Low-High Orders of Directed Graphs: Incremental Algorithms and Applications

Η Μεταπτυχιακή Εργασία Εξειδίκευσης

υποβάλλεται στην ορισθείσα

από τη Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Μηχανικών Η/Υ και Πληροφορικής

Εξεταστική Επιτροπή

από την

Αικατερίνη Καρανάσιου

ως μέρος των υποχρεώσεων για την απόκτηση του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΗΝ ΘΕΩΡΙΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Πανεπιστήμιο Ιωαννίνων

Σεπτέμβριος 2016

Acknowledgements

I would like to express my gratitude to my supervisor, Assistant Prof. Loukas Georgiadis for the patient guidance, encouragement and advice he has provided throughout my time as his student. His positive outlook in my research inspired me and gave me confidence.

My sincere gratitude goes to my co-authors, Giuseppe F. Italiano, Giannis Konstantinos, Luigi Laura, Charis Papadopoulos, and Nikos Parotsidis, for their valuable collaboration.

I am thankful also to my family for their continued support and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Karanasiou Aikaterini.

MSc, Computer Science and Engineering Department, University of Ioannina, Greece, September 2016.

Tilte of Dissertation: Low-High Orders of Directed Graphs: Incremental Algorithms and Applications.

Thesis Supervisor: Loukas Georgiadis.

A number of diverse natural or man-made systems are modeled as graphs, capturing both the structure and the dynamics of the underlying system. Examples include but are not limited to the world-wide web, transportation, communication and social networks, databases, biological systems, circuits, and the control-flow of computer programs. For this reason, graphs play an important role in many academic disciplines, including mathematics,computer science, and social sciences. Connectivity problems hold central role in the area of graph theory and graph algorithms, with numerous practical applications such as routing, navigation and reliable communication.

In this thesis we deal with problems related to connectivity in directed graphs. A flow graph $G = (V, E, s)$ is a directed graph with a distinguished start vertex s. The dominator tree D of G is a tree rooted at s, such that a vertex v is an ancestor of a vertex w if and only if all paths from s to w include v. The dominator tree is a central tool in program optimization and code generation, and has many applications in other diverse areas including constraint programming, circuit testing, biology, and in algorithms for graph connectivity problems. A low-high order of G is a preorder *delta* of D that certifies the correctness of D, and has further applications in connectivity and path-determination problems.

In the first part of the thesis, we consider how to maintain efficiently a low-high order of a flow graph incrementally under edge insertions. We present two algorithms that run in $O(mn)$ total time for a sequence of m edge insertions in an initially empty flow graph with n vertices. Moreover, we provide applications of this result to other incremental problems in directed graphs.

In the second part of the thesis, we apply low-high orders to a type of network design problems. Given a directed graph $G = (V, E)$, our goal is to compute the smallest spanning subgraph of G that maintains the 2-vertex-connectivity relations in G. First, we

deal the case where G is 2-vertex-connected and we wish to compute the smallest 2-vertex-connected spanning subgraph of G. We provide provide a linear-time algorithm that computes a 2-approximation for this problem, improving significantly the best previous approximation ratio achievable in linear time which was 3. Then we deal with the more general case, where G is not 2-vertex-connected, and provide linear-time 6-approximation algorithms.

We complement our theoretical study of the above problems with an extensive empirical evaluation of our algorithms, using large real-world graphs taken from a variety of application areas. The experimental results show that our algorithms are not only theoretically-efficient but also perform very well in practice.

ΠΕΡΙΛΗΨΗ

Καρανάσιου Αικατερίνη.

MSc, Τμήμα Μηχανικών Η/Υ & Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Σεπτέμβριος 2016.

Τίτλος Διατριβής: Χαμηλές-Υψηλές Διατάξεις σε Κατευθυνόμενα Γραφήματα: Δυναμικοί Αλγόριθμοι και Εφαρμογές.

Επιβλέπων: Λουκάς Γεωργιάδης.

