies that s is reachable from all vertices of H_s^R in $H_s^R \setminus (x, y)$, except from x, which cannot reach s in $H_s^R \setminus (x, y)$. Then Lemma 4.15 implies that all vertices of H_{sr} can reach $d(r)$ in $H_{sr} \setminus (x, y)$, except possibly x. Now, as an implication of Lemma 4.6, we have that $d(r)$ can reach all vertices of H_{sr} in $H_{sr} \setminus (x, y)$. We conclude that either $H_{sr} \setminus (x, y)$ is strongly connected, or its strongly connected components are $\{x\}$ and $H_{sr} \setminus \{x\}$. (1.4) Let x be oa and y be oa. But this implies that both x and y are auxiliary vertices of H_s, which is impossible to occur.

(2) Let x be an ordinary vertex of H_{sr} and let y be an auxiliary vertex of H_{sr}. Since $y \neq d(r)$ in H_s^R, this means that y is a marked vertex of H_s^R and $x = d(y)$ in H_s^R. Thus (y, x) is an edge of H_s. Now we distinguish four different cases, depending on whether x and y are ordinary or auxiliary in H_s. (2.1) Let x be oo and y be ao. Then we can argue as in (1.1) in order to show that $d(r)$ is reachable from every vertex of H_{sr} in $H_{sr} \setminus (x, y)$. Now, as an implication of Lemma 4.7, we have that $d(r)$ reaches every vertex of H_{sr} in $H_{sr} \setminus (x, y)$, except y, which is unreachable from $d(r)$ in $H_{sr} \setminus (x, y)$. Thus we conclude that the strongly connected components of $H_{sr} \setminus (x, y)$ are $\{y\}$ and $H_{sr} \setminus \{y\}$. (2.2) Let x be oo and y be aa. Then we can argue as in (2.1) in order to conclude that the strongly connected components of $H_{sr} \setminus (x, y)$ are $\{y\}$ and $H_{sr} \setminus \{y\}$. (2.3). Let x be oa and y be ao. Then, by Lemma 4.7 we have that s can reach all vertices of H_s in $H_s \setminus (y, x)$, except x, which is unreachable from s in

$H_s \setminus (y, x)$. This implies that s can be reached by all vertices of H_s^R in $H_s^R \setminus (x, y)$, except from x, which cannot reach s in $H_s^R \setminus (x, y)$. By Lemma 4.15, this implies that $d(r)$ is reachable from all vertices of H_{sr} in $H_{sr} \setminus (x, y)$, except from x, which cannot reach $d(r)$ in $H_{sr} \setminus (x, y)$. Now, as an implication of Lemma 4.7 we have that $d(r)$ can reach all vertices of H_{sr} in $H_{sr} \setminus (x, y)$, except y, which is unreachable from $d(r)$ in $H_{sr} \setminus (x, y)$. We conclude that the strongly connected components of $H_{sr} \setminus (x, y)$ are $\{x\}$, $\{y\}$ and $H_{sr} \setminus \{x, y\}$. (2.4) Let x be oa and y be aa. But this implies that both x and y are auxiliary vertices of H_s, which is impossible to occur.

(3) Let x be an auxiliary vertex of H_{sr} and let y be an ordinary vertex of H_{sr}. Since $x \neq d(r)$ in H_s^R, this means that x is a marked vertex of H_s^R with $d(x) \in T(r)$ in H_s^R. Now we distinguish four different cases, depending on whether x and y are ordinary or auxiliary in H_s. In any case, as an implication of Lemma 4.6 we have that $d(r)$ can reach every vertex of H_{sr} in $H_{sr} \setminus (x, y)$. (3.1) Let x be ao and y be oo. Let (x_0, y) be an edge of H_s^R that corresponds to (x, y) in H_{sr}. Then (y, x_0) is an edge of H_s. If x_0 is an ordinary vertex of H_s, then by Lemma 4.6 we have that s reaches all vertices of H_s in $H_s \setminus (y, x_0)$. This implies that every vertex of H_s^R can reach s in $H_s^R \setminus (x_0, y)$. Then Lemma 4.16 implies that every vertex of H_{sr} can reach $d(r)$ in $H_{sr} \setminus (x, y)$, and so we have that $H_{sr} \setminus (x, y)$ is strongly connected. Similarly, if x_0 is an auxiliary vertex of H_s, then, by Lemma 4.7, we have that s reaches all vertices of H_s in $H_s \setminus (y, x_0)$, except x_0, which is unreachable from s in $H_s \setminus (y, x_0)$. This implies that every vertex of H_s^R can reach s in $H_s^R \setminus (x_0, y)$, except x_0, which cannot reach s in $H_s^R \setminus (x_0, y)$. Since x is an ordinary vertex of H_s, we have that $x \neq x_0$, and so x can reach s in $H_s^R \setminus (x_0, y)$. Then Lemma 4.16 implies that every vertex of H_{sr} can reach $d(r)$ in $H_{sr} \setminus (x, y)$, and so we have that $H_{sr} \setminus (x, y)$ is strongly connected. (3.2) Let x be ao and y be oa. Let $\{(x_1, y), \ldots, (x_k, y)\}$ be the set of all the incoming edges to y in H_s^R. Then $\{(y, x_1), \ldots, (y, x_k)\}$ is a collection of edges of H_s that stem from y. Since H_s contains no critical vertex, we have that y is a marked vertex of G_s with $d(y) \in T(s)$. By Lemma 4.9 we have that s can reach all vertices of H_s in $H_s \setminus \{(y, x_1), \ldots, (y, x_k)\}$. This implies that s can be reached from all vertices of H_s^R in $H_s^R \setminus \{(x_1, y), \ldots, (x_k, y)\}$. Furthermore, since y is dominated by $d(r)$ in H_s^R, we have that there is a path from s to $d(r)$ in H_s^R that avoids y; therefore, this path also avoids all edges $\{(x_1, y), \ldots, (x_k, y)\}$. Thus we infer that $d(r)$ is reachable from all vertices of H_s^R in $H_s^R \setminus \{(x_1, y), \ldots, (x_k, y)\}$. In particular, let P be a path from x to $d(r)$ in $H_s^R \setminus \{(x_1, y), \ldots, (x_k, y)\}$. Then, by Lemma 4.2 the corresponding path P_H in

H_{sr} has the same endpoints as P and avoids the edge (x, y). Thus, even by removing (x, y) from H_{sr}, all vertices from H_{sr} can reach $d(r)$. We conclude that $H_{sr} \setminus (x, y)$ is strongly connected. (3.3) Let x be aa and y be oo. Let (x_0, y) be an edge of H_s^R that corresponds to (x, y) in H_{sr}. Then (y, x_0) is an edge of H_s. Now, if x_0 is ordinary in H_s or it is auxiliary in H_s but $x_0 \neq x$, then we can argue as in (3.1) in order to establish that $H_{sr} \setminus (x, y)$ is strongly connected. Otherwise, suppose that $x_0 = x$. Then, since x is auxiliary in H_s and y is ordinary in H_s (and H_s contains no critical vertex), we have that $y = d(x)$ in G_s. Now, by Lemma 4.7 we have that s reaches all vertices of H_s in $H_s \setminus (y, x)$, except x, which is unreachable from s in $H_s \setminus (y, x)$. This implies that every vertex of H_s^R can reach s in $H_s^R \setminus (x, y)$, except x, which cannot reach s in $H_s^R \setminus (x, y)$. Now Lemma 4.16 implies that every vertex of H_{sr} can reach $d(r)$ in $H_{sr} \setminus (x, y)$, except possibly x. Thus, we have that either $H_{sr} \setminus (x, y)$ is strongly connected, or its strongly connected components are $\{x\}$ and $H_{sr} \setminus \{x\}$. (3.4) Let x be aa and y be oa. Then we can use the same argument that we used in (3.2) in order to conclude that $H_{sr} \setminus (x, y)$ is strongly connected.

(4) Since $y \neq d(r)$ in H_s^R, it is impossible that both x and y are auxiliary in H_{sr}. $\square$

Lemma 4.18. *Let G_s be a strongly connected digraph, let r be a marked vertex of G_s, and let r' be a marked vertex of H_r^R. Let (x, y) be a strong bridge of $H_{rr'}$ such that neither x nor y is $d(r')$ in H_r^R. Furthermore, if $r' = d(r)$ in G_s, then we also demand that neither x nor y is r'. Then the strongly connected components of $H_{rr'} \setminus (x, y)$ are given by one of the following:*

(i) $\{x\}$ and $H_{rr'} \setminus \{x\}$, where x is auxiliary in H_r and $y = d(x)$ in G_s (see case (1.3) below)

(ii) $\{x\}$ and $H_{rr'} \setminus \{x\}$, where x is auxiliary in $H_{rr'}$ and (x, y) is the only outgoing edge of x in $H_{rr'}$ (see case (3.3) below)

(iii) $\{y\}$ and $H_{rr'} \setminus \{y\}$, where y is auxiliary in $H_{rr'}$ and $x = d(y)$ in H_r^R (see cases (2.1) and (2.2) below)

(iv) $\{y\}$ and $H_{rr'} \setminus \{y\}$, where x is auxiliary in H_r and $y = d(x)$ in G_s (see case (2.3) below)

(v) $\{x\}$, $\{y\}$ and $H_{rr'} \setminus \{x, y\}$, where x is auxiliary in H_r, y is auxiliary in $H_{rr'}$, $x = d(y)$ in H_r^R and $y = d(x)$ in G_s (see case (2.3) below)

(vi) $\{x\}$, $\{y\}$, $\{d(r)\}$ *and* $\{r\}$, *where* $r' = d(r)$, x *is auxiliary in* H_r, y *is auxiliary in* $H_{rr'}$, $x = d(y)$ *in* H_r^R, $y = d(x)$ *in* G_s, *and* $(d(r), x)$ *is the only kind of outgoing edge of* $d(r)$ *in* $H_{rr'}$ *(see case* (2.3) *below)*

Proof. Let (x, y) be an edge of $H_{rr'}$ such that neither x nor y is $d(r')$ in H_r^R, and also, if $r' = d(r)$ in G_s, then neither x nor y is r'. Notice that, if $r' = d(r)$ in G_s, then r' is auxiliary in H_r, and therefore r' is an oa-vertex of $H_{rr'}$. Conversely, $d(r)$ appears as an ordinary vertex in $H_{rr'}$ only when $r' = d(r)$. Therefore, if x or y is ordinary in $H_{rr'}$, then we can immediately infer, due to our demand in the statement of the Lemma, that x or y, respectively, is not $d(r)$. Now we distinguish four cases, depending on whether x and y are ordinary or auxiliary in $H_{rr'}$.

