2-Connectivity in Directed Graphs

MASTER THESIS

submitted to the designated
by the General Assembly Composition of the Department
of Computer Science & Engineering inquiry committee

from

Nikos Parotsidis

in fulfilment of the requirements for the

MASTER'S DEGREE IN COMPUTER SCIENCE
WITH EXPERTISE IN
COMPUTER SCIENCE THEORY

May 2015

2-Συνεκτικότητα σε Κατευθυνόμενα Γραφήματα

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην
ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Μηχανικών Η/Υ & Πληροφορικής
Εξεταστική Επιτροπή

από τον

Νικόλαο Παροτσίδη

ως μέρος των υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ
ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ
ΣΤΗΝ ΘΕΩΡΙΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Μάιος 2015

Acknowledgements

I would like to express my gratitude to my supervisor, Assistant Prof. Loukas Georgiadis, who gave me the opportunity to work on this exciting area and guided me in my very first steps in research.

I'm thankful to my co-authors, William Di Luigi, Prof. Giuseppe F. Italiano, and Dr. Luigi Laura, for accepting me in this great group within which we were able to produce the results that are included in this master thesis.

Special thanks are also intended to all the people of the third floor of our Department for the great atmosphere and the interesting discussions of all kind. My sincere thanks to Dr. Maria Chroni for her help and her pleasant attitude.

To my family and friends...

Contents

List of Figures

List of Tables

Abstract

Graphs are a fundamental mathematical tool used to model diverse entities such as the world-wide web, transportation, communication and social networks, databases, biological systems, VLSI circuits, and the control-flow of computer programs. It is of great interest to answer relational queries and solve problems on graphs, since very often such problems appear in practice and require sophisticated approaches. Connectivity holds a central role in the area of network and graph algorithms, with numerous practical applications. Such topics have been studied for decades and many significant results have been produced; nevertheless, many important problems remain open.

In the 2-connectivity problems in graphs, which are part of the greater family of connectivity problems, the objective is to compute the 2-connected subgraphs on the input graph, the so-called 2-connected components. A graph is called 2-connected if the removal of any edge (or vertex) leaves the graph connected (strongly connected when dealing with directed graph). The problem is further subdivided, depending on whether the elements to be removed refer to an edge or a vertex; the variations are denoted as 2-edge-connectivity and 2-vertex-connectivity, respectively. These concepts extend in both undirected and directed graphs.

In this master thesis we study a variety of 2-connectivity problems on directed graphs. Specifically, we introduce the notion of 2-*edge-connected blocks* and 2-*vertex-connected blocks* in directed graphs. We say that two vertices v and w are 2-edge-connected (resp., 2-vertex-connected) if there are two edge-disjoint (resp., internally vertex-disjoint) directed paths from v to w and two edge-disjoint (resp., internally vertex-disjoint) directed paths from w to v. We define a 2-*edge-connected block* (resp., a 2-vertex-connected block) of a digraph as a maximal subset S of vertices such that every distinct pair of vertices $v, w \in S$ is 2-edge-connected (resp., 2-vertex-connected).

The main results of this study are the two linear time algorithms to compute the 2-edge-connected and 2-vertex-connected blocks of a directed graph. These two algorithms are not only theoretically optimal, in terms of asymptotic running time, but also improve significantly over previous bounds. Although, the two algorithms follow the same high level approach, the description and the analysis of each of the algorithms is presented separately since, as we show, there are different technical difficulties and different structural properties that need to be tackled in each case. Furthermore, in both cases, we can augment the algorithms that compute the 2-edge-connected (resp., 2-vertex-connected) blocks with the computation of a subgraph of the initial graph that preserves the 2-edge-

connected (resp., 2-vertex-connected) blocks of the initial graph and has $O(n)$ edges. We call such a subgraph *sparse certificate* of the 2-edge-connected blocks (resp., 2-vertex-connected blocks).

Despite the fact that in our study we present linear time (asymptotically optimal) algorithms for computing the 2-edge-connected and 2-vertex-connected blocks, it is unknown, and of great interest, whether the 2-edge-connected and the 2-vertex-connected components can be computed in linear time as well. In this work we furthermore present new algorithms for computing the 2-edge-connected and the 2-vertex-connected components in $O(mn)$ worst-case time, where m is the number of edges and n is the number of vertices in the graph. The $O(mn)$ theoretical bound was the best known until very recently, where a new algorithm with worst-case time complexity $O(n^2)$ was proposed.

Finally, we engineered the implementations of our new algorithms for all the variations of the 2-connectivity problems that we consider and performed a thorough experimental study comparing our new algorithms against previously known solutions. Our experiments suggest that, in most cases, our new algorithms perform substantially better than the known algorithms.

ΠΕΡΙΛΗΨΗ

Οι γράφοι αποτελούν ένα θεμελιώδες μαθηματικό εργαλείο που χρησιμοποιείται κατά κόρον για την μοντελοποίηση διαφόρων οντοτήτων όπως τον Παγκόσμιο Ιστό, δίκτυα μεταφορών, τηλεπικοινωνιακά και κοινωνικά δίκτυα, βάσεις δεδομένων, βιολογικά συστήματα, κυκλώματα VLSI, καθώς και διαγράμματα ροής προγραμμάτων. Η απάντηση σχεσιακών ερωτημάτων και η επίλυση προβλημάτων σε γράφους, υφίσταται μεγάλου ενδιαφέροντος, δεδομένου ότι πολύ συχνά τέτοιου είδους προβλήματα εμφανίζονται στην πράξη και απαιτούν εξελιγμένες προσεγγίσεις. Τα προβλήματα συνεκτικότητας παίζουν κεντρικό ρόλο στην περιοχή των γράφων και των δικτύων, με πληθώρα πρακτικών εφαρμογών. Τέτοιου είδους θέματα έχουν μελετηθεί σε βάθος δεκαετιών και έχουν εμφανιστεί πολλά σημαντικά αποτελέσματα, παρά το γεγονός αυτό σημαντικό πολλά προβλήματα της περιοχής παραμένουν ανοιχτά.

Ένα υποσύνολο της ευρύτερης οικογένειας των προβλημάτων συνεκτικότητας αποτελούν τα προβλήματα 2-συνεκτικότητας σε γράφους, στα οποία ο στόχος είναι ο υπολογισμός των 2-συνεκτικών υπογράφων του γράφου εισόδου, τις λεγόμενες 2-συνεκτικές συνιστώσες. Ένας γράφος ονομάζεται 2-συνεκτικός εάν η αφαίρεση οποιασδήποτε ακμής (ή κορυφής) αφήνει τον υπολειπόμενο γράφο συνεκτικό (ισχυρά συνεκτικό όταν αναφερόμαστε σε κατευθυνόμενους γράφους). Τα προβλήματα 2-συνεκτικότητας υποδιαιρούνται περαιτέρω ανάλογα με το αν τα στοιχεία προς διαγραφή αφορούν σε ακμές ή κορυφές και αναφέρονται στην εύρεση των 2-συνεκτικών συνιστωσών ως προς τις ακμές και των 2-συνεκτικών συνιστωσών ως προς τους κόμβους. Οι παραπάνω έννοιες εκτείνονται τόσο σε μη-κατευθυνόμενους όσο και σε κατευθυνόμενους γράφους.

Στην παρούσα διπλωματική εργασία μελετάμε μια σειρά από προβλήματα 2-συνεκτικότητας σε κατευθυνόμενους γράφους. Πιο συγκεκριμένα, εισαγάγουμε την έννοια των 2-συνεκτικών μπλοκ ως προς τις ακμές και των 2-συνεκτικών μπλοκ ως προς τις κορυφές σε κατευθυνόμενους γράφους. Λέμε ότι δύο κορυφές v και w είναι 2-συνεκτικές ως προς τις ακμές (αντιστ., 2-συνεκτικές ως προς τις κορυφές) αν υπάρχουν δύο κατευθυνόμενα μονοπάτια ξένα ως προς τις ακμές (αντιστ., εσωτερικά ξένα ως προς τις κορυφές) από την κορυφή v προς την w και δύο κατευθυνόμενα μονοπάτια ξένα ως προς τις ακμές (αντιστ., εσωτερικά ξένα ως προς τις κορυφές) από την κορυφή w προς την v. Ορίζουμε ένα 2-συνεκτικό μπλοκ ως προς τις ακμές (αντίστ., ένα 2-συνεκτικό μπλοκ προς τις κορυφές) ως το μεγιστοτικό σύνολο κόμβων S τέτοιο ώστε οποιεσδήποτε δύο κορυφές $v, w \in S$ να είναι 2-συνεκτικές ως προς τις ακμές (αντιστ., 2-συνεκτικές ως προς τις κορυφές).

Τα σημαντικότερα αποτελέσματα αυτής της μελέτης είναι η εύρεση δύο γραμμικών αλγορίθμων για τον υπολογισμό των 2-συνεκτικών μπλοκ ως προς τις ακμές και των 2-συνεκτικών

μπλοκ ως προς τις κορυφές ενός κατευθυνόμενου γράφου. Οι δύο αυτοί αλγόριθμοι δεν είναι μόνο θεωρητικά βέλτιστοι, όσων αφορά την ασυμπτωτική τους πολυπλοκότητα, αλλά βελτιώνουν επίσης σημαντικά τον προηγούμενο καλύτερο χρόνο για τον υπολογισμό αυτών των σχέσεων. Παρά το γεγονός ότι οι δύο αλγόριθμοι ακολουθούν την ίδια προσέγγιση υψηλού επιπέδου, η περιγραφή και η ανάλυση του κάθε αλγόριθμου παρουσιάζεται ξεχωριστά διότι, όπως δείχνουμε, εμφανίζονται διαφορετικές τεχνικές δυσκολίες και δομικά χαρακτηριστικά που χρήζουν ειδικής αντιμετώπισης σε κάθε περίπτωση. Επιπλέον, τόσο στον αλγόριθμο για τον υπολογισμό των 2-συνεκτικών μπλοκ ως προς τις ακμές όσο και στον αλγόριθμο για τον υπολογισμό των 2-συνεκτικών μπλοκ ως προς τις κορυφές, μπορούμε να ενσωματώσουμε τον υπολογισμό ενός αραιού υπογράφου του γράφου εισόδου που διατηρεί αντίστοιχα τα 2-συνεκτικά μπλοκ ως προς τις ακμές και τα 2-συνεκτικά μπλοκ ως προς τις κορυφές του αρχικού γράφου και έχει $O(n)$ ακμές. Καλούμε ένα τέτοιο υπογράφο αραιό πιστοποιητικό των 2-συνεκτικών μπλοκ ως προς τις ακμές (αντίστοιχα των 2-συνεκτικών μπλοκ ως προς τις κορυφές).

Παρά το γεγονός ότι στη μελέτη μας παρουσιάζουμε γραμμικού χρόνου αλγόριθμους για τον υπολογισμό των 2-συνεκτικών μπλοκ ως προς τις ακμές και 2-συνεκτικών μπλοκ ως προς τις κορυφές, είναι άγνωστο, και παρουσιάζει μεγάλο ενδιαφέρον, αν οι 2-συνεκτικές συνιστώσες ως προς τις ακμές και οι 2-συνεκτικές συνιστώσες ως προς τους κόμβους μπορούν να υπολογιστούν σε γραμμικό χρόνο. Στην παρούσα εργασία περιγράφουμε επιπλέον νέους αλγόριθμους για τον υπολογισμό των 2-συνεκτικών συνιστωσών ως προς τις ακμές και των 2-συνεκτικών συνιστωσών ως προς τις κορυφές με $O(mn)$ χρόνο εκτέλεσης στην χειρότερη περίπτωση, όπου m είναι το πλήθος των ακμών και n είναι το πλήθος των κορυφών του γράφου. Ήταν άγνωστο αν αυτές οι σχέσεις μπορούν να υπολογιστούν ταχύτερα μέχρι πολύ πρόσφατα, όταν ένας νέος αλγόριθμος προτάθηκε και πετυχαίνει χρόνο χειρότερης περίπτωσης $O(n^2)$.

Τέλος, υλοποιήσαμε τους αλγόριθμους που προτείναμε για όλα τα προβλήματα 2-συνεκτικότητας που θεωρήσαμε και εκτελέσαμε μια εκτενή πειραματική μελέτη στην οποία συγκρίναμε τις μεθόδους μας με τους γνωστούς αλγόριθμους για κάθε πρόβλημα. Τα πειράματά μας δείχνουν ότι, στις περισσότερες περιπτώσεις, οι νέοι αλγόριθμοι πετυχαίνουν σημαντικά καλύτερες επιδόσεις από τους γνωστούς αλγορίθμους.

Chapter 1

Introduction and Theoretical Background

1.1 Graphs and applications

Graphs are a fundamental mathematical tool which is used for representing elements and the pairwise relations between them. Formally, a graph G is defined by a pair of sets V and E (i.e. $G = (V, E)$), where V is the set of *vertices* (elements of the graph), and E is the set of pairwise connections between the vertices, which are called *edges*. The set of edges E may contain either unordered of ordered pairs of vertices, distinguishing undirected from directed graphs, respectively. In directed graphs, an edge $(u, v) \in E$, represents a directed connection from vertex u to vertex v; (u, v) is an outgoing edge from u and incoming to v. We call u and v the source and the destination, respectively, of an edge (u, v). In undirected graphs, each edge $(u, v) \in E$ has no direction and is both outgoing and incoming to u and v. We refer the reader that is interested in an extensive graph terminology to the literature, as for instance in [5].

There is a great variety of practical problems that can be formulated as graphs. The representation of a problem with graphs, offers a unique plethora of algorithms and techniques for studying and extracting information from the graph. In computer science,

graphs are used to represent communication or transportation networks, data organization, computational devices, the flow of computation, etc. For instance, the link structure of a website can be represented by a directed graph, in which the vertices represent web pages and directed edges represent links from one page to another. A similar approach can be taken to problems in transportation, biology, computer chip design, and many other fields. The development of algorithms to handle graphs is therefore of major interest in computer science. The transformation of graphs is often formalized and represented by graph rewrite systems. Complementary to graph transformation systems focusing on rule-based in-memory manipulation of graphs are graph databases geared towards transaction-safe, persistent storing and querying of graph-structured data.

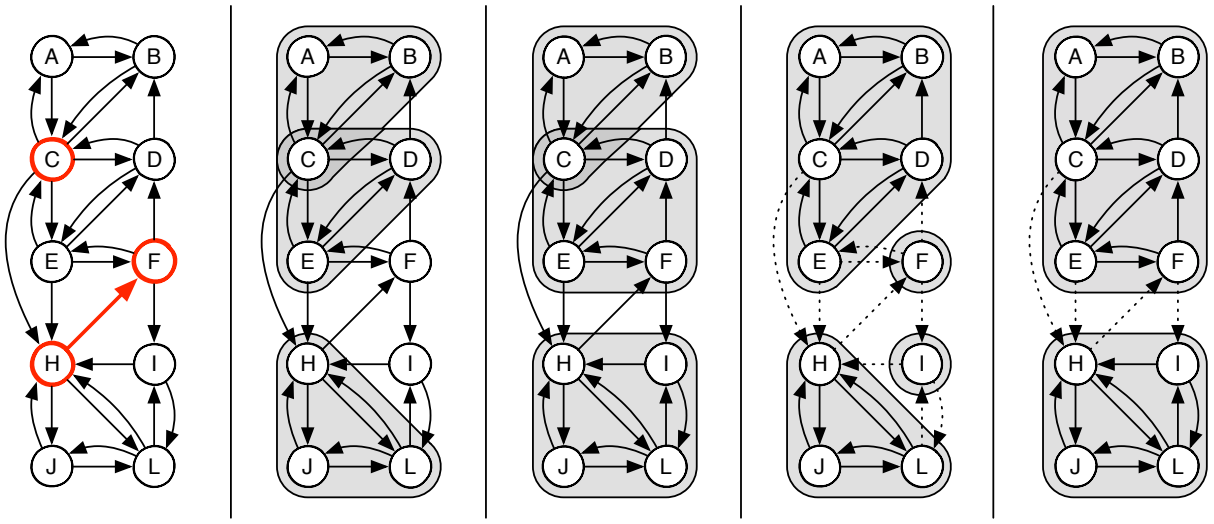
1.2 Theoretical background

An *undirected path* (resp., *directed path*) in G is a sequence of vertices v_1, v_2, ..., v_k, such that edge $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k-1$. A path P is called *simple* if all the vertices in P, except the first and the last, are distinct. In this thesis we will consider only simple paths and we will refer to them simply as paths. An undirected graph G is *connected* if there is an undirected path from each vertex to every other vertex. The *connected components* of an undirected graph are its maximal connected subgraphs. A directed graph G is *strongly connected* if there is a directed path from each vertex to every other vertex. The *strongly connected components* of a directed graph are its maximal connected subgraphs.

Let $G = (V, E)$ be an *undirected* (resp., *directed*) graph, with m edges and n vertices. Throughout the paper, we use interchangeably the term directed graph and digraph. Edge and vertex connectivity are fundamental concepts in graph theory with numerous practical applications [2, 30]. As an example, we mention the computation of disjoint paths in routing and reliable communication, both in undirected and directed graphs [18, 21].

1.2.1 2-connectivity

Given an undirected graph $G = (V, E)$, an edge is a *bridge* if its removal increases the number of connected components of G. Graph G is 2-edge-connected if it has no bridges. The 2-edge-connected components of G are its maximal 2-edge-connected subgraphs. Two vertices v and w are 2-edge-connected if there are two edge-disjoint paths between v and w: we denote this relation by $v \leftrightarrow_{2e} w$. Equivalently, by Menger's Theorem [28], v and w are 2-edge-connected if the removal of any edge leaves them in the same connected component. Analogous definitions can be given for 2-vertex connectivity. In particular, a vertex is an *articulation point* if its removal increases the number of connected components of G. A graph G is 2-vertex-connected if it has at least three vertices and no articulation points. The 2-vertex-connected components of G are its maximal 2-vertex-connected subgraphs.



(a) G (b) $2VCC(G)$ (c) $2VCB(G)$ (d) $2ECC(G)$ (e) $2ECB(G)$

Figure 1.1: (a) A strongly connected digraph G, with strong articulation points and strong bridges shown in red (better viewed in color); (b) The 2-vertex-connected components of G; (c) The 2-vertex-connected blocks of G; (d) The 2-edge-connected components of G; (e) The 2-edge-connected blocks of G.

Note that the condition on the minimum number of vertices in a 2-vertex-connected graph disallows degenerate 2-vertex-connected components consisting of one single edge. Two vertices v and w are 2-vertex-connected if there are two internally vertex-disjoint paths between v and w: we denote this relation by $v \leftrightarrow_{2\text{v}} w$. If v and w are 2-vertex-connected then Menger's Theorem implies that the removal of any vertex different from v and w leaves them in the same connected component. The converse does not necessarily hold, since v and w may be adjacent but not 2-vertex-connected. It is easy to show that $v \leftrightarrow_{2\text{e}} w$ (resp., $v \leftrightarrow_{2\text{v}} w$) if and only if v and w are in a same 2-edge-connected (resp., 2-vertex-connected) component. All bridges, articulation points, 2-edge- and 2-vertex-connected components of undirected graphs can be computed in linear time essentially by the same algorithm [33].

The notions of 2-edge and 2-vertex connectivity were naturally extended to directed graphs in [22]. Given a digraph G, an edge is a *strong bridge* if its removal increases the number of strongly connected components of G. Respectively, a vertex is a *strong articulation point* if its removal increases the number of strongly connected components of G. A digraph G is 2-edge-connected if it has no strong bridges; G is 2-vertex-connected if it has at least three vertices and no strong articulation points. The 2-edge-connected (resp., 2-vertex-connected) components of G are its maximal 2-edge-connected (resp., 2-vertex-connected) subgraphs. Again, the condition on the minimum number of vertices disallows for degenerate 2-vertex-connected components consisting of two mutually adjacent vertices (i.e., two vertices v and w and the two edges (v, w) and (w, v)).

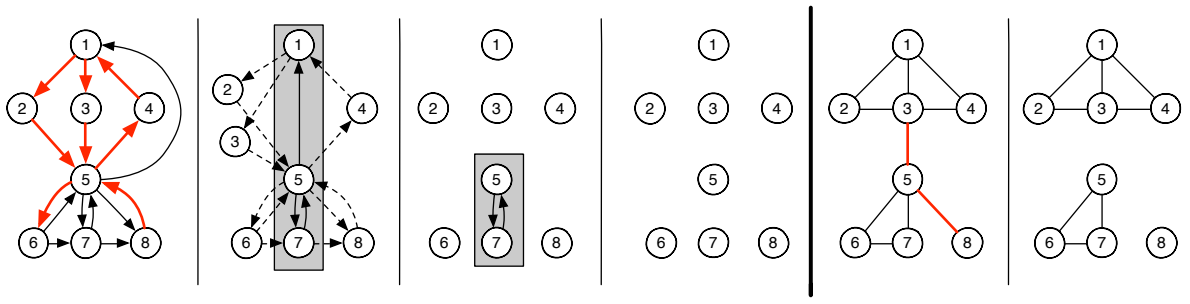
Similarly to the undirected case, we say that two vertices v and w are 2-edge-connected,

and we denote this relation by $v \leftrightarrow_{2e} w$, if there are two edge-disjoint directed paths from v to w and two edge-disjoint directed paths from w to v. (Note that a path from v to w and a path from w to v need not be edge-disjoint). It is easy to see that $v \leftrightarrow_{2e} w$ if and only if the removal of any edge leaves v and w in the same strongly connected component. We define a 2-*edge-connected block* of a digraph $G = (V, E)$ as a maximal subset $B \subseteq V$ such that $u \leftrightarrow_{2e} v$ for all $u, v \in B$. Analogous definitions can be given for 2-vertex connectivity. We say that two vertices v and w are 2-vertex-connected, and we denote this relation by $v \leftrightarrow_{2v} w$, if there are two internally vertex-disjoint directed paths from v to w and two internally vertex-disjoint directed paths from w to v. (Note that a path from v to w and a path from w to v need not be vertex-disjoint). As in the 2-edge connectivity, $v \leftrightarrow_{2v} w$ implies that the removal of any vertex different from v and w leaves v and w in the same strongly connected component. We define a 2-*vertex-connected block* of a digraph $G = (V, E)$ as a maximal subset $B \subseteq V$ such that $u \leftrightarrow_{2v} v$ for all $u, v \in B$. The 2-connectivity blocks relations were first considered by Reif and Spirakis in [32].

It can be easily seen that, differently from undirected graphs, in digraphs 2-edge- and 2-vertex-connected blocks do not correspond to 2-edge- and 2-vertex-connected components, as illustrated in Figure 1.1. Two vertices may be 2-edge-connected (resp., 2-vertex-connected) but lie in different 2-edge-connected (resp., 2-vertex-connected) components. Furthermore, these notions seem to have a much richer (and more complicated) structure in digraphs. Just to give an example, we observe that while in the case of undirected connected graphs the 2-edge-connected components (which correspond to the 2-edge-connected blocks) are exactly the connected components left after the removal of all bridges, for directed strongly connected graphs the 2-edge-connected components, the 2-edge-connected blocks, and the strongly connected components left after the removal of all strong bridges are not necessarily the same. These observations are illustrated in Figure 1.2. Put in other words, differently from the undirected case, in digraphs 2-vertex- (resp., 2-edge-) connected components do not encompass the notion of pairwise 2-vertex (resp., 2-edge) connectivity among its vertices. We note that pairwise 2-connectivity may be relevant in several applications, where one is interested in local properties, e.g., checking whether two vertices are 2-connected, rather than in global properties.

1.2.2 Related work

Following the discussion from Section 1.2.1, it is not surprising that 2-connectivity problems on directed graphs appear to be more difficult than on undirected graphs. For undirected graphs it has been known for over 40 years how to compute all bridges, articulation points, 2-edge- and 2-vertex-connected components in linear time, by simply using depth first search [33]. In the case of digraphs, however, the very same problems have been much more challenging. Indeed, it has been shown only few years ago that all strong bridges and strong articulation points of a digraph can be computed in linear time [22]. Furthermore, the best current bound for computing the 2-edge- and the 2-vertex-connected components in digraphs is not even linear, but it is $O(n^2)$, and it was achieved only very recently by



(a) G (b) $2ECB(G)$ (c) $G/SB(G)$ (d) $2ECC(G)$ (e) U (f) $2ECC(U)$

Figure 1.2: (a) A digraph G with strong bridges shown in red; (b) The 2-edge-connected blocks of G; (c) The strongly connected components left after removing all the strong bridges from G; (d) The 2-edge-connected components of G. (e) An undirected graph U with bridges shown in red; (f) The 2-edge-connected components of U, corresponding to the 2-edge-connected blocks and to the connected components left after the removal of all bridges of U.

Henzinger et al. [20], improving previous $O(mn)$ time bounds [24, 31]. A simple algorithm for computing the 2-edge-connected components can be obtained by repeatedly removing all the strong bridges in the graph (and repeating this process until no strong bridges are left). Since at each round all the strong bridges can be computed in $O(m+n)$ time [22] and there can be at most $O(n)$ rounds, the total time taken by this algorithm is $O(mn)$. As for 2-vertex connectivity, Erusalimskii and Svetlov [6] proposed an algorithm that reduces the problem of computing the 2-vertex-connected components of a digraph to the computation of the 2-vertex-connected components in an undirected graph, but did not analyze the running time of their algorithm. Their reduction is achieved by repeatedly computing the strongly connected components of all subgraphs $G \setminus v$, for every vertex v, and deleting the edges that connect different strongly connected components. This process is repeated until no edge is removed in all current subgraphs $G \setminus v$; the 2-vertex-connected components of the resulting digraph G are identical to the 2-vertex-connected components of the undirected version of G. Jaberi [24] showed that the algorithm of Erusalimskii and Svetlov has $O(nm^2)$ running time, and proposed two different algorithms with running time $O(mn)$.The first algorithm decomposes the digraph by repeatedly removing a strong articulation point at a time. The second algorithm divides the digraph using a dominator tree [25]. The computation of the k-edge-connected components of a digraph was considered by Matula and Vohra [27], where they gave an $O(n^3)$-time algorithm.