Τα γραφήματα είναι μια απο τις πιο θεμελιώδης κατηγορίες δεδομένων στην επιστήμη των υπολογιστών. Αυτό έχει σαν αποτέλεσμα οι αλγόριθμοι για τον χειρισμό τους, και για τον υπολογισμό σχέσεων σε αυτά, να είναι ιδιαίτερα σημαντικοί στον κλάδο της πληροφορικής. Μια κατηγορία γραφημάτων είναι τα συνεκτικά γραφήματα.

Η συνεκτικότητα των γραφημάτων βρίσκει εφαρμογές σε διάφορους τομείς όπως για παράδειγμα σε δίκτυα επικοινωνιών, στην θεωρία ηλεκτρικών κυκλωμάτων και σε πλοήγηση δικτύων. Η μελέτη και η ανάλυση ενός γραφήματος για την -ως προς τις ακμες- και -ως προς τους κόμβους- συνεκτικότητα του αποτελεί ενα σημαντκό ερευνητικό πεδίο της θεωρίας γραφημάτων.

Σε αυτή τη διπλώματική ερευνώνται προβλήματα συνεκτικότητας σε κατευθυνόμενα γραφήματα. Ένα γράφημα ροής $G = (V, E, s)$ ειναι ενα κατευθυνομένο γραφημα με εναν διακεκριμένο κατευθυντήριο κόμβο s. Ένα δέντρο κυριαρχίας D ενός γραφήματος ροής G είναι ενα δέντρο με αφετηριακό κόμβο s, τέτοιο ώστε κάθε κόμβος v είναι πρόγονος ενός κόμβου w αν και μόνο αν όλα τα μονοπάτια απο τον κόμβο s στον κόμβο w περιέχουνε τον κόμβο v. Τα δέντρα κυριαρχίας έχουν διάφορες εφαρμογές τόσο στην επιστήμη της πληροφορικής όσο και σε άλλες επιστήμες, όπως για παράδειγμα στην βιολογία. Υψηλή-χαμηλή διάταξη δ ενός γραφήματος ροής G είναι μια προδιάταξη του δέντρου κυριαρχίας D του G, η οποία αποτελεί ένα πιστοποιητικό ορθότητας του D και έχει διάφορες εφαρμογές στην συνεκτικότητα κατευθυνόμενων γραφημάτων.

Στο πρώτο μέρος της διπλωματικής αυτής, ερευνάται η δυναμική ενημέρωση μιας ψηλήςχαμηλής διάταξης δ ενός γραφήματος ροής G μετα από μια εισαγωγή ακμής στο γράφημα. Το αποτέλεσμα που παρουσιάζεται ειναι 2 αλγόριθμοι οι οποίοι επιτυγχάνουν χρόνο εκτέλεσης της τάξεως mn μετά απο m εισαγωγές ακμών σε ένα αρχικά άδειο γράφημα n κόμβων.

Στο δεύτερο μέρος της εργασίας, θεωρείται το πρόβλημα έυρεσης ενός ελάχιστου υπογραφήματος του γραφήματος G σε δυο περιπτώσεις. Στην πρώτη περίπτωση έχοντας ένα

γράφημα G το οποίο ειναι 2-συνεκτικό ως προς τους κόμβους, παρουσιάζεται ένας αλγόριθμος με λόγο προσέγγισης 2 και γραμμικό χρόνο εκτέλεσης. Στην δεύτερη περίπτωση το γράφημα G δεν ειναι 2-συνεκτικό ως προς τους κόμβους και απαιτείται το υπογράφημα να διατηρεί κάποια συγκεκριμενα χαρακτηριστικά συνεκτικότητας. Για την τελευταία περίπτωση παρέχονται αλγορίθμοι με λόγο προσέγγισης 6 και χρόνο εκτέλεσης γραμμικό.

Τέλος, οι αλγόριθμοι που προτείνονται αναλύονται πειραματικά σε γραφήματα που προκύπτουν απο πραγματικές εφαρμογές, και παρουσιάζονται τα αποτέλεσμα της πειραματικής αυτής μελέτης.