(1) Let both x and y be ordinary vertices of $H_{rr'}$. This implies that (x, y) is also an edge of H_r^R, and therefore (y, x) is an edge of H_r. Now we distinguish four cases, depending on whether x and y are ordinary or auxiliary in H_r. Notice that, in any case, as an implication of Lemma 4.6, we have that $d(r')$ reaches all vertices of $H_{rr'}$ in $H_{rr'} \setminus (x, y)$. (1.1) Let x be oo and y be oo. Then, as an implication of Lemma 4.6, we have that $d(r)$ reaches all vertices of H_r in $H_r \setminus (y, x)$. This implies that all vertices of H_r^R can reach $d(r)$ in $H_r^R \setminus (x, y)$. By Lemma 4.15, this implies that all vertices of $H_{rr'}$ can reach $d(r')$ in $H_{rr'}$ in $H_{rr'} \setminus (x, y)$. We conclude that $H_{rr'} \setminus (x, y)$ is strongly connected. (1.2) Let x be oo and y be oa. Then we can argue as in (1.1) in order to conclude that $H_{rr'} \setminus (x, y)$ is strongly connected. (1.3) Let x be oa and y be oo. Since $x \neq d(r)$, we must have that x is a marked vertex of G_s and $y = d(x)$ in G_s. Then, by Lemma 4.7, we have that r can reach every vertex of H_r in $H_r \setminus (y, x)$, except x, which is unreachable from r in $H_r \setminus (y, x)$, and possibly $d(r)$. This implies that r is reachable from every vertex of H_r^R in $H_r^R \setminus (x, y)$, except from x, which cannot reach r in $H_r^R \setminus (x, y)$, and possibly $d(r)$. Now let us assume, for the sake of contradiction, that $d(r)$ cannot reach r in $H_r^R \setminus (x, y)$, and also that $d(r) \in H_{rr'}$. Since r is reachable from every vertex of $H_r^R \setminus \{x, d(r)\}$ in $H_r^R \setminus (x, y)$, we must have that $(d(r), x)$ is the only kind of outgoing edge of $d(r)$ in H_r^R. Now, since $x \in H_{rr'}$ and the immediate dominator of $d(r)$ in H_r^R is r, we have that x is dominated by $d(r)$ in H_r^R; and since $(d(r), x)$ is an edge of H_r^R, we must have that $d(r)$ is the immediate dominator of x in H_r^R. Now, since $y \in H_{rr'}$, we must also have that y is dominated by $d(r)$ in H_r^R. And then, since $(d(r), x)$ is the only kind of outgoing edge of $d(r)$ in H_r^R, and (x, y) is the only outgoing edge of x in H_r^R (since x is marked in G_s and $y = d(x)$ in G_s), we must have that (x, y) is a bridge of H_r^R, and so y is a marked vertex of H_r^R and

x is the immediate dominator of y in H_r^R. Since y is a marked vertex of H_r^R, but also y is ordinary in $H_{rr'}$, we must have that $r' = y$. But then x is the critical vertex of r' in $H_{rr'}$ - which contradicts the fact that x is ordinary in $H_{rr'}$. Thus we have that either $d(r)$ can reach r in $H_r^R \setminus (x,y)$, or that $d(r) \notin H_{rr'}$. In any case, since r is reachable from every vertex of H_r^R in $H_r^R \setminus (x,y)$, except from x, which cannot reach r in $H_r^R \setminus (x,y)$, by Lemma 4.15 we must have that $d(r')$ is reachable from every vertex of $H_{rr'}$ in $H_{rr'} \setminus (x,y)$, except possibly from x. We conclude that either $H_{rr'} \setminus (x,y)$ is strongly connected, or its strongly connected components are $\{x\}$ and $H_{rr'} \setminus \{x\}$. (1.4) Let x be oa and y be oa. This means that both x and y are auxiliary in H_r, which implies that $y = d(r)$ in G_s. But since y is ordinary in $H_{rr'}$, this case is not permitted to occur.

(2) Let x be an ordinary vertex of $H_{rr'}$ and let y be an auxiliary vertex of $H_{rr'}$. Since $y \neq d(r')$ in H_r^R, this means that y is a marked vertex of H_r^R and $x = d(y)$ in H_r^R. Thus (y,x) is an edge of H_r. Now we distinguish four different cases, depending on whether x and y are ordinary or auxiliary in H_r. (2.1) Let x be oo and y be ao in $H_{rr'}$. Then we can argue as in (1.1) in order to get that all vertices of $H_{rr'}$ can reach $d(r')$ in $H_{rr'}$ in $H_{rr'} \setminus (x,y)$. Now, as an implication of Lemma 4.7, we have that $d(r')$ reaches all vertices of $H_{rr'}$ in $H_{rr'} \setminus (x,y)$, except y, which is unreachable from $d(r')$ in $H_{rr'} \setminus (x,y)$. We conclude that the strongly connected components of $H_{rr'} \setminus (x,y)$ are $\{y\}$ and $H_{rr'} \setminus \{y\}$. (2.2) Let x be oo and y be ao in $H_{rr'}$. Then we can argue as in (2.1) in order to conclude that the strongly connected components of $H_{rr'} \setminus (x,y)$ are $\{y\}$ and $H_{rr'} \setminus \{y\}$. (2.3) Let x be oa and y be ao. First observe that, as an implication of Lemma 4.7, we have that $d(r')$ reaches all vertices of $H_{rr'}$ in $H_{rr'} \setminus (x,y)$, except y, which is unreachable from $d(r')$ in $H_{rr'} \setminus (x,y)$. Now, we can argue as in (1.3) in order to get that r is reachable from every vertex of H_r^R in $H_r^R \setminus (x,y)$, except from x, which cannot reach r in $H_r^R \setminus (x,y)$, and possibly $d(r)$. Then, if we have that either $d(r)$ can also reach r in $H_r^R \setminus (x,y)$, or $d(r) \notin H_{rr'}$, by Lemma 4.15 we must have that $d(r')$ is reachable from every vertex of $H_{rr'}$ in $H_{rr'} \setminus (x,y)$, except possibly from x. Thus, we have that the strongly connected components of $H_{rr'} \setminus (x,y)$ are either $\{y\}$ and $H_{rr'} \setminus \{y\}$, or $\{x\}$, $\{y\}$ and $H_{rr'} \setminus \{x,y\}$. Otherwise, let us assume that $d(r)$ cannot reach r in $H_r^R \setminus (x,y)$, and also that $d(r) \in H_{rr'}$. Since r is reachable from every vertex of $H_r^R \setminus \{x, d(r)\}$ in $H_r^R \setminus (x,y)$, we must have that $(d(r),x)$ is the only kind of outgoing edge of $d(r)$ in H_r^R. Now, since $x \in H_{rr'}$ and the immediate dominator of $d(r)$ in H_r^R is r, we have that x is dominated by $d(r)$ in H_r^R; and since $(d(r),x)$ is an edge of H_r^R, we

must have that $d(r)$ is the immediate dominator of x in H_r^R. Now, since $y \in H_{rr'}$, we must also have that y is dominated by $d(r)$ in H_r^R. And then, since $(d(r), x)$ is the only kind of outgoing edge of $d(r)$ in H_r^R, and (x, y) is the only outgoing edge of x in H_r^R (since x is marked in G_s and $y = d(x)$ in G_s), we must have that (x, y) is a bridge of H_r^R, and so y is a marked vertex of H_r^R and x is the immediate dominator of y in H_r^R. Since x is an ordinary vertex of $H_{rr'}$, and y is marked vertex of H_r^R such that $x = d(y)$ in H_r^R, we must have that either x is marked in H_r^R and $r' = x$, or that $r' = d(x)$ in H_r^R. In any case, recall that $(d(r), x)$ is the only kind of outgoing edge of $d(r)$ in H_r^R, and (x, y) is the only outgoing edge of x in H_r^R. Thus, if $r' = x$, then the vertex set of $H_{rr'}$ is $\{d(r), x, y\}$. Furthermore, the strongly connected components of $H_{rr'} \setminus (x, y)$ in this case are $\{x\}$, $\{y\}$ and $\{d(r)\}$ $(= H_{rr'} \setminus (x, y))$. Otherwise, if $r' = d(r)$, then the vertex set of $H_{rr'}$ is $\{r, d(r), x, y\}$. Furthermore, the strongly connected components of $H_{rr'} \setminus (x, y)$ in this case are $\{x\}$, $\{y\}$, $\{d(r)\}$ and $\{r\}$. (2.4) Let x be *oa* and y be *aa*. This means that both x and y are auxiliary in H_r, which implies that $x = d(r)$ in G_s. But since x is ordinary in $H_{rr'}$, this case is not permitted to occur.

(3) Let x by an auxiliary vertex of $H_{rr'}$ and let y be an ordinary vertex of $H_{rr'}$. Since $x \neq d(r')$ in H_r^R, this means that x is a marked vertex of H_r^R with $d(x) \in T(r')$ in H_r^R. Now we distinguish four different cases, depending on whether x and y are ordinary or auxiliary in H_r. In any case, as an implication of Lemma 4.6, we have that $d(r')$ can reach every vertex of $H_{rr'}$ in $H_{rr'} \setminus (x, y)$. (3.1) Let x be *ao* and y be *oo*. Let (x_0, y) be an edge of H_r^R that corresponds to (x, y) in $H_{rr'}$. Then (y, x_0) is an edge of H_r. If x_0 is an ordinary vertex of H_r, then by Lemma 4.6 we have that r reaches all vertices of H_r in $H_r \setminus (y, x_0)$. This implies that every vertex of H_r^R can reach r in $H_r^R \setminus (x_0, y)$. Then Lemma 4.16 implies that every vertex of $H_{rr'}$ can reach $d(r')$ in $H_{rr'} \setminus (x, y)$, and so we have that $H_{rr} \setminus (x, y)$ is strongly connected. Similarly, if x_0 is an auxiliary vertex of H_r, then, by Lemma 4.7, we have that r reaches all vertices of H_r in $H_r \setminus (y, x_0)$, except x_0, which is unreachable from r in $H_r \setminus (y, x_0)$, and possibly $d(r)$. This implies that every vertex of H_r^R can reach r in $H_r^R \setminus (x_0, y)$, except x_0, which cannot reach r in $H_r^R \setminus (x_0, y)$, and possibly $d(r)$. Since x is an ordinary vertex of H_r, we have that $x \neq x_0$, and so x can reach r in $H_r^R \setminus (x_0, y)$. Now Lemma 4.16 implies that x can reach $d(r')$ in $H_{rr'} \setminus (x, y)$. Thus, every vertex of $H_{rr'}$ can reach $d(r')$ in $H_{rr'} \setminus (x, y)$, and so we have that $H_{rr'} \setminus (x, y)$ is strongly connected. (3.2) Let x be *ao* and y be *oa*. Let $\{(x_1, y), \dots, (x_k, y)\}$ be the set of all the incoming edges to y in H_r^R. Then $\{(y, x_1), \dots, (y, x_k)\}$ is a collection of edges of H_r that stem from y. Since y is