A simple algorithm for computing the 2-edge- or 2-vertex-connected blocks of a digraph takes $O(mn)$ time: given a vertex v, one can find in linear time all the vertices that are 2-edge- or 2-vertex-connected with v with the help of dominator trees. Since in the worst case this step must be repeated for all vertices v, the total time required by the simple algorithm is $O(mn)$. Very recently, and independently of our work, Jaberi [23]

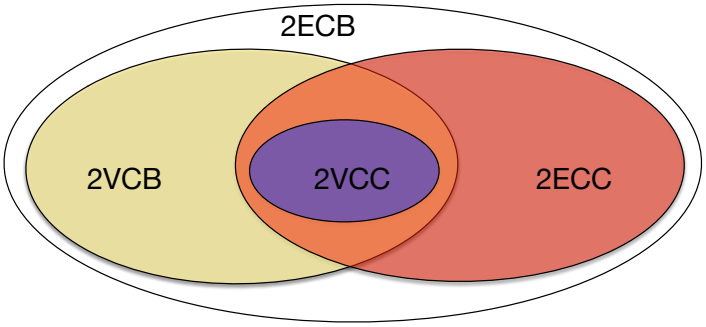


Figure 1.3: The relation among various notions of 2-connectivity in directed graphs.

presented algorithms for computing the 2-vertex-connected and 2-edge-connected blocks. His algorithms require $O(n \cdot \min\{m, b^*n\})$ time for computing the 2-edge-connected blocks and $O(n \cdot \min\{m, (a^* + b^*)n\})$ time for computing the 2-vertex-connected blocks, where a^* and b^* are respectively the number of strong articulation points and strong bridges in the digraph G. Since both a^* and b^* can be as large as $O(n)$, both bounds are $O(mn)$ in the worst case.

From the above discussion it is clear that, differently from the case of undirected graphs, for digraphs there is a huge gap between the $O(m + n)$ time bound for computing all connectivity cuts (strong bridges and strong articulation points), and the $O(mn)$ time bound for computing the connectivity blocks or components (2-edge- and 2-vertex-connected blocks and 2-edge- and 2-vertex-connected components). Thus, it seems quite natural to ask whether the $O(mn)$ bound is a natural barrier for those problems, or whether they could be solved faster in linear time.

1.2.3 Digraphs, dominators, articulation points, and bridges

In this section we introduce some terminology that will be useful throughout the paper. A *flow graph* is a digraph such that every vertex is reachable from a distinguished start vertex. Let $G = (V, E)$ be the input digraph, which we assume to be strongly connected. (If not, we simply treat each strongly connected component separately.) For any vertex $s \in V$, we denote by $G(s) = (V, E, s)$ the corresponding flow graph with start vertex s; all vertices in V are reachable from s since G is strongly connected. The *dominator relation* in $G(s)$ is defined as follows: A vertex u is a *dominator* of a vertex w (u *dominates* w) if every path from s to w contains u; u is a *proper dominator* of w if u dominates w and $u \neq w$. The dominator relation is reflexive and transitive. Its transitive reduction is a rooted tree, the *dominator tree* $D(s)$: u dominates w if and only if u is an ancestor of w in $D(s)$. If $w \neq s$, $d(w)$, the parent of w in $D(s)$, is the *immediate dominator* of w: it is the unique proper dominator of w that is dominated by all proper dominators of w. An edge (u, w) is a *bridge* in $G(s)$ if all paths from s to w include (u, w). Throughout the paper, to avoid danger of ambiguity we use consistently the term *bridge* to refer to a

bridge of a flow graph $G(s)$ and the term *strong bridge* to refer to a strong bridge in the original graph G.

Lengauer and Tarjan [25] presented an algorithm for computing dominators in $O(m\alpha(n, m/n))$ time for a flow graph with n vertices and m edges, where α is a functional inverse of Ackermann's function [36]. Subsequently, several linear-time algorithms were discovered [1, 3, 4, 8, 9, 15]. Tarjan [34] showed that the bridges of flow graph $G(s)$ can be computed in $O(m)$ time given $D(s)$. He also presented an $O(m\alpha(n, m/n))$-time algorithm to compute bridges that uses static tree set union to contract strongly connected subraphs in G [37]. The Gabow-Tarjan static tree disjoint set union algorithm [10] reduces the running time of this algorithm to $O(m)$ on a RAM. Buchsbaum et al. [3] gave an $O(m)$-time pointer-machine algorithm.

Italiano et al. [22] showed that the strong articulation points of G can be computed from the dominator trees of $G(s)$ and $G^R(s)$, where s is an arbitrary start vertex and G^R is the digraph that results from G after reversing edge directions; similarly, the strong bridges of G correspond to the bridges of $G(s)$ and $G^R(s)$. This gives the following bound on the number of strong bridges.

Lemma 1.1. *Any digraph with n vertices has at most $2n - 2$ strong bridges.*

Experimental studies for algorithms that compute dominators, strong bridges, and strong articulation points are presented in [7, 13, 14, 17]. The experimental results show that the corresponding fast algorithms given in [8, 22, 25, 37] perform very well in practice even on very large graphs.

1.3 Contribution

In this work, we present the first linear-time algorithms to compute the 2-edge-connected and the 2-vertex-connected blocks of a digraph. Our algorithms are not only asymptotically optimal, but also improve significantly over previous bounds. Furthermore, the ability to compute the 2-edge-connected and the 2-vertex-connected blocks of a digraph in linear time seems a significant step, especially as it was the first real progress on this extremely natural problem, starting from the foundational work done 40 years ago for undirected graphs.

Our algorithms follow the high-level approach for computing the 2-edge-connected blocks and the 2-vertex-connected blocks. However, the algorithm for computing the 2-vertex-connected blocks is much more involved and requires several novel ideas and non-trivial techniques to achieve the claimed bounds. In particular, we discuss the main technical difficulties that need to be tackled in Section 3.2.

1.3.1 2-edge-connectivity

We present the first linear-time algorithm to compute the 2-edge-connected blocks of a digraph [11]. Our algorithm, as mentioned above, is asymptotically optimal and improves

significantly over the previous known bounds $O(mn)$. Our approach hinges on two different algorithms. The first is a simple iterative algorithm that builds the 2-edge-connected blocks by removing one strong bridge at a time. The second algorithm is more involved and recursive: the main idea is to consider simultaneously how several distinct strong bridges partition vertices with the help of dominator trees. Although both algorithms run in $O(mn)$ time in the worst case, we show that a sophisticated combination of the iterative and the recursive method is able to achieve the claimed linear-time bound.

Using our algorithm for 2-edge-connected blocks, we can preprocess a digraph in linear time, and then answer in constant time queries on whether any two vertices are 2-edge-connected. We also show how to compute in linear time a sparse certificate for 2-edge-connected blocks, i.e., a subgraph of the input graph that has $O(n)$ edges and maintains the same 2-edge-connected blocks as the input graph.

1.3.2 2-vertex-connectivity

We complete the picture on 2-connectivity for digraphs by presenting the first algorithm for computing the 2-vertex-connected blocks in $O(m+n)$ time [12]. Our bound is asymptotically optimal and it improves sharply over a previous $O(mn)$ time bound by Jaberi [23]. As a side result, our algorithm constructs an $O(n)$-space data structure that reports in constant time if two vertices are 2-vertex-connected. Additionally, when two query vertices v and w are not 2-vertex-connected, our data structure can produce, in constant time, a "witness" by exhibiting a vertex (i.e., a strong articulation point) or an edge (i.e., a strong bridge) that separates them. We are also able to compute in linear time a sparse certificate for 2-vertex connectivity, i.e., a subgraph of the input graph that has $O(n)$ edges and maintains the same 2-vertex connectivity properties.

We additionally contribute in the area of 2-vertex-connectivity, by providing a new $O(mn)$-time algorithm for computing the 2-vertex-connected components of a digraph, that refines the dominator tree division used by Jaberi [24]. Our new algorithms does not decrease the best known asymptotic bound, but we show that it performs very well in practice.

1.3.3 2-connectivity in practice

We also consider the computation of the 2-edge- and 2-vertex-connected blocks and components of a digraph in practice [26], and present efficient implementations of the algorithms introduced in this work, and also compare them to known algorithms [6, 24].

We evaluate the efficiency of our algorithms experimentally on large digraphs taken from a variety of application areas. To the best of our knowledge, this is the first empirical study for these problems. Our extensive experimental study sheds light on the relative difficulty of computing various notions of 2-connectivity in directed graphs. Specifically, we compare the performance of the linear-time algorithms for computing the 2-edge- and 2-vertex-connected blocks of a digraph with simpler algorithms that iterate over the strong

articulation points and strong bridges of the digraph. We also consider the computation of the 2-vertex-connected components of a digraph and compare the performance of our new algorithm and the algorithms of Erusalimskii and Svetlov [6] and Jaberi [24]. Our results show that algorithms that apply a dominator-tree-based division of the input digraph perform well in practice and are more robust than their simpler competitors. The experimental results also suggest that the 2-edge- and 2-vertex-connected components of digraphs that arise in many practical applications can be found efficiently, despite the fact that currently the best bound for their computation is $O(n^2)$.

Chapter 2

Computing 2-Edge-Connected Blocks

2.1 Introduction and properties

First, recall the following definitions from Section 1.2.1. We say that two vertices v and w are 2-edge-connected, and we denote this relation by $v \leftrightarrow_{2e} w$, if there are two edge-disjoint directed paths from v to w and two edge-disjoint directed paths from w to v. (A path from v to w and a path from w to v need not be edge-disjoint.) It is easy to see that $v \leftrightarrow_{2e} w$ if and only if the removal of any edge leaves v and w in the same strongly connected component. We define a 2-*edge-connected block* of a digraph $G = (V, E)$ as a maximal subset $B \subseteq V$ such that $u \leftrightarrow_{2e} v$ for all $u, v \in B$.

Theorem 2.1. *The 2-edge-connected blocks of a digraph $G = (V, E)$ form a partition of V.*

Proof. We show that $\leftrightarrow_{2e}$ is an equivalence relation. The relation is by definition reflexive and symmetric, so it remains to show that it is also transitive when G has at least three vertices. Let u, v, and w be three distinct vertices such that $u \leftrightarrow_{2e} v$ and $v \leftrightarrow_{2e} w$. Consider any u-w cut (U, W), where $u \in U$ and $w \in W$. Let k' be the number of edges

directed from U to W. We will show that $k' \geq 2$. If $v \in U$, then $v \leftrightarrow_{2e} w$ implies that $k' \geq 2$. Otherwise, $v \in W$, and $u \leftrightarrow_{2e} v$ implies that $k' \geq 2$. A completely analogous argument applies to the edges directed from W to U. The fact that $u \leftrightarrow_{2e} w$ now follows from Menger's Theorem [28]. ∎

Throughout, we use the notation $[v]_{2e}$ to denote the 2-edge-connected block containing vertex $v \in V$. We can generalize the 2-edge-connected relation for $k \geq 2$ edge-disjoint paths: the proof of Theorem 2.1 can be extended to show that this relation also defines a partition of V into k-edge-connected blocks. By Theorem 2.1, once the 2-edge-connected blocks are available, it is easy to test in constant time if two vertices are 2-edge-connected.

Next we develop algorithms that compute the 2-edge-connected blocks of a digraph G. Clearly, we can assume that G is strongly connected, so $m \geq n$. If not, then we process each strongly connected component separately; if $u \leftrightarrow_{2e} v$ then u and v are in the same strongly connected component S of G, and moreover, any vertex on a path from u to v or from v to u also belongs in S. We begin with a simple algorithm that removes a single strong bridge at a time. In order to get a more efficient solution, we need to consider simultaneously how different strong bridges partition the vertex set. We present a recursive algorithm that does this with the help of dominator trees. Although both these algorithms run in $O(mn)$ time in the worst case, we finally show that a careful combination of them is able to achieve linear time.

2.2 A simple algorithm

Let u and v be two distinct vertices in G. We say that a strong bridge e *separates* u *from* v if all paths from u to v contain edge e. In this case, u and v must belong to different strongly connected components of $G \setminus e$. This simple observation gives a characterization of the 2-edge-connected blocks in terms of the strong bridges. In particular, one can obtain the 2-edge-connected blocks of G by simply computing the strongly connected components of $G \setminus e$ for every strong bridge e, as illustrated by Algorithm Simple2ECB in Figure 2.1.

Lemma 2.1. *Algorithm Simple2ECB runs in $O(mb^*)$ time, where b^* is the number of strong bridges of G.*

Proof. The strong bridges of G can be computed in linear time by [22]. In each iteration of Step 3, we can compute the strongly connected components of $G \setminus e$ in linear time [33]. As we discover the i-th strongly connected component, we assign label i ($i \in \{1, \ldots, n\}$) to the vertices in S_i. Then, the refinement of the current blocks in Step 3.1 can be done in $O(n)$ time with bucket sorting. So each iteration takes $O(m)$ time. ∎

Note that the above bound is $O(mn)$ in the worst case, since for any digraph $b^* \leq 2n-2$ by Lemma 1.1. We remark that deleting all strong bridges (at once) will not produce a correct result, as it can be easily seen from Figures 1.2(b) and 1.2(c). Despite the fact

Algorithm Simple2ECB: Computation of the 2-edge-connected blocks of a strongly connected digraph $G = (V, E)$

Step 1: Initialize the 2-edge-connected blocks as $[v]_{2e} = V$. (Start from the trivial partition containing only one block.)

Step 2: Compute the strong bridges of G.

Step 3: For each strong bridge e do:

Step 3.1: Compute the strongly connected components $S_1, \ldots, S_k$ of $G \backslash e$.

Step 3.2: Let $\{[v_1]_{2e}, \ldots, [v_l]_{2e}\}$ be the current 2-edge-connected blocks. Refine the partition into blocks by computing the intersections $[v_i]_{2e} \cap S_j$ for all $i = 1, \ldots, l$ and $j = 1, \ldots, k$.

Figure 2.1: Algorithm Simple2ECB

that removing a single strong bridge at a time does not yield an efficient algorithm, we will make use of this idea, in a more restricted way, in the linear-time algorithm described in Section 2.4.

2.3 A recursive algorithm

In order to obtain a faster algorithm we need to determine how multiple strong bridges affect the partition of the vertices into blocks. We achieve this by selecting an arbitrary start vertex s and by using the dominator tree $D(s)$ of the flow graph $G(s)$. We do this as follows. First we consider the computation of the 2-edge-connected block that contains a specific vertex v. Let w be a vertex other than v. We say that w is 2-*edge-connected from* v if there are two edge disjoint paths from v to w. Analogously, we say that w is 2-*edge-connected to* v if there are two edge disjoint paths from w to v. We divide the computation of $[v]_{2e}$ in two parts, where the first part finds the set of vertices $[v]_{\overrightarrow{2e}}$ that are 2-edge-connected from v, and the second part finds the set $[v]_{\overleftarrow{2e}}$ of vertices that are 2-edge-connected to v. Then $[v]_{2e}$ is formed by the intersection of these two sets.

Consider the computation of $[v]_{\overrightarrow{2e}}$. An efficient way to compute this set is based on the dominators and bridges of the flow graph $G(v)$. In particular, we compute the dominator tree $D(v)$ and identify the bridges of $G(v)$. Then for each bridge $e = (u, w)$ we have $d(w) = u$, i.e., each bridge is also an edge in the dominator tree; we mark w in $D(v)$.

Lemma 2.2. *$w \in [v]_{\overrightarrow{2e}}$ if and only if w is not dominated in $G(v)$ by a marked vertex.*

Proof. We have that $w \notin [v]_{\overrightarrow{2e}}$ if and only if there is an edge (strong bridge) that separates

v from w in G. Then $e = (x, y)$ is such an edge if and only if it is a bridge in $G(v)$, so y is a marked ancestor of w in $D(v)$. ∎

Lemma 2.2 implies a straightforward linear-time algorithm to compute $[v]_{\overrightarrow{2e}}$, given the dominator tree $D(v)$ of $G(v)$. We use the same algorithm to compute $[v]_{\overleftarrow{2e}}$, but operate on the reverse graph $G^R(v)$ and its dominator tree $D^R(v)$. That is, we identify the bridges of the flow graph $G^R(v)$, and for each bridge $e = (u, w)$ we mark w in $D^R(v)$. Note that a vertex w that is marked in $D(v)$ may not be marked in $D^R(v)$ and vice versa.

Corollary 2.1. *$w \in [v]_{2e}$ if and only if w is not dominated in $G(v)$ and in $G^R(v)$ by any marked vertex. Moreover, $[v]_{2e}$ can be computed in $O(m)$ time.*

Note that all the 2-edge-connected blocks $[v]_{2e}$ can be computed in $O(mn)$ time by applying Corollary 2.1 to all vertices v. We describe next a more complicated algorithm, which avoids repeated applications of Corollary 2.1. This algorithm will still require $O(mn)$ time but it will be a useful ingredient for our linear-time algorithm.

Let s be an arbitrarily chosen start vertex. We first observe that the bridges in the dominator trees $D(s)$ and $D^R(s)$ of $G(s)$ and $G^R(s)$, respectively, partition the vertices into sets that contain the 2-edge-connected blocks. More precisely, identify the bridges of $G(s)$ (resp., $G^R(s)$), and for each bridge $e = (u, w)$ mark w in $D(s)$ (resp. $D^R(s)$) as above. Now delete all bridges from $D(s)$ and $D^R(s)$: namely, remove from $D(s)$ all edges $(d(v), v)$ such that v is marked in $D(s)$, and remove from $D^R(s)$ all edges $(d^R(v), v)$ such that v is marked in $D^R(s)$. This decomposes the dominator trees $D(s)$ and $D^R(s)$ into forests of rooted trees, where each tree is rooted either at a marked vertex or at the start vertex s. In the following, we refer to this as the *canonical decomposition* of the dominator tree $D(s)$, and use the notation $T(v)$ to denote the tree containing vertex v in this decomposition. Note that $T(v)$ is a subtree of $D(s)$ and its root r_v is either a marked vertex or the start vertex s. Similarly, we denote by $T^R(v)$ the tree containing vertex v in the canonical decomposition of the dominator tree $D^R(s)$. Figure 2.2 shows an example of a flow graph $G(s)$, its dominator tree $D(s)$ and the canonical decomposion of $D(s)$ into the subtrees $T(v)$ induced by the removal of all bridges of the flow graph $G(s)$.

In the following lemmas, we assume that s is an arbitrarily chosen start vertex in G, $G(s)$ is the flow graph with start vertex s, $G^R(s)$ is the flow graph obtained from $G(s)$ after reversing edge directions, $D(s)$ and $D^R(s)$ are the dominator trees of $G(s)$ and $G^R(s)$ respectively, and $T(v)$ and $T^R(v)$ are the subtrees containing vertex v in the canonical decompositions of $D(s)$ and $D^R(s)$ respectively (i.e., induced by the removal of all bridges in $D(s)$ and $D^R(s)$).

Lemma 2.3. *Let v and w be two different vertices in G. Then $[v]_{2e} = [w]_{2e}$ only if $T(v) = T(w)$ and $T^R(v) = T^R(w)$.*

Proof. We show that $[v]_{2e} = [w]_{2e}$ implies $T(v) = T(w)$. Then the same argument applied on $G^R(s)$ shows that $T^R(v) = T^R(w)$. Suppose by contradiction that $[v]_{2e} = [w]_{2e}$ but $T(v) \neq T(w)$, i.e., $w \notin T(v)$. Assume without loss of generality that r_v is not an ancestor

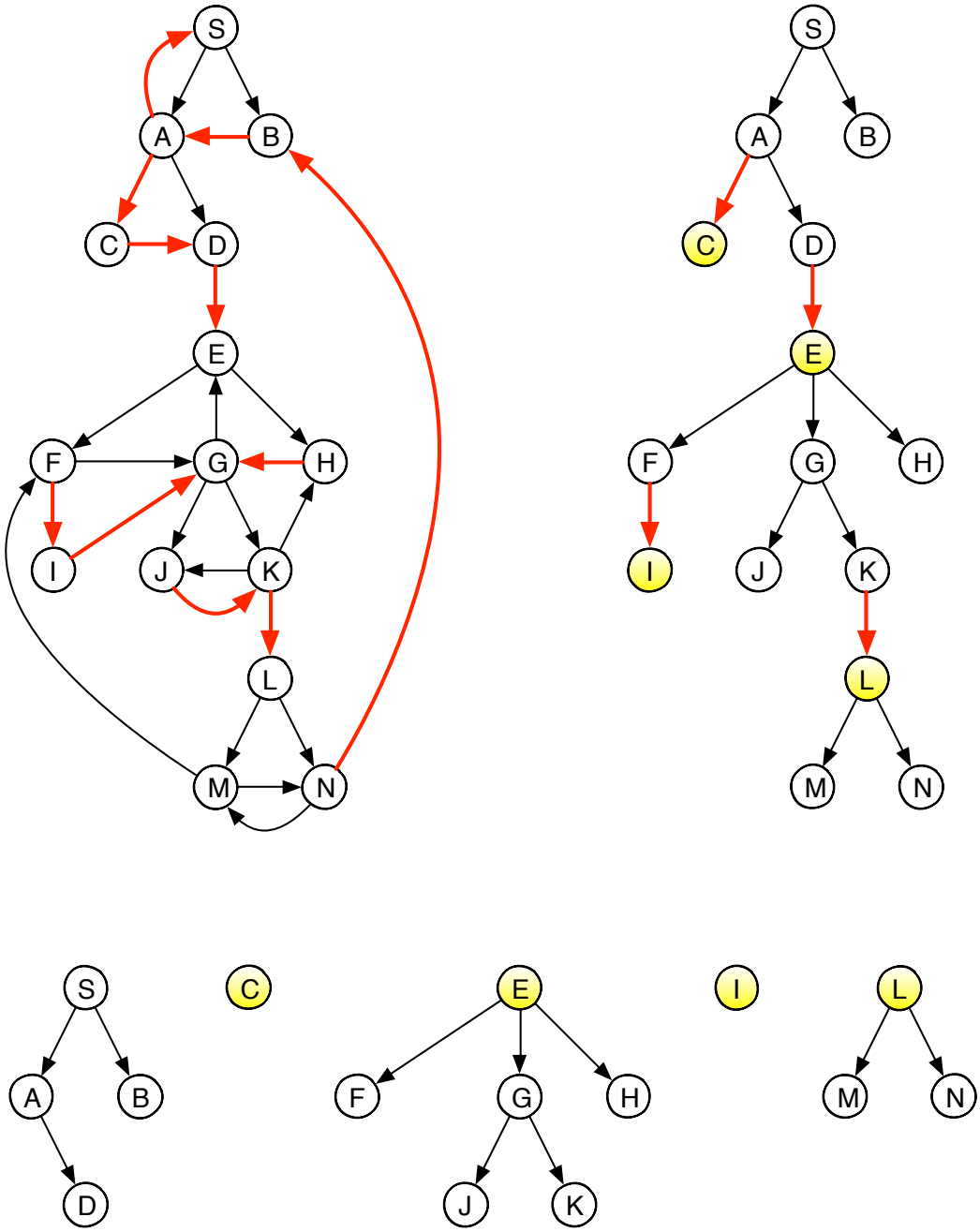


Figure 2.2: A flow graph $G(s)$, its dominator tree $D(s)$ and its canonical decomposition into the subtrees $T(v)$ induced by the bridges of $G(s)$. Strong bridges of the original graph G and bridges of the flow graph $G(s)$ are shown in red; marked vertices are shown in yellow. (Better viewed in color.)

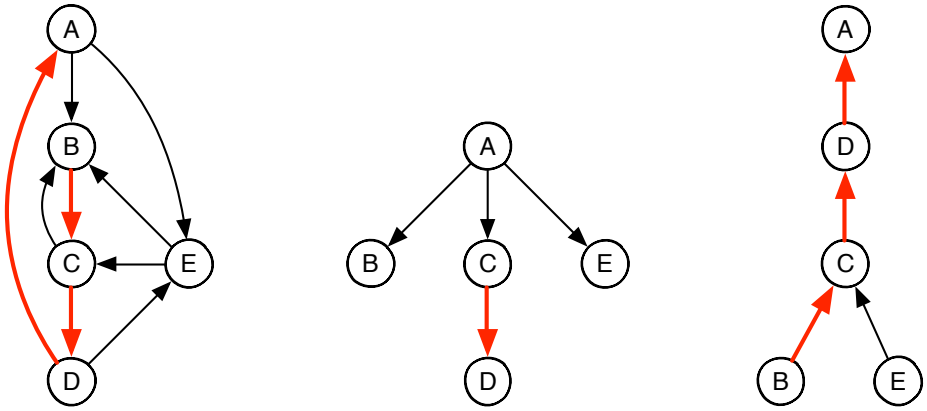


Figure 2.3: A strongly connected digraph G and its dominator trees $D(A)$ and $D^R(A)$ rooted at vertex A. (The edges of the dominator tree $D^R(A)$ are shown directed from child to parent.) Strong bridges are shown in red (better viewed in color). Vertices C and E lie in the same subtree in both $D(A)$ and $D^R(A)$ but they are not 2-edge-connected, as they are separated by the strong bridge (C, D).

of r_w in $D(s)$ (if not, swap v and w). Note that the edge $e = (d(r_v), r_v)$ must be a bridge in $G(s)$. Since $[v]_{2e} = [w]_{2e}$, then there must be a path P in G from w to v that avoids edge e. Since r_v is not an ancestor of r_w in $D(s)$, there is a path Q in G from s to w that avoids e. If $v \in Q$ then the part of Q from s to v avoids e, which contradicts the fact that e is a bridge in $G(s)$, i.e., it induces a cut that separates s from v in G. If $v \notin Q$, then Q followed by P ($Q \cdot P$) gives a path from s to v in G that avoids the bridge e, again a contradiction. ∎

Note that Lemma 2.3 provides a necessary condition for two vertices to be 2-edge-connected. This is not a sufficient condition, however, as two vertices may be separated by a strong bridge and still lie in the same subtree in both the canonical decompositions of $D(s)$ and of $D^R(s)$ (see Figure 2.3). The main challenge in this approach is thus to discover which vertices in the same subtree are separated by a strong bridge. To tackle this challenge, we provide some key observations regarding edges and paths that connect different subtrees $T(r)$. We will use the *parent property* of dominator trees [16], that we state next.

Lemma 2.4. (Parent property of the dominator tree [16].) *For all $(v, w) \in E$, $d(w)$ is an ancestor of v in $D(s)$.*

Now we prove some structural properties for paths that connect vertices in different subtrees.