Chapter 1

Introduction

1.1 Graphs and connectivity

Graphs can be used to model many types of relations and processes in physical, biological, social and information systems and many practical problems can be represented by them. More specifically in computer science, graphs are used to represent networks of communication, data organization, computational devices.

Connectivity is one of the basic concepts of graph theory. Edge and vertex connectivity are fundamental concepts in graph theory with numerous practical applications. As an example, we mention the computation of disjoint paths in routing and reliable communication, both in undirected and directed graphs.

Throughout this master thesis, we assume that the reader is familiar with the standard graph terminology and we are dealing only with directed graphs.

1.2 2-connectivity in directed graphs

Let $G = (V, E)$ be a directed graph (digraph), with m edges and n vertices. G is *strongly connected* if there is a directed path from each vertex to every other vertex. The *strongly connected components* of G are its maximal strongly connected subgraphs. A vertex (resp., an edge) of G is a *strong articulation point* (resp., a *strong bridge*) if its removal increases

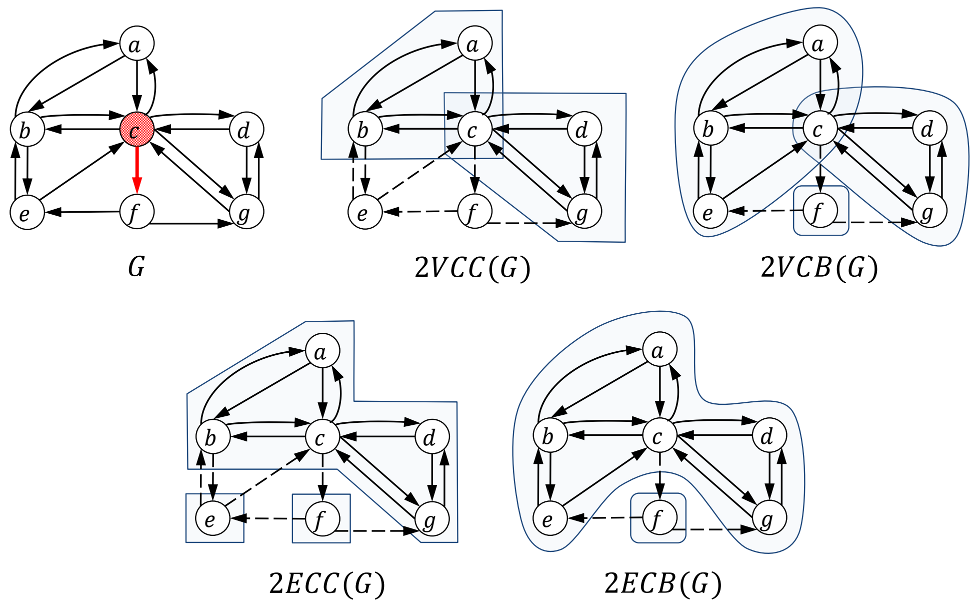

Figure 1.1: A strongly connected digraph G with a strong bridge (c, f) and a strong articulation point c shown in red (better viewed in color), the 2-vertex-connected components and blocks of G, and the 2-edge-connected components and blocks of G. Vertex f forms a trivial 2-edge-connected and 2-vertex-connected block.

the number of strongly connected components. A digraph G is 2*-vertex-connected* if it has at least three vertices and no strong articulation points; G is *2-edge-connected* if it has no strong bridges. The 2*-vertex-* (resp., *2-edge-*) *connected components* of G are its maximal 2-vertex- (resp., 2-edge-) connected subgraphs. Let v and w be two distinct vertices: v and w are 2*-vertex-connected* (resp., 2*-edge-connected*), denoted by $v \leftrightarrow_{2\mathrm{v}} w$ (resp., $v \leftrightarrow_{2\mathrm{e}} w$), if there are two internally vertex-disjoint (resp., two edge-disjoint) directed paths from v to w and two internally vertex-disjoint (resp., 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 either vertex- or edge- disjoint). A 2*-vertex-connected block* (resp., 2*-edge-connected block*) of a digraph $G = (V, E)$ is a maximal subset $B \subseteq V$ such that $u \leftrightarrow_{2\mathrm{v}} v$ (resp., $u \leftrightarrow_{2\mathrm{e}} v$) for all $u, v \in B$. Note that, as a (degenerate) special case, a 2-vertex- (resp., 2-edge-) connected block might consist of a singleton vertex only: we denote this as a *trivial* 2*-vertex-* (resp., 2*-edge-*) *connected block*. In the following, we will consider only non-trivial 2-vertex- and 2-edge- connected blocks. Since there is no danger of ambiguity, we will call them simply 2-vertex- and 2-edge-connected blocks.