auxiliary in H_r and ordinary in $H_{rr'}$, we have that $y \neq d(r)$ in G_s, and therefore y is a marked vertex of G_s such that $d(y) \in T(r)$. Thus, by Lemma 4.9, we have that r can reach all vertices of H_r in $H_r \setminus \{(y, x_1), \dots, (y, x_k)\}$. This implies that r can be reached from all vertices of H_r^R in $H_r^R \setminus \{(x_1, y), \dots, (x_k, y)\}$. Furthermore, since y is dominated by $d(r')$ in H_r^R, we have that there is a path from r to $d(r')$ in H_r^R that avoids y; therefore, this path also avoids all edges $\{(x_1, y), \dots, (x_k, y)\}$. Thus we infer that $d(r')$ is reachable from all vertices of H_r^R in $H_r^R \setminus \{(x_1, y), \dots, (x_k, y)\}$. In particular, let P be a path from x to $d(r')$ in $H_r^R \setminus \{(x_1, y), \dots, (x_k, y)\}$. Then, by Lemma 4.2 the corresponding path P_H in $H_{rr'}$ has the same endpoints as P and avoids the edge (x, y). Thus, even by removing (x, y) from $H_{rr'}$, all vertices from $H_{rr'}$ can reach $d(r')$. We conclude that $H_{rr'} \setminus (x, y)$ is strongly connected. (3.3) Let x be aa and y be oo. Let (x_0, y) be an edge of H_r^R that corresponds to (x, y) in $H_{rr'}$. Then (y, x_0) is an edge of H_r. Now, if x_0 is ordinary in H_r or it is auxiliary in H_r but $x_0 \neq x$, then we can argue as in (3.1) in order to establish that $H_{rr'} \setminus (x, y)$ is strongly connected. Otherwise, suppose that $x_0 = x$. First, let us assume that $d(r) \notin H_{rr'}$. Now, since x is auxiliary in H_r and y is ordinary in H_r (and $x \neq d(r)$), we have that $y = d(x)$ in G_s. Now, by Lemma 4.7 we have that r reaches all vertices of H_r in $H_r \setminus (y, x)$, except x, which is unreachable from r in $H_r \setminus (y, x)$, and possibly $d(r)$. This implies that every vertex of H_r^R can reach r in $H_r^R \setminus (x, y)$, except x, which cannot reach r in $H_r^R \setminus (x, y)$, and possibly $d(r)$. Now Lemma 4.16 implies that every vertex of $H_{rr'}$ can reach $d(r')$ in $H_{rr'} \setminus (x, y)$, except possibly x and $d(r)$. Since we have assumed that $d(r) \notin H_{rr'}$, we conclude that either $H_{rr'} \setminus (x, y)$ is strongly connected, or its strongly connected components are $\{x\}$ and $H_{rr'} \setminus \{x\}$. Now let us assume that $d(r) \in H_{rr'}$. There are two cases to consider: either $r' = d(r)$, or $d(r)$ is the immediate dominator of r' in H_r^R, and so it is contained in $H_{rr'}$ as the critical vertex. In either case, we have that $x \neq d(r)$. Thus we can argue as previously, in order to get that every vertex of $H_{rr'}$ can reach $d(r')$ in $H_{rr'} \setminus (x, y)$, except x, which cannot reach $d(r')$ in $H_{rr'}$, and possibly $d(r)$. Now, if $d(r)$ is the critical vertex of $H_{rr'}$, then we have that $d(r) = d(r')$, and so we conclude that the strongly connected components of $H_{rr} \setminus (x, y)$ are $\{x\}$ and $H_{rr} \setminus \{x\}$. Otherwise, let $r' = d(r)$. Then, since $H_{rr'}$ is strongly connected, we have that every vertex of $H_{rr'}$ reaches $d(r')$ in $H_{rr'}$. And since $y = d(r)$ is forbidden (since $r' = d(r)$), we have that (x, y) is not an incoming edge to $d(r)$. Thus, every vertex of $H_{rr'}$ can reach $d(r')$ in $H_{rr'} \setminus (x, y)$, except possibly x. We conclude that the strongly connected components of $H_{rr'} \setminus (x, y)$ are $\{x\}$ and $H_{rr'} \setminus \{x\}$. (3.4) Let x be aa and y

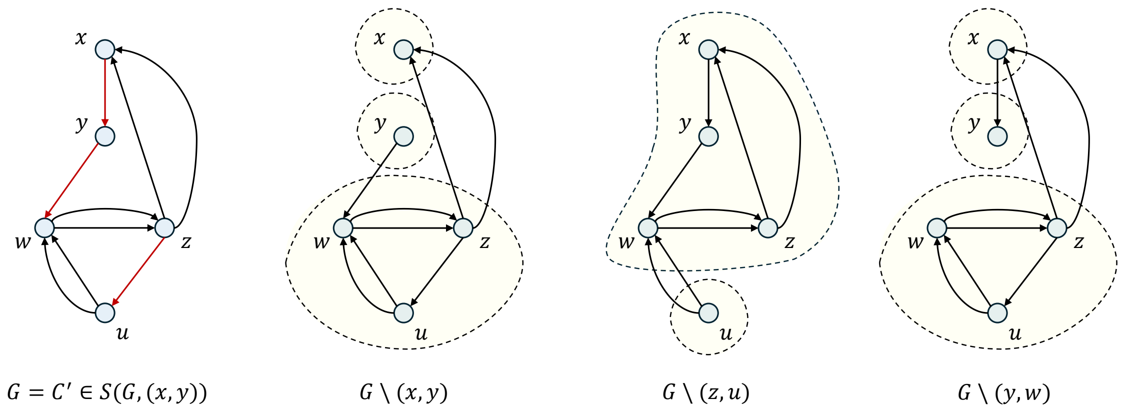

Figure 4.6: A strongly connected graph G with its strong bridges shown in red color. $C = \{w, u, z\}$ is a SCC of $G \setminus (x, y)$ and the corresponding graph $C' \in S(G, (x, y))$ is identical to G. Each SCC of $G \setminus e$ that does not contain x or y, for any strong bridge e of G, lies entirely within C.

be oa. Then we can use the same argument that we used in (3.2) in order to conclude that $H_{rr'} \setminus (x, y)$ is strongly connected.

(4) Since $y \neq d(r')$ in H_r^R, it is impossible that both x and y are auxiliary in $H_{rr'}$. □

The following Lemma shows that the S-operation maintains the strongly connected components of a digraph upon removal of strong bridges. (See also Figure 4.6.)

Lemma 4.19. *Let G be a strongly connected digraph and let (x, y) be a strong bridge of G. Let C be a strongly connected component of $G \setminus (x, y)$, and let C' be the corresponding graph in $S(G, (x, y))$. Let e be an edge of C'.*

(1) Suppose first that both endpoints of e lie in C. Then e is a strong bridge of C' if and only if e is a strong bridge of G. Furthermore, let $C_1, \dots, C_k$ be the strongly connected components of $G \setminus e$. Then at most one of $C_1, \dots, C_k$ contains both x and y. All the other strongly connected components of $G \setminus e$ either lie entirely within C, or they do not intersect with C' at all. Now let $i \in \{1, \dots, k\}$. Then, if C_i contains both x and y, but no vertex of C, then $\{x\}$ and $\{y\}$ are two distinct strongly connected components of $C' \setminus e$. Otherwise, $C_i \cap C'$ (if it is non-empty) is a strongly connected component of $C' \setminus e$.

(2) Now suppose that at least one endpoint p of e lies outside of C. Then e is a strong bridge of C' if and only if it is the only incoming or outgoing edge of p in C'.

Furthermore, the strongly connected components of $C' \setminus e$ are $\{x\}$ (if $x \notin C$), $\{y\}$ (if $y \notin C$), and C.

Proof. (1) Let e be an edge of C' such that both endpoints of e lie in C. Let u, v be two vertices of C. Then, as a consequence of Lemma 4.3, we have that u reaches v in $C' \setminus e$ if and only if u reaches v in $G \setminus e$ $(*)$. This implies that e is a strong bridge of C' if and only if e is a strong bridge of G. Now let C_0 be a strongly connected component of $G \setminus e$. First, let us assume that C_0 contains both x and y, but no vertex of C. This means that neither x nor y is a vertex of C. Then, by the definition of C', we have that (y, x) is not an edge of C'. However, since both endpoints of e lie in C, we have that $e \neq (x, y)$, and so $(x, y) \in C' \setminus e$. Now let's assume, for the sake of contradiction, that y can reach x in $C' \setminus e$. Then there must be vertices $z, w \in C$ such that $(y, z) \in C'$, $(w, x) \in C'$, and z, w remain strongly connected in $C' \setminus e$. The existence of the edge $(y, z) \in C'$ implies that there is a path P_1 from y to z in G, that uses no vertices of C except z. Similarly, the existence of the edge $(w, x) \in C'$ implies that there is a path P_2 from w to x in G, that uses no vertices of C except w. Now, since z can reach w in $C' \setminus e$, Lemma 4.3 implies that there is a path P from z to w in $G \setminus e$. Thus the concatenation $P_1 + P + P_2$ is a path from y to x in $G \setminus e$ that uses at least one vertex from C. But then the existence of (x, y) in $G \setminus e$ implies that $\{x, y, z, w\} \subseteq C_0$ - a contradiction. Thus we have that $\{x\}$ and $\{y\}$ are two distinct strongly connected components of $C' \setminus e$.

Now let us assume that C_0 contains both x and y, and also a vertex z of C (of course, it is possible that $z = x$ or $z = y$). Then $(*)$ implies that all vertices of $C_0 \cap C$ remain strongly connected in $C' \setminus e$. Furthermore, there is no vertex in $C \setminus C_0$ that is strongly connected with vertices of $C_0 \cap C$ in $C' \setminus e$. Now, since z is strongly connected with x in $G \setminus e$, there is a path P_1 from z to x in $G \setminus e$. Similarly, there is a path P_2 from y to z in $G \setminus e$. Now consider the concatenation $P = P_1 + (x, y) + P_2$, and let $\tilde{P}$ be the compressed path of P in C'. Then, by Lemma 4.3, $\tilde{P}$ is a path from z to z in C', that uses the edge (x, y) and avoids the edge e. Thus, z is strongly connected with both x and y in $C' \setminus e$. This means that $C_0 \cap C'$ is a strongly connected component of $C' \setminus e$.

Finally, let us assume that C_0 contains a vertex z of C, but not both x and y. We will show that C_0 lies entirely within C. So let us assume, for the sake of contradiction, that there is a vertex $w \in C_0 \setminus C$. Then, since z and w are strongly connected in $G \setminus e$, there is a path P_1 from z to w in $G \setminus e$. Similarly, there is a path P_2 from w to z in

$G \setminus e$. Now, since $z \in C$ but $w \notin C$, at least one of P_1 and P_2 must use the edge (x, y). But then the existence of the concatenation $P = P_1 + P_2$ implies that z is strongly connected with both x and y in $G \setminus e$ - a contradiction. Thus we have that $C_0 \subseteq C$. Now we will show that C_0 is a strongly connected component of $C' \setminus e$. By $(*)$ we have that all vertices of C_0 are strongly connected in $C' \setminus e$. Furthermore, there is no vertex in $C \setminus C_0$ that is strongly connected with vertices of C_0 in $C' \setminus e$. Thus, C_0 is contained in a strongly connected component C'_0 of $C' \setminus e$ such that $C_0 \subseteq C'_0 \subseteq C_0 \cup \{x, y\}$. Now let us assume that both x and y are contained in C'_0. Then, since $z \in C'_0$, there is a path P' in $C' \setminus e$ that starts from z, ends in z, and uses the edge (x, y). By Lemma 4.3, there is a path P in G such that $\tilde{P} = P'$. This implies that P starts from z, ends in z, uses the edge (x, y) and avoids e. But this means that z is strongly connected with both x and y in $G' \setminus e$ - a contradiction. Thus C'_0 cannot contain both x and y. Now, if C_0 contains either x or y, then C'_0 also contains x or y, respectively, and we are done (i.e., we have $C'_0 = C_0$, because C'_0 cannot contain both x and y). Thus it is left to assume that C_0 contains neither x nor y. Now there are three cases to consider: either $x \in C$ and $y \notin C$, or $x \notin C$ and $y \in C$, or $C \cap \{x, y\} = \emptyset$. Let us assume first that $x \in C$ and $y \notin C$. Now suppose, for the sake of contradiction, that $x \in C'_0$. This means that there is a path P' from z to z in $C' \setminus e$ that contains x. Then, by Lemma 4.3, we have that there is a path P in G such that $\tilde{P} = P'$. But this means that z is strongly connected with x in $G \setminus e$ - contradicting our assumption that $x \notin C_0$. Thus $x \notin C'_0$. Now suppose, for the sake of contradiction, that $y \in C'_0$. But since $y \notin C$, in order for $z \in C'_0$ to reach y in C', we must necessarily pass from the edge (x, y). Then we can see that x and y are both strongly connected with z in $C' \setminus e$, contradicting the fact that C'_0 cannot contain both x and y. Thus $y \notin C'_0$, and so we have that $C'_0 = C_0$. Similarly, we can see that if $x \notin C$ and $y \in C$, then C'_0 contains neither x nor y, and so we have $C'_0 = C_0$. Finally, let us assume that neither x nor y is in C. Now, if either x or y is in C'_0, then y or x, respectively, must also be in C'_0, since (x, y) is the unique outgoing (resp. incoming) edge of x (resp. y) in C'. But this contradicts the fact that C'_0 cannot contain both x and y. Thus we conclude that neither x nor y is in C'_0, and so $C'_0 = C_0$.