Lemma 2.5. *Let $e = (u, v)$ be an edge of G such that $T(u) \neq T(v)$ and let r_v be the root of $T(v)$. Then either $u = d(v)$ and e is a bridge in the flow graph $G(s)$, or u is a proper descendant of r_v in $D(s)$.*

Proof. If e is a bridge in $G(s)$ then $u = d(v)$ and the lemma holds. Suppose that e is not a bridge, so $u \neq d(v)$. If v is an ancestor of u in $D(s)$ then the lemma holds. If not, then by Lemma 2.4, $d(v)$ is a proper ancestor of u in $D(s)$. We show that $d(v) \in T(v)$, which implies the lemma. Assume by contradiction that $d(v) \notin T(v)$. Then $(d(v), v)$ is a bridge and $v = r_v$. Since v is not an ancestor of u in $D(s)$, there is a path P from s to u that does not contain v. Then $P \cdot e$ is a path from s to v that avoids the bridge $(d(v), v)$, a contradiction. ∎

Lemma 2.6. *Let r be a marked vertex in $D(s)$. Let v be any vertex that is not a descendant of r in $D(s)$. Then there is path from v to r that does not contain any vertex in $T(r) \setminus r$. Moreover, all simple paths from v to any vertex in $T(r)$ contain the edge $(d(r), r)$.*

Proof. Since v is not a descendant of r in $D(s)$, $v \notin T(r)$. Graph G is strongly connected, so it contains a path from v to r. Let P be any such path. Let $e = (u, w)$ be the first edge on P such that $w \in T(r)$. Then by Lemma 2.5, either $e = (d(r), r)$ or u is a proper descendant of r. In the first case the lemma holds. Suppose u is a proper descendant of r. Since v is not a descendant of r in $D(s)$, there is a path Q from s to v in G that does not contain r. Then Q followed by the part of P from v to w gives a path from s to $w \in T(r)$ that avoids $d(r)$, a contradiction. ∎

Auxiliary graphs. We now introduce the notion of *auxiliary graphs* that plays a crucial role in our algorithm. Auxiliary graphs represent a decomposition of the input digraph G into smaller digraphs (not necessarily subgraphs of G) that maintain the original 2-edge-connected blocks of G. Let r be either a non-leaf marked vertex or the start vertex s in the dominator tree $D(s)$, and let $T(r)$ be the subtree with root r in the canonical decomposition of the dominator tree $D(s)$. For each such subtree $T(r)$, we define the *auxiliary graph $G_r = (V_r, E_r)$ of r* as follows. The vertex set V_r of G_r consists of all the vertices in $T(r)$, referred to as *ordinary* vertices, and a set of *auxiliary* vertices, which are obtained by contracting vertices in $V \setminus T(r)$, as follows. Let v be a vertex in $T(r)$. We say that v is a *boundary vertex in $T(r)$* if v has a marked child in $D(s)$. Let w be a marked child of a boundary vertex v: all the vertices that are descendants of w in $D(s)$ are contracted into w. All vertices in $V \setminus T(r)$ that are not descendants of r are contracted into $d(r)$ ($r \neq s$ if any such vertex exists). During those contractions, parallel edges are eliminated. This is necessary in order to obtain the size bound given in Lemma 2.7. Figure 2.4 shows a flow graph, its dominator tree and an auxiliary graph.

Lemma 2.7. *If $G(s)$ has b bridges then all the auxiliary graphs G_r have at most $n + 2b$ vertices and $m + 2b$ edges in total.*

Proof. Every vertex appears as an ordinary vertex only in one auxiliary graph. A marked vertex in $D(s)$ corresponds to a bridge in $G(s)$, so there are $b \leq n - 1$ marked vertices. Since we have one auxiliary graph for each marked vertex, the total number of the auxiliary vertices $d(r)$ is b. Each marked vertex v can also appear in at most one other auxiliary graph as a child of a boundary vertex. So the total number of vertices is at most $n + 2b$.

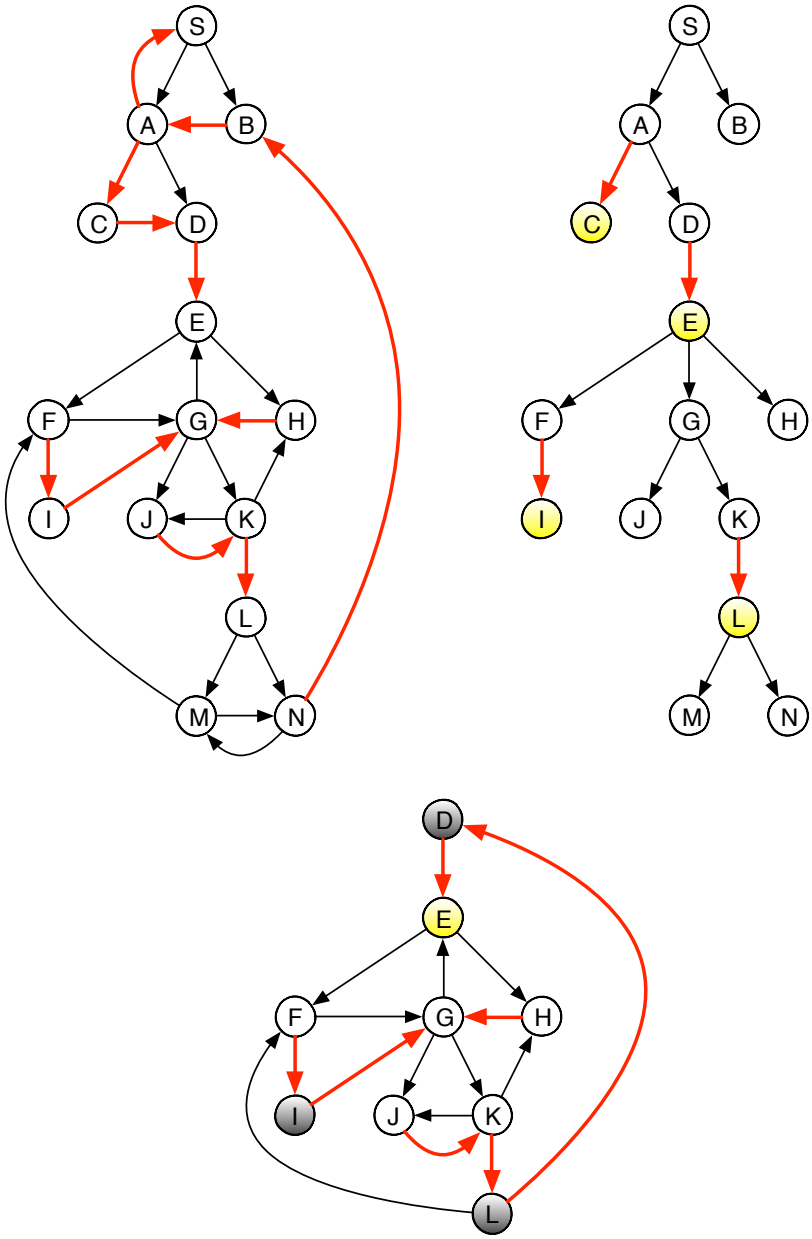


Figure 2.4: The flow graph $G(S)$ and its dominator tree $D(S)$ from Figure 2.2, together with the auxiliary graph of vertex E. Strong bridges are red, marked vertices are yellow, and auxiliary vertices are gray. (Better viewed in color.) Edge (L, D) is a shortcut edge of type (c) that corresponds to a path in G from L to D, e.g., L, N, B, A, D. Edges (L, F) and (I, G) are shortcut edges of type (b).

Next we bound the total number of edges. The total number of edges between ordinary vertices in all auxiliary graphs is at most $m - b$. Each bridge can appear in at most two auxiliary graphs. Finally, the number of edges connecting auxiliary vertices is at most b, since each such edge corresponds to a unique occurrence of a marked vertex as an auxiliary vertex. So we have at most $m + 2b$ edges in total. ∎

Lemma 2.8. *Let v and w be two vertices in a subtree $T(r)$. Any path P from v to w in G has a corresponding path P_r from v to w in the auxiliary graph G_r, and vice versa. Moreover, P_r contains a strong bridge if and only if P does.*

Proof. The first part of the lemma follows immediately from the definition of auxiliary graph G_r. For the second part, let P be a path from v to w in G and let P_r be its corresponding path in G_r. Let $e = (x, y)$ be a strong bridge on P. We consider the following cases:

- $x \in T(r)$ and $y \in T(r)$. Then (x, y) is also an edge on P_r. Moreover, by the definition of G_r, the edge (x, y) is a strong bridge in G_r.
- $x \in T(r)$ and $y \notin T(r)$. By Lemma 2.5, either $x = d(y)$ or x is a proper descendant of r_y in the dominator tree $D(s)$. In the former case, $e = (d(y), y)$ is a strong bridge in G_r that is contained in all paths from v to w. In the latter case, Lemma 2.6 implies that all paths from v to w in G contain the strong bridge $(d(r), r)$. By the construction of the auxiliary graphs, this is also true for all paths from v to w in G_r.
- $x \notin T(r)$ and x a descendant of r in $D(s)$. Then v is not an ancestor of w in $D(s)$, since otherwise, by Lemma 2.4, there would be a path from v to w that avoids e. Let $t \in T(r)$ be the boundary vertex that is an ancestor of x, and let z be the child of t that is an ancestor of x. By Lemma 2.6, all paths from v to x in G, and thus all paths from v to w, contain the strong bridge (t, z). By the construction of the auxiliary graphs this is also true for all paths from v to w in G_r.
- $x \notin T(r)$ and x not a descendant of r in $D(s)$. In this case, Lemma 2.6 implies that all paths from x to w in G contain the strong bridge $(d(r), r)$. Hence, all paths from v to w in G contain the strong bridge $(d(r), r)$, and so do all paths from v to w in G_r by the construction of the auxiliary graphs.

Thus, in every case we have that P_r contains a strong bridge, and so the lemma follows. ∎

Corollary 2.2. *Each auxiliary graph G_r is strongly connected.*

Proof. It follows immediately from Lemma 2.8 and the fact that G is strongly connected. ∎

Now we are ready to show that we can compute the 2-edge-connected blocks in each auxiliary graph independently of each other.

Lemma 2.9. *Let v and w be any two distinct vertices of G. Then v and w are 2-edge-connected in G if and only if they are both ordinary vertices in an auxiliary graph G_r and they are 2-edge-connected in G_r.*

Proof. By Lemma 2.3, v and w are 2-edge-connected in G only if they belong to the same subtree $T(r)$, in which case they are both ordinary vertices of G_r. If v and w are 2-edge-connected in G then Lemma 2.8 implies that they are also 2-edge-connected in G_r. Now suppose that there is a strong bridge that separates v from w in G. The case analysis in the proof of Lemma 2.8 shows that all paths from v to w in G_r also have a strong bridge in common. The same argument applies if there is a strong bridge that separates w from v in G, and the lemma follows. ∎

We next show how to construct the auxiliary graphs $G_r = (V_r, E_r)$ efficiently. The vertex set V_r contains the set V_r^o of *ordinary* vertices (i.e., the vertices of $T(r)$), and the set V_r^a of *auxiliary* vertices. The edge set E_r contains all edges in $G = (V, E)$ induced by the ordinary vertices (i.e., edges $(u, v) \in E$ such that $u \in T(r)$ and $v \in T(r)$), together with some edges that have at most one endpoint in $T(r)$ and are either bridges of $G(s)$ or *shortcut* edges that correspond to paths in G. We define shortcut edges as follows. Let v be a *boundary vertex in* $T(r)$ (i.e., v has a marked child in $D(s)$). For each marked child w of v in $D(s)$ we add a copy of w in V_r^a, and add the edge (v, w) in E_r. Also, if r is marked ($r \neq s$) then we add a copy of $d(r)$ in V_r^a, and add the edge $(d(r), r)$ in E_r. We also add in E_r the following shortcut edges for edges (u, v) of the following type: (a) If u is ordinary and v is not a descendant of r, then we add the shortcut edge $(u, d(r))$. (b) If v is ordinary and u is a proper descendant in $D(s)$ of a boundary vertex w, then we add the shortcut edge (z, v) where z is the child of w that is an ancestor of u in $D(s)$. (c) Finally, if u is a proper descendant in $D(s)$ of a boundary vertex w and v is not a descendant of r, then we add the shortcut edge $(z, d(r))$, where z is the child of w that is an ancestor of u in $D(s)$. We note that, according to the definition, in this construction we do not keep multiple (parallel) shortcut edges (see Figure 2.4).

To complete our construction of the auxiliary graphs, we need to specify how to compute the shortcut edges of each type (a), (b), and (c). Suppose (u, v) is an edge of type (a). Then v is not a descendant of r in $D(s)$, which can be tested using an $O(1)$-time test of the ancestor-descendant relation. There are several simple $O(1)$-time tests of this relation [35]. The most convenient one for us is to number the vertices of $D(s)$ from 1 to n in preorder, and to compute the number of descendants of each vertex v, which we denote by $size(v)$. Then v is a descendant of r if and only if $pre(r) \leq pre(v) < pre(r) + size(r)$. Next suppose that (u, v) is of type (b). Then u is a proper descendant of a boundary vertex w in $D(s)$. To compute the shortcut edge of (u, v) we need to find the child z of w that is an ancestor of u in $D(s)$. To that end, we create a list A_r that contains the edges (u, v) of type (b) such that $v \in T(r)$, and sort A_r in increasing preorder of u. We create a second list A'_r that contains the children in $D(s)$ of the boundary vertices in $T(r)$, and sort A'_r in increasing preorder. Then, the shortcut edge of (u, v) is (z, v), where z is the last vertex in the sorted list A'_r such that $pre(z) \leq pre(u)$. Thus the shortcut edges of

type (b) can be computed in linear time by bucket sorting and merging. In order to do this fast for all auxiliary graphs, we sort all the lists at the same time as follows. First, we create a unified list A containing the triples $(r, pre(u), v)$, for each type (b) edge (u, v) in the auxiliary graph G_r. Next we sort A in increasing order of the two first elements. We also create a second list A' with pairs $(r, pre(w))$, where w is a proper descendant of a boundary vertex in $T(r)$, and sort the pairs in increasing order. Finally, we compute the shortcut edges of each auxiliary graph G_r by merging the sorted sublists of A and A' that correspond to the same root r. Then, the shortcut edge for the triple $(r, pre(u), v)$ is (z, v), where $(r, pre(z))$ is the last pair in the sorted sublist of A' with root r such that $pre(z) \leq pre(u)$.

Finally, consider the edges of type (c). For each such edge (u, v) we need to add the edge $(z, d(r))$ in each G_r, where u is a proper descendant of a boundary vertex $w \in T(r)$, v is not a descendant of r in $D(s)$, and z is the child of w that is an ancestor of u in $D(s)$. We compute these edges for all auxiliary graphs G_r as follows. First, we create a compressed tree $\widehat{D}(s)$ that contains only s and the marked vertices. A marked vertex v becomes child of its nearest marked ancestor u, or of s if u does not exist. This can be easily done in $O(n)$ time during the preorder traversal of $D(s)$. Next we process all edges (u, v) such that v is not a descendant of r_u in $D(s)$. At each node $w \neq s$ in $\widehat{D}(s)$ we store a label $\ell(w)$ which is the minimum $pre(r_v)$ of an edge (u, v) of type (c) such that $u \in T(w)$; we let $\ell(w) = pre(w)$ if no such edge exists. Using these labels we compute for each $w \neq s$ in $\widehat{D}(s)$ the values $low(w) = \min\{\ell(v) \mid v \text{ is a descendant of } w \text{ in } \widehat{D}(s)\}$. These computations can be done in $O(m)$ time by processing the tree $\widehat{D}(s)$ in a bottom-up order. Now consider the auxiliary graph G_r. We process the children in $D(s)$ of the boundary vertices in $T(r)$. Note that these children are marked, so they have a *low* value. For each such child z we test if G_r has a shortcut edge $(z, d(r))$: If $low(z) < pre(r)$ then we add the edge $(z, d(r))$. This leads to the following lemma.

Lemma 2.10. *We can compute all auxiliary graphs G_r in $O(m)$ time.*

We also describe an alternative way to compute the type (b) shortcut edges of the auxiliary graphs, by replacing sorting with vertex contractions. To that end, we use a *disjoint set union data structure* [36], which maintains a collection of disjoint sets, each with a representative element, under three operations:

make-set(x)**:** Create a new set $\{x\}$ with representative x. Element x must be in no existing set.

find(x)**:** Return the representative element of the set containing element x.

unite(x, y)**:** Unite the sets containing elements x and y and give the new set the representative of the old set containing x.

Using this data structure, we can compute the type (b) shortcut edges of all auxiliary graphs in a single bottom-up pass of $D(s)$. To initialize the sets, we perform *make-set*(v)

Algorithm Rec2ECB: Recursive computation of the 2-edge-connected blocks for the ordinary vertices of a strongly connected digraph $H = (V, E)$

Step 1: Choose an arbitrary ordinary vertex $s \in V^o$ as a start vertex. Compute the dominator trees $D(s)$ and $D^R(s)$ and the bridges of the flow graphs $H(s)$ and $H^R(s)$.

Step 2: Compute the number b of bridges (x, y) in $H(s)$ such that y is an ancestor of an ordinary vertex in $D(s)$. Compute the number b^R of bridges (x, y) in $H^R(s)$ such that y is an ancestor of an ordinary vertex in $D^R(s)$.

Step 3: If $b = b^R = 0$ then return $[s]_{2e} = V^o$.

Step 4: If $b^R > b$ then swap H and H^R. Find the canonical decomposition of $D(s)$ into the subtrees $T(r)$ and compute the corresponding auxiliary graphs H_r. Compute recursively the 2-edge-connected blocks for each auxiliary graph H_r with at least two ordinary vertices.

Figure 2.5: Algorithm Rec2ECB

for every vertex $v \in V$. The unite operations are also executed in a bottom-up order, and each such operation has the effect of contracting a vertex v to its ancestor u in $D(s)$, such that $v \in T(u)$, or v is a child of a boundary vertex in $T(u)$. The details of these computations are as follows. We process all marked vertices r in a bottom-up order of $D(s)$, and compute the type (b) shortcut edges of the corresponding auxiliary graph $G_r = (V_r, E_r)$. To do this, we process the edges entering each vertex $v \in T(r)$. For each such edge (u, v) we compute $z = \mathit{find}(u)$. If $z \neq v$ and $z \in T(r)$ then (z, v) is a type (b) shortcut edge of G_r. The former condition means that u is not a descendant of v, while the second condition can only be violated for $v = r$. After we have processed all edges entering $T(r)$, we execute $\mathit{unite}(r, \mathit{find}(v))$ for each vertex $v \in T(r)$ and each vertex $v \in V_r^a$ that is a child in $D(s)$ of a boundary vertex in $T(r)$. This construction runs $O(m)$ time plus the time for $n-1$ *unite* operations and, by Lemma 3.13, at most $4m$ *find* operations. If *unite* and *find* are implemented using compressed trees with balanced linking [36], the total time for these disjoint set union operations is $O(m\alpha(m, n))$. Since the set of *unite* operations is known in advance we have an instance of the static tree disjoint set union problem, which is solvable in $O(m)$ time on a RAM [10].

Lemma 2.9 allows us to compute the 2-edge-connected blocks of each auxiliary graph separately. Algorithm Rec2ECB, described in Figure 2.5, applies this idea recursively, until all ordinary vertices in each graph H are 2-edge-connected. Since, by Lemma 2.3,

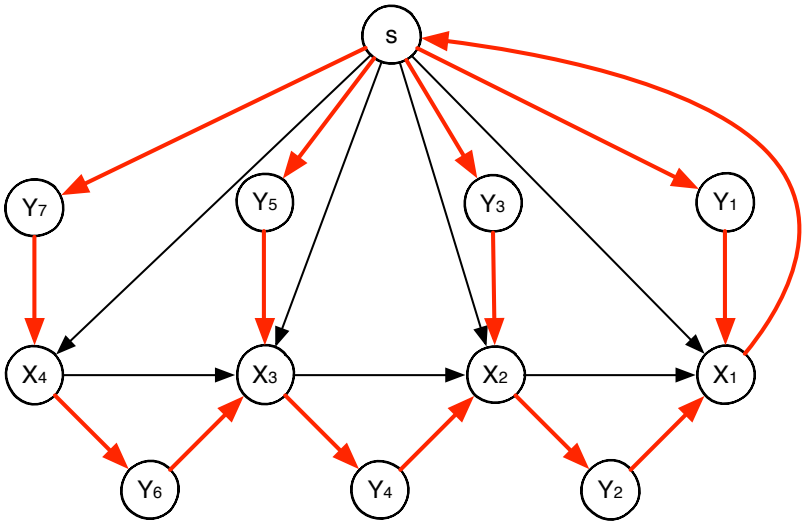


Figure 2.6: An input digraph with $n = \Theta(k)$ vertices that causes k recursive calls of Algorithm Rec2ECB (see Figures 2.7 and 2.8). Vertices X_1, X_2, ..., X_k are not 2-edge-connected but Algorithm Rec2ECB requires k recursive calls to separate them into different blocks. (In this figure $k = 4$.) Each new partition is induced by the strong bridge (X_1, S).

an auxiliary vertex and an ordinary vertex of H are not 2-edge-connected, we only need to consider the strong bridges that separate ordinary vertices of H. In order to find if H contains such strong bridges, we choose an arbitrary ordinary vertex s of H as a start vertex and compute the bridges and the dominator trees $D(s)$ and $D^R(s)$ of the flow graphs $H(s)$ and $H^R(s)$, respectively. Then Corollary 2.1 implies that H contains a strong bridge that separates two ordinary vertices if and only if $H(s)$ or $H^R(s)$ contains a bridge (x, y) such that y is an ancestor of an ordinary vertex. Otherwise, all ordinary vertices are 2-vertex-connected to and from s, so $[s]_{2e} = V^o$. We remark that in the first call of Algorithm Rec2ECB, the input graph is G and all the vertices are considered ordinary, i.e., $H = G$ and $V^o = V$.

Lemma 2.11. *Algorithm Rec2ECB runs in $O(mn)$ time.*

Proof. Each recursive call refines the current partition of V, thus we have at most $n - 1$ recursive calls. By [3, 34] and Lemma 2.10, the total work per recursive call is $O(m)$. ∎

We note that the bound stated in Lemma 2.11 is tight. The same strong bridge can be used repeatedly to separate different pairs of vertices in successive recursive calls (see Figures 2.6, 2.7 and 2.8). Despite the fact that Algorithm Rec2ECB only achieves an $O(mn)$ time bound, it will be the basis of our linear-time algorithm that we develop in the next section.

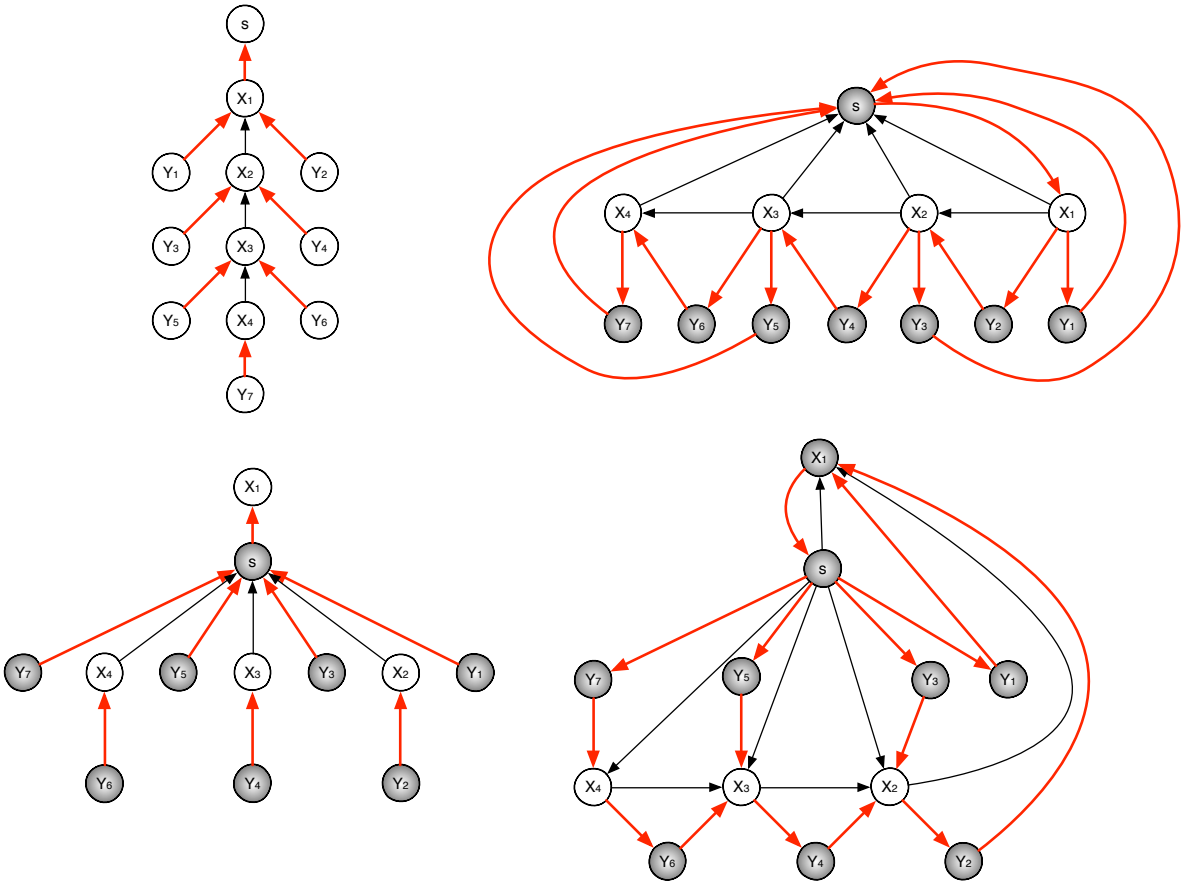


Figure 2.7: The first two recursive calls of Algorithm Rec2ECB, with input the digraph of Figure 2.6. In the left column we see the dominator tree used to compute the next partition, whilst in the right column there is the auxiliary graph containing the majority of ordinary vertices, that will be the input digraph in the next recursive call.