The computation of 2-edge- and 2-vertex- connected blocks has been shown recently in linear time [23, 24], and the best current bound for this computation is not even linearbut it is $O(n^2)$ [35].

1.3 Terminology

Dominators A *flow graph* is a digraph such that every vertex is reachable from a distinguished start vertex. Let $G = (V, E)$ be a strongly connected digraph. 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 in $G(s)$ can be represented by a rooted tree, the *dominator tree* $D(s)$, such that u dominates w if and only if u is an ancestor of w in $D(s)$. If $w \neq s$, we denote by $d(w)$ the parent of w in $D(s)$. The dominator tree is a central tool in program optimization and code generation [11], and it has applications in other diverse areas including constraint programming [46], circuit testing [5], theoretical biology [2], memory profiling [42], the analysis of diffusion networks [34], and in connectivity problems [20, 21, 23, 22, 30, 36, 37, 38, 39]. Lengauer and Tarjan [41] presented an algorithm for computing dominators in $O(m\alpha(m, n))$ time for a flow graph with n vertices and m edges, where α is a functional inverse of Ackermann's function [50]. Subsequently, several linear-time algorithms were discovered [3, 8, 14, 16].

An edge (u, w) is a *bridge* in $G(s)$ if all paths from s to w include (u, w).[1] Italiano et al. [37] gave linear-time algorithms for computing all the strong bridges and all the strong articulation points of a digraph G. Their algorithms use the dominators and the bridges of flow graphs $G(s)$ 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.

Low-High Order A *low-high order* δ *of* G [28] is a preorder of the dominator tree D of G such for all reachable vertices $v \neq s$, $(d(v), v) \in E$ or there are two edges $(u, v) \in E$, $(w, v) \in E$ such that u and w are reachable, u is less than v ($u <_\delta v$), v is less than w ($v <_\delta w$), and w is not a descendant of v in D. See Figure 1.2. Every flow graph G has a low-high order, computable in linear-time [28]. Low-high orders provide a correctness certificate for dominator trees that is straightforward to verify [54]. By augmenting an algorithm that computes the dominator tree D of a flow graph G so that it also computes a low-high order of G, one obtains a *certifying algorithm* to compute D. (A *certifying algorithm* [43] outputs both the solution and a correctness certificate, with the property that it is straightforward to use the certificate to verify that the computed solution is correct.) Low-high orders also have applications in path-determination problems [53] and in fault-tolerant network design [6, 7, 32].

Divergent Spanning trees A notion closely related to low-high orders is that of divergent spanning trees [28]. Let V_r be the set of reachable vertices, and let $G[V_r]$ be the flow graph with start vertex s that is induced by V_r. Two spanning trees B and R of

[1] Throughout, 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.

$G[V_r]$, rooted at s, are *divergent* if for all v, the paths from s to v in B and R share only the dominators of v; B and R are *strongly divergent* if for every pair of vertices v and w, either the path in B from s to v and the path in R from s to w share only the common dominators of v and w, or the path in R from s to v and the path in B from s to w share only the common dominators of v and w. In order to simplify our notation, we will refer to B and R, with some abuse of terminology, as strongly divergent spanning trees of G. Every flow graph has a pair of strongly divergent spanning trees. Given a low-high order of G, it is straightforward to compute two strongly divergent spanning trees of G in $O(m)$ time [28]. Divergent spanning trees can be used in data structures that compute pairs of vertex-disjoint s-t paths in 2-vertex connected digraphs (for any two query vertices s and t) [20], in fast algorithms for approximating the smallest 2-vertex-connected spanning subgraph of a digraph [21], and in constructing sparse subgraphs of a given digraph that maintain certain connectivity requirements [23, 38, 39].