(2) Now let e be an edge of C' such that at least one endpoint p of e lies outside of C. Then we have that either $p = x$ or $p = y$. Let us assume that $p = x$ (the case $p = y$ is treated similarly). Now suppose that $e = (x, z)$, for some $z \in C'$. Then, by the definition of C', we have that $z = y$. Also, (x, y) is the only outgoing edge of x in C'.

Thus, the strongly connected components of $C' \setminus (x,y)$ are $\{x\}$, $\{y\}$ (if $y \notin C$), and C. Now suppose that $e = (z,x)$, for some $z \in C'$. Then, by the definition of C', we have that $z \in C$. Thus, if there is another incoming edge (w,x) of x in C', then $C' \setminus (z,x)$ is not a strong bridge of C', since z can still reach w in $C' \setminus (z,x)$ using edges of C. Thus (z,x) is a strong bridge of C' if and only if it is the only incoming edge of x in C'. In this case, the strongly connected components of $C' \setminus (z,x)$ are $\{x\}$, $\{y\}$ (if $y \notin C$), and C. □

Corollary 4.5. *Let G_s be a strongly connected digraph and let r be a marked vertex of H_s^R. Let C be a strongly connected component of $H_{sr} \setminus (d(r),r)$, and let H be the corresponding graph in $S(H_{sr},(d(r),r))$, where $d(r)$ is the immediate dominator of r in H_s^R. Let e be a strong bridge of H. Then the strongly connected components of $H \setminus e$ are given by one of the following:*

(i) $\{x\}$ *and* $H \setminus \{x\}$, *where* x *is auxiliary in* H_s *and* $y = d(x)$ *in* G_s

(ii) $\{x\}$ *and* $H \setminus \{x\}$, *where* x *is auxiliary in* H_{sr} *and* (x,y) *is the only outgoing edge of* x *in* H_{sr}

(iii) $\{y\}$ *and* $H \setminus \{y\}$, *where* y *is auxiliary in* H_{sr} *and* $x = d(y)$ *in* H_s^R

(iv) $\{x\}$, $\{y\}$ *and* $H \setminus \{x,y\}$, *where* x *is auxiliary in* H_s, y *is auxiliary in* H_{sr}, $x = d(y)$ *in* H_s^R *and* $y = d(x)$ *in* G_s

(v) $\{d(r)\}$, $\{r\}$ *(if* $r \notin C$*), and* C

Proof. Since $(d(r),r)$ is the only outgoing edge of $d(r)$ in H_{sr}, observe that one of the strongly connected components of $H_{sr} \setminus (d(r),r)$ is $\{d(r)\}$. It is trivial to verify the Corollary for this case, and so we may assume that $C \neq \{d(r)\}$. Now let us assume first that both endpoints of e lie in C. Then case (1) of Lemma 4.19 applies, and thus e is subject to one of cases (i) to (iv) of Lemma 4.17. Thus we get cases (i) to (iv) in the statement of the Corollary. Now, if one of the endpoints of e lies outside of C, then case (2) of Lemma 4.19 applies, and thus we get case (v) in the statement of the Corollary. □

Corollary 4.6. *Let G_s be a strongly connected digraph, let r be a marked vertex of G_s, and let r' be a marked vertex of H_r^R. Let C be a strongly connected component of $H_{rr'} \setminus (d(r'),r')$, and let H be the corresponding graph in $S(H_{rr'},(d(r'),r'))$, where $d(r')$ is the immediate* *dominator of r' in H_r^R. Let e be a strong bridge of H. Then the strongly connected components of $H \setminus e$ are given by one of the following:*

(i) $\{x\}$ *and* $H \setminus \{x\}$*, where* x *is auxiliary in* H_r *and* $y = d(x)$ *in* G_s

(ii) $\{x\}$ *and* $H \setminus \{x\}$*, where* x *is auxiliary in* $H_{rr'}$ *and* (x, y) *is the only outgoing edge of* x *in* $H_{rr'}$

(iii) $\{y\}$ *and* $H \setminus \{y\}$*, where* y *is auxiliary in* $H_{rr'}$ *and* $x = d(y)$ *in* H_r^R

(iv) $\{x\}$*,* $\{y\}$ *and* $H \setminus \{x, y\}$*, where* x *is auxiliary in* H_r*,* y *is auxiliary in* $H_{rr'}$*,* $x = d(y)$ *in* H_r^R *and* $y = d(x)$ *in* G_s

(v) $\{x\}$*,* $\{y\}$*,* $\{d(r)\}$ *and* $\{r\}$*, where* $r' = d(r)$*,* x *is auxiliary in* H_r*,* y *is auxiliary in* $H_{rr'}$*,* $x = d(y)$ *in* H_r^R*,* $y = d(x)$ *in* G_s*, and* $(d(r), x)$ *is the only kind of outgoing edge of* $d(r)$ *in* $H_{rr'}$

(vi) $\{d(r')\}$*,* $\{r'\}$ *(if* $r' \notin C$*), and* C

Proof. Arguing as in the proof of Corollary 4.5, we can see that this Corollary is a consequence of Lemma 4.19 and Lemma 4.18. □

4.4 Computing 2-edge twinless strongly connected components

We assume that G is a twinless strongly connected digraph since otherwise we can compute the twinless strongly connected components in linear time and process each one separately. We let E_t denote the set of twinless strong bridges of G, and let E_s denote the set of strong bridges of G. (Note that $E_s \subseteq E_t$.) Algorithm 4.3 is a simple $O(mn)$-time algorithm for computing the 2-edge twinless strongly connected components of G. (It is essentially the same as in [49].)

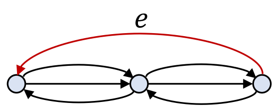

Figure 4.7: Example of an edge e (colored red) that is a twinless strong bridge but not a strong bridge. Note that the deletion of e leaves the digraph strongly connected by not twinless strongly connected.

Algorithm 4.3: Partition vertices of G with respect to its twinless strong bridges.

```
1 initialize a partition of the vertices $\mathcal{P} = \{V(G)\}$
2 compute the set $E_t$ of the twinless strong bridges of $G$
3 foreach $e \in E_t$ do
4     compute the twinless strongly connected components $C_1, \ldots, C_k$ of $G \setminus e$
5     let $\mathcal{P} = \{S_1, \ldots, S_l\}$ be the current partition of the vertices in $V(G)$
6     refine the partition by computing the intersections $C_i \cap S_j$ for all
        $i = 1, \ldots, k$ and $j = 1, \ldots, l$
7 end
```

Our goal is to provide a faster algorithm by processing separately the edges in $E_t \setminus E_s$ and the edges in E_s. That is, we first partition the vertices according to the twinless strong bridges of G that are not strong bridges, and then we refine this partition by considering the effect of strong bridges. We call the first partition the one that is "*due to the twinless strong bridges that are not strong bridges*", and the second partition the one that is "*due to the strong bridges*".

4.4.1 Computing the partition of the 2eTSCCs due to the twinless strong bridges that are not strong bridges

Let e be an edge in $E_t \setminus E_s$. (See Figure 4.7.) Then the TSCCs of $G \setminus e$ are given by the 2-edge-connected components of $G^u \setminus \{e^u\}$, where e^u is the undirected counterpart of e [50]. Thus, we can simply remove the bridges of $G^u \setminus \{e^u\}$, in order to get the partition into the TSCCs that is due to e. To compute the partition that is due to all edges in $E_t \setminus E_s$ at once, we may use the cactus graph Q which is given by contracting the 3-edge-connected components of G^u into single nodes [73]. Q comes together with a function $\phi : V(G^u) \to V(Q)$ (the quotient map) that maps every vertex of G^u to the node of Q that contains it, and induces a natural correspondence between edges of G^u and edges of Q. The cactus graph of the 3-edge-connected components provides a clear representation of the 2-edge cuts of an undirected graph; by definition, it has the property that every edge of it belongs to exactly one cycle. Thus, Algorithm 4.4 shows how we can compute the partition of 2eTSCCs that is due to the edges in $E_t \setminus E_s$.

Algorithm 4.4: Compute the partition of 2eTSCCs of G that is due to the twinless strong bridges that are not strong bridges.

```
1 compute the cactus $Q$ of the 3-edge-connected components of $G^u$, and let
    $\phi : V(G^u) \to V(Q)$ be the quotient map
2 foreach edge $e$ of $Q$ do
3     if $e$ corresponds to a single edge of $G$ that has no twin and is not a strong
        bridge then  remove from $Q$ the edges of the cycle that contains $e$
4 end
5 let $Q'$ be the graph that remains after all the removals in the previous step
6 let $C_1, \dots, C_k$ be the connected components of $Q'$
7 return $\phi^{-1}(C_1), \dots, \phi^{-1}(C_k)$
```

Proposition 4.7. *Algorithm 4.4 is correct and runs in linear time.*

Proof. The correctness of Algorithm 4.4 is easily established due to the structure of the cactus of the 3-edge-connected components. Since the strong bridges of a directed graph can be computed in linear time [51], and the 3-edge-connected components of an undirected graph can also be computed in linear time (see e.g., [74, 52]), we have that Algorithm 4.4 runs in linear time. □

4.4.2 Computing the partition of the 2eTSCCs due to the strong bridges

Let G_s be a strongly connected digraph, and let H be either (1) H_{ss}, or (2) $\widetilde{H}_{rr}$, or (3) $S(H_{sr}, (d(r), r))$, or (4) $S(H_{rr'}, (d(r'), r'))$ (for some marked vertices r and r' in G_s and H^R_r, respectively). Depending on whether H satisfies (1), (2), (3), or (4), we

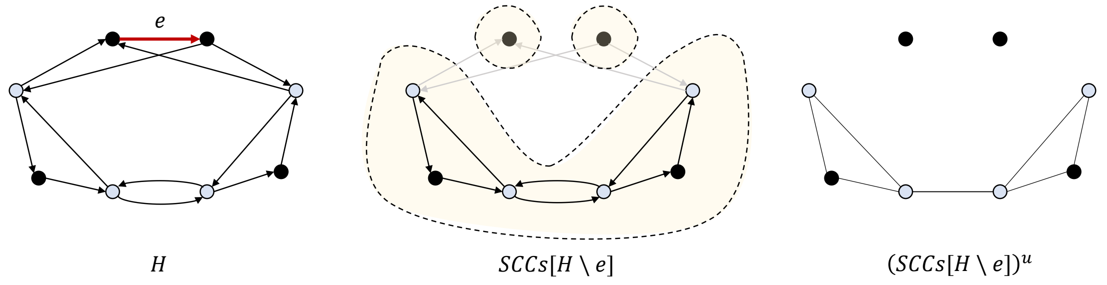

Figure 4.8: An auxiliary graph $H = H_{uu}$, corresponding to the digraph of Figure 4.2, with auxiliary vertices colored black. Edge e (shown in red) is a strong bridge of H. The underlying undirected graph of the SCCs of $H \setminus e$.

have a clear picture of the structure of the strongly connected components of H upon removal of a strong bridge, thanks to Lemma 4.11, Lemma 4.14, Corollary 4.5, or Corollary 4.6, respectively. We can apply this information in order to compute the partition of the 2-edge twinless strongly connected components of H due to the strong bridges as follows.