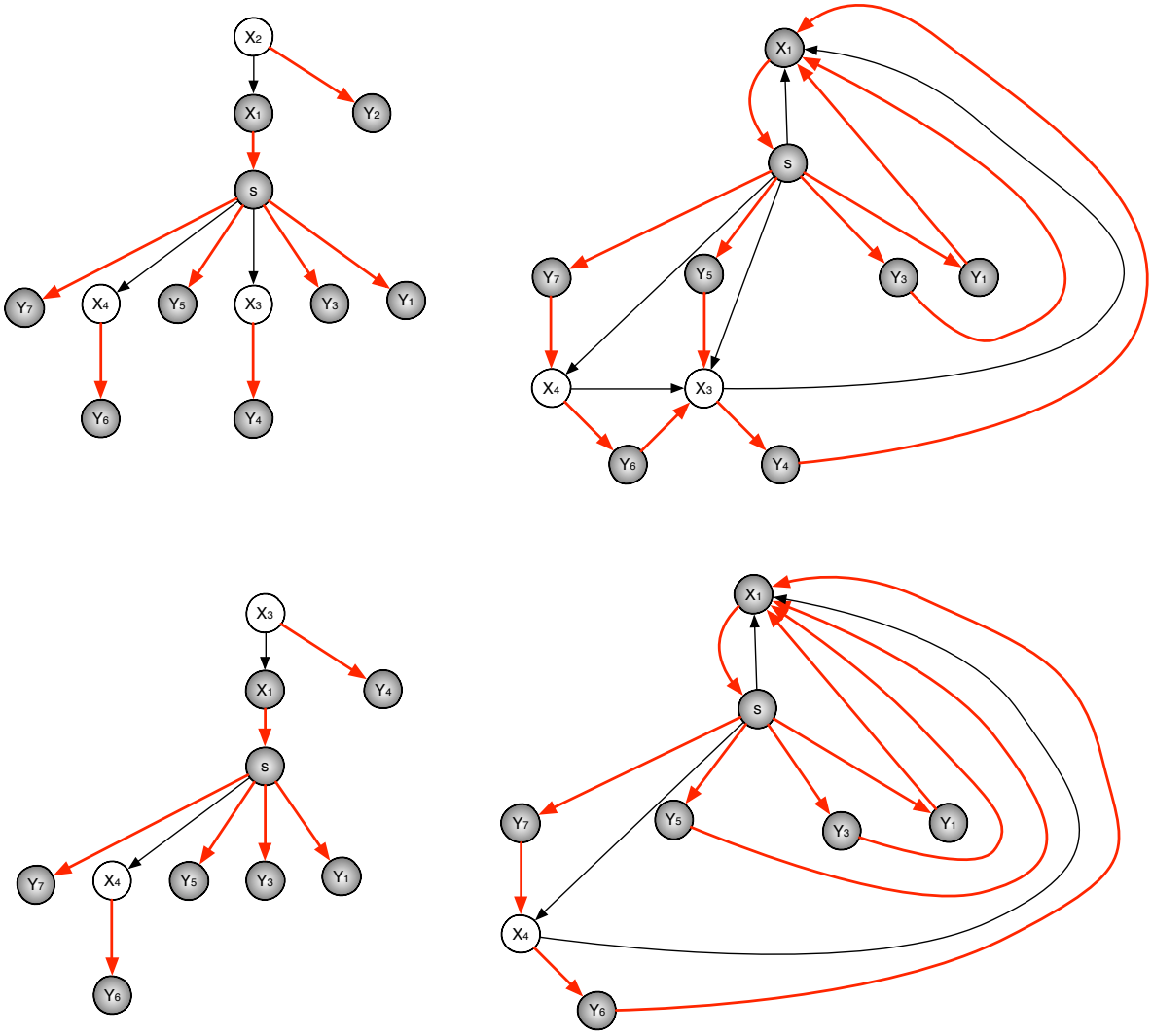


Figure 2.8: The other two recursive calls of Algorithm Rec2ECB, with input the digraph of Figure 2.6 (the first two calls are shown in Figure 2.7). In the left column we see the dominator tree used to compute the next partition, whilst in the right column there is the auxiliary graph containing the majority of ordinary vertices, that will be the input digraph in the next recursive call.

Algorithm Fast2ECB: Linear-time computation of the 2-edge-connected blocks of a strongly connected digraph $G = (V, E)$

Step 1: Choose an arbitrary vertex $s \in V$ as a start vertex. Compute the dominator tree $D(s)$ and the bridges of the flow graph $G(s)$.

Step 2: Partition $D(s)$ into subtrees $T(r)$ and compute the corresponding auxiliary graphs G_r.

Step 3: For each auxiliary graph $H = G_r$ do:

- **Step 3.1:** Compute the dominator tree $D_H^R(r)$ and the bridges of $H^R(r)$. Let $d_H^R(q)$ be the parent of $q \neq r$ in $D_H^R(r)$.
- **Step 3.2:** Partition $D_H^R(r)$ into the subtrees $T_H^R(q)$. Compute the corresponding auxiliary graphs H_q^R with $q \neq r$.
- **Step 3.3:** Set $[r]_{2e}$ to consist of the ordinary vertices in $T_H^R(r)$.
- **Step 3.4:** For each auxiliary graph H_q^R with $q \neq r$ do:
  - **Step 3.4.1:** Compute the strongly connected components $S_1, S_2, \ldots, S_k$ of $H_q^R \setminus (d_H^R(q), q)$.
  - **Step 3.4.2:** Partition the ordinary vertices of H_q into blocks according to each S_j, $j = 1, \ldots, k$; For each ordinary vertex v, $[v]_{2e}$ contains the ordinary vertices in the strongly connected component of v.

Figure 2.9: Algorithm Fast2ECB

2.4 A linear-time algorithm

Although Algorithms Simple2ECB and Rec2ECB run in $O(mn)$ time, we show that a careful combination of them gives a linear-time algorithm. The critical observation, proved in Lemma 2.12 below, is that if a strong bridge separates different pairs of vertices in successive recursive calls (which causes the worst-case behavior of Algorithm Rec2ECB, as shown in Figures 2.6, 2.7 and 2.8), then it will appear as the strong bridge entering the root of a subtree in the canonical decomposition of a dominator tree.

Algorithm Fast2ECB, described in Figure 2.9, applies the observation above together with all the tools we developed in the previous sections, and achieves the computation of the 2-edge-connected blocks in linear time. In essence, it runs Algorithm Rec2ECB but stops the recursion at depth 2. Two vertices that are not 2-edge-connected but have not been separated yet, i.e., they are ordinary vertices of an auxiliary graph computed at recursion depth 2, can be separated by running Algorithm Simple2ECB for the specific

auxiliary graph. However, as we show in the proof of Lemma 2.12, it suffices to remove only one strong bridge of that auxiliary graph, so we only need to execute Step 3 of Algorithm Simple2ECB once.

Lemma 2.12. *Algorithm Fast2ECB is correct.*

Proof. Let u and v be any vertices. If u and v are 2-edge-connected in G, then by Lemma 2.9 they are 2-edge-connected in both auxiliary graphs of G and G_r that contain them as ordinary vertices. This implies that the algorithm will correctly include them in the same block. So suppose that u and v are not 2-edge-connected. Then, without loss of generality, we can assume that all paths from u to v contain a common strong bridge. We argue that the blocks of u and v will be separated in some step of the algorithm. If u and v are located in different subtrees of $D(s)$ then the claim is true. If they are in the same subtree then they appear in an auxiliary graph $H = G_r$ as ordinary vertices. By Lemma 2.9, H contains a strong bridge that is contained in all paths from u to v. Let H^R be the reverse graph of H. Let $D_H^R(r)$ be the dominator tree of $H^R(r)$. If u and v are located in different subtrees of D_H^R then the claim is true. Suppose then that they are located in a subtree with root q. By Corollary 2.1, $q \neq r$. Let $p = d_H^R(q)$ be the parent of q in $D_H^R(r)$. Then (q, p) is a strong bridge of H. We claim that $H \setminus (q, p)$ does not contain any path from u to v. To prove the claim, we consider two cases. First suppose that all paths from v to u in H^R contain a bridge $(d_H^R(x), x)$ of $D_H^R(r)$ such that x is ancestor of u. Then (q, p) must appear in all paths from u to v in H. If not, then $(p, q) \neq (d_H^R(x), x)$, and there is a path π in H^R from x to u that avoids (p, q). Since x is an ancestor of p, there is a path π' in H^R from r to x that also avoids (p, q). So $\pi' \cdot \pi$ gives a path from r to u in H^R that avoids (p, q), a contradiction. Now suppose that there is no bridge $(d_H^R(x), x)$ of $D_H^R(r)$ with x an ancestor of u that is contained in all paths from v to u in H^R. Let e be a strong bridge that separates u from v in H. Then $e \neq (q, p)$, so there is a path π in H from u to r that avoids e. But H contains a path π' from r to v that avoids e. Then $\pi \cdot \pi'$ is a path from u to v in H that does not contain e, a contradiction. ∎

Finally, we show that the algorithm indeed runs in linear time.

Lemma 2.13. *Algorithm Fast2ECB runs in $O(m)$ time.*

Proof. We analyze the total time spent on each step that Algorithm Fast2ECB executes. Step 1 takes $O(m)$ time by [3], and Step 2 takes $O(m)$ time by Lemma 2.10. From Lemma 2.7 we have that the total number of vertices and the total number of edges in all auxiliary graphs H of G are $O(n)$ and $O(m)$ respectively. Therefore, the total number of strong bridges in these auxiliary graphs is $O(n)$ by Lemma 1.1. Then, by Lemma 2.7, the total size (number of vertices and edges) of all auxiliary graphs H_q^R for all H, computed in Step 3.2, is still $O(m)$ and they are also computed in $O(m)$ total time by Lemma 2.10. So Steps 3.1 and 3.4 take $O(m)$ time in total as well. ∎

2.5 Sparse certificate for the 2-edge-connected blocks

We now show how to compute in linear time a sparse certificate for the 2-edge-connected blocks, i.e., a subgraph $C(G)$ of the input graph G that has $O(n)$ edges and maintains the same 2-edge-connected blocks as the input graph. Such a sparse certificate allows us to speed up computations, such as finding the actual edge-disjoint paths that connect a pair of vertices (see, e.g., [29]). As throughout the Chapter, we can assume without loss of generality that G is strongly connected, in which case subgraph $C(G)$ will also be strongly connected (see the proof of Lemma 2.14 below). The certificate uses the concept of *independent spanning trees* [16]. In this context, a spanning tree T of a flow graph $G(s)$ is a tree with root s that contains a path from s to v for all vertices v. Two spanning trees B and R rooted at s are *independent* if for all vertices v, the paths from s to v in B and R share only the dominators of v. Every flow graph $G(s)$ has two such spanning trees, computable in linear time [16]. Moreover, the computed spanning trees are *maximally edge-disjoint*, meaning that the only edges they have in common are the bridges of $G(s)$.

The sparse certificate can be constructed during the computation of the 2-edge-connected blocks, by extending Algorithm Fast2ECB. We now sketch the main modifications needed. During the execution of Algorithm Fast2ECB, we maintain a list (multiset) L of the edges to be added in $C(G)$. The same edge may be inserted into L multiple times, but the total number of insertions will be $O(n)$. Then we can use radix sort to remove duplicate edges in $O(n)$ time. We initialize L to be empty. During Step 1 of Algorithm Fast2ECB we compute two independent spanning trees, $B(G(s))$ and $R(G(s))$ of $G(s)$ and insert their edges into L. Next, in Step 3.1 we compute two independent spanning trees $B(H^R(r))$ and $R(H^R(r))$ for each auxiliary graph $H^R(r)$. For each edge (u, v) of these spanning trees, we insert a corresponding edge into L as follows. If both u and v are ordinary vertices in $H^R(r)$, we insert (u, v) into L since it is an original edge of G. Otherwise, u or v is an auxiliary vertex and we insert into L a corresponding original edge of G. Such an original edge can be easily found during the construction of the auxiliary graphs. Finally, in Step 3.4, we compute two spanning trees for every connected component S_i of each auxiliary graph $H^R_q \setminus (p, q)$ as follows. Let H_{S_i} be the subgraph of H_q that is induced by the vertices in S_i. We choose an arbitrary vertex $v \in S_i$ and compute a spanning tree of $H_{S_i}(v)$ and a spanning tree of $H^R_{S_i}(v)$. We insert in L the original edges that correspond to the edges of these spanning trees.

Lemma 2.14. *The sparse certificate $C(G)$ has the same 2-edge-connected blocks as the input digraph G.*

Proof. It suffices to show that the execution of Algorithm Fast2ECB on $C(G)$ and produces the same 2-edge-connected blocks as the execution of Algorithm Fast2ECB on G. The correctness of Algorithm Fast2ECB implies that it produces the same result regardless of the choice of start vertex s. So we assume that both executions choose the same start vertex s. We will refer to the execution of Algorithm Fast2ECB with input G (resp. $C(G)$) as Fast2ECB(G) (resp. Fast2ECB($C(G)$)).

First we note that $C(G)$ is strongly connected. This follows from the fact that $C(G)$ contains a spanning tree of $G(s)$, and that it also contains edges that correspond to a spanning tree for the reverse of each auxiliary graph G_r; if (u, v) is a shortcut edge in such a spanning tree for the reverse auxiliary graph H^R, then $C(G)$ will contain original edges that form a path from v to u. Moreover, the fact that $C(G)$ contains two independent spanning trees of G implies that G and $C(G)$ have the same dominator tree and bridges with respect to the start vertex s that are computed in Step 1. Hence, the subtrees $T(r)$ computed of Step 2 of Algorithm Fast2ECB are the same in both executions Fast2ECB(G) and Fast2ECB($C(G)$). The same argument as in Step 1 implies that in Steps 3.1 and 3.2, both executions Fast2ECB(G) and Fast2ECB($C(G)$) compute the same partitions $T^R(r)$ of each auxiliary graph $H^R(r)$. Finally, by construction, the strongly connected components of each auxiliary graph $H_q^R \setminus (p, q)$ are the same in both executions of Fast2ECB(G) and Fast2ECB($C(G)$).

We conclude that Fast2ECB(G) and Fast2ECB($C(G)$) compute the same 2-edge-connected blocks as claimed. ∎

The fact that $C(G)$ has $O(n)$ edges follows from Lemmas 1.1 and 2.7. Therefore, we have the following result.

Corollary 2.3. *We can compute in linear time a sparse certificate for the 2-edge-connected blocks of a digraph.*

Chapter 3

Computing 2-Vertex-Connected Blocks

3.1 Introduction and properties

First, we recall some definitions from Section 1.2.1. Let v and w be two distinct vertices in a digraph. By Menger's Theorem [28], $v \leftrightarrow_{2e} w$ if and only if the removal of any edge leaves v and w in the same strongly connected component, i.e., two vertices are 2-edge-connected if and only if they are resilient to the deletion of a single edge. The situation for 2-vertex connectivity is more complicated. Indeed, Menger's Theorem implies that $v \leftrightarrow_{2v} w$ only if the removal of any vertex different from v and w leaves them in the same strongly connected component, while the converse holds only when v and w are not adjacent. For instance, two mutually adjacent vertices are left in the same strongly

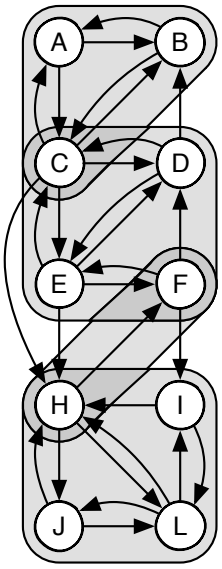


Figure 3.1: The vertex-resilient blocks of the digraph of Figure 1.1.

connected component by the removal of any other vertex, although they are not necessarily 2-vertex-connected. To handle this situation, we use the following notation, which was also considered in [23]. Vertices v and w are said to be *vertex-resilient*, denoted by $v \leftrightarrow_{\mathrm{vr}} w$ if the removal of any vertex different from v and w leaves v and w in the same strongly connected component. We define a *vertex-resilient block* of a digraph $G = (V, E)$ as a maximal subset $B \subseteq V$ such that $u \leftrightarrow_{\mathrm{vr}} v$ for all $u, v \in B$. See Figure 3.1. Note that, as a (degenerate) special case, a vertex-resilient block might consist of a singleton vertex only: we denote this as a *trivial vertex-resilient block*. Clearly, for any vertex v, the singleton set $\{v\}$ is a trivial vertex-resilient block of G if and only if there is no vertex $u \neq v$ such that $u \leftrightarrow_{\mathrm{vr}} v$.In the following, we will consider only non-trivial vertex-resilient blocks. Since there is no danger of ambiguity, we will call them simply vertex-resilient blocks. We remark that two vertices v and w that are vertex-resilient are not necessarily 2-vertex-connected: this is indeed the case for vertices H and F in the digraph of Figure 1.1(a). If, however, v and w are not adjacent then $v \leftrightarrow_{2\mathrm{v}} w$ if and only if $v \leftrightarrow_{\mathrm{vr}} w$.

We next provide some basic properties of the vertex-resilient blocks and the 2-vertex-connected blocks. In particular, we show that any digraph has at most $n-1$ vertex-resilient (resp., 2-vertex-connected) blocks and, moreover, that there is a forest representation of these blocks that enables us to test vertex-resilience (resp., 2-vertex-connectivity) between any two vertices in constant time. This structure is reminiscent of the representation used by Westbrook and Tarjan [38] for the biconnected components of an undirected graph.

Lemma 3.1. *Let u, v, x, and y be distinct vertices such that $u \leftrightarrow_{\mathrm{vr}} x$, $v \leftrightarrow_{\mathrm{vr}} x$, $u \leftrightarrow_{\mathrm{vr}} y$ and $v \leftrightarrow_{\mathrm{vr}} y$. Then also $x \leftrightarrow_{\mathrm{vr}} y$ and $u \leftrightarrow_{\mathrm{vr}} v$.*

Proof. Assume, for contradiction, that x and y are not vertex-resilient. Then there is a

strong articulation point w such that every path from y to x contains w, or every path from x to y contains w (or both). Without loss of generality, suppose that w is contained in every path from y to x. Since u and v are distinct, we can assume that $w \neq u$. (If $w = u$ then we swap the role of u and v.) Then, $y \leftrightarrow_{\text{vr}} u$ implies that there is a path P from y to u that avoids w, and similarly, $u \leftrightarrow_{\text{vr}} x$ implies that there is a path Q from u to x that avoids w. So, P followed by Q gives a path from y to x that does not contain w, a contradiction. Hence $x \leftrightarrow_{\text{vr}} y$. The fact that $u \leftrightarrow_{\text{vr}} v$ follows by repeating the same argument for u and v. ∎

Corollary 3.1. *Let B and B' be two distinct vertex-resilient blocks of a digraph $G = (V, E)$. Then $|B \cap B'| \leq 1$.*

Proof. Follows immediately from Lemma 3.1. ∎

We denote by $VRB(u)$ the vertex-resilient blocks that contain u. Define the *block graph* $F = (V_F, E_F)$ of G as follows. The vertex set V_F consists of the vertices in V and also contains one *block node* for each vertex-resilient block of G. The edge set E_F consists of the edges $\{u, B\}$ where $B \in VRB(u)$. Thus, F is an undirected bipartite graph. Next we show that it is also acyclic.

Lemma 3.2. *Let u and v be any vertices that are connected by a path P in F. Then, for any vertex $w \in V$ not on P, u and v are strongly connected in digraph $G \setminus w$.*

Proof. It suffices to show that G contains a path Q from u to v that avoids w. The same argument shows that G contains a path from v to u that avoids w. Let $P = (u_1 = u, B_1, u_2, B_2, \ldots, u_{k+1} = v)$. Then $u_i \leftrightarrow_{\text{vr}} u_{i+1}$, for $1 \leq i \leq k$, so there is a path P_i in G from v_i to v_{i+1} that avoids w. Then the catenation of paths $P_1, \ldots, P_k$ gives a path in G from u to v that avoids w. ∎

Lemma 3.3. *Graph F is acyclic.*

Proof. Suppose, for contradiction, that F contains a cycle C. We show that all vertices $w \in C \cap V$ belong to the same vertex-resilient block B. Let $u, v \in V$ be two vertices on a minimal cycle C of F that are adjacent to a block node B. (Such u, v, and B exist since F is bipartite.) Then, u and v cannot be the only vertices in V that are on C, since otherwise they would be adjacent to another block B' on C, violating Corollary 3.1. Therefore, C contains a vertex $w \in V \setminus \{u, v\}$. Clearly, $w \notin B$, otherwise the edge $\{w, B\}$ would exist contradicting the minimality of C. Hence, there is a vertex $z \in B$ such that all paths from z to w contain a common strong articulation point or all paths from w to z contain a common strong articulation point. Suppose, without loss of generality, that a vertex x is contained in every path from z to w. Let P be the path that results from C by removing B. Let P_u and P_v be the subpaths of P from u to w and from v to w, respectively. Then $x \notin P_u$ or $x \notin P_v$ (or both). Suppose $x \notin P_u$; if not then swap the role of u and v. Then, by Lemma 3.2 there is a path Q in G from u to w that avoids x. Also, since $u \leftrightarrow_{\text{vr}} z$, there is a path Q' in G from z to u that avoids x. Then the catenation of Q' and Q gives a path in G from z to w that avoids x, a contradiction. ∎

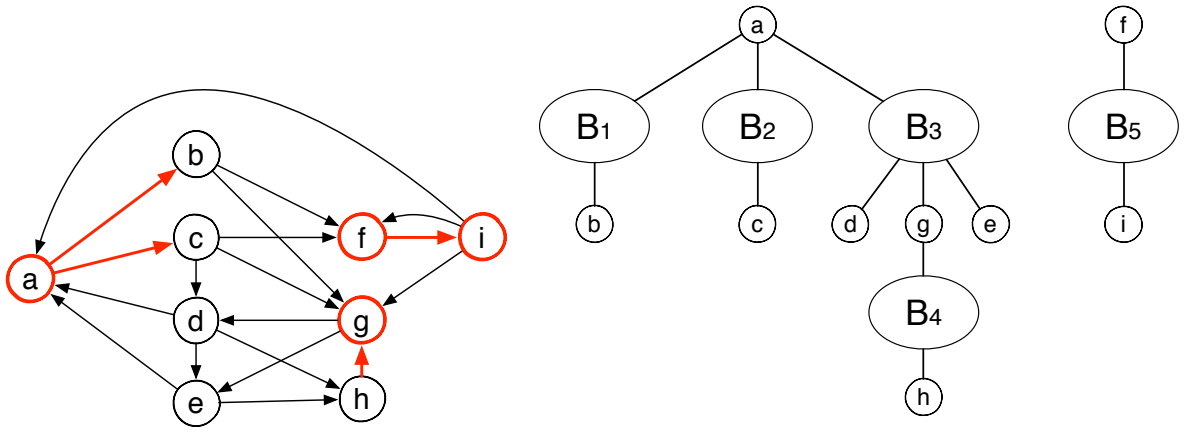


Figure 3.2: A digraph G and its vertex-resilient block forest F. The strong articulation points and the strong bridges of G are shown in red. (Better viewed in color.)

Lemma 3.4. *The number of vertex-resilient blocks in a digraph G is at most $n-1$.*

Proof. We prove the lemma by showing that forest F contains at most $n-1$ block nodes. Since F is a forest we can root each tree T of F at some arbitrary vertex $r \in V$. Every level of T contains either only vertices of V or only block nodes, because F is bipartite. Moreover, every block node is adjacent to at least two vertices of V, due to the fact that each (non-trivial) vertex-resilient block in G contains at least 2 vertices. Hence, every leaf of T is a vertex in V. Now consider a partition of T into vertex disjoint paths $P_1, P_2, \ldots, P_k$, such that each P_i leads from some vertex or block node to a leaf descendant. The number of block nodes in each P_i is at most equal to $|P_i \cap V|$. Also, in the path P_i starting at r the number of block nodes in P_i is less than $|P_i \cap V|$. We conclude that there at most $n-1$ block nodes in F. ∎

Lemma 3.5. *The total number of vertices in all vertex-resilient blocks is at most $2n-2$.*

Proof. By Lemmas 3.3 and 3.4, the block graph F is a forest with at most $2n-1$ vertices. Each occurrence of a vertex v in a block B corresponds to an edge $\{v, B\}$ of F. Therefore, the total number of vertices in all vertex-resilient blocks equals the number of edges in F, and the lemma follows. ∎

Lemma 3.6. *Let u and v be any vertices that are not vertex-resilient but are connected by a path P in F. Then, for any vertex $w \in V \setminus \{u, v\}$ on P, u and v are not strongly connected in digraph $G \setminus w$.*

Proof. We prove the lemma by contradiction. Let P be a path that connects u and v in F. By Lemma 3.3, this path is unique for u and v. First suppose that P contains only one other vertex $w \in V \setminus \{u, v\}$, so $P = (u, B, w, B', v)$. Then $u \leftrightarrow_{\text{vr}} w$ and $w \leftrightarrow_{\text{vr}} v$. Now suppose that u and v are strongly connected in $G \setminus w$. This fact, together with Lemma

3.2, imply that u and v are strongly connected in $G \setminus x$ for all $x \in V \setminus \{u, v\}$. But this contradicts the assumption that u and v are not vertex-resilient.

Now suppose that path P contains more than one vertex in $V \setminus \{u, v\}$. Let $P = (u = w_0, B_1, w_1, \ldots, B_k, w_k, B_{k+1}, v = w_{k+1})$, where $k > 1$. By the argument above, w_{i-1} and w_{i+1} are not strongly connected in $G \setminus w_i$ for all $i \in \{1, \ldots, k\}$. Suppose that u and v are strongly connected in $G \setminus w_i$ for a fixed $i \in \{1, \ldots, k\}$. By Lemma 3.2, u and w_{i-1}, and w_{i+1} and v, are strongly connected in $G \setminus w_i$. But then, w_{i-1} and w_{i+1} are also strongly connected in $G \setminus w_i$, a contradiction. ∎

We consider F as a forest of rooted trees by choosing an arbitrary vertex as the root of each tree. Then $u \leftrightarrow_{\text{vr}} w$ if and only if u and w are siblings or one is the grandparent of the other. See Figure 3.2. We can perform both tests in constant time simply by storing the parent of each vertex in F. Thus, we can test in constant time if two vertices are vertex-resilient. Note that we cannot always apply Lemma 3.6 to find a strong articulation point that separates two vertices u and w that are not vertex-resilient. Indeed, two vertices that are strongly connected but not vertex-resilient may not even be connected by a path in the forest F (see, e.g., vertices f and h in Figure 3.2). So if we wish to return a witness that u and w are not vertex-resilient, we cannot rely on F. We deal with this problem in Section 3.5.