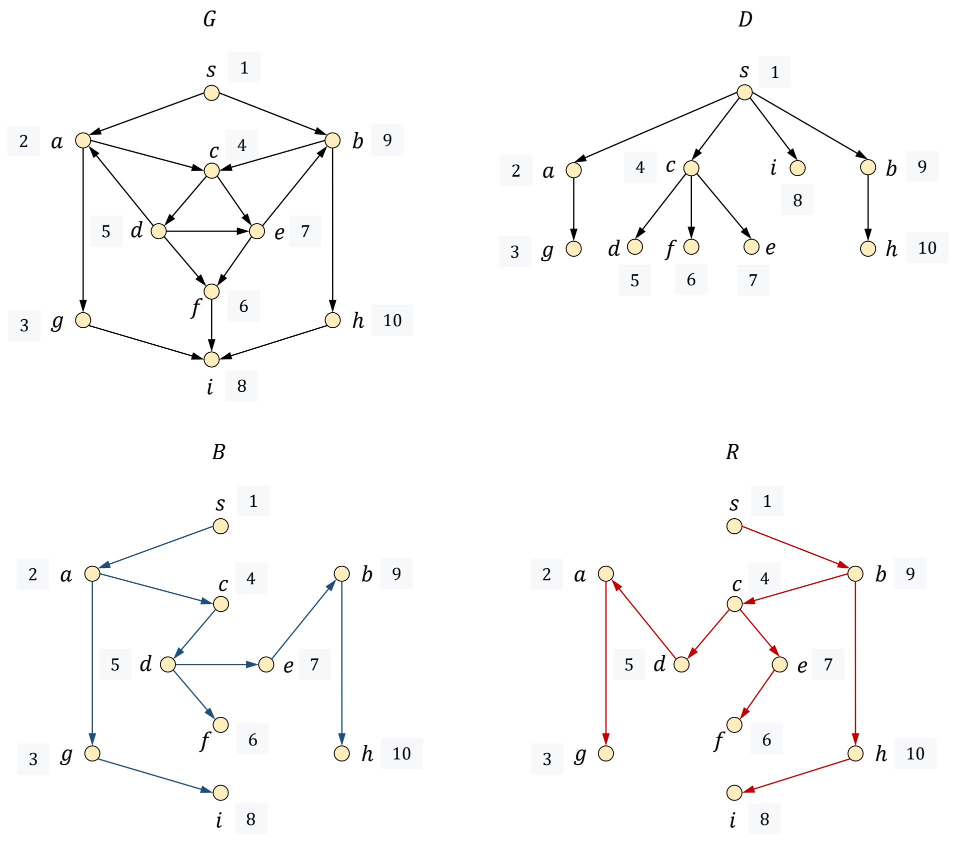

Figure 1.2: A flow graph G, dominator tree D, and two strongly divergent spanning trees B and R. The numbers correspond to a preorder numbering of D that is a low-high order of G.