Let $e = (x, y)$ be a strong bridge of H. Observe that in every case that appears in Lemma 4.11, Lemma 4.14, Corollary 4.5, or Corollary 4.6, the strongly connected components of $H \setminus e$ are given by one of the following: (i), $\{x\}$ and $H \setminus \{x\}$, (ii), $\{y\}$ and $H \setminus \{y\}$, (iii), $\{x\}$, $\{y\}$ and $H \setminus \{x, y\}$, or (iv) $\{x\}$, $\{y\}$, $\{d(r)\}$ and $\{r\}$. In every case in which $\{x\}$ or $\{y\}$ appears as a separate strongly connected component, we have that x, y, or both, respectively, are not oo vertices, and so they are not in the same 2-edge twinless strongly connected component with any oo vertex of H. (We may ignore case (iv), because this is due to a singular case in which H consists of four vertices, where only one of them is oo.) Now, depending on whether we are in case (i), (ii), or (iii), we define $X_e = \{x\}$, $X_e = \{y\}$, or $X_e = \{x, y\}$, respectively. In other words, X_e is the set of the non-oo vertices that get (strongly) disconnected from the rest of the graph upon removal of e. Then we have that X_e satisfies the following:

(1) $H[V \setminus X_e]$ is a strongly connected component of $H \setminus e$

(2) X_e contains no oo vertex of H

Now we can apply the following procedure to compute the partition of 2-edge twinless strongly connected components of (the oo vertices of) H due to the strong bridges. Initially, we let $\mathcal{P}$ be the trivial partition of V (i.e., $\mathcal{P} = \{V\}$). Now, for every strong bridge e of H, we compute the TSCCs of $H \setminus X_e$, and we refine $\mathcal{P}$ according to those TSCCs. By [50], the computation of the TSCCs of $H \setminus X_e$ is equivalent to determining the 2-edge-connected components of $H^u \setminus X_e$. Observe that this procedure does not run in linear time in total, since it has to be performed for every strong bridge e of H.

Our goal is to perform the above procedure for all strong bridges e of H at once. We can do this by first taking H^u, and then shrinking every X_e in H^u into a single marked vertex, for every strong bridge e of H. Let H' be the resulting graph. Then we simply compute the marked vertex-edge blocks of H'. (See Figure 4.9 for an example of the process of contracting all X_e into single marked vertices.) This is shown in Algorithm 4.5. We note that given an auxiliary graph H as above, we can compute all sets X_e in linear time, by first computing all strong bridges of H [51], and then checking which case of Lemma 4.11, Lemma 4.14, Corollary 4.5, or Corollary 4.6 applies for each strong bridge (which can be easily checked in $O(1)$ time).

Algorithm 4.5: A linear-time algorithm for computing the partition of 2-edge twinless strongly connected components of an auxiliary graph H due to the strong bridges

```
input : An auxiliary graph $H$ equipped with the following information: for
        every strong bridge $e$ of $H$, the set $X_e$ defined as above
output : The partition of 2-edge twinless strongly connected components of
        the ordinary vertices of $H$ due to the strong bridges
1  begin
2      compute the underlying undirected graph $H^u$
3      foreach strong bridge $e$ of $H$ do
4          contract $X_e$ into a single vertex in $H^u$, and mark it
5      end
6      let $H'$ be the graph with the marked contracted vertices derived from $H^u$
7      compute the partition $\mathcal{B}_{ve}$ of the marked vertex-edge blocks of $H'$
8      let $\mathcal{O}$ be the partition of $V$ consisting of the set of the ordinary vertices of
        $H$ and the set of the auxiliary vertices of $H$
9      return $\mathcal{B}_{ve}$ refined by $\mathcal{O}$
10 end
```

The following two lemmas describe conditions under which the simultaneous contraction of several subsets of the vertex set of an undirected graph maintains the marked vertex-edge blocks.

Lemma 4.20. *Let $G = (V, E)$ be a connected undirected graph and let $X_1, \ldots, X_k$ be a collection of disjoint subsets of V, such that $G \setminus X_i$ is connected for every $i \in \{1, \ldots, k\}$. Let also u and v be two vertices of G that do not belong to any of $X_1, \ldots, X_k$. Suppose that u and v belong to the same 2-edge-connected component of $G \setminus X_i$, for every $i \in \{1, \ldots, k\}$. Now let G' be the graph that is derived from G by contracting every X_i, for $i \in \{1, \ldots, k\}$, into a single marked vertex. Then u and v belong to the same marked vertex-edge block of G'.*

Proof. Let z be a marked vertex of G' that corresponds to a set $X \in \{X_1, \ldots, X_k\}$. Then we have that u and v belong to the same 2-edge-connected component of $G \setminus X$. Thus there is a cycle C in $G \setminus X$ that contains u and v, but no vertex of X. Then C corresponds to a cycle C' of G' that contains both u and v and avoids z. Thus u and v are also 2-edge-connected in $G' \setminus z$. Due to the generality of z, we have that u and v belong to the same marked vertex-edge block of G'. □

Lemma 4.21. *Let $G = (V, E)$ be a connected undirected graph and let $X_1, \ldots, X_k$ be a collection of disjoint subsets of V, such that $G \setminus X_i$ is connected for every $i \in \{1, \ldots, k\}$. Let also u and v be two vertices of G that do not belong to any of $X_1, \ldots, X_k$. Suppose that there is a $j \in \{1, \ldots, k\}$ such that u and v belong to different 2-edge-connected components of $G \setminus X_j$. Now let G' be the graph that is derived from G by contracting every X_i, for $i \in \{1, \ldots, k\}$, into a single marked vertex. Then the following is a sufficient condition to establish that u and v belong to different marked vertex-edge blocks of G': Let T be the tree of the 2-edge-connected components of $G \setminus X_j$; then, for every $i \in \{1, \ldots, k\} \setminus \{j\}$, we have that either (1) X_i lies entirely within a node of T, or (2) a part of X_i constitutes a leaf of T, and the remaining part of X_i lies within the adjacent node of that leaf in T.*

Proof. Let z be the marked vertex of G' that corresponds to X_j. Let T' be the tree of the 2-edge-connected components of $G' \setminus z$. We will show that u and v belong to different nodes of T'. First, we have that u and v belong to different nodes of T. Now, T' is derived from T in the same way that G' is derived from G, i.e., by contracting the nodes of T that contain all the different parts of X_i into a single node, for every $i \in \{1, \ldots, k\} \setminus \{j\}$. Now take a $X_i \in \{X_1, \ldots, X_k\} \setminus \{X_j\}$. If (1) holds for X_i, then

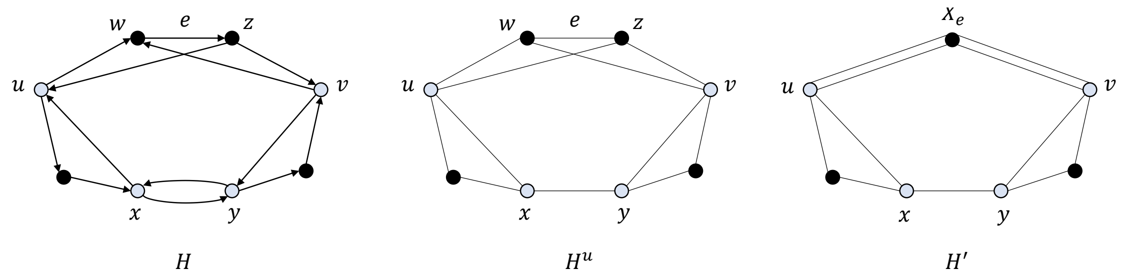

Figure 4.9: An auxiliary graph $H = H_{uu}$, corresponding to the digraph of Figure 4.2, with auxiliary vertices colored black. The underlying undirected graph H^u of H, and the corresponding graph H' resulting from H^u after shrinking the vertex sets X_e for all strong bridges e; here $X_e = \{w, z\}$. Note that we have two parallel edges $\{X_e, u\}$ and $\{X_e, v\}$.

the contraction of X_i in G' induces no change on T. If (2) holds for X_i, then the contraction of X_i in G' induces the following change on T: a leaf of T, that contains neither u nor v, is contracted with its adjacent node in T. Thus we can see that u and v belong to different nodes of T'. $\square$

Proposition 4.8. *Let G_s be a strongly connected digraph, and let H be a digraph that satisfies one of the following:*

(1) $H = H_{ss}$

(2) $H = \widetilde{H}_{rr}$, *where r is a marked vertex of G_s*

(3) $H \in S(H_{sr}, (d(r), r))$, *where r is a marked vertex of G_s and $(d(r), r)$ is the critical edge of H_{sr}*

(4) $H \in S(H_{rr'}, (d(r'), r'))$, *where r is a marked vertex of G_s, r' is a marked vertex of H_r^R, and $(d(r'), r')$ is the critical edge of $H_{rr'}$*

Then Algorithm 4.5 correctly computes the partition of 2-edge twinless strongly connected components of the oo vertices of H due to the strong bridges.

Proof. We will demonstrate how to handle case (1), by relying on Lemma 4.11. Cases (2), (3) and (4) can be handled using similar arguments, by relying on Lemma 4.14, Corollary 4.5, and Corollary 4.6, respectively. In any case, we have to establish the following. Let $\mathcal{E}$ be the collection of the strong bridges of H. Then the collection

$\{X_e \mid e \in \mathcal{E}\}$ consists of sets that are either singletons or 2-element sets. First we show that the 2-element sets in $\{X_e \mid e \in \mathcal{E}\}$ are pairwise disjoint. Then we show that, despite the simultaneous contractions of the sets in $\{X_e \mid e \in \mathcal{E}\}$, the marked vertex-edge blocks of the underlying graph of H are maintained. This is achieved by showing that the collection $\{X_e \mid e \in \mathcal{E}\}$ satisfies the preconditions in Lemmas 4.20 and 4.21.