Now we turn to 2-vertex-connected blocks and provide some properties that enable us to compute them via the vertex-resilient blocks.

Lemma 3.7. *Let v and w be two distinct vertices of G such that $v \leftrightarrow_{\text{vr}} w$. Then, v and w are not 2-vertex connected if and only if at least one of the edges (v, w) and (w, v) is a strong bridge in G.*

Proof. Menger's Theorem [28] implies that if v and w are not adjacent then $v \leftrightarrow_{\text{2v}} w$ if and only if $v \leftrightarrow_{\text{vr}} w$. If, on the other hand, $v \leftrightarrow_{\text{vr}} w$ but v and w are not 2-vertex-connected, then at least one of the edges (v, w) and (w, v) exists in G and is a strong bridge. ∎

The following corollary, which relates 2-vertex-connected, 2-edge-connected and vertex-resilient blocks, is an immediate consequence of Lemma 3.7.

Corollary 3.2. *For any two distinct vertices v and w, $v \leftrightarrow_{\text{2v}} w$ if and only if $v \leftrightarrow_{\text{vr}} w$ and $v \leftrightarrow_{\text{2e}} w$.*

By Corollary 3.2 we have that the 2-vertex-connected blocks are refinements of the vertex-resilient blocks, formed by the intersections of the vertex-resilient blocks and the 2-edge-connected blocks of the digraph G. Since the 2-edge-connected blocks are a partition of the vertices of G, these intersections partition each vertex-resilient block. From this property we conclude that Lemmas 3.3 and 3.4 also hold for the 2-vertex-connected blocks, i.e., they can also be represented by a bipartite forest of $O(n)$ size.

3.2 Additional challenges computing the 2-vertex-connected blocks

Our algorithm for computing vertex-resilient blocks of a digraph follows the high-level approach of the algorithm we described in Chapter 2 for computing the 2-edge-connected blocks. However, the algorithm for computing the 2-vertex-connected blocks is much more involved and requires several novel ideas and non-trivial techniques to achieve the claimed bounds. In particular, the main technical difficulties that need to be tackled when following the approach of the algorithm in Chapter 2 are the following:

- First, the algorithm for computing the 2-edge-connected blocks maintains a partition of the vertices into approximate blocks, and refines this partition as the algorithm progresses. Unlike 2-edge-connected blocks, however, vertex-resilient and 2-vertex-connected blocks do not partition the vertices of a digraph, and therefore it is harder to maintain approximate blocks throughout the algorithm's execution. To cope with this problem, we show that these blocks can be maintained using THE more complicated forest representation from Section 3.1, and we define a set of suitable operations on this representation in order to refine and split blocks. We believe that our forest representation of the 2-vertex-connected blocks of a digraph can be of independent interest.

- Second, in Chapter 2 we used a properly defined *canonical decomposition* of the input digraph G, in order to obtain smaller *auxiliary* digraphs (not necessarily subgraphs of G) that maintain the original 2-edge-connected blocks of G. A key property of this decomposition was the fact that any vertex in an auxiliary graph G_r is reachable from a vertex outside G_r only through a single strong bridge. In the computation of the 2-vertex-connected blocks, we have to decompose the graph according to strong articulation points, and so the above crucial property is completely lost. To overcome this problematic issue, we need to design and to implement efficiently a different and more sophisticated decomposition.

- Third, differently from 2-edge connectivity, 2-vertex connectivity in digraphs is plagued with several degenerate special cases, which are not only more tedious but also more cumbersome to deal with. For instance, the algorithm in Section 2.4 exploits implicitly the property that two vertices v and w are 2-edge-connected if and only if the removal of any edge leaves v and w in the same strongly connected component. Unfortunately, this property no longer holds for 2-vertex connectivity, as for instance two mutually adjacent vertices are always left in the same strongly connected component by the removal of any other vertex, but they are not necessarily 2-vertex-connected. To handle this more complicated situation, we introduce the notion of *vertex-resilient blocks* and prove some useful properties about the vertex-resilient and 2-vertex-connected blocks of a digraph.

Another difference with Chapter 2 is that now we are able to provide a witness for two vertices not being 2-vertex-connected. This approach can be applied to provide a witness

Algorithm SimpleVRB: Computation of the vertex-resilient blocks of a strongly connected digraph $G = (V, E)$

Step 1: Compute the strong articulation points of G.

Step 2: Initialize the current set of blocks as $\mathcal{B} = \{V\}$. (Start from the trivial set containing only one block.)

Step 3: For each strong articulation point x do:

Step 3.1: Compute the strongly connected components $S_1, \ldots, S_k$ of $G \setminus x$. Let $\mathcal{S}$ be the partition of $V \setminus x$ defined by the strongly connected components S_i.

Step 3.2: Execute *refine*$(\mathcal{B}, \mathcal{S}, x)$.

Figure 3.3: Algorithm SimpleVRB

for two vertices not being 2-edge-connected, thus extending the result in 2.

3.3 A simple algorithm

Algorithm SimpleVRB, illustrated in Figure 3.3, is an immediate application of the characterization of the vertex-resilient blocks in terms of strong articulation points. Let u and v be two distinct vertices. We say that a strong articulation point x *separates* u *from* v if all paths from u to v contain x. In this case u and v belong to different strongly connected components of $G \setminus x$. This observation implies that we can compute the vertex-resilient blocks by computing the strongly connected components of $G \setminus x$ for every strong articulation point x. To do this efficiently we define an operation that refines the currently computed blocks. Let $\mathcal{B}$ be a set of blocks, let $\mathcal{S}$ be a partition of a set $U \subseteq V$, and let x be a vertex not in U.

refine$(\mathcal{B}, \mathcal{S}, x)$: For each block $B \in \mathcal{B}$, substitute B by the sets $B \cap (S \cup \{x\})$ of size at least two, for all $S \in \mathcal{S}$.

In Section 3.6, where we will compute the 2-vertex-connected blocks from the vertex-resilient blocks and the 2-edge-connected blocks, we will use the notation *refine*$(\mathcal{B}, \mathcal{S})$ as a shorthand for *refine*$(\mathcal{B}, \mathcal{S}, x)$ with $x = null$.

Lemma 3.8. *Let N be the total number of elements in all sets of $\mathcal{B}$ ($N = \sum_{B \in \mathcal{B}} |B|$), and let K be the number of elements in U. Then, the operation refine$(\mathcal{B}, \mathcal{S}, x)$ can be executed in $O(N + K)$ time.*

Proof. A simple way to achieve the claimed bound is to number the sets of the partition $\mathcal{S}$, each with a distinct integer id in the interval $[1, K]$. Consider a block B. Each element

$v \in B$ is assigned a label that is equal to the id of the set $S \in \mathcal{S}$ that contains v if $v \in U$, and zero otherwise. Then, the computation of the sets $B \cap (S \cup \{x\})$ for all $S \in \mathcal{S}$ can be done in $O(|B|)$ time with bucket sorting. ∎

Lemma 3.9. *The block graph F of the set of blocks $\mathcal{B}$ maintained by algorithm SimpleVRB is a forest throughout the execution of the algorithm.*

Proof. The lemma follows by induction on the number of *refine* operation executed. Initially, $\mathcal{B}$ contains a single block V, so F is a forest. For the induction step, consider an execution of *refine*$(\mathcal{B}, \mathcal{S}, x)$. Let B be a block of $\mathcal{B}$ that is split into blocks $B_1, B_2, \ldots, B_l$ as a result of this operation. Let T be the tree of F that contains B before the refine operation. We can view T as being rooted at some arbitrary vertex $r \in V$. Let u be the parent of B in T, and let $v_1, v_2, \ldots, v_\delta$ be the children of B. Suppose first that x is not a child of B. Then, since $\mathcal{S}$ is a partition, each child v_i of B is contained in at most one new block B_j. Moreover, if $u = x$ then u will be the parent of all blocks $B_1, B_2, \ldots, B_l$ after the operation. Otherwise, if $u \neq x$, then u is also contained in at most one new block B_j that will remain in T with parent u, and the rest of the new blocks will be detached from T. Finally, suppose that x is a child of B. Then x is contained in all new blocks $B_1, B_2, \ldots, B_l$, but at most one of these new blocks contains u. If such a block $B_j \supseteq \{u, x\}$ exists, then B_j is the new parent of x, and all other blocks $B_i \neq B_j$ become children of x. Thus, in all cases F remains a forest. ∎

Lemma 3.10. *Algorithm SimpleVRB runs in $O(mp^*)$ time, where p^* is the number of strong articulation points of G. This is $O(mn)$ in the worst case.*

Proof. The strong articulation points of G can be computed in linear time by [22]. In each iteration of Step 3, we can compute the strongly connected components of $G \setminus x$ in linear time [33]. As we discover the i-th strongly connected component, we assign label i ($i \in \{1, \ldots, n\}$) to the vertices in S_i. By Lemma 3.9, the block graph corresponding to the set of blocks $\mathcal{B}$ that the algorithm maintains is a forest. Hence, by Lemma 3.4, $\mathcal{B}$ has at most $n - 1$ blocks, and by Lemma 3.5, the total number of elements in all blocks is at most $2n - 2$. So, by Lemma 3.8, each iteration of Step 3 takes $O(n)$ time. This yields the desired $O(mp^*)$ running time, where p^* is the number of strong articulation points of G. Since a digraph may have up to n strong articulation points, this is $O(mn)$ in the worst case. ∎

3.4 Linear-time algorithm

We will show how to obtain a faster algorithm by using dominator trees and auxiliary graphs, as we did in the computation of the 2-edge-connected blocks in Chapter 2. As already mentioned, auxiliary graphs need to be defined in a substantially different way, which complicates several technical details.

As a warm up, first consider the computation of $VRB(v)$, i.e., the vertex-resilient blocks that contain a specific vertex v. Consider the flow graph $G(v)$ with start vertex v and its reverse digraph $G^R(v)$, obtained after reversing edge directions. Let w be a vertex other than v. Clearly, v and w are vertex-resilient if and only if v is the only proper dominator of w in both $G(v)$ and $G^R(v)$, i.e., $d(w) = v$ and $d^R(w) = v$. Now let u be a sibling of w in both $D(v)$ and $D^R(v)$. The fact that $d^R(w) = v$ and $d(u) = v$ implies that for any vertex $x \in V \setminus \{v, w, u\}$ there is path from w to u through v that avoids x. So w and u are in a common vertex-resilient block that contains v if and only if they lie in the same strongly connected component of $G \setminus v$. This observation implies the following linear-time algorithm to compute the vertex-resilient blocks that contain v. Compute the dominator trees $D(v)$ and $D^R(v)$ of $G(v)$ and $G^R(v)$ respectively. Let $C(v)$ (resp., $C^R(v)$) be the set of children of v in $D(v)$ (resp., $D^R(v)$). Set $U = C(v) \cap C^R(v)$ and initialize the set of blocks $\mathcal{B} = \{U\}$. Compute the strongly connected blocks $S_1, S_2, \ldots, S_k$ of $G \setminus v$. Let $\mathcal{S}$ be the set that contains the nonempty restrictions of the S_i sets to U, i.e., $\mathcal{S}$ contains the nonempty sets $S_i \cap U$. Finally, execute *refine*$(\mathcal{B}, \mathcal{S}, v)$.

Note that all the vertex-resilient blocks can be computed in $O(mn)$ time by applying the above algorithm to all vertices v. To avoid the repeated applications of this algorithm we develop a new concept of *auxiliary graphs* for 2-vertex connectivity. Before doing that, we state two properties regarding information that a dominator tree can provide about vertex-resilient blocks and paths.

Lemma 3.11. *Let $G = (V, E)$ be a strongly connected digraph, and let $s \in V$ be an arbitrary start vertex. Any two vertices x and y are vertex-resilient only if they are siblings in $D(s)$ or one is the immediate dominator of the other in $G(s)$.*

Proof. Immediate. ∎

Lemma 3.12. *Let r be a vertex, and let v be any vertex that is not a descendant of r in $D(s)$. Then there is a path from v to r that does not contain any proper descendants of r in $D(s)$. Moreover, all simple paths from v to any descendant of r in $D(s)$ contain r.*

Proof. Let P be any path from v to r. (Such a path exists since digraph G is strongly connected.) Let u be the first vertex on P such that u is a descendant of r. Then either $u = r$ or u is a proper descendant of r. In the first case the lemma holds. Suppose u is a proper descendant of r. Since v is not a descendant of r in $D(s)$, there is a path Q from s to v in G that does not contain r. Then Q followed by the part of P from v to u is a path from s to u that avoids r, a contradiction. ∎

3.4.1 Auxiliary graphs

As in Chapter 2, *auxiliary graphs* are a key concept in our algorithm that provides a decomposition of the input digraph G into smaller digraphs (not necessarily subgraphs of G) that maintain the original vertex-resilient blocks. In Chapter 2 we used a *canonical decomposition* of the input digraph, in order to obtain auxiliary graphs that maintain

the 2-edge-connected blocks. A key property of this decomposition was the fact that any vertex in an auxiliary graph G_r is reachable from a vertex outside G_r only though a single strong bridge. In the computation of the vertex-resilient blocks, however, we have to decompose the input digraph according to strong articulation points, and thus the above property is completely lost. To overcome this critical issue, we apply a different and more involved decomposition.

Let s be an arbitrarily chosen start vertex in G. Recall that we denote by $G(s)$ the flow graph with start vertex s, by $G^R(s)$ the flow graph obtained from $G(s)$ after reversing edge directions, by $D(s)$ and $D^R(s)$ the dominator trees of $G(s)$ and $G^R(s)$ respectively, and by $C(v)$ and $C^R(v)$ the set of children of v in $D(s)$ and $D^R(s)$ respectively.

For each vertex r, let $C^k(r)$ denote the level k descendants of r, i.e., $C^0(r) = \{r\}$, $C^1(r) = C(r)$, etc. For each vertex $r \neq s$ that is not a leaf in $D(s)$ we build the *auxiliary graph* $G_r = (V_r, E_r)$ *of* r as follows. The vertex set of G_r is $V_r = \cup_{k=0}^{3} C^k(r)$ and it is partitioned into a set of *ordinary* vertices $V_r^o = C^1(r) \cup C^2(r)$ and a set of *auxiliary* vertices $V_r^a = C^0(r) \cup C^3(r)$. The auxiliary graph G_r results from G by contracting the vertices in $V \setminus V_r$ as follows. All vertices that are not descendants of r in $D(s)$ are contracted into r. For each vertex $w \in C^3(r)$, we contract all descendants of w in $D(s)$ into w. See Figure 3.4. We use the same definition for the auxiliary graph G_s of s, with the only difference that we let s be an ordinary vertex. Also note that when we form G_s from G, no vertex is contracted into s. In order to bound the size of all auxiliary graphs, we eliminate parallel edges during those contractions.

Lemma 3.13. *The auxiliary graphs G_r have at most $4n$ vertices and $4m + n$ edges in total.*

Proof. A vertex of G may appear in at most four auxiliary graphs. Therefore, the total number of edges in all auxiliary graphs excluding type-(b) shortcut edges (u, v) with $u \notin V_r$ is at most $4m$. A type-(b) shortcut edge (u, v) with $u \notin V_r$ of G_r corresponds to a unique vertex in $C^3(r)$, so there are at most n such edges. ∎

Lemma 3.14. *Let v and w be two vertices in V_r. Any path P from v to w in G has a corresponding path P_r from v to w in G_r, and vice versa. Moreover, if v and w are both ordinary vertices in G_r, then P_r contains a strong articulation point if and only if P does.*

Proof. The correspondence between paths in G and paths in G_r follows from the definition of the auxiliary graph. Next we prove the second part of the lemma. Let P_r be the path in G_r that corresponds to a path P from v to w in G, where both v and w are ordinary vertices in G_r. By the construction of the auxiliary graph, we have that if P_r contains a strong articulation point then so does P. For the contraposition, suppose P contains a strong articulation point x. Consider the following cases:

- $x \in V_r$. Then, by the construction of the auxiliary graph, we have $x \in P_r$.

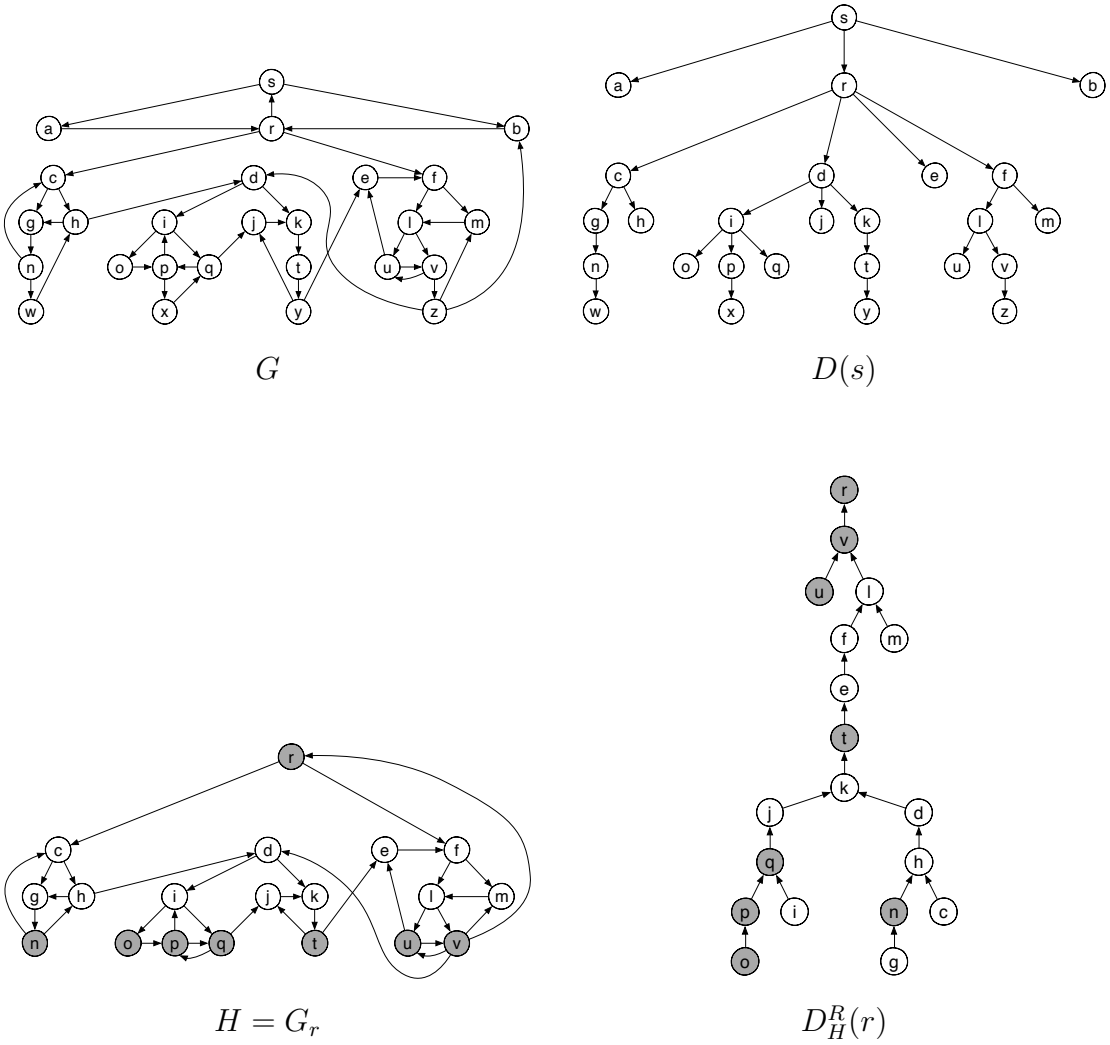


Figure 3.4: A strongly connected graph G, the dominator tree $D(s)$ of flow graph $G(s)$, the auxiliary graph $H = G_r$ and the dominator tree $D^R_H(r)$ of the flow graph $H^R(r)$. (The edges of the dominator tree $D^R_H(r)$ are shown directed from child to parent.) The auxiliary vertices of H are shown gray.

- x is a descendant of a vertex $z \in C^3(r)$. Vertex z is a strong articulation point since it is either x or a proper descendant of x. Then, by Lemma 3.12, the part of P from v to x contains z. So, P_r also contains z by the construction of the auxiliary graph.

- x is not a descendant of r. In this case, we have $r \neq s$. Since v and w are ordinary vertices of G_r, $C^1(r)$ is not empty and therefore r is a strong articulation point. By Lemma 3.12, the part of P from x to w contains r. So, P_r also contains r by the construction of the auxiliary graph.

Hence, in every case P_r contains a strong articulation point and the lemma follows. ∎

Corollary 3.3. *Each auxiliary graph G_r is strongly connected.*

Proof. Follows from the construction of G_r, Lemma 3.14, and the fact that G is strongly connected. ∎

The next lemma shows that auxiliary graphs maintain the vertex-resilient relation of the original digraph.

Lemma 3.15. *Let v and w be any two distinct vertices of G. Then v and w are vertex-resilient in G if and only if they are both ordinary vertices in an auxiliary graph G_r and they are vertex-resilient in G_r.*

Proof. Suppose first that v or w is s. Without loss of generality assume $v = s$. Then by Lemma 3.11 we have that $w \in C^1(r)$, so v and w are both ordinary vertices of G_s. Now consider that $v, w \in V \setminus s$. From Lemma 3.11 we have that v and w belong in a set $C^1(r) \cup C^2(r)$ so they are both ordinary vertices of G_r. Clearly if all paths from v to w in G_r contain a common vertex (strong articulation point), then so do all paths from v to w in G by Lemma 3.14. Now we prove the converse. Suppose all paths from v to w in G contain a common vertex u. If $u \in V_r$ then also all paths from v to w in G_r contain u by the proof of Lemma 3.14. So suppose $u \notin V_r$. Then v is not an ancestor of w in $D(s)$, since otherwise there would be a path from v to w that avoids u.

First consider that u is a (proper) descendant of r in $D(s)$. Since v is not an ancestor of w in $D(s)$, there is a vertex $x \in C^3(r)$ that is an ancestor of u. By Lemma 3.12, all paths from v to u in G, and thus all paths from v to w, contain x. By Lemma 3.14 this is also true for all paths from v to w in G_r.

Finally, if u is not a descendant of r, Lemma 3.12 implies that all paths from u to w in G contain vertex r. Hence, all paths from v to w in G contain r, and so do all paths from v to w in G_r by Lemma 3.14. ∎

Now we specify how to compute all the auxiliary graphs $G_r = (V_r, E_r)$ in $O(m + n)$ time. Observe that the edge set E_r contains all edges in $G = (V, E)$ induced by the vertices in V_r (i.e., edges $(u, v) \in E$ such that $u \in V_r$ and $v \in V_r$). We also add in E_r the following types of *shortcut* edges that correspond to paths in G. (a) If G contains an edge (u, v) such that $u \notin V_r$ is a descendant of r in $D(s)$ and $v \in V_r$ then we add the shortcut edge (z, v) where z the is an ancestor of u in $D(s)$ such that $z \in C^3(r)$. (b) If G contains an edge (u, v) such that u but not v is a descendant of r in $D(s)$ then we add the shortcut edge (z, r) where z the nearest ancestor of u in $D(s)$ such that $z \in V_r$ ($z = u$ if $u \in V_r$). We note that we do not keep multiple (parallel) shortcut edges. See Figure 3.4. We use the same definition for the auxiliary graph G_s of s, with the only difference that we let s be an ordinary vertex. We also note that G_s does not contain type-(b) shortcut edges.

To construct the auxiliary graphs $G_r = (V_r, E_r)$ we need to specify how to compute the shortcut edges of type (a) and (b). To do this efficiently we need to test ancestor-descendant relations in $D(s)$. There are several simple $O(1)$-time tests of this relation [35]. The most convenient one for us is to number the vertices of $D(s)$ from 1 to n in preorder, and to compute the number of descendants of each vertex. Then, vertex v is a descendant

of r if and only if $pre(r) \leq pre(v) < pre(r) + size(r)$, where, for any vertex x, $pre(x)$ and $size(x)$ are, respectively, the preorder number and the number of descendants of x in $D(s)$.

Suppose (u, v) is an edge of type (a). We need to find the ancestor z of u in $D(s)$ such that $z \in C^3(r)$. We process all such arcs of G_r as follows. We create a list A_r that contains the edges (u, v) of type (a), and sort A_r in increasing preorder of u. We create a second list A'_r that contains the vertices in $C^3(r)$, and sort A'_r in increasing preorder. Then, the shortcut edge of (u, v) is (z, v), where z is the last vertex in the sorted list A'_r such that $pre(z) \leq pre(u)$. Thus the shortcut edges of type (a) can be computed by bucket sorting and merging. In order to do this fast for all auxiliary graphs, we sort all the lists at the same time as follows. First, we create a unified list A containing the triples $(r, pre(u), v)$, for each type (a) edge (u, v) in the auxiliary graph G_r. Next we sort A in increasing order of the two first elements. We also create a second list A' with pairs $(r, pre(w))$, where $w \in C^3(r)$, and sort the pairs in increasing order. Finally, we compute the shortcut edges of each auxiliary graph G_r by merging the sorted sublists of A and A' that correspond to the same root r. Then, the shortcut edge for the triple $(r, pre(u), v)$ is (z, v), where $(r, pre(z))$ is the last pair in the sorted sublist of A' with root r such that $pre(z) \leq pre(u)$.

Now we consider the edges of type (b). For each vertex $w \in C^3(r)$ we need to test if there is an edge (u, v) in G such that u is a proper descendant of w and v is not a descendant of r in $D(s)$. In this case, we add in G_r the edge (w, r). To do this test efficiently, we assign to each edge (u, v) a tag $t(u, v)$ which we set equal to the preorder number of the nearest common ancestor of u and v in $D(s)$. We can do this easily by using the parent property and the $O(1)$-time test of the ancestor-descendant relation as follows: $t(u, v) = pre(u)$ if u is an ancestor of v in $D(s)$, $t(u, v) = pre(v)$ if v is an ancestor of u in $D(s)$, and $t(u, v) = pre(d(v))$ otherwise. At each vertex $w \neq s$ in $D(s)$ we store a label $\ell(w)$ which is the minimum tag of among the edges (w, v). Using these labels we compute for each $w \neq s$ in $D(s)$ the values $low(w) = \min\{\ell(v) \mid v \text{ is a descendant of } w \text{ in } D(s)\}$. These computations can be done in $O(m)$ time by processing the tree $D(s)$ in a bottom-up order. Now consider the auxiliary graph G_r. We process the vertices in $C^3(r)$. For each such vertex w we add the shortcut edge (w, r) if $low(w) < pre(r)$.