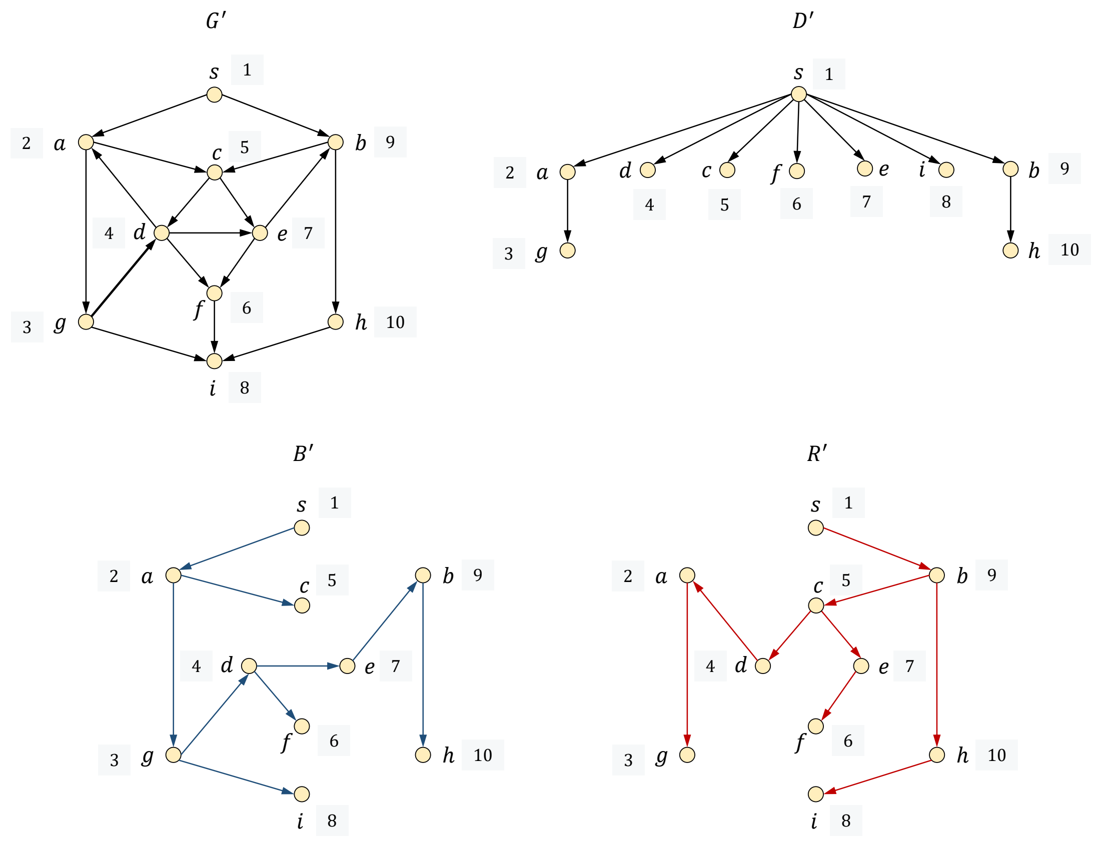

Figure 1.3: The flow graph of Figure 1.2 after the insertion of edge (g, d), and its updated dominator tree D' with a low-high order, and two strongly divergent spanning trees B' and R'.

1.4 Our contribution

1.4.1 Incremental low-high order algorithms

In the second chapter we consider the maintenance of a low-high order of a flow graph through a sequence of edge insertions. We present algorithms that run in $O(mn)$ total time for a sequence of m edge insertions in an initially empty flow graph with n vertices. These immediately provide the first incremental *certifying algorithms* for maintaining the dominator tree in $O(mn)$ total time, and also imply incremental algorithms for other problems. Hence, we provide a substantial improvement over the $O(m^2)$ simple-minded algorithms, which recompute the solution from scratch after each edge insertion.

So, we present two algorithms that achieve this bound ($O(mn)$), a simple and a more sophisticated. Both algorithms combine the incremental dominators algorithm of [25] with the linear-time computation of two divergent spanning trees from [28]. Our more sophisticated algorithm also applies a slightly modified version of a static low-high algorithm from [28] on an auxiliary graph. Although both algorithms have the same worst-case running time, our experimental results show that the sophisticated algorithm is by far superior in practical scenarios.

We note that the incremental dominators problem arises in various applications, such as incremental data flow analysis and compilation [10, 19, 47, 48], distributed authorization [44], and in incremental algorithms for maintaining 2-connectivity relations in directed graphs [30]. We provide also some applications of our algorithms to other incremental problems in digraphs. More specifically, we show how our result on incremental low-high order maintenance implies the following incremental algorithms that also run in $O(mn)$ total time for a sequence of m edge insertions.