Now, in case (1), by Lemma 4.11 we have that the sets X_e, for all strong bridges e of H_{ss}, are given by cases (i) to (iv) in the statement of this Lemma. To be precise, in cases (i) and (ii) we set $X_e \leftarrow \{x\}$; in case (iii) we set $X_e \leftarrow \{y\}$; and in case (iv) we set $X_e \leftarrow \{x, y\}$. Thus, the last case is the only one in which X_e is a 2-element set.

Now let e, e' be two strong bridges of H_{ss} such that X_e and $X_{e'}$ are two distinct 2-element sets. We will show that $X_e \cap X_{e'} = \emptyset$. So let $X_e = \{x, y\}$ and $X_{e'} = \{z, w\}$. Since these sets can occur only in case (iv) of Lemma 4.11, we may assume, without loss of generality, that $e = (x, y)$ and $e' = (z, w)$. Then we have that y and w are auxiliary, but not critical, vertices of H_{ss}. Since, then, y and w have only one incoming edge in H_{ss}, we must have that either $y = w$ and $x = z$, or $y \neq w$. The first case means that $e = e'$, and so it is impossible to occur. Thus we have $y \neq w$. By Lemma 4.11, we have that $y = d(x)$ in G_s and $w = d(z)$ in G_s. Thus, $y \neq w$ implies that $x \neq z$. Notice that none of the cases $x = w$ and $y = z$ can occur, because there is no edge between auxiliary but not critical vertices. We conclude that $\{x, y\} \cap \{z, w\} = \emptyset$.

Now let H' be the graph that is derived from H^u after we have contracted every X_e into a single marked vertex. We will show that two oo-vertices of H_{ss} are twinless strongly connected in $H_{ss} \setminus e$ for any strong bridge e, if and only if they belong to the same marked vertex-edge block of H'. So let u, v be two oo-vertices of H_{ss} that are twinless strongly connected in $H_{ss} \setminus e$ for any strong bridge e. Then, by [50] we have that, for every strong bridge e of H_{ss}, u and v belong to the same 2-edge-connected component of $H^u_{ss} \setminus X_e$. Then Lemma 4.20 implies that u and v belong to the same marked vertex-edge block of H'. Conversely, let u, v be two oo-vertices of H_{ss} such that there is a strong bridge e for which u and v are not twinless strongly connected in $H_{ss} \setminus e$. By [50], this means that u and v belong to different 2-edge-connected components of $H^u_{ss} \setminus X_e$. Now let e' be a strong bridge of H_{ss} such that $X_{e'} \neq X_e$. We will show that $X_{e'}$ satisfies one of conditions (1) and (2) of Lemma 4.21. Now, if $X_{e'}$ is a singleton set, then there is nothing to show, since there is no contraction induced by $X_{e'}$ in this case. So let $X_{e'} = \{x, y\}$. This can occur only in case (iv), and so we may assume, without loss of generality, that $e' = (x, y)$. Now, if x and y belong to the same twinless strongly connected component of $H_{ss} \setminus e$, then condition (1) of Lemma 4.21 is satisfied. Otherwise, since x and y are connected with the edge (x, y), we must have that (y, x) exists. But since (x, y) is the only incoming edge of y in H_{ss} (since $x = d(y)$ in H_s^R), we must have that $\{y\}$ constitutes a singleton twinless strongly connected component of $H_{ss} \setminus e$, and so condition (2) of Lemma 4.21 is satisfied. Thus, Lemma 4.21 ensures that, by contracting every $\{X_e \mid e \in \mathcal{E}\}$ into a single marked vertex in H^u, u and v belong to different marked vertex-edge blocks of H'. $\square$

The final 2eTSCCs of (the subset of the ordinary vertices of) an auxiliary graph are given by the mutual refinement of the partitions computed by Algorithms 4.4 and 4.5. (The mutual refinement of two partitions can be computed in linear time using bucket sort.) Hence, by Corollary 4.4 and Propositions 4.7 and 4.8, we have that the 2eTSCCs of a strongly connected digraph can be computed in linear time.

It remains to establish that Algorithm 4.5 runs in linear time. For this, we provide a linear-time procedure for Step 7. Observe that the marked vertices of H' have the property that their removal from H leaves the graph strongly connected, and thus they are not articulation points of the underlying graph H^u. This allows us to reduce the computation of the marked vertex-edge blocks of H' to the computation of marked vertex-edge blocks in biconnected graphs. Specifically, we first partition H' into its biconnected components, which can be done in linear time [75]. Then we process each biconnected component separately, and we compute the marked vertex-edge blocks that are contained in it. Finally, we "glue" the marked vertex-edge blocks of all biconnected components, guided by their common vertices that are articulation points of the graph. In the next section, we provide a linear-time algorithm for computing the marked vertex-edge blocks of a biconnected graph.

sectionComputing marked vertex-edge blocks Let G be a biconnected undirected graph. An SPQR tree $\mathcal{T}$ for G represents the triconnected components of G [53, 54]. Each node $\alpha \in \mathcal{T}$ is associated with an undirected graph G_α. Each vertex of G_α corresponds to a vertex of the original graph G. An edge of G_α is either a *virtual edge* that corresponds to a separation pair of G, or a *real edge* that corresponds to an edge of the original graph G. The node α, and the graph G_α associated with it, has one of the following types:

- If α is an S-node, then G_α is a cycle graph with three or more vertices and edges.

- If α is a P-node, then G_α is a multigraph with two vertices and at least 3 parallel edges.
- If α is a Q-node, then G_α is a single real edge.
- If α is an R-node, then G_α is a simple triconnected graph.

Each edge $\{\alpha, \beta\}$ between two nodes of the SPQR tree is associated with two virtual edges, where one is an edge in G_α and the other is an edge in G_β. If $\{u, v\}$ is a separation pair in G, then one of the following cases applies:

(a) u and v are the endpoints of a virtual edge in the graph G_α associated with an R-node α of $\mathcal{T}$.

(b) u and v are vertices in the graph G_α associated with a P-node α of $\mathcal{T}$.

(c) u and v are vertices in the graph G_α associated with an S-node α of $\mathcal{T}$, such that either u and v are not adjacent, or the edge $\{u, v\}$ is virtual.

In case (c), if $\{u, v\}$ is a virtual edge, then u and v also belong to a P-node or an R-node. If u and v are not adjacent then $G \setminus \{u, v\}$ consists of two components that are represented by two paths of the cycle graph G_α associated with the S-node α and with the SPQR tree nodes attached to those two paths. Gutwenger and P. Mutzel [76] showed that an SPQR tree can be constructed in linear time, by extending the triconnected components algorithm of Hopcroft and Tarjan [77].

Let $e = \{x, y\}$ be an edge of G such that $\{v, e\}$ is a vertex-edge cut-pair of G. Then, $\mathcal{T}$ must contain an S-node α such that v, x and y are vertices of G_α and $\{x, y\}$ is not a virtual edge. The above observation implies that we can use $\mathcal{T}$ to identify all vertex-edge cut-pairs of G as follows. A vertex-edge cut-pair (v, e) is such that $v \in V(G_\alpha)$ and e is a real edge of G_α that is not adjacent to v, where α is an S-node [52, 78]. Now we define the *split operation* of v as follows. Let e_1 and e_2 be the edges incident to v in G_α. We split v into two vertices v_1 and v_2, where v_1 is incident only to e_1 and v_2 is incident only to e_2. (In effect, this makes S a path with endpoints v_1 and v_2.) To find the connected components of $G \setminus \{v, e\}$, we execute a split operation on v and delete e from the resulting path. Note that $e \neq e_1, e_2$, and e does not have a copy in any other node of the SPQR tree since it is a real edge. Then, the connected components of $G \setminus \{v, e\}$ are represented by the resulting subtrees of $\mathcal{T}$.

Here, we need to partition the ordinary vertices of G according to the vertex-edge cut-pairs (v,e), where v is a marked auxiliary vertex. To do this efficiently, we can process all vertices simultaneously as follows. First, we note that we only need to consider the marked vertices that are in S-nodes that contain at least one real edge. Let α be such an S-node. We perform the split operation on each marked (auxiliary) vertex v, and then delete all the real edges of α. This breaks $\mathcal{T}$ into subtrees, and the desired partition of the ordinary vertices is formed by the ordinary vertices of each subtree. See Algorithm 4.6.

Algorithm 4.6: Partition the ordinary vertices of a biconnected graph G according to a set of marked vertex-edge cuts.

```
input : An SPQR tree $\mathcal{T}$ of $G$, a set of marked vertices $v \in V(G_\alpha)$ and real
        edges $e \in E(G_\alpha)$, for all $S$-nodes $\alpha$ of $\mathcal{T}$.
output: A partition of the ordinary vertices of $G$, so that two ordinary
        vertices in the same set of the partition remain in the same connected
        component of $G \setminus (v,e)$, for any vertex-edge cut $(v,e)$ such that $v$ is
        marked.
1  begin
2      foreach $S$-node $\alpha$ of $\mathcal{T}$ that contains a marked vertex and a real edge do
3          perform a split operation on each marked vertex of $\alpha$
4          delete all real edges of $G_\alpha$
5      end
6      break $\mathcal{T}$ into connected subtrees $\mathcal{T}_1, \dots, \mathcal{T}_\lambda$ after the above operations
7      foreach subtree $\mathcal{T}_i$ do
8          put all the ordinary vertices of $\mathcal{T}_i$ into the same set of the partition
9      end
10 end
```

Theorem 4.9. *The marked vertex-edge blocks of an undirected graph can be computed in linear time.*

Proof. First, we establish the correctness of Algorithm 4.6. To see why this algorithm works, consider two ordinary vertices u and w of G. First observe that if u and w are in different connected components of $G \setminus \{v,e\}$ for some marked vertex-edge cut (v,e), then u and w will end up in different subtrees of $\mathcal{T}$. Now suppose that u and w

remain in the same connected component of $G \setminus \{v, e\}$, for any vertex-edge cut (v, e) with v marked. Let β and γ be the nodes of $\mathcal{T}$ such that $u \in V(G_\beta)$ and $w \in V(G_\gamma)$. Consider any S-node α that lies on the path of $\mathcal{T}$ between β and γ and contains at least one marked vertex and at least one real edge. (If no such S-node exists, then clearly u and w cannot be separated by any marked vertex-edge cut.) Let e_u and e_v be the virtual edges of $E(G_\alpha)$ that correspond to the paths from β to α and from γ to α, respectively. Also, let P_1 and P_2 be the two paths that connect e_u and e_v in G_α. Without loss of generality, we can assume that P_1 contains a marked vertex v. Then, P_2 cannot contain any real edge e, since otherwise (v, e) would be a marked vertex-edge cut separating u and w. Hence, all real edges are on P_1. But then, for the same reason, P_2 cannot contain a marked vertex. Hence, all marked vertices are also on P_1. This implies that β and γ remain in the same subtree of $\mathcal{T}$ after the execution of Algorithm 4.6. The same arguments work if u or w (or both) are vertices of $V(G_\alpha)$. Also, it is easy to see that Algorithm 4.6 runs in linear time. $\square$

Now, since Algorithm 4.6 computes the marked vertex-edge blocks in linear time, we have that Algorithm 4.5 also runs in linear time. Hence, we obtain the following result:

Theorem 4.10. *The 2-edge twinless strongly connected components of a directed graph can be computed in linear time.*

Finally, by the reduction of Section 4.2, we have:

Theorem 4.11. *The edge-resilient strongly orientable blocks of a mixed graph can be computed in linear time.*

4.5 Concluding remarks

In this work we studied the notion of edge-resilient strongly orientable blocks of a mixed graph G. Each such block C has the property that for any (directed or undirected) edge e of G, there is an orientation R of $G \setminus e$ that maintains the strong connectivity of the vertices in C. Our main contribution was to provide a linear-time algorithm that compute the such blocks of maximal size. We note that if we change in the definition of edge-resilient strongly orientable blocks the assumption that the edges that we allow to fail are both directed and undirected, and we demand that they are only directed, or only undirected, then we can easily modify Algorithm 4.2 so that we can compute those blocks in linear time (using again the reduction to computing 2-edge twinless strongly connected components).