Lemma 3.16. *We can compute all auxiliary graphs G_r in $O(m)$ time.*

We also describe an alternative way to compute the type (a) shortcut edges of the auxiliary graphs, by replacing sorting with vertex contractions. To that end, we use a *disjoint set union data structure* [36], which maintains a collection of disjoint sets, each with a representative element, under three operations:

make-set(x)**:** Create a new set $\{x\}$ with representative x. Element x must be in no existing set.

find(x)**:** Return the representative element of the set containing element x.

unite(x, y)**:** Unite the sets containing elements x and y and give the new set the representative of the old set containing x.

Using this data structure, we can compute the type (a) shortcut edges of all auxiliary graphs in a single bottom-up pass of $D(s)$. To initialize the sets, we perform *make-set*(v) for every vertex $v \in V$. The unite operations are also executed in a bottom-up order, and each such operation has the effect of contracting a vertex to its parent in $D(s)$. The details of these computations are as follows. We visit all vertices r in a bottom-up order of $D(s)$, and compute the type (a) shortcut edges of the corresponding auxiliary graph $G_r = (V_r, E_r)$. To do this, we process the edges entering each vertex $v \in V_r$. For each such edge (u, v) we compute $z = \mathit{find}(u)$. If $z \neq v$ and $z \in V_r$ then (z, v) is a type (a) shortcut edge of G_r. The former condition means that u is not a descendant of v, while the second condition can only be violated for $v = r$. After we have processed all edges entering V_r, we execute *unite*$(d(v), v)$ for each vertex $v \in C^3(s)$. This construction runs $O(m)$ time plus the time for $n - 1$ *unite* operations and, by Lemma 3.13, at most $4m$ *find* operations. If *unite* and *find* are implemented using compressed trees with balanced linking [36], the total time for these disjoint set union operations is $O(m\alpha(m, n))$. Since the set of *unite* operations is known in advance we have an instance of the static tree disjoint set union problem, which is solvable in $O(m)$ time on a RAM [10].

3.4.2 The algorithm

Our linear-time algorithm FastVRB is illustrated in Figure 3.5. It uses two levels of auxiliary graphs and applies one iteration of Algorithm SimpleVRB for each auxiliary graph of the second level. The algorithm uses different dominator trees, and applies Lemma 3.11 in order to identify the vertex-resilient blocks. Since different dominator trees may define different blocks (which by Lemma 3.11 are supersets of the vertex-resilient blocks), we will use an operation that we call *split* to combine the different blocks.

We begin by computing the dominator tree $D(s)$ for an arbitrary start vertex s. For any vertex v, we let $\widehat{C}(v)$ denote the set containing v and the children of v in $D(s)$, i.e., $\widehat{C}(v) = C(v) \cup \{v\}$. Lemma 3.11 gives an initial division of the vertices into blocks that are supersets of the vertex-resilient blocks. Specifically, the vertex-resilient blocks that contain v are subsets of $\widehat{C}(v)$ or $\widehat{C}(d(v))$ (for $v \neq s$).

During the course of the algorithm, each vertex v becomes associated with a set of blocks $\mathcal{B}(v)$ that contain v, which are subsets of $\widehat{C}(v)$ and $\widehat{C}(d(v))$ if $v \neq s$. The blocks are refined by applying the *refine* operation of Section 3.3 and operation *split* that we define next, and at the end of the algorithm each set of blocks $\mathcal{B}(v)$ will be equal to $VRB(v)$.

Let B be a block and T be a rooted tree with vertex set $V(T) \supseteq B$. For any vertex $v \in V(T)$, let $\widehat{C}_T(v)$ be the set containing v and the children of v in T.

split(B, T)**:** Return the set that consists of the blocks $B \cap \widehat{C}_T(v)$ of size at least two, for all $v \in V(T)$.

Algorithm FastVRB: Linear-time computation of the vertex-resilient blocks of a strongly connected digraph $G = (V, E)$

Step 1: Choose an arbitrary vertex $s \in V$ as a start vertex. Compute the dominator tree $D(s)$. For any vertex v, let $\widehat{C}(v)$ be the set containing v and the children of v in $D(s)$. Initialize the block forest F by associating block $\widehat{C}(v)$ with every vertex $w \in \widehat{C}(v)$, for all vertices v that are not a leaves in $D(s)$.

Step 2: Compute the auxiliary graphs G_r for all vertices r that are not leaves in $D(s)$.

Step 3: Process the vertices of $D(s)$ in bottom-up order. For each auxiliary graph $H = G_r$ with r not a leaf in $D(s)$ do:

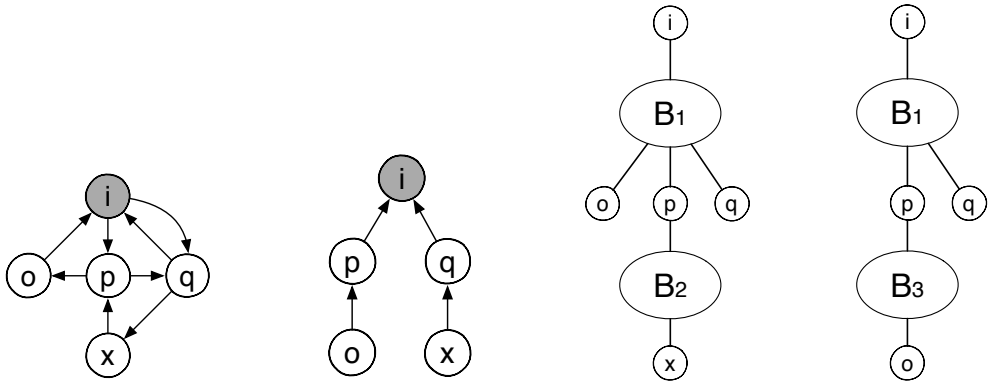
- **Step 3.1:** Compute the dominator tree $T = D_H^R(r)$.
- **Step 3.2:** Compute the set $\mathcal{B}$ of blocks that contain vertices in $C(r)$.
- **Step 3.3:** For each block $B \in \mathcal{B}$ execute *split*(B, T).
- **Step 3.4:** Compute the auxiliary graphs H_q^R for all vertices q that are not leaves in T.
- **Step 3.5:** For each auxiliary graph H_q^R with q not a leaf do:
  - **Step 3.5.1:** Compute the set $\mathcal{B}_q$ of blocks that contain at least two ordinary vertices in H_q^R.
  - **Step 3.5.2:** Compute the set $\mathcal{S}$ of the strongly connected components of $H_q^R \setminus q$.
  - **Step 3.5.3:** Refine the blocks in $\mathcal{B}_q$ by executing *refine*$(\mathcal{B}_q, \mathcal{S}, q)$.

Figure 3.5: Algorithm FastVRB

Lemma 3.17. *Let N be the number of vertices in $V(T)$. Then, the operation split(B, T) can be executed in $O(N)$ time.*

Proof. We number the vertices of T in preorder. Let $pre(v)$ be the preorder number of $v \in V(T)$. Let $t(v)$ be the parent of $v \neq r$ in T, where r is the root of T. We associate each vertex $v \neq r$ in B with two labels $\ell_1(v) = pre(t(v))$ and $\ell_2(v) = pre(v)$, and create two corresponding pairs $\langle \ell_1(v), v \rangle$ and $\langle \ell_2(v), v \rangle$. Also, if $r \in B$, we associate r with one label $\ell_2(r) = pre(r)$, and create a corresponding pair $\langle \ell_2(r), r \rangle$. Each block created by the *split* operation consists of a set of at least two vertices $v \in B$ that are associated with a specific label. We can find these blocks by sorting the pairs $\langle \ell_j(v), v \rangle$ by label, which can be done in $O(N)$ time with bucket sort. ∎

Lemma 3.18. *The block graph F of the set of blocks $\mathcal{B}$ maintained by algorithm FastVRB is a forest throughout the execution of the algorithm.*



$H^R = G_i^R$ $T = D_H^R(i)$ F F'

Figure 3.6: The reverse auxiliary graph $H^R = G_i^R$ of the flow graph $G(s)$ of Figure 3.4 and its dominator tree $T = D_H^R(i)$; F and F' are, respectively, the block forest before and after the execution of *split*(B, T). Only the affected portion of the block forest is shown.

Proof. We describe how a *split* operation can be simulated by a sequence of *refine* operations. The result then follows from Lemma 3.9. Consider the *split*(B, T) operation for a block B and a rooted tree T with vertex set $V(T) = \{v_1, v_2, \ldots, v_\delta\} \supseteq B$. For any vertex $v_i \in V(T)$, we let V_i be the set of descendant of v_i in T. We can achieve the effect of *split*(B, T) by executing a sequence of δ *refine* operations. The i-th operation in this sequence is *refine*$(\mathcal{B}_{i-1}, \mathcal{S}_i, v_i)$, where $\mathcal{B}_{i-1}$ is the set of blocks computed by the first $i-1$ operations, and $\mathcal{S}_i$ is the partition of $V(T) \setminus v_i$ formed by the sets $V_i \setminus v_i$ and $V(T) \setminus V_i$. Initially we set $\mathcal{B}_0 = \{B\}$. This sequence computes exactly the blocks $B \cap \widehat{C}_T(v_i)$ of size at least two, for all $v_i \in V(T)$. ∎

At a high level, the algorithm begins with a "coarse" block tree, induced by the $\widehat{C}(v)$ sets of $D(s)$, which is then refined by the blocks defined from the dominator trees of the auxiliary graphs. An example of this process is shown in Figure 3.6. The final vertex-resilient block forest is then computed by considering the strongly connected components of the second level auxiliary graphs, after removing their designated start vertex. The algorithms need to keep track of the blocks that contain a specific vertex, and, conversely, of the vertices that are contained in a specific block. To facilitate this search we explicitly store the adjacency lists of the current block forest F. Recall that F is bipartite, so the adjacency list of a vertex v stores the blocks that contain v, and the adjacency list of a block node B stores the vertices in B. Initially F contains one block for each set $\widehat{C}(v)$, for all vertices v that are not leaves in $D(s)$. These blocks are later refined by executing the *split* and *refine* operations, which by Lemmas 3.9 and 3.18 maintain the invariant that F is a forest. This fact implies that Lemma 3.5 holds, so the total number of vertices and edges in F is $O(n)$. So when we execute a *split* or a *refine* operation we can update the adjacency lists of F, while maintaining the bounds given in Lemmas 3.8 and 3.17.

Lemma 3.19. *Algorithm FastVRB is correct.*

Proof. Let u and v be any vertices. If u and v are vertex-resilient in G, then by Lemma 3.15 they are vertex-resilient in both auxiliary graphs of G and G_r that contain them as ordinary vertices. This implies that the algorithm will correctly include them in the same block in Step 1 and will not separate them in Steps 3.3 and 3.5. So suppose that u and v are not vertex-resilient. Then, without loss of generality, we can assume that all paths from u to v contain a common strong articulation point. Thus, $d(v) \neq u$. We argue that all the blocks that contain u and all the blocks that contain v will be separated in some step of the algorithm.

First we observe that u and v can appear together in at most one of the blocks constructed in Step 1. Also, u and v can remain in at most one block after each *split* operation (u and v can have at most one identical label $\ell_i(u) = \ell_j(v)$). So suppose that u and v are still contained in one common block just before the execution of Step 3.5. We will show that u and v will be separated after the *refine* operation executed in Step 3.5.3. Since u and v were not separated by a *split* operation, they are either siblings or one is the parent of the other in $D_H^R(r)$. Also, since $d(v) \neq u$ we have the following cases.

(a) $d(u) = v$. Then u and v are both ordinary vertices of the auxiliary graph $H = G_r$ with $r = d(v)$. Lemma 3.15 implies that G_r contains a strong articulation point x that separates u from v. We argue that x is a proper ancestor of u in $D_H^R(r)$. If not, then H^R contains a path P^R from r to u that avoids x. Since $d(v) = r$, H contains a path Q from r to v that avoids x. Thus $P \cdot Q$ is a path in H from u to v that avoids x, a contradiction. Now we claim that $q = d_H^R(u)$ is also a strong articulation point that separates u from v. Suppose the claim is false. Then $x \neq q$, so x is a proper ancestor of q in $D_H^R(r)$. Let P be a path from u to v that avoids q. Then x is on P since x separates u from v. Let P_x be the part of P from u to x. Also, since x is a proper ancestor of q in $D_H^R(r)$, H^R has a path Q^R from r to x that avoids q. Then $P_x \cdot Q$ is a path in H from u to r that avoids q, a contradiction. The claim implies that u and v are located in different strongly connected components of $H_q^R \setminus q$, so they are contained in different blocks computed in Step 3.5.3.

(b) $d(v) = d(u) = r$. Then u and v are both ordinary vertices of the auxiliary graph $H = G_r$. Lemma 3.15 implies that G_r contains a strong articulation point x that separates u from v. By the same arguments as in case (a), it follows that $q = d_H^R(u)$ is a strong articulation point that separates u from v. So again u and v will be located in different blocks after Step 3.5.3. ∎

Lemma 3.20. *Algorithm FastVRB runs in $O(m)$ time.*

Proof. We account for the total time spent on each step that Algorithm FastVRB executes. Step 1 takes $O(m)$ time by [3], and Step 2 takes $O(m)$ time by Lemma 3.16. From Lemma 3.13 we have that the total number of vertices and the total number of edges in all auxiliary graphs H of G are $O(n)$ and $O(m)$ respectively. Then, again by Lemma 3.13, the total size (number of vertices and edges) of all auxiliary graphs H_q^R for all H, computed in Step 3.4, is still $O(m)$ and they are also computed in $O(m)$ total time by Lemma 3.16. Now consider the *split* operations. All these operations that occur during Step 3.3 for a specific auxiliary graph G_r operate on the same tree T, which can be preprocessed once,

as in Lemma 3.17, for all *split* operations. Therefore, the total preprocessing time for all *split* operations is $O(n)$. Excluding the preprocessing time for T, a $split(B, T)$ operation takes time proportional to the number of vertices in B. Therefore all *split* operations take $O(n)$ time in total by Lemmas 3.5 and 3.17. In Step 3.5.1 we examine the adjacency lists of the ordinary vertices $v \in H_q^R$ and find the corresponding blocks that contain at least such two ordinary vertices. Then we examine the adjacency lists of each such block. So, the adjacency lists of each vertex v and each block that contains v can be examined at most three times. Hence, Step 3.5.1 takes $O(n)$ time in total. Finally, Steps 3.5.2 and 3.5.3 take $O(m)$ time in total by [33] and Lemmas 3.5 and 3.8. ∎

3.5 Queries

Algorithm FastVRB computes the vertex-resilient blocks of the input digraph G and stores them in the block forest F of Section 3.1, which makes it straightforward to test in constant time if two query vertices v and w are vertex-resilient. Here we show that if v and w are not vertex-resilient, then we can report a witness of this fact, that is, a strong articulation point x such that v and w are not in the same strongly connected component of $G \setminus x$. Using this witness, it is straightforward to verify in $O(m)$ time that v and w are not vertex-resilient; it suffices to check that v is not reachable from w in $G \setminus x$ or vice versa.

To obtain this witness, we would like to apply Lemma 3.6, but this requires v and w to be in the same tree of the block forest. Fortunately, we can find the witness fast by applying Lemmas 3.11 and 3.12, which use information computed during the execution of FastVRB. We do that as follows. First consider the simpler case where $v = s$. If Lemma 3.11 does not hold for s and w in $D(s)$ then $d(w) \neq s$ is a strong articulation point that separates s from w. Otherwise, $s = d(w)$, and s and w are both ordinary vertices in the auxiliary graph $H = G_s$. Then s and w cannot satisfy Lemma 3.11 in $D_H^R(s)$, so $d_H^R(w)$ is a strong articulation point that separates w from s. Now consider the case where $v, w \in V \setminus s$. Suppose first that v and w do not satisfy Lemma 3.11 in $D(s)$. Then $d(w)$ is not an ancestor of v or $d(v)$ is not an ancestor of w (or both). Assume, without loss of generality, that $d(w)$ is not an ancestor of v. By Lemma 3.12, all paths from v to w pass through $d(w)$, so $d(w)$ is a strong articulation point that separates v from w. On the other hand, if Lemma 3.11 holds for v and w in $D(s)$, then v and w are both ordinary vertices in an auxiliary graph $H = G_r$, where $r = d(v)$ if $v = d(w)$, $r = d(w)$ if $w = d(v)$, and $r = d(v) = d(w)$ otherwise. By Lemma 3.15, v and w are not vertex-resilient in H. If they violate Lemma 3.11 for $D_H^R(r)$ then we can find a strong articulation point that separates them as above. Finally, assume that Lemma 3.11 holds for v and w in $D_H^R(r)$. Now v and w are both ordinary vertices in an auxiliary graph H_q^R. From the proof of Lemma 3.19 we have that $q = d_H^R(v)$ or $q = d_H^R(w)$ and that q is a strong articulation point that separates v and w.

All the above tests can be performed in constant time. It suffices to store the dominator tree $D(s)$ of $G(s)$, and the dominator trees $D_H^R(r)$ of all auxiliary graphs $H^R = G_r^R$. The

space required for these data structures is $O(n)$ by Lemma 3.13.

Theorem 3.1. *Let G be a digraph with n vertices and m edges. We can compute the vertex-resilient blocks of G in $O(m+n)$ time and store them in a data structure of $O(n)$ space. Given this data structure, we can test in $O(1)$ time if any two vertices are vertex-resilient. Moreover, if the two vertices are not vertex-resilient, then we can report in $O(1)$ time a strong articulation point that separates them.*

3.6 Computing the 2-vertex-connected blocks

We can compute the 2-vertex-connected blocks of the input digraph $G = (V, E)$ by applying Corollary 3.2 as follows. Given the vertex-resilient blocks $\mathcal{B}$ and the 2-edge-connected blocks $\mathcal{S}$ of G, we simply execute *refine*$(\mathcal{B}, \mathcal{S})$. This takes $O(n)$ time by Lemma 3.8. Also, since the 2-vertex-connected blocks have a block forest representation, we can test if two given vertices are 2-vertex-connected in $O(1)$ time as described in Section 3.1.

If we only wish to answer queries of whether two vertices v and w are 2-vertex-connected, without computing explicitly the 2-vertex and the 2-edge-connected blocks, then we can use a simpler alternative, as suggested by Lemma 3.7. First, we test if v and w are vertex-resilient in $O(1)$-time as in Section 3.5, and if they are not, then we can report a strong articulation point that separates them. If, on the other hand, v and w are vertex-resilient then we need to check if G contains (v, w) or (w, v) as a strong bridge. We can do this easily using the same information as in Section 3.5, namely the dominator tree $D(s)$ of $G(s)$, and the dominator trees $D_H^R(r)$ of all auxiliary graphs $H^R = G_r^R$. For instance, if (v, w) is a strong bridge in G, then it will appear as an edge in one of the dominator trees. Therefore, it suffices to mark the edges of dominator trees that are strong bridges, and then check if v is the parent of w or w is the parent of v in $D(s)$ or in $D_H^R(r)$, where $H = G_r$ is the auxiliary graph of G such that $r = d(v)$ if $v = d(w)$, $r = d(w)$ if $w = d(v)$, and $r = d(v) = d(w)$ otherwise.

Theorem 3.2. *Let G be a digraph with n vertices and m edges. We can compute the 2-vertex-connected blocks of G in $O(m+n)$ time and store them in a data structure of $O(n)$ space. Given this data structure, we can test in $O(1)$ time if any two vertices are 2-vertex-connected. Moreover, if the two vertices are not 2-vertex-connected, then we can report in $O(1)$ time a strong articulation point or a strong bridge that separates them.*

3.7 Sparse certificate for the vertex-resilient blocks and the 2-vertex-connected blocks

Here we show how to extend Algorithm FastVRB so that it also computes in linear time a sparse certificate for the vertex-resilient and the 2-vertex-connected relations. That is, we compute a subgraph $C(G)$ of the input graph G that has $O(n)$ edges and main-

tains the same vertex-resilient and 2-vertex-connected blocks as the input graph. We can achieve this by applying the same approach we used in Section 2.5 for computing a sparse certificate for the 2-edge-connected blocks.

As throughout the Chapter we can assume without loss of generality that G is strongly connected, in which case subgraph $C(G)$ will also be strongly connected. The certificate uses the concept of *independent spanning trees* [16]. A spanning tree T of a flow graph $G(s)$ is a tree with root s that contains a path from s to v for all vertices v. Two spanning trees B and R rooted at s are *independent* if for all v, the paths from s to v in B and R share only the dominators of v. Every flow graph $G(s)$ has two such spanning trees, computable in linear time [16]. Moreover, the computed spanning trees are *maximally edge-disjoint*, meaning that the only edges they have in common are the bridges of $G(s)$.

During the execution of Algorithm FastVRB, we maintain a list (multiset) L of the edges to be added in $C(G)$. The same edge may be inserted into L multiple times, but the total number of insertions will be $O(n)$. Then we can use radix sort to remove duplicate edges in $O(n)$ time. We initialize L to be empty. During Step 1 of Algorithm FastVRB we compute two independent spanning trees, $B(G(s))$ and $R(G(s))$ of $G(s)$ and insert their edges into L. Next, in Step 3.1 we compute two independent spanning trees $B(H^R(r))$ and $R(H^R(r))$ for each auxiliary graph $H^R(r)$. For each edge (u, v) of these spanning trees, we insert a corresponding edge into L as follows. If both u and v are ordinary vertices in $H^R(r)$, we insert (u, v) into L since it is an original edge of G. Otherwise, u or v is an auxiliary vertex and we insert into L a corresponding original edge of G. Such an original edge can be easily found during the construction of the auxiliary graphs. Finally, in Step 3.5, we compute two spanning trees for every connected component S_i of each auxiliary graph $H_q^R \setminus q$ as follows. Let H_{S_i} be the subgraph of H_q that is induced by the vertices in S_i. We choose an arbitrary vertex $v \in S_i$ and compute a spanning tree of $H_{S_i}(v)$ and a spanning tree of $H_{S_i}^R(v)$. We insert in L the original edges that correspond to the edges of these spanning trees.

Lemma 3.21. *The sparse certificate $C(G)$ has the same vertex-resilient blocks and 2-vertex-connected blocks as the input digraph G.*

Proof. We first argue that the execution of Algorithm FastVRB on $C(G)$ and produces the same vertex-resilient blocks as the execution of Algorithm FastVRB on G. The correctness of Algorithm FastVRB implies that it produces the same result regardless of the choice of start vertex s. So we assume that both executions choose the same start vertex s. We will refer to the execution of Algorithm FastVRB with input G (resp. $C(G)$) as FastVRB(G) (resp. FastVRB($C(G)$)).

First we note that $C(G)$ is strongly connected since it contains a spanning tree of $G(s)$ and a spanning tree for the reverse of each auxiliary graph G_r. Moreover, the fact that $C(G)$ contains two independent spanning trees of G implies that G and $C(G)$ have the same dominator tree with respect to the start vertex s that are computed in Step 1. Hence, the auxiliary graphs computed in Step 2 of Algorithm FastVRB have the same sets of ordinary and auxiliary vertices in both executions FastVRB(G) and FastVRB($C(G)$).

Hence, Step 3.1 computes the same dominator trees $D_H(r)$ and $D_H^R(r)$ in both executions, and therefore Steps 3.2 and 3.3 compute the same blocks. The same argument as in Steps 1 and 2 implies that both executions FastVRB(G) and FastVRB($C(G)$) compute in Step 3.4 auxiliary graphs H_q^R with the same sets of ordinary and auxiliary vertices. Finally, by construction, the strongly connected components of each auxiliary graph $H_q^R \setminus q$ are the same in both executions of FastVRB(G) and FastVRB($C(G)$).

We conclude that FastVRB(G) and FastVRB($C(G)$) compute the same vertex-resilient blocks as claimed. Next, observe that since the independent spanning trees computed in Steps 1 and 3.1 of the extended version of FastVRB are maximally edge-disjoint, $C(G)$ maintains the same strong bridges as G. Then, by Corollary 3.2, $C(G)$ also has the same 2-vertex-connected blocks as G. ∎

Chapter 4

Experimental evaluation

4.1 Introduction

In this Chapter we consider the computation of the 2-edge- and 2-vertex-connected blocks and components of a digraph in practice, and present efficient implementations of the algorithms introduced in this work, and also compare them to known algorithms [6, 24]. We also provide a new $O(mn)$-time algorithm for computing the 2-vertex-connected components of a digraph, that refines the dominator tree division used by Jaberi [24], and a simple $O(mn)$-time algorithm for computing the 2-edge-connected components of a digraph. We evaluate the efficiency of our algorithms experimentally on large digraphs taken from a variety of application areas. To the best of our knowledge, this is the first empirical study for these problems. Our extensive experimental study sheds light on the relative difficulty of computing various notions of 2-connectivity in directed graphs. More

specifically, we compare the performance of the linear-time algorithms for computing the 2-edge- and 2-vertex-connected blocks of a digraph with simpler algorithms that iterate over the strong articulation points and strong bridges of the digraph. We also consider the computation of the 2-vertex-connected components of a digraph and compare the performance of our new algorithm and the algorithms of Erusalimskii and Svetlov [6] and Jaberi [24]. Our results show that algorithms that apply a dominator-tree-based division of the input digraph perform well in practice and are more robust than their simpler competitors. The experimental results also suggest that the 2-edge- and 2-vertex-connected components of digraphs that arise in many practical applications can be found efficiently, despite the fact that the theoretically asymptotic bound of the algorithms we considered is $O(mn)$.

4.2 Overview of algorithms

In this section we overview the algorithms that were implemented for our experimental study, which are briefed in Table 4.1. To compute the 2-edge-connected blocks, we implemented the algorithms Simple2ECB and Fast2ECB that were presented in Chapter 2. We also implemented a fast algorithm that computes a sparse certificate for the 2-edge-connected blocks, i.e., a subgraph of the input graph that has $O(n)$ edges and maintains the same 2-edge-connected blocks as the input graph. This certificate is produced by extending Fast2ECB with the use of independent spanning trees [16], as detailed in Chapter 2. We refer to the resulting algorithm as SC2ECB.