- First, we give an algorithm that maintains, after each edge insertion, two strongly divergent spanning trees of G, and answers the following queries in constant time: (i) For any two query vertices v and w, find a path π_{sv} from s to v and a path π_{sw} from s to w, such that π_{sv} and π_{sw} share only the common dominators of v and w. We can output these paths in $O(|\pi_{sv}| + |\pi_{sw}|)$ time. (ii) For any two query vertices v and w, find a path π_{sv} from s to v that avoids w, if such a path exists. We can output this path in $O(|\pi_{sv}|)$ time.

- Then we provide an algorithm for an incremental version of the fault-tolerant reachability problem [6, 7]. We maintain a flow graph $G = (V, E, s)$ with n vertices through a sequence of m edge insertions, so that we can answer the following query in $O(n)$ time. Given a spanning forest $F = (V, E_F)$ of G rooted at s, find a set of edges $E' \subseteq E \setminus E_F$ of minimum cardinality, such that the subgraph $G' = (V, E_F \cup E', s)$ of G has the same dominators as G.

- Finally, given a digraph G, we consider how to maintain incrementally a spanning subgraph of G with $O(n)$ edges that preserves the 2-edge-connectivity relations in G.

1.4.2 Sparse subgraphs for 2-connectivity in directed graphs

In the third chapter we investigate problems where we wish to find a smallest spanning subgraph of G (i.e., with minimum number of edges) that maintains certain 2-connectivity requirements in addition to strong connectivity. Problems of this nature are fundamental in network design, and have several practical applications [45]. Specifically, we consider computing a smallest strongly connected spanning subgraph of a digraph G that maintains the following properties: the pairwise 2-vertex-connectivity of G, i.e., the 2-vertex-connected blocks of G (2VC-B); the 2-vertex-connected components of G (2VC-C); both the 2-vertex-connected blocks and components of G (2VC-B-C). This complements a previous study of the edge-connectivity versions of these problems [26], that the authors refer to as 2EC-C (maintaining 2-edge-connected components), 2EC-B (maintaining 2-edge-connected blocks), and 2EC-B-C (maintaining 2-edge-connected blocks and components). Finally, we also consider computing a smallest spanning subgraph of G that maintains all the 2-connectivity relations of G ($\mathsf{2C}$), that is, simultaneously the 2-vertex-connected and the 2-edge-connected components and blocks. Note that all these problems are NP-hard [18, 26], so one can only settle for efficient approximation algorithms. Computing small spanning subgraphs is of particular importance when dealing with large-scale graphs, say graphs having hundreds of million to billion edges. In this framework, one big challenge is to design linear-time algorithms, since algorithms with higher running times might be practically infeasible on today's architectures.

Chapter 2

Incremental low-high order

In this chapter,we first consider how to maintain efficiently a low-high order of a flow graph incrementally under edge insertions. We present two algorithms that run in $O(mn)$ total time for a sequence of m edge insertions in an initially empty flow graph with n vertices. Then we show how to apply low-high orders to obtain a linear-time 2-approximation algorithm for the smallest 2-vertex-connected spanning subgraph problem (2VCSS). Finally, we present efficient implementations of our new algorithms for the incremental low-high and 2VCSS problems.

2.1 Incremental algorithms

We first review some useful facts for updating a dominator tree after an edge insertion [4, 25, 47]. Let (x, y) be the edge to be inserted. We consider the effect of this insertion when both x and y are reachable. Let G' be the flow graph that results from G after inserting (x, y). Similarly, if D is the dominator tree of G before the insertion, we let D' be the the dominator tree of G'. Also, for any function f on V, we let f' be the function after the update. We say that vertex v is *affected* by the update if $d(v)$ (its parent in D) changes, i.e., $d'(v) \neq d(v)$. We let A denote the set of affected vertices. Note that we can have $D'[s, v] \neq D[s, v]$ even if v is not affected. We let $nca(x, y)$ denote the nearest common ancestor of x and y in the dominator tree D. We also denote by $\mathit{depth}(v)$ the depth of a reachable vertex v in D. There are affected vertices after the insertion of (x, y) if and only if $\mathit{nca}(x, y)$ is not a descendant of $d(y)$ [47]. A characterization of the affected vertices is provided by the following lemma, which is a refinement of a result in [4].