Chapter 5

Conclusions and Open Problems

From Chapter 3 and Chapter 4, according to the experimental analysis, we conclude that the algorithm of Gabow is the fastest algorithm (among the included algorithms) for computing the edge-connectivity of a directed graph as well as for computing a maximum packing of edge-disjoint arborescences of a directed graph. We leave as an interesting open problem whether an efficient implementation of the algorithm of Bhalgat et al. [37] can be faster in practice than the algorithm of Gabow for computing a maximum packing of edge-disjoint arborescences. Furthermore, another interesting direction is to provide implementations that exploit parallelism and may achieve improved performance in multi core architectures.

Additionally, realated to orientation problems from Chapter 5, we presented a linear time algorithm that computes edge-resilient strongly orientable blocks of a mixed graph. We may also introduce the similar concept of the 2*-edge strongly orientable blocks* of a mixed graph. These are the maximal sets of vertices $C_1, \ldots, C_k$ with the property that, for every $i \in \{1, \ldots, k\}$, there is an orientation R of G such that all vertices of C_i are 2-edge strongly connected in R. There are some relations between the 2-edge strongly orientable blocks and the edge-resilient strongly orientable blocks. First, we can easily see that every 2-edge strongly orientable block lies within an edge-resilient strongly orientable block. Moreover, both these concepts coincide with the 2-edge strongly connected components in directed graphs. However, Figure 5.1 shows that these concepts do not coincide in general. Despite these connections, the efficient computation of the 2-edge strongly orientable blocks seems to be an even more challenging problem than computing the edge-resilient strongly orientable

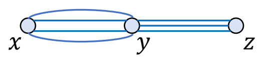

Figure 5.1: In this example $\{x, y\}$ is a 2-edge strongly orientable block. No orientation can make z and y (or z and x) 2-edge strongly connected. $\{x, y, z\}$ is an edge-resilient strongly orientable block.

blocks, since the former generalizes the notion of 2-edge strong orientations of a mixed graph.

We note that our techniques may be useful for solving other connectivity problems in mixed graphs, such as related connectivity augmentation problems [79, 72, 80].

Furthermore, an undirected graph has a strongly connected orientation if and only if it is 2-edge-connected. For higher edge-connectivity, this theorem was generalized by Nash-Williams as follows. Let G be an undirected graph and k a positive integer. Then G has a k-edge-connected orientation if and only if G is $2k$-edge-connected [81]. To find such an orientation we can use a special case of submodular flow [62] in which we assign at each edge a cost of orientation at each direction. Then we can find the minimum total cost of orienting the edges satisfying the constraint that the resulting graph is k-edge-connected. The analogous problem for vertex-connectivity turns out to be much more complicated. An undirected graph admits a strongly 2-vertex-connected orientation if and only if it is 4-edge-connected and every vertex deleted subgraph is 2-edge-connected [81]. The problem of deciding whether a given undirected graph G has a k-vertex connected orientation is NP-hard for any $k > 2$ [81].

The problem of finding whether a *mixed graph* has a 2-edge-connected orientation can be solved in polynomial time [46], due to Andras Frank who obtained a pretty technical characterization of mixed graphs admitting a k-edge connected orientation for all $k \in Z+$ using the theory of generalized polymatroids. On the other hand, the problem of finding whether a *mixed graph* has a 2-vertex-connected orientation is NP-hard [81].

The same problems appear if we consider the notion of *rooted connected graphs*. In particular, let $r \in V$ be a fixed vertex called the root of the graph. The r-rooted edge connectivity is the minimum number of edges that have to be removed so that there is some vertex in $V \setminus r$ that r cannot reach. A graph G is rooted connected if the root vertex r can reach any other vertex v. Similarly, the r-rooted vertex connectivity is the minimum number of vertices (excluding r) that have to be removed so that r cannot reach some vertex in the residual graph. Deciding whether a rooted mixed graph has a rooted k-vertex-connected orientation at a given vertex is NP-complete for any $k > 2$ [82]. To the best of our knowledge, we are not aware of sufficient conditions that guarantee the existence of a rooted 2-edge-connectivity orientation of an undirected or of a mixed graph. Furthermore, the sufficient conditions are not known for a rooted a 2-vertex-connectivity orientation of a given undirected graph. Finally, there is no proof of whether the problem of deciding the existence of a rooted 2-vertex-connectivity orientation of a mixed graph is NP-hard. In the tables below we can see more clearly which problems can be solved in polynomial time and which problems are NP-hard. The cells with the question marks indicate that the specific case is not known yet.

Table 5.1: {2-edge, 2-vertex}-connected-orientation and rooted-{2-edge, 2-vertex}-connected-orientation in undirected and mixed graphs.

Graph	2EC-orientation	2VC-orientation	rooted-2EC-orientation	rooted-2VC-orientation
Undirected	polynomial	polynomial	?	?
Mixed	polynomial	NP-hard	?	?

Table 5.2: {k-edge, k-vertex}-connected-orientation and rooted-{k-edge, k-vertex}-connected-orientation in undirected and mixed graphs for general $k > 2$.

Graph	kEC-orientation	kVC-orientation	rooted-kEC-orientation	rooted-kVC-orientation
Undirected	polynomial	NP-hard	?	?
Mixed	polynomial	NP-hard	?	NP-hard

Bibliography

[1] H. N. Gabow, “A matroid approach to finding edge connectivity and packing arborescences,” *Journal of Computer and System Sciences*, vol. 50, pp. 259–273, 1995.

[2] S. Chechik, T. D. Hansen, G. F. Italiano, V. Loitzenbauer, and N. Parotsidis, “Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs,” in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’17, (Philadelphia, PA, USA), pp. 1900–1918, Society for Industrial and Applied Mathematics, 2017.

[3] S. Forster, D. Nanongkai, L. Yang, T. Saranurak, and S. Yingchareonthawornchai, “Computing and testing small connectivity in near-linear time and queries via fast local cut algorithms,” in *Proc. of the 2020 ACM-SIAM Symp. on Discrete Algorithms*, pp. 2046–2065, 2020.

[4] R. E. Tarjan, “A good algorithm for edge-disjoint branching,” *Information Processing Letters*, vol. 3, no. 2, pp. 51–53, 1974.

[5] P. Tong and E. Lawler, “A faster algorithm for finding edge-disjoint branchings,” *Information Processing Letters*, vol. 17, no. 2, pp. 73–76, 1983.

[6] L. Georgiadis, D. Kefallinos, L. Laura, and N. Parotsidis, “An experimental study of algorithms for computing the edge connectivity of a directed graph,” in *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021* (M. Farach-Colton and S. Storandt, eds.), pp. 85–97, SIAM, 2021.

[7] S. Forster and L. Yang, “A faster local algorithm for detecting bounded-size cuts with applications to higher-connectivity problems,” *CoRR*, vol. abs/1904.08382, 2019.

[8] J. Edmonds, “Edge-disjoint branchings,” *Combinatorial Algorithms*, pp. 91–96, 1972.

[9] L. Georgiadis, D. Kefallinos, A. Mpanti, and S. D. Nikolopoulos, “An Experimental Study of Algorithms for Packing Arborescences,” in *20th International Symposium on Experimental Algorithms (SEA 2022)* (C. Schulz and B. Uçar, eds.), vol. 233 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 14:1–14:16, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[10] L. Georgiadis, D. Kefallinos, and E. Kosinas, “On 2-strong connectivity orientations of mixed graphs and related problems,” in *34th International Workshop on Combinatorial Algorithms (IWOCA 2023)*, 2023.

[11] K. Hanauer, M. Henzinger, and C. Schulz, “Recent advances in fully dynamic graph algorithms – a quick reference guide,” *ACM J. Exp. Algorithmics*, vol. 27, dec 2022.

[12] K. Menger, “Zur allgemeinen kurventheorie,” *Fundamenta Mathematicae*, vol. 10, no. 1, pp. 96–115, 1927.

[13] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis, “2-edge connectivity in directed graphs,” *ACM Transactions on Algorithms*, vol. 13, no. 1, pp. 9:1–9:24, 2016. Announced at SODA 2015.

[14] D. R. Ford and D. R. Fulkerson, *Flows in Networks*. USA: Princeton University Press, 2010.

[15] S. Even and R. E. Tarjan, “Network flow and testing graph connectivity,” *SIAM Journal on Computing*, vol. 4, no. 4, pp. 507–518, 1975.

[16] Y. Dinic, “Algorithm for solution of a problem of maximum flow in networks with power estimation,” *Soviet Math. Dokl.*, vol. 11, p. 1277–1280, 1970.

[17] C. P. Schnorr, “Bottlenecks and edge connectivity in unsymmetrical networks,” *SIAM Journal on Computing*, vol. 8, p. 265–274, 1979.

[18] Y. Mansour and B. Schieber, “Finding the edge connectivity of directed graphs,” *J. Algorithms*, vol. 10, p. 76–85, Mar. 1989.

[19] L. Georgiadis, G. F. Italiano, and N. Parotsidis, "Strong connectivity in directed graphs under failures, with applications," *SIAM J. Comput.*, vol. 49, no. 5, pp. 865–926, 2020.

[20] M. Henzinger, S. Krinninger, and V. Loitzenbauer, "Finding 2-edge and 2-vertex strongly connected components in quadratic time," in *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, (ICALP 2015), pp. 713–724, 2015.

[21] J. Edmonds, "Submodular functions, matroids, and certain polyhedra," *Combinatorial Structures and their Applications*, pp. 69–81, 1970.

[22] D. Nanongkai, T. Saranurak, and S. Yingchareonthawornchai, "Computing and testing small vertex connectivity in near-linear time and queries," *CoRR*, vol. abs/1905.05329, 2019.

[23] M. Ghaffari, K. Nowicki, and M. Thorup, "Faster algorithms for edge connectivity via random 2-out contractions," in *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, (USA), p. 1260–1279, Society for Industrial and Applied Mathematics, 2020.

[24] D. R. Karger, "Minimum cuts in near-linear time," *Journal of the ACM*, vol. 47, p. 46–76, Jan. 2000.

[25] M. Henzinger, S. Rao, and D. Wang, "Local flow partitioning for faster edge connectivity," *SIAM Journal on Computing*, vol. 49, no. 1, p. 1–36, 2020.

[26] K.-I. Kawarabayashi and M. Thorup, "Deterministic edge connectivity in near-linear time," *Journal of the ACM*, vol. 66, Dec. 2018.