We computed the vertex-resilient blocks by implementing algorithms Simple2VRB and Fast2VRB that we developed in Chapter 3. As previously mentioned, the vertex-resilient blocks are at the heart of the computation of the 2-vertex-connected blocks of a digraph: indeed, the 2-vertex-connected blocks of a digraph can be obtained by combining the computation of the vertex-resilient blocks and the computation of the 2-edge-connected blocks. For this reason, in our experiments we did not consider explicitly algorithms for computing 2-vertex-connected blocks. Similarly to the 2-edge-connectivity case, the fast algorithm for vertex-resilient blocks (Fast2VRB) can be extended with the use of independent spanning trees to obtain a sparse certificate for vertex-resilient and 2-vertex-connected blocks. This was not considered in our experiments. Also, we did not include the algorithms of Jaberi [23] for 2-edge- and 2-vertex-connected blocks because of their large requirements in storage space.

To compute the 2-edge-connected components, we implemented a new simple algorithm, called 2ECC, which repeatedly removes all strong bridges, and which will be described in detail in Section 4.2.1. To compute the 2-vertex-connected components, we implemented the algorithm by Erusalimskii and Svetlov [6], which we refer to as 2VCC-ES, and the two algorithms by Jaberi [24] which we refer to as 2VCC-J1 and 2VCC-J2. We also implemented a new algorithm, called 2VCC, which will be described in detail in Section 4.2.2.

Algorithm	Problem solved	Technique	Complexity	Reference
Simple2ECB	2-edge-connected blocks	Remove one strong bridge at a time	$O(mb)$	Section 2.2
Fast2ECB	2-edge-connected blocks	Dominator-tree division and auxiliary graphs	$O(m+n)$	Section 2.4
SC2ECB	Sparse certificate for 2-edge-connected blocks	Extend Fast2ECB using independent spanning trees	$O(m+n)$	Section 2.5
SimpleVRB	Vertex-resilient blocks	Remove one strong articulation point at a time	$O(mp)$	3
FastVRB	Vertex-resilient blocks	Dominator-tree division and auxiliary graphs	$O(m+n)$	3
2ECC	2-edge-connected components	Repeatedly remove all strong bridges	$O(mn)$	Section 4.2.1
2VCC-ES	2-vertex-connected components	Remove one vertex and the edges that connect different strongly connected components at a time	$O(m^2n)$	[6]
2VCC-J1	2-vertex-connected components	Remove one strong articulation point at a time	$O(mn)$	[24]
2VCC-J2	2-vertex-connected components	Dominator-tree division and induced subgraphs	$O(mn)$	[24]
2VCC	2-vertex-connected components	Refined dominator-tree division and induced subgraphs	$O(mn)$	Section 4.2.2

Table 4.1: An overview of the algorithms considered in our experimental study. The worst-case bounds refer to a digraph with n vertices, m edges, p strong articulation points, and b strong bridges. Note that $p \leq n$, $b \leq 2(n-1)$.

It can be shown that the $O(mn)$ bounds for all the 2-vertex- and 2-edge-connected components algorithms are tight (see Figure 4.1). This implies that there is currently a big gap between the $O(m+n)$ time bound for computing the 2-connected blocks, and the $O(mn)$ time bound for computing the 2-connected components. For the latter problems, the main additional difficulty encountered is due to the fact that the deletion of a strong articulation point (resp., a strong bridge) may cause other vertices to become strong articulation points (resp., strong bridges) in the remaining graph, as in the example considered in Figure 4.1. Very recently, and after this experimental study has been conducted, Henzinger et al. [20] showed how to compute the 2-edge- and the 2-vertex-connected components in $O(n^2)$ time using a hierarchical sparsification technique [19].

We refer the interested reader to the references given in Table 4.1 for a complete description of the algorithms considered. Before describing in detail the new Algorithms 2VCC and 2ECC, we observe that all the algorithms shown in Table 4.1 are roughly based on two different approaches: repeatedly removing strong articulation points (or strong bridges) and using dominator tree divisions. This will be briefly discussed below.

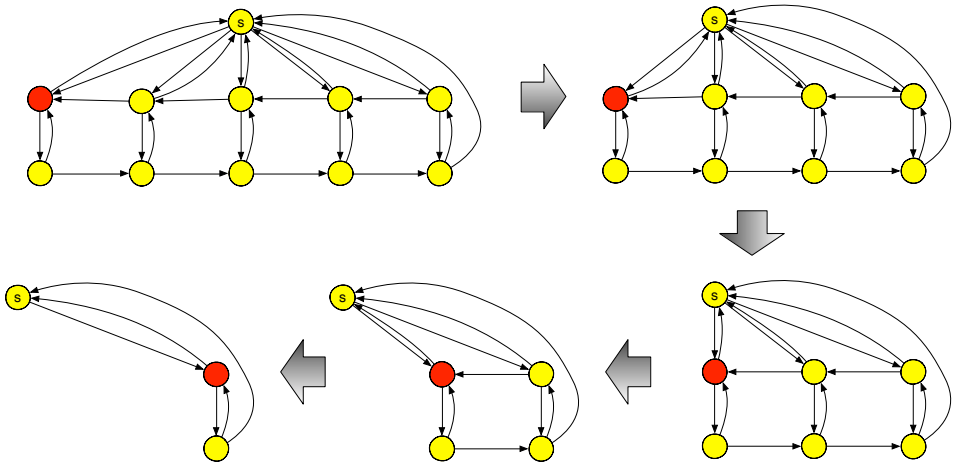


Figure 4.1: An example that elicits the worst-case behavior of the algorithms that compute the 2-vertex-connected components. The input digraph has only one strong articulation point (shown in red), and its removal creates one new strong articulation point. This process continues until only one vertex (s) remains. A similar example can be constructed for the case of 2-edge-connected components.

As mentioned in the Section 1.2.3, there are several linear-time algorithms for computing the dominator tree of a digraph [1, 3, 4, 8, 9, 15]. Despite this fact, in our experiments we used the simple version of the Lengauer-Tarjan algorithm [25] which runs in time $O(m\alpha(n, m/n))$ time, where α is a functional inverse of Ackermann's function [36]. We use the simple version of the Lengauer-Tarjan algorithm since it was reported to run faster in practice comparing to the other linear-time algorithms [7, 13, 14].

Vertex or edge removal. These algorithms remove one strong articulation point or one (or more) strong bridge(s) at a time. After removing such vertices or edges, the algorithms compute the strongly connected components of the remaining digraph, and update the division of vertices into components or blocks. This category includes algorithms SimpleE2ECB, 2ECC, SimpleVRB, 2VCC-ES and 2VCC-J1. Note that a significant difference between the computation of 2-vertex-connected components and 2-edge-connected components is that in the former we can only remove one strong articulation point (of the current digraph) at a time, while in the latter we can remove all strong bridges (of the current digraph) at once.

Dominator tree division. These are algorithms that use dominator trees to divide the input digraph into smaller graphs that maintain the desired relation (2-vertex- or 2-edge-connected components or blocks). This category includes algorithms Fast2ECB, SC2ECB, FastVRB, 2VCC-J2, and 2VCC. In the case of 2-vertex- or 2-edge-connected components, we only need to consider paths that contain vertices in the same division, so we use a dominator tree to divide the input digraph into induced subgraphs. For 2-vertex- or

Algorithm 2ECC: Computation of the 2-edge-connected components of a strongly connected digraph $G = (V, E)$ by removing strong bridges

Step 1: Compute the set B of the strong bridges of G. If B is empty then return V. (G is 2-edge-connected.)

Step 2: Compute the strongly connected components $S_1, \ldots, S_k$ of $G \setminus B$.

Step 3: For each strongly connected component S_i do:

Step 3.1: Compute the digraph G_i induced by S_i.

Step 3.2: Compute recursively the 2-edge-connected components of G_i.

Figure 4.2: Algorithm 2ECC

2-edge-connected blocks, we construct auxiliary graphs that take into account paths that contain vertices in other divisions. These are formed by augmenting induced subgraphs with auxiliary edges and vertices that correspond to such paths, as detailed in Chapters 2 and 3.

Finally, we note that algorithms 2ECC, 2VCC-J1, 2VCC-J2, and 2VCC are recursive.

4.2.1 Computing 2-edge-connected components

Here we describe a simple $O(mn)$-time algorithm that computes the 2-edge-connected components of a strongly connected digraph G. Our algorithm 2ECC, described in Figure 4.2, applies the definition of the 2-edge-connected components in terms of strong bridges, and uses the fact that the 2-edge-connected components are vertex-disjoint.

Lemma 4.1. *Algorithm 2ECC is correct.*

Proof. The algorithm clearly computes subgraphs of the input digraph $G = (V, E)$ that are 2-edge-connected. So we need to argue that each such subgraph is maximal. Let $C \subseteq V$ be the vertices in a 2-edge-connected component of G, which, by definition, is the induced subgraph $G(C) = (C, E(C))$ of G, where $E(C) = E \cap (C \times C)$. Since $G(C)$ is 2-edge-connected it is strongly connected and contains no strong bridges. This implies that Algorithm 2ECC does not remove any edge in $E(C)$. Therefore, Algorithm 2ECC does not partition $G(C)$ into smaller subgraphs. ∎

Lemma 4.2. *Algorithm 2ECC runs in $O(mn)$ time.*

Proof. By [22, 33], each recursive call runs in time that is linear in the total size of the input digraph. Since the 2-edge-connected components form a partition of the vertices, the depth of the recursion is n, and the total size of all induced subgraphs in each recursion level is $O(m)$. The bound follows. ∎

Algorithm 2VCC: Computation of the 2-vertex-connected components of a strongly connected digraph $G = (V, E)$ via dominator trees

Step 1: Choose an arbitrary start vertex $s \in V$. Compute the dominator trees $D(s)$ and $D^R(s)$.

Step 2: If $G \setminus s$ is strongly connected and $d(v) = d^R(v) = s$, for all vertices $v \neq s$, then return G. (G is 2-vertex-connected.)

Step 3: Compute the subgraphs $G(u, v)$ of G with at least three vertices.

Step 4: For each subgraph $G(u, v)$ with $u \neq v$ do:

Step 4a: Compute the strongly connected components of $G(u, v)$.

Step 4b: Compute recursively the 2-vertex-connected components of each strongly connected component.

Step 5: For each subgraph $G(v, v)$ do:

Step 5a: Compute the strongly connected components of $G(v, v) \setminus v$.

Step 5b: Process each strongly connected component S of $G(v, v) \setminus v$ as follows: If there are two arcs from v to S and two arcs from S to v then compute recursively the 2-vertex-connected components of the subgraph induced by $S \cup \{v\}$. Otherwise, compute recursively the 2-vertex-connected components of the subgraph induced by S.

Figure 4.3: Algorithm 2VCC

4.2.2 Computing 2-vertex-connected components

In this section we describe a new algorithm that computes the 2-vertex-connected components of a strongly connected digraph G. Algorithm 2VCC, which is a refinement of an algorithm by Jaberi [24], is illustrated in Figure 4.3. As Jaberi's algorithm, we use the dominator tree of $G(s)$, for an arbitrary start vertex s, to divide G into subgraphs that contain all the 2-vertex-connected components of G. The division is based on the following lemma, which is a restatement of a key lemma in [24]:

Lemma 4.3. *Let $G = (V, E)$ be a strongly connected digraph, and let $s \in V$ be an arbitrary start vertex. Any three vertices x, y and z (not necessarily distinct) belong to a common 2-vertex-connected component Σ of G only if they are all siblings in $D(s)$ or one is the immediate dominator of the other two in $G(s)$.*

Let $D(s)$ (resp., $D^R(s)$) be the dominator tree of $G(s)$ (resp., of the reverse digraph $G^R(s)$). We obtain a refined division of G into subgraphs using the dominator trees $D(s)$ and $D^R(s)$ concurrently. For any vertex v, let $C(v)$ (resp. $C^R(v)$) denote the set

of children of v in $D(s)$ (resp. $D^R(s)$). Also, let $d(v)$ (resp. $d^R(v)$) be the parent of $v \neq s$ in $D(s)$ (resp. $D^R(s)$). For any pair of vertices u and v we identify the vertices in $C(u,v) = C(u) \cap C^R(v)$. Also, if $u = v$ or $u \in C^R(v)$ then we include u in $C(u,v)$, and if $v \in C(u)$ then we include v in $C(u,v)$. Let $G(u,v)$ be the subgraph of G induced by the vertices in $C(u,v)$. We say that $G(u,v)$ is an *induced subgraph* of G.

Lemma 4.4. *Let x and y be any vertices in G such that they are in a 2-vertex-connected component Σ of G. Then x and y are vertices of a subgraph $G(u,v)$.*

Proof. We apply Lemma 4.3 to $G(s)$ and $G^R(s)$. Since $x, y \in \Sigma$, x and y are either siblings in $D(s)$, or $d(x) = y$ or $d(y) = x$. Also x and y are either siblings in $D^R(s)$, or $d^R(x) = y$ or $d^R(y) = x$. Now consider the relation between x and y in the dominator trees $D(s)$ and $D^R(s)$. We have the following cases:

(i) x and y are siblings in both $G(s)$ and $G^R(s)$. Then $d(x) = d(y)$ and $d^R(x) = d^R(y)$, so $\{x, y\} \subseteq C(d(x), d^R(x))$.

(ii) x and y are siblings in $G(s)$ and $d^R(x) = y$. Then $x \in C(d(x), y)$. But since $y \in C(d(x))$ we also have $y \in C(d(x), y)$.

(iii) $d(x) = y$ and $d^R(x) = y$. Then $x \in C(y, y)$, which, by definition, contains y.

(iv) $d(x) = y$ and $d^R(y) = x$. Since Σ has at least three vertices, consider a vertex $z \in \Sigma \setminus \{x, y\}$. By Lemma 4.3, vertex z can be neither a sibling of y nor the parent of y in $D(s)$. So z must be a sibling of x in $D(s)$. Similarly, we conclude that z is a sibling of y in $D^R(s)$. Hence $z \in C(y, x)$. But since $y \in C(d^R(x))$ and $x \in C(d(y))$, we also have $x, y \in C(y, x)$.

The remaining cases are analogous (with the role of x and y interchanged), so the lemma follows. ∎

Algorithm 2VCC, as described in Figure 4.3, applies Lemma 4.4 and the above division into subgraphs $G(u,v)$.

Lemma 4.5. *Algorithm 2VCC is correct.*

Proof. Lemma 4.4 implies that every 2-vertex-connected component of G is a subgraph of an induced subgraph $G(u,v)$. Since each $G(u,v)$ is a subgraph of G it cannot contain a 2-vertex-connected subgraph H that is not a subgraph of G. Therefore, the induced subgraphs $G(u,v)$ maintain the same 2-vertex-connected components as the original digraph G. ∎

Next we bound the running time of our algorithm. First, we provide a bound on the size of all induced subgraphs $G(u,v)$.

Lemma 4.6. *The induced subgraphs $G(u,v)$ have at most $4n-3$ vertices and m edges in total.*

Proof. Any vertex $x \neq s$ appears in at most four sets $C(u, v)$, namely $C(x, x)$, $C(d(x), d^R(x))$, $C(d(x), x)$, and $C(x, d^R(x))$. Vertex s can only appear in $C(s, s)$, so the total size of all sets $C(u, v)$ is at most $4n - 3$. The bound on the total number of edges follows from the claim that any two distinct vertices x and y can appear together in at most one set $C(u, v)$. To prove this claim, assume, without loss of generality, that $y \neq d(x)$. (If $y = d(x)$, switch the role of x and y.) By the definition of $C(u, v)$, one of the following happens: (i) x and y are siblings in both $D(s)$ and $D^R(s)$, (ii) $d(y) = x = u$ and x and y are siblings in $D^R(s)$, (iii) $d(y) = x = u$ and $d^R(x) = y = v$, or (iv) $d(y) = d^R(y) = x$. For any pair of vertices x and y at most one of the above cases applies, which proves the claim. ∎

Given the dominator trees $D(s)$ and $D^R(s)$, we can compute all sets $C(u, v)$ of size at least three, as follows. We number the vertices in $D(s)$ and $D^R(s)$ in preorder. Let $pre(v)$ ($pre^R(v)$) be the preorder number of a vertex $v \in D(s)$ ($D^R(s)$). We label each vertex v with the pair $\langle pre(d(v)), pre^R(d^R(v)) \rangle$. We can sort the labels lexicographically in $O(n)$ time by radix sort. Then we group the vertices with identical labels. If there is at least one vertex with label $\langle pre(u), pre^R(v) \rangle$ we also test if $d(v) = u$, in which case we include v in $C(u, v)$, and test if $d^R(u) = v$, in which case we include u in $C(u, v)$. As we discover the distinct labels $\langle pre(u), pre^R(v) \rangle$ we number the corresponding sets $C(u, v)$ in increasing order. We use these numbers in order to partition the adjacency list of each vertex, which gives a representation of the $G(u, v)$ subgraphs. The correctness of this method follows by Lemma 4.4.

Lemma 4.7. *We can compute all induced subgraphs $G(u, v)$ with at least three vertices in $O(m + n)$ time.*

Lemma 4.8. *Algorithm 2VCC runs in $O(mn)$ time.*

Proof. By [3, 33] and Lemma 4.7, each recursive call runs in time that is linear in the size (number of vertices and edges) of the input digraph. The depth of the recursion is at most n, and the total size of all subgraphs constructed in Step 3 in each recursion level is $O(m)$ by Lemma 4.6. The bound follows. ∎

4.3 Empirical analysis

For the experimental evaluation we use the same graph datasets as in [14], shown in Table 4.2, and process only the largest strongly connected component (SCC) of each graph. We wrote our implementations in `C++`, using `g++ v.4.6.4` with full optimization (flag `-O3`) to compile the code. We report the running times on a GNU/Linux machine, with Ubuntu (12.04LTS): 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

Dataset	n	m	file size	δ_{avg}	p	b	type
rome99	3.3k	8.8k	98k	2.64	0.8k	1.4k	road network
p2p-gnutella25	5.1k	17.9k	199k	3.48	1.9k	2.1k	peer2peer
Oracle-16k	10.4k	29,9k	320k	2.88	2.4k	12.4k	memory profiling
s38584	16.3k	26.0k	321k	1.59	10.5k	16.4k	circuit
web-NotreDame	48.7k	267k	3.4M	5.49	9.0k	31.1k	web graph
soc-Epinions1	32.2k	442k	5.1M	13.74	8.1k	20.9k	social network
USA-road-NY	264k	730k	11M	2.76	46.4k	105k	road network
USA-road-BAY	321k	794k	12M	2.47	84.6k	197k	road network
Amazon0302	241k	1.1M	17M	4.67	69.6k	73.3k	prod. co-purchase
wiki-Talk	111k	1.4M	18M	12.93	14.8k	85.5k	social network
web-Stanford	150k	1.5M	22M	10.47	20.2k	64.6k	web graph
Amazon0601	395k	3.3M	48M	8.35	69.3k	83.9k	prod. co-purchase
web-BerkStan	334k	4.5M	66M	13.50	53.6k	164k	web graph
Oracle-4M	2.8M	8.4M	137M	2.95	1.0M	1.3M	memory profiling
SAP-4M	4.0M	11.9M	181M	2.91	1.9M	4.4M	memory profiling
Oracle-11M	6.4M	15.9M	261M	2.47	3.1M	6.9M	memory profiling
SAP-11M	11.1M	36.3M	673M	3.26	4.9M	12.3M	memory profiling
LiveJournal	3.8M	65.3M	1G	17.06	649k	1.3M	social network
USA-road	23.9M	57.7M	1.1G	2.40	6.2M	14.5M	road network

Table 4.2: Real-world graphs sorted by file size; n is the number of vertices, m the number of edges, and δ_{avg} is the average vertex degree; p and b denote, respectively, the number of strong articulation points and strong bridges. All characteristics refer to the largest SCC of each graph

core. We report CPU times measured with the `getrusage` function. All the running times reported in our experiments were averaged over ten different runs.

In addition to the running time for the algorithms of Table 4.1, we also report the running time of the (simple) Lengauer-Tarjan algorithm for computing dominators, which we refer to as LT. We use this as a baseline, since we can compute efficiently both the strong bridges and the strong articulation points of a digraph using dominators [22]. Moreover, as another reference baseline, we provide the running time for executing a depth-first search (DFS) traversal of the graphs. (We note that LT also uses DFS.)

4.3.1 2-connectivity structure of the considered digraphs

Tables 4.3 and 4.4 provide some statistics about the number and size of the 2-vertex- and 2-edge-connected components and blocks of the digraphs in our collection, where the size is measured as the number of vertices. We will use these data in order to interpret the performance of the tested algorithms. Recall that a 2-edge- or a 2-vertex-connected block has at least two vertices, while a 2-edge- or a 2-vertex-connected component has

	2-edge-connected blocks			2-edge-connected components		
Graph	Max Size	Avg. Size	#	Max Size	Avg. Size	#
rome99	2543	2543	1	2255	566.5	4
p2p-gnutella25	3116	3116	1	–	–	0
Oracle-16k	738	61.38	13	24	12	4
s38584	427	47.23	13	–	–	0
web-NotreDame	14749	32.31	762	3780	35.32	481
soc-Epinions1	18046	260.89	70	17512	300.35	59
USA-road-NY	207128	299.08	710	207128	299.08	710
USA-road-BAY	212199	149.15	1501	212199	149.15	1501
Amazon0302	140200	24.57	7283	81423	12	12874
wiki-Talk	50335	8391.33	6	49503	16503	3
web-Stanford	58599	76.89	1224	21767	29.12	1708
Amazon0601	305850	87.75	3730	296281	75.14	4300
web-BerkStan	128156	64.4	2930	56166	27.84	4744
Oracle-4M	434386	1408.02	1192	64397	1560.86	535
SAP-4M	141521	39.59	5325	2501	15.1	1883
Oracle-11M	389352	18.98	44921	3591	10.22	44127
SAP-11M	751793	36.66	25774	6340	20.12	1479
LiveJournal	2931062	747.73	3950	2914807	779.16	3771
USA-road	16051070	158.7	105704	16051070	158.7	105704

Table 4.3: Size (maximum and average) and number of the 2-edge-connected blocks and components. The size of a block or a component is measured as the number of its vertices.

	vertex-resilient blocks			2-vertex-connected blocks			2-vertex-connected components		
Graph	Max Size	Av. Size	#	Max Size	Av. Size	#	Max Size	Av. Size	#
rome99	2542	5.29	771	2542	1272.00	2	2249	453.39	5
p2p-gnutella25	3113	4.49	1246	3113	779.75	4	–	–	0
Oracle-16k	581	2.24	2548	581	6.01	156	24	8.2	5
s38584	375	2.38	1378	375	12.07	53	–	–	0
web-NotreDame	5381	3.07	19223	5381	8.78	3053	1462	19.53	847
soc-Epinions1	17560	3.41	12626	17560	53.12	349	17113	84.83	210
USA-road-NY	206871	5.94	53474	206871	273.73	776	206871	273.73	776
USA-road-BAY	211590	4.19	100513	211590	138.09	1622	211590	138.09	1622
Amazon0302	123592	4.11	74848	123592	13.83	13375	55414	7.81	19789
wiki-Talk	50187	2.93	53551	50187	438.75	115	49427	1768.39	28
web-Stanford	26194	3.91	40776	26194	12.19	7668	10893	16.41	2936
Amazon0601	287619	6.02	78139	287619	37.83	8784	276049	35.02	9340
web-BerkStan	64022	3.37	109112	64022	10.34	17984	29145	15.69	8104
Oracle-4M	129071	3.51	710717	129071	3.80	596741	64397	1560.86	535
SAP-4M	119712	2.64	271914	119712	8.71	26219	2501	15.1	1883
Oracle-11M	283036	2.45	1471583	283036	6.91	135601	3591	9.58	47430
SAP-11M	640932	3.04	752260	640932	8.31	124349	6340	20.12	1479
LiveJournal	2882722	5.39	862546	2882722	122.78	24219	2868808	153.7	19202
USA-road	16019892	4.24	7390323	16019892	148.80	112780	16019892	148.80	112780

Table 4.4: Size (maximum and average) and number of the vertex-resilient blocks and the 2-vertex-connected blocks and components. The size of a block or a component is measured as the number of its vertices.

at least three vertices. (A 2-edge-connected component may have only two vertices if there are parallel edges.) As it can be seen from Tables 4.3 and 4.4, the size and number of components and blocks varies, but for most digraphs the difference between the total number and average size of the 2-edge-connected blocks and that of the 2-edge-connected components is not too large. The same holds for the 2-vertex-connected blocks and components. The vertex-resilient blocks, on the other hand, are much more numerous than the 2-vertex-connected blocks and components, and consequently their average size is smaller.

Note that, in particular, for the USA road networks (USA-road-NY, USA-road-BAY, and USA-road), their 2-edge-connected blocks are identical to their 2-edge-connected components, and, similarly, their 2-vertex-connected blocks are identical to their 2-vertex-connected components. This is due to the fact the USA road networks in our data set are essentially undirected graphs: for every edge (u, v), there is the opposite directed edge (v, u) as well. As we will see later (see Tables 4.5 and 4.6), this has implications also on the practical performance of our algorithms. Indeed, for the USA road networks the simpler $O(mn)$ algorithms (2VCC, 2ECC) for computing components run faster than the more complicated $O(m)$ algorithms (FastVRB, Fast2ECB) for computing blocks. This is due to the fact that in the undirected road networks the simple algorithms (2VCC, 2ECC) have a very small recursion depth, as it will be shown later (see Table 4.7).

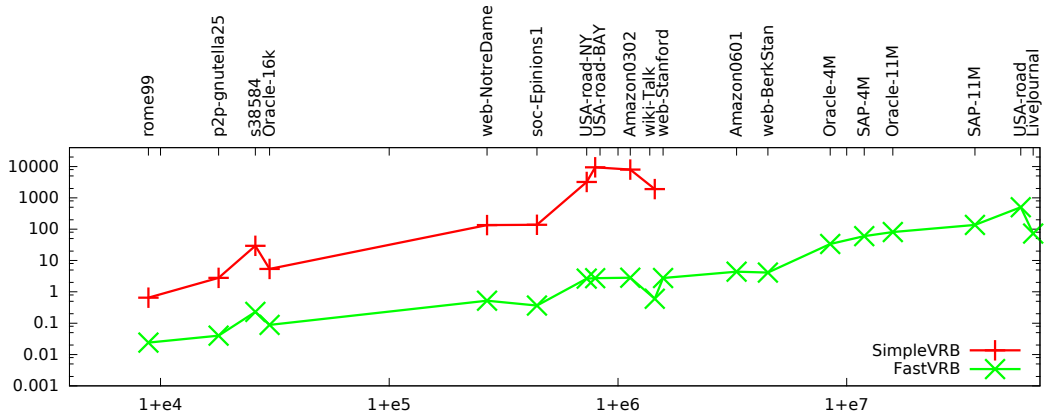


Figure 4.4: Vertex-resilient blocks. Running times, in seconds, and number of edges shown in log scale.

Graph	2VCC-ES	2VCC-J1	2VCC-J2	2VCC	SimpleVRB	FastVRB	DFS	LT
rome99	10.71	2.88	0.07	0.04	0.64	0.02	0.01	0.01
p2p-gnutella25	10.78	12.86	0.06	0.03	2.78	0.04	0.01	0.01
Oracle-16k	30.85	0.23	0.02	0.02	5.38	0.08	0.01	0.01
s38584	24.25	2.70	0.03	0.02	29.62	0.22	0.01	0.01
web-NotreDame	1528.75	70.30	0.65	0.29	135.28	0.52	0.01	0.05
soc-Epinions1	1629.30	653.93	1.30	0.77	137.74	0.36	0.01	0.04
USA-road-NY	12052.07	13779.92	0.64	0.92	3225.83	2.65	0.03	0.16
USA-road-BAY	14134.73	16059.15	0.89	1.09	9401.70	2.73	0.05	0.18
Amazon0302	>12h	14514.38	8.17	7.51	>12h	2.81	0.07	0.27
wiki-Talk	18119.10	3567.23	3.52	2.84	>12h	0.60	0.05	0.19
web-Stanford	>12h	910.86	4.22	2.14	>12h	2.76	0.06	0.22
Amazon0601	>12h	81021.47	17.33	13.91	>12h	4.40	0.15	0.51
web-BerkStan	>12h	12025.61	5.50	6.22	>12h	4.13	0.12	0.40
Oracle-4M	>12h	1055.93	49.78	8.53	>12h	33.61	0.34	1.60
SAP-4M	>12h	244.37	16.29	9.86	>12h	60.02	0.46	2.17
Oracle-11M	>12h	8494.18	154.65	16.48	>12h	80.98	0.61	3.06
SAP-11M	>12h	21907.76	44.75	18.59	>12h	136.19	0.75	6.01
LiveJournal	>12h	>12h	757.21	491.47	>12h	72.35	3.22	16.20
USA-road	>12h	>12h	107.72	68.85	>12h	499.70	3.71	17.34

Table 4.5: Experimental comparison of algorithms for computing the 2-vertex-connected components and the vertex-resilient blocks; running time are in seconds, codes running longer than 12 hours were terminated.

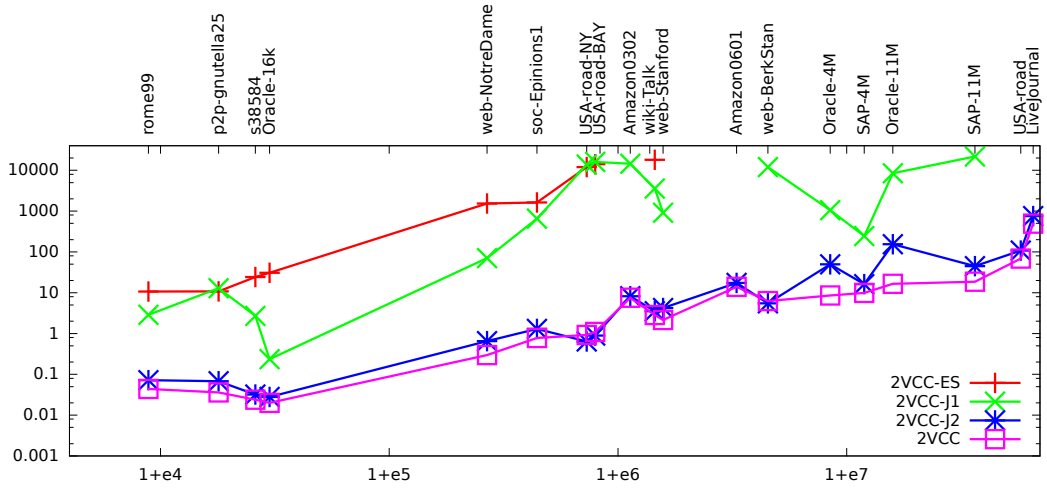


Figure 4.5: 2-vertex-connected components. Running times, in seconds, and number of edges shown in log scale.

4.3.2 Vertex-resilient blocks

Here we compare the performance of two algorithms, SimpleVRB and FastVRB, for computing the vertex-resilient blocks of a digraph. Figure 4.4 gives a plot (in log-log scale) of the corresponding running times shown in Table 4.5. As it can be observed, SimpleVRB is not competitive even for the smallest graphs in our data set. We also notice from Figure 4.4 that FastVRB scales quite nicely with the graph size.

4.3.3 2-vertex-connected components

We evaluate the performance of four algorithms that compute the 2-vertex-connected components of a digraph: the algorithm of Erusalimskii and Svetlov [6], 2VCC-ES, two algorithms, 2VCC-J1 and 2VCC-J2, proposed by Jaberi [24], and our new algorithm, 2VCC, described in Figure 4.3. The results are shown in the first four columns of Table 4.5. As expected, 2VCC-ES is not a viable approach for large graphs. Algorithm 2VCC-J1 performs better than 2VCC-ES, but still it does not scale well with the graph size and it is not competitive with the fastest algorithms. Algorithms 2VCC-J2 and 2VCC, on the other hand, work well in practice, with the latter being the clear winner; 2VCC is faster than 2VCC-J2 by a factor close to 2 on average. In fact for two graphs (Oracle-4M and Oracle-11M) it performs significantly better. This is due to the fact that 2VCC reduces the recursion depth, sometimes substantially, by using the refined dominator-tree division, and by treating the strongly connected components of the induced subgraphs in a more efficient way. We will investigate this behavior more closely later (see Table 4.7).

4.3.4 2-edge-connected blocks

Now we evaluate two algorithms, Simple2ECB and Fast2ECB, for computing the 2-edge-connected blocks of a digraph. The results are shown in Table 4.6, and Figure 4.6 gives

Graph	Simple2ECB	Fast2ECB	SC2ECB	2ECC	DFS	LT
rome99	1.96	0.01	0.01	0.02	0.01	0.01
p2p-gnutella25	4.57	0.01	0.01	0.01	0.01	0.01
Oracle-16k	39.78	0.03	0.04	0.01	0.01	0.01
s38584	82.12	0.04	0.05	0.01	0.01	0.01
web-NotreDame	323.88	0.14	0.22	0.24	0.01	0.05
soc-Epinions1	409.58	0.17	0.32	0.34	0.01	0.04
USA-road-NY	16018.53	0.48	0.91	0.58	0.03	0.16
USA-road-BAY	24306.32	0.70	1.14	0.63	0.04	0.18
Amazon0302	>12h	0.59	1.66	3.44	0.07	0.27
wiki-Talk	>12h	0.66	1.32	1.44	0.04	0.18
web-Stanford	>12h	0.75	1.39	2.24	0.06	0.22
Amazon0601	>12h	1.78	3.82	4.98	0.15	0.51
web-BerkStan	>12h	1.43	2.81	2.36	0.12	0.40
Oracle-4M	>12h	7.30	11.05	2.78	0.34	1.60
SAP-4M	>12h	19.02	24.50	5.26	0.46	2.17
Oracle-11M	>12h	18.07	24.12	5.16	0.61	3.06
SAP-11M	>12h	44.43	54.90	14.84	0.75	6.01
LiveJournal	>12h	46.23	83.62	163.77	3.22	16.20
USA-road	>12h	68.49	107.60	43.08	3.71	17.34

Table 4.6: Experimental comparison of algorithms for computing the 2-edge-connected components and blocks; running time are in seconds, codes running longer than 12 hours were terminated.

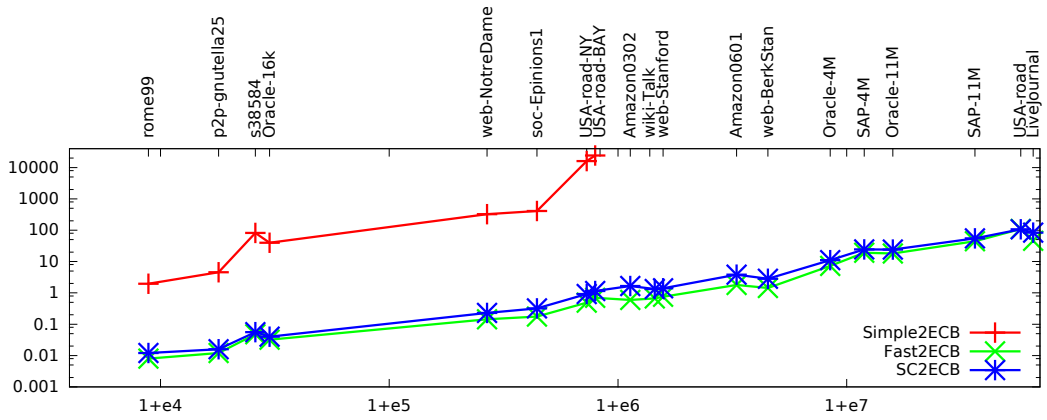


Figure 4.6: 2-edge-connected blocks. Running times, in seconds, and number of edges shown in log scale.

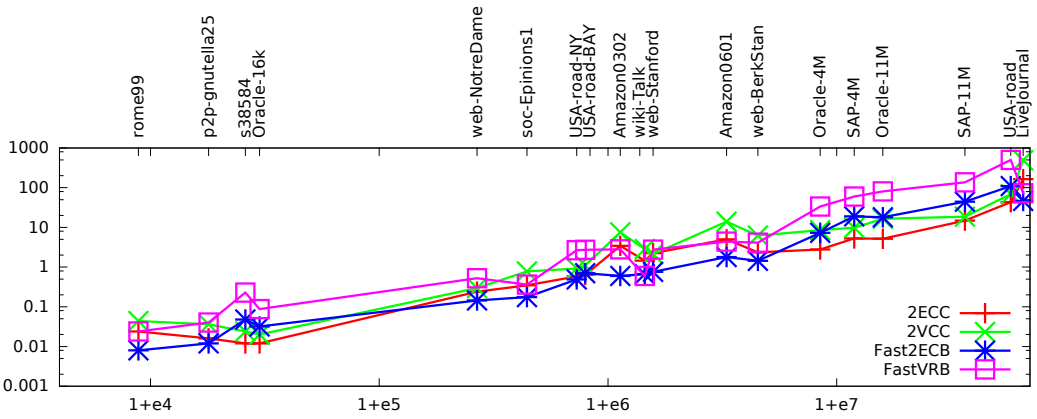


Figure 4.7: Best algorithms for each problem. Running times, in seconds, and number of edges shown in log scale.

the corresponding plot. We observe a behavior analogous to the case of the vertex-resilient blocks. Specifically, algorithm Fast2ECB outperforms Simple2ECB by at least three orders of magnitude. The running time of Simple2ECB depends on the number of strong bridges in the input graph, and it does not scale well with the graph size. We also evaluate the overhead of constructing a sparse certificate for the 2-edge-connected blocks. Algorithm SC2ECB is an extended version of Fast2ECB that computes, in addition to the 2-edge-connected blocks, such a sparse certificate. The running time of SC2ECB is within a factor of 1.67 on average compared to Fast2ECB. The quality of the sparse certificate is measured in the second column of Table 4.7, and, as expected, it depends on the average degree δ_{avg} of the input digraph. (The percentage of the edges that are included in the certificate decreases with δ_{avg}.)

4.3.5 2-edge-connected components

Now we compare the performance of algorithm 2ECC with the best algorithms for computing the 2-edge-connected blocks and the 2-vertex-connected components and blocks, in order to get a view of the relative difficulty of computing the various notions of 2-vertex- and 2-edge connectivity in a digraph. As it can be seen from Figure 4.7, which shows a plot of the corresponding running times, all algorithms have close performance. This is somewhat surprising, since there is a big asymptotical gap: the 2-vertex- and 2-edge-connected components are computed in $O(mn)$ time, while the 2-vertex- and 2-edge-connected blocks are computed faster in $O(m+n)$ time.

In order to take a closer look at this phenomenon, we show in Table 4.7 the number of recursive calls and the recursion depth for the above algorithms. It can be observed that the recursion depth achieved by algorithms 2ECC and 2VCC is remarkably low. This behavior can be explained in light of the digraph statistics discussed in Section 4.3.1.

Now we compare the recursive calls performed by 2VCC-J2 and 2VCC. Algorithm 2VCC-J2 requires more recursive calls, but performs less work per call, so it remains competitive. There are two reasons that justify the large difference in the recursion depth

		Number of recursive calls			Recursion depth		
Graph	% size SC (m)	2VCC	2VCC-J2	2ECC	2VCC	2VCC-J2	2ECC
rome99	89.46%	23	39	17	9	11	10
p2p-gnutella25	74.53%	7	9	7	5	8	5
Oracle-16k	68.88%	75	277	39	8	33	10
s38584	89.00%	1	1	1	1	1	1
web-NotreDame	50.89%	2834	12598	1042	13	153	22
soc-Epinions1	23.32%	848	4212	69	4	61	3
USA-road-NY	84.99%	777	7777	711	1	3	1
USA-road-BAY	88.23%	1623	17510	1502	1	4	1
Amazon0302	61.47%	32605	36013	16534	17	27	11
wiki-Talk	24.50%	3797	33290	8	4	444	4
web-Stanford	29.03%	6840	24952	2421	15	180	20
Amazon0601	40.14%	15774	19858	4626	6	9	6
web-BerkStan	22.60%	20982	59789	6814	18	248	22
Oracle-4M	81.72%	10714	311902	1801	3	793	3
SAP-4M	54.28%	16521	35370	7209	6	248	4
Oracle-11M	69.76%	90766	1100721	50446	21	808	22
SAP-11M	49.53%	74564	92026	31338	4	251	4
LiveJournal	20.24%	47534	221321	4144	6	265	7
USA-road	88.52%	112781	1273500	105705	1	6	1

Table 4.7: Some algorithm statistics: Total number of recursive calls and recursion depth. The second column gives the percentage of the edges that are included in the sparse certificate by SC2ECB.

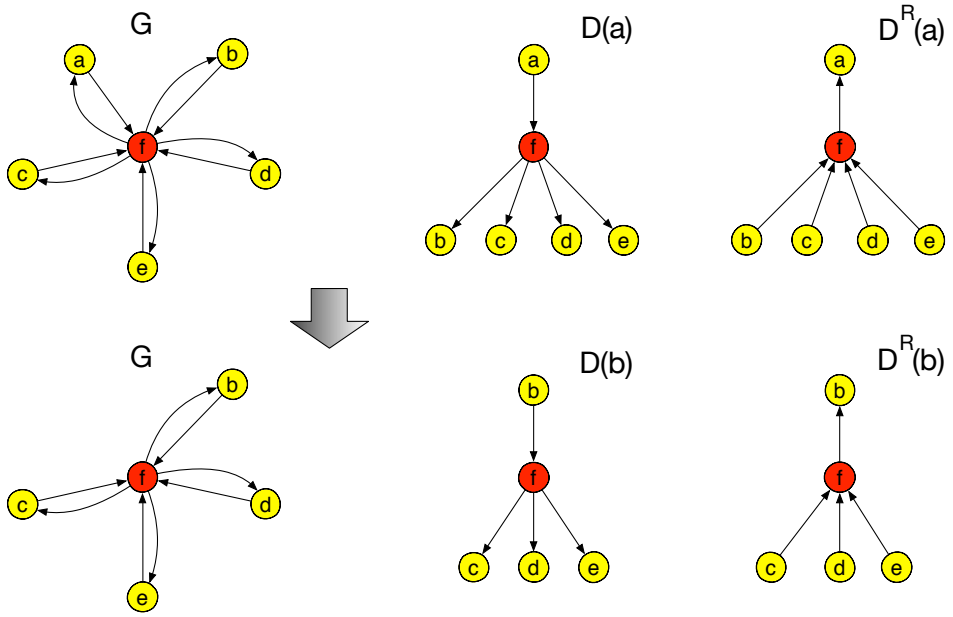


Figure 4.8: An input digraph that elicits $O(n)$ recursion depth for Algorithm 2VCC-J2. Algorithm 2VCC, on the other hand, requires only one recursive call.

between algorithms 2VCC and 2VCC-J2. The first one, as already mentioned, is that 2VCC uses a refined division that uses the dominator tree of both the forward and the reverse digraph. The second reason has to do with the way these two algorithms compute the subgraphs that are used as inputs in the recursive calls. More specifically, Algorithm 2VCC-J2 uses a dominator tree $D(s)$ (of the forward or the reverse digraph) to partition the graph G into subgraphs and perform the recursive calls. For each strong articulation point u in $D(s)$, 2VCC-J2 computes the strongly connected components of the subgraph induced by $C(u) \cup u$ (i.e., u and its children in $D(s)$) and executes recursively on each such strongly connected component. The problem with this approach is that it does not take into account the connectivity between the vertices in $C(u)$; all the paths between vertices in $C(u)$ may go through u. Such a bad instance is illustrated in Figure 4.8. In this example, digraph G has no 2-vertex-connected components but 2VCC-J2 discovers this after $n - 2$ recursive calls. Algorithm 2VCC, on the other hand, uses a different approach (in Steps 4 and 5) that yields a much smaller number of recursion calls and recursion depth.

Next we compare the performance of the fastest algorithms for each task with our baseline algorithm LT. Compared to the baseline, 2ECC and Fast2ECB are slower on average by a factor of 5.35 and 4.61, respectively. Algorithms 2VCC and FastVRB are slower on average than LT by a factor of 11.13 and 16.79, respectively. This confirms experimentally the intuition that the vertex connectivity problems are more complicated. The fact that FastVRB is the slowest among these four algorithms can be attributed to two facts. First, it constructs more complicated auxiliary graphs than Fast2ECB. Second, the vertex-resilient (and the 2-vertex-connected) blocks have a more complex structure

than the 2-edge-connected blocks. In order to allow fast operations on the vertex-resilient blocks, FastVRB maintains them in a forest data structure. This incurs some significant overhead, since, as shown in Table 4.4, the input digraphs have a much higher number of vertex-resilient blocks.

Chapter 5

Conclusion

In this master thesis we have studied 2-connectivity problems in directed graphs. In particular, we have presented two linear-time algorithms for computing the 2-*edge-connected blocks* and the 2-*vertex-connected blocks* relations among vertices. These two algorithms are not only theoretically optimal, but also improve significantly over previous bounds.

In the case of the 2-edge-connectivity. Once the 2-edge-connected blocks of a digraph G are available, it is straightforward to check in constant time if any two vertices are 2-edge-connected. Moreover, in the case of the 2-vertex connectivity, we showed how to represent these relations with a data structure of $O(n)$ size, so that it is also straightforward to check in constant time if any two vertices are vertex-resilient or 2-vertex-connected. Moreover, if the answer to such a query is negative, then we can provide a witness of this fact in constant time, i.e., a vertex (strong articulation point) or an edge (strong bridge) of G that separates the two query vertices. Furthermore, we showed how to compute a *sparse certificate* for 2-edge-connected and the 2-vertex connected blocks, i.e., a subgraph of the input graph that has $O(n)$ edges and maintains the same 2-edge-connected 2-vertex connected blocks, respectively, as the input graph.

For the 2-vertex-connected components we introduced an algorithm that divides the input digraph into induced digraphs using two dominator trees (one for the original digraph and one for its reversal). For the 2-edge-connected components we presented a simple algorithm that removes all strong bridges from the current digraph at a time.

We conducted an extensive experimental study of algorithms that compute the 2-vertex- and 2-edge-connected components and blocks. The algorithms we tested fall into two broad categories: algorithms that remove one strong articulation point or one (or more) strong bridge(s) at a time, and algorithms that use a dominator-tree-based division of the input digraph. The former includes $O(mn)$-time algorithms for all tasks, while the latter includes two linear-time algorithms for computing the 2-vertex- and 2-edge-connected blocks, and two $O(mn)$-time algorithms for computing the 2-vertex-connected components. Our experimental results showed that the dominator-tree-based algorithms perform well in practice and are more robust. The results also suggest that the 2-vertex- and 2-edge-connected components of digraphs that arise in many practical applications

can be found efficiently, despite the fact that the theoretically asymptotic bound of the algorithms that we considered is $O(mn)$. The best practical performances for both these problems were achieved by the two new algorithms.

We leave as an open question if the 2-edge-connected or the 2-vertex-connected components of a digraph can be computed in linear time. The best current bound for both problems is $O(n^2)$.

Bibliography

[1] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.

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

[3] 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*, 38(4):1533–1573, 2008.

[4] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–96, 1998. Corrigendum in 27(3):383-7, 2005.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1991.

[6] Ya. M. Erusalimskii and G. G. Svetlov. Bijoin points, bibridges, and biblocks of directed graphs. *Cybernetics*, 16(1):41–44, 1980.

[7] D. Firmani, G. F. Italiano, L. Laura, A. Orlandi, and F. Santaroni. Computing strong articulation points and strong bridges in large scale graphs. In *Proc. 10th Int'l. Symp. on Experimental Algorithms*, pages 195–207, 2012.

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

[9] H. N. Gabow. A poset approach to dominator computation. Unpublished manuscript 2010, revised unpublished manuscript, 2013.

[10] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–21, 1985.

[11] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. In *Proc. 26th ACM-SIAM Symp. on Discrete Algorithms*, pages 1988–2005, 2015.

[12] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *Proc. 42th Int'l. Coll. on Automata, Languages, and Programming*, 2015.

[13] 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*, pages 284–295, 2013.

[14] L. Georgiadis, L. Laura, N. Parotsidis, and R. E. Tarjan. Loop nesting forests, dominators, and applications. In *Proc. 13th Int'l. Symp. on Experimental Algorithms*, pages 174–186, 2014.

[15] L. Georgiadis and R. E. Tarjan. Finding dominators revisited. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms*, pages 862–871, 2004.

[16] L. Georgiadis and R. E. Tarjan. Dominator tree certification and independent spanning trees. *CoRR*, abs/1210.8303, 2012.

[17] L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications (JGAA)*, 10(1):69–94, 2006.

[18] Y. Guo, F. Kuipers, and P. Van Mieghem. Link-disjoint paths for reliable qos routing. *International Journal of Communication Systems*, 16(9):779–798, 2003.

[19] M. Henzinger, V. King, and T. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999.

[20] M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. 42nd International Colloquium on Automata, Languages, and Programming (ICALP 2015)*, 2015.

[21] A. Itai and M. Rodeh. The multi-tree approach to reliability in distributed networks. *Information and Computation*, 79(1):43–59, 1988.

[22] G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447(0):74–84, 2012.

[23] R. Jaberi. Computing the 2-blocks of directed graphs. *CoRR*, abs/1407.6178, 2014.

[24] R. Jaberi. On computing the 2-vertex-connected components of directed graphs. *CoRR*, abs/1401.6000, 2014.

[25] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–41, 1979.

[26] W. Di Luigi, L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-connectivity in directed graphs: An experimental study. In *Proc. 17th Wks. on Algorithm Engineering and Experiments*, pages 173–187, 2015.

[27] D. W. Matula and R. V. Vohra. Calculating the connectivity of a directed graph. Technical Report 386, Institute for Mathematics and Application, University of Minnesota, 1988.

[28] K. Menger. Zur allgemeinen kurventheorie. *Fund. Math.*, 10:96–115, 1927.

[29] H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.

[30] H. Nagamochi and T. Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Cambridge University Press, 2008. 1st edition.

[31] H. Nagamochi and T. Watanabe. Computing k-edge-connected components of a multigraph. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E76–A(4):513—-517, 1993.

[32] J. H. Reif and P. G. Spirakis. Strong k-connectivity in digraphs and random digraphs. Technical Report TR-25-81, Harvard University, 1981.

[33] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[34] R. E. Tarjan. Edge-disjoint spanning trees, dominators, and depth-first search. Technical report, Stanford University, Stanford, CA, USA, 1974.

[35] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

[36] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[37] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–85, 1976.

[38] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(5&6):433–464, 1992.

Author's Publications

1. W. Di Luigi, L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-Connectivity in Directed Graphs: An Experimental Study. In *Proceedings of the 17th SIAM Meeting on Algorithm Engineering and Experimentation*, pages 173-187, 2015.

2. L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge Connectivity in Directed Graphs. In *Proceedings of the 26th ACM-SIAM Symposium on Discrete Algorithms*, pages 1988-2005, 2015.

3. L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *Proceedings of the 42th International Colloquium on automata, Languages, and Programming*, 2015.

4. L. Georgiadis, L. Laura, N. Parotsidis, and R. E. Tarjan. Dominator certification and independent spanning trees: An experimental study. In *Proceedings of the 12th International Symposium on Experimental Algorithms*, pages 284–295, 2013.

5. L. Georgiadis, L. Laura, N. Parotsidis, and R. E. Tarjan. Loop nesting forests, dominators, and applications. In *Proceedings of the 13th International Symposium on Experimental Algorithms*, pages 174–186, 2014.

6. N. Parotsidis, and L. Georgiadis. Dominators in Directed Graphs: A Survey of Recent Results, Applications, and Open Problems. In *Proceedings of the 2nd International Symposium on Computing in Informatics and Mathematics*, pages 15–20, 2013.

7. N. Parotsidis, E. Pitoura, and P. Tsaparas. Selecting Shortcuts for a Smaller World. In *Proceedings of the 15th SIAM International Conference on Data Mining, 2015.*

Short Vita

Nikos Parotsidis holds a BSc degree (2013) in Computer Science from the Department of Computer Science & Engineering, University of Ioannina, Greece. His research interests are focused, but not limited, to the design and analysis of algorithms, algorithms engineering, graph theory, complexity theory, approximation algorithms, and algorithmic data mining. Nikos has been an assistant in the laboratories of the undergraduate course on Data Structures (Fall 2014 and Fall 2015) at the Department of Computer Science & Engineering, University of Ioannina. He has also participated in the Marie Curie Reintegration project JMUGCS (Jointly Mining User Generated Content Sources).