[27] C. Demetrescu, A. Goldberg, and D. Johnson, "9th DIMACS implementation challenge - shortest paths." http://users.diag.uniroma1.it/challenge9/index.shtml, Oct. 2006.

[28] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.

[29] L. Georgiadis, L. Laura, N. Parotsidis, and R. E. Tarjan, "Dominator certification and independent spanning trees: An experimental study," in *Proc. 12th Int'l. Symp. on Experimental Algorithms*, pp. 284–295, 2013.

[30] P. Boldi and S.Vigna, "Laboratory for web algorithmics (LAW)." http://law.di.unimi.it/datasets.php, 2002.

[31] M. Franck and S. Yingchareonthawornchai, "Engineering nearly linear-time algorithms for small vertex connectivity," in *19th International Symposium on Experimental Algorithms (SEA 2021)*, Leibniz International Proceedings in Informatics (LIPIcs), p. 1–18, 2021.

[32] S. Fujishige and N. Kamiyama, "The root location problem for arc-disjoint arborescences," *Discrete Applied Mathematics*, vol. 160, no. 13, pp. 1964–1970, 2012.

[33] L. Lovász, "On two minimax theorems in graph," *Journal of Combinatorial Theory, Series B*, vol. 21, no. 2, pp. 96–103, 1976.

[34] Y. Shiloach, "Edge-disjoint branching in directed multigraphs," *Information Processing Letters*, vol. 8, no. 1, pp. 24–27, 1979.

[35] S. Fujishige, "A note on disjoint arborescences," *Combinatorica*, vol. 30, no. 2, p. 247–252, 2010.

[36] N. Kamiyama, N. Katoh, and A. Takizawa, "Arc-disjoint in-trees in directed graphs," *Combinatorica*, vol. 29, p. 197–214, 2009.

[37] A. Bhalgat, R. Hariharan, T. Kavitha, and D. Panigrahi, "Fast edge splitting and edmonds' arborescence construction for unweighted graphs," in *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, (USA), p. 455–464, Society for Industrial and Applied Mathematics, 2008.

[38] A. A. Benczúr and D. R. Karger, "Augmenting undirected edge connectivity in Õ(n2) time," *Journal of Algorithms*, vol. 37, no. 1, pp. 2–36, 2000.

[39] H. N. Gabow, "Efficient splitting off algorithms for graphs," in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, (New York, NY, USA), p. 696–705, Association for Computing Machinery, 1994.

[40] C. Demetrescu, A. Goldberg, and D. Johnson, "9th DIMACS Implementation Challenge: Shortest Paths." http:// www.dis.uniroma1.it/~challenge9/, 2007.

[41] CAD Benchmarking Lab, "ISCAS'89 benchmark information." http://www.cbl.ncsu.edu/www/CBL_Docs/iscas89.html.

[42] K. Bhawalkar, J. Kleinberg, K. Lewi, T. Roughgarden, and A. Sharma, "Preventing unraveling in social networks: The anchored k-core problem," *SIAM Journal on Discrete Mathematics*, vol. 29, no. 3, pp. 1452–1475, 2015.

[43] F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis, "The core decomposition of networks: theory, algorithms and applications," *The VLDB Journal*, vol. 29, no. 1, p. 61–92, 2020.

[44] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[45] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications (Springer Monographs in Mathematics)*. Springer, 1st ed. 2001. 3rd printing ed., 2002.

[46] A. Frank, *Connections in Combinatorial Optimization*. Oxford University Press, first ed., 2011.

[47] C. S. J. A. Nash-Williams, "On orientations, connectivity and odd-vertex-pairings in finite graphs," *Canadian Journal of Mathematics*, vol. 12, p. 555–567, 1960.

[48] A. Schrijver, *Combinatorial Optimization - Polyhedra and Efficiency*. Springer, 2003.

[49] R. Jaberi, "2-edge-twinless blocks," *Bulletin des Sciences Mathématiques*, vol. 168, p. 102969, 2021.

[50] S. Raghavan, "Twinless strongly connected components," in *Perspectives in Operations Research: Papers in Honor of Saul Gass' 80th Birthday* (F. B. Alt, M. C. Fu, and B. L. Golden, eds.), pp. 285–304, Boston, MA: Springer US, 2006.

[51] G. F. Italiano, L. Laura, and F. Santaroni, "Finding strong bridges and strong articulation points in linear time," *Theoretical Computer Science*, vol. 447, pp. 74–84, 2012.

[52] L. Georgiadis and E. Kosinas, "Linear-Time Algorithms for Computing Twinless Strong Articulation Points and Related Problems," in *31st International Symposium on Algorithms and Computation (ISAAC 2020)* (Y. Cao, S.-W. Cheng, and M. Li, eds.), vol. 181 of *Leibniz International Proceedings in Informatics (LIPIcs)*,

(Dagstuhl, Germany), pp. 38:1–38:16, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[53] G. D. Battista and R. Tamassia, “On-line maintenance of triconnected components with SPQR-trees,” *Algorithmica*, vol. 15, p. 302–318, Apr. 1996.

[54] G. D. Battista and R. Tamassia, “On-line planarity testing,” *SIAM Journal on Computing*, vol. 25, p. 956–997, Oct. 1996.

[55] A. Aamand, N. Hjuler, J. Holm, and E. Rotenberg, “One-way trail orientations,” in *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, vol. 107 of *LIPIcs*, pp. 6:1–6:13, 2018.

[56] A. Conte, R. Grossi, A. Marino, R. Rizzi, and L. Versari, “Directing road networks by listing strong orientations,” in *Combinatorial Algorithms*, pp. 83–95, Springer International Publishing, 2016.

[57] F. Boesch and R. Tindell, “Robbins’s theorem for mixed multigraphs,” *The American Mathematical Monthly*, vol. 87, no. 9, pp. 716–719, 1980.

[58] M. Elberfeld, D. Segev, C. R. Davidson, D. Silverbush, and R. Sharan, “Approximation algorithms for orienting mixed graphs,” *Theoretical Computer Science*, vol. 483, pp. 96–103, 2013. Special Issue Combinatorial Pattern Matching 2011.

[59] F. R. K. Chung, M. R. Garey, and R. E. Tarjan, “Strongly connected orientations of mixed multigraphs,” *Networks*, vol. 15, no. 4, pp. 477–484, 1985.

[60] A. Frank, “An algorithm for submodular functions on graphs,” in *Bonn Workshop on Combinatorial Optimization*, vol. 66 of *North-Holland Mathematics Studies*, pp. 97–120, North-Holland, 1982.

[61] H. N. Gabow, “A framework for cost-scaling algorithms for submodular flow problems,” in *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pp. 449–458, 1993.

[62] S. Iwata and Y. Kobayashi, “An algorithm for minimum cost arc-connectivity orientations,” *Algorithmica*, vol. 56, p. 437–447, 2010.

[63] A. Bhalgat and R. Hariharan, "Fast edge orientation for unweighted graphs," in *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pp. 265–272, 2009.

[64] H. N. Gabow, "Efficient splitting off algorithms for graphs," in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, p. 696–705, 1994.

[65] H. Nagamochi and T. Ibaraki, "Deterministic Õ(nn) time edge-splitting in undirected graphs," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, p. 64–73, 1996.

[66] L. Georgiadis, E. Kipouridis, C. Papadopoulos, and N. Parotsidis, "Faster computation of 3-edge-connected components in digraphs," in *Proceedings of 34th ACM-SIAM Symposium on Discrete Algorithms (SODA23)*, pp. 2489–2531, 2023.

[67] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp. 121–41, 1979.

[68] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, 1975.

[69] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup, "Dominators in linear time," *SIAM Journal on Computing*, vol. 28, no. 6, pp. 2117–32, 1999.

[70] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook, "Linear-time algorithms for dominators and other path-evaluation problems," *SIAM Journal on Computing*, vol. 38, no. 4, pp. 1533–1573, 2008.

[71] W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan, "Finding dominators via disjoint set union," *Journal of Discrete Algorithms*, vol. 23, pp. 2–20, 2013.

[72] H. N. Gabow, "The minset-poset approach to representations of graph connectivity," *ACM Transactions on Algorithms*, vol. 12, pp. 24:1–24:73, Feb. 2016.

[73] Y. Dinitz, "The 3-edge-components and a structural description of all 3-edge-cuts in a graph," in *Proceedings of the 18th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG '92, (Berlin, Heidelberg), p. 145–157, Springer-Verlag, 1992.

[74] Y. H. Tsin, “Yet another optimal algorithm for 3-edge-connectivity,” *Journal of Discrete Algorithms*, vol. 7, no. 1, pp. 130 – 146, 2009. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP).

[75] R. E. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[76] C. Gutwenger and P. Mutzel, “A linear time implementation of spqr-trees,” in *Graph Drawing* (J. Marks, ed.), (Berlin, Heidelberg), pp. 77–90, Springer Berlin Heidelberg, 2001.

[77] J. E. Hopcroft and R. E. Tarjan, “Dividing a graph into triconnected components,” *SIAM Journal on Computing*, vol. 2, no. 3, pp. 135–158, 1973.

[78] I. Heinrich, T. Heller, E. Schmidt, and M. Streicher, “2.5-connectivity: Unique components, critical graphs, and applications,” in *Graph-Theoretic Concepts in Computer Science* (I. Adler and H. Müller, eds.), (Cham), pp. 352–363, Springer International Publishing, 2020.

[79] J. Bang-Jensen, A. Frank, and B. Jackson, “Preserving and increasing local edge-connectivity in mixed graphs,” *SIAM Journal on Discrete Mathematics*, vol. 8, no. 2, pp. 155–178, 1995.

[80] D. Gusfield, “Optimal mixed graph augmentation,” *SIAM Journal on Computing*, vol. 16, no. 4, pp. 599–612, 1987.

[81] F. Hörsch and Z. Szigeti, “A note on 2-vertex-connected orientations.” https://arxiv.org/abs/2112.07539, 2021.

[82] O. D. de Gevigney, *Graphs Orientations : structures and algorithms*. PhD thesis, UNIVERSITE DE GRONOBLE, France, May 2014.

Author's Publications

- Loukas Georgiadis, Dionysios Kefallinos and Evangelos Kosinas, On 2-Strong Connectivity Orientations of Mixed Graphs and Related Problems, IWOCA2023.
- Loukas Georgiadis, Dionysios Kefallinos, Anna Mpanti, Stavros D. Nikolopoulos, An Experimental Study of Algorithms for Packing Arborescences. SEA 2022.
- Loukas Georgiadis, Dionysios Kefallinos, Luigi Laura, Nikos Parotsidis, An Experimental Study of Algorithms for Computing the Edge Connectivity of a Directed Graph. ALENEX 2021.

Short Biography

Dionysios Kefallinos was born in Zakynthos in 1993. He received his B.Sc. degree in Computer Science and Engineering (2017) from the Department of Computer Science and Engineering of the University of Ioannina, Greece. In 2019 he received his postgraduate degree in the Department of Computer Science and Engineering of the Polytechnic School of the University of Ioannina, acquiring the postgraduate degree (MSc) in Theory of Computer Science. His Master Thesis, entitled “A simpler algorithm for computing the kernel of a polyhedron”, was conducted under the supervision of Professor Leonidas Palios. His research interests are focused on the design and analysis of algorithms, data structures, computational complexity, algorithmic graph theory and computational Geometry.