Lemma 2.1. ([25]) *Suppose x and y are reachable vertices in G. A vertex v is affected after the insertion of edge (x, y) if and only if $\mathit{depth}(\mathit{nca}(x, y)) < \mathit{depth}(d(v))$ and there is a path π in G from y to v such that $\mathit{depth}(d(v)) < \mathit{depth}(w)$ for all $w \in \pi$. If v is affected, then it becomes a child of $\mathit{nca}(x, y)$ in D', i.e., $d'(v) = \mathit{nca}(x, y)$.*

The algorithm (DBS) in [25] applies Lemma 2.1 to identify affected vertices by starting a search from y (if y is not affected, then no other vertex is). To do this search for affected vertices, it suffices to maintain the outgoing and incoming edges of each vertex. These sets are organized as singly linked lists, so that a new edge can be inserted in $O(1)$ time. The dominator tree D is represented by the parent function d. We also maintain the depth in D of each reachable vertex. We say that a vertex v is *scanned*, if the edges leaving v are examined during the search for affected vertices, and that it is *visited* if there is a scanned vertex u such that (u, v) is an edge in G. By Lemma 2.1, a visited vertex v is scanned if $\mathit{depth}(\mathit{nca}(x, y)) < \mathit{depth}(d(v))$.

Lemma 2.2. ([25]) *Let v be a scanned vertex. Then v is a descendant of an affected vertex in D.*

2.1.1 Simple algorithm

In this algorithm we maintain, after each insertion, a subgraph $H = (V, E_H)$ of G with $O(n)$ edges that has the same dominator tree as G. Then, we can compute a low-high order δ of H in $O(|E_H|) = O(n)$ time. Note that δ is also a valid low-high order of G. Subgraph H is formed by the edges of two divergent spanning trees B and R of G. After the insertion of an edge (x, y), where both x and y are reachable, we form a graph H' by inserting into H a set of edges $\mathit{Last}(A)$ found during the search for affected vertices. Specifically, $\mathit{Last}(A)$ contains edge (x, y) and, for each affected vertex $v \neq y$, the last edge on a path π_{yv} that satisfies Lemma 2.1. Then, we set $H' = H \cup \mathit{Last}(A)$. Finally, we compute a low-high order and two divergent spanning trees of H', which are also valid for G'. Algorithm SimpleInsertEdge describes this process.

Note that when only x is reachable before the insertion, we re-initialize our algorithm by running the linear-time algorithm of [28, Section 6], which returns both a low-high order and two divergent spanning trees.

Lemma 2.3. *Algorithm SimpleInsertEdge is correct.*

Proof. It suffices to show that subgraph H' of G', computed in line 9, has the same dominator tree with G'. Note that graph H, formed by two divergent spanning trees B and R of G in line 7, has the same dominator tree D as G. Hence, since $\mathit{Last}(A)$ contains

Algorithm 1: SimpleInsertEdge(G, D, δ, B, R, e)

```
   Input: Flow graph $G = (V, E, s)$, its dominator tree $D$, a low-high order $\delta$ of $G$,
          two divergent spanning trees $B$ and $R$ of $G$, and a new edge $e = (x, y)$.
   Output: Flow graph $G' = (V, E \cup (x, y), s)$, its dominator tree $D'$, a low-high
           order $\delta'$ of $G'$, and two divergent spanning trees $B'$ and $R'$ of $G'$.
 1 Insert $e$ into $G$ to obtain $G'$.
 2 if $x$ is unreachable in $G$ then return $(G', D, \delta, B, R)$
 3 else if $y$ is unreachable in $G$ then
 4     $(D', \delta', B', R') \leftarrow$ ComputeLowHigh$(G')$
 5     return $(G', D', \delta', B', R')$
 6 end
 7 Let $H = B \cup R$.
 8 Compute the updated dominator tree $D'$ of $G'$ and return a list $A$ of the affected
   vertices, and a list $Last(A)$ of the last edge entering each $v \in A$ in a path of
   Lemma 3.2.
 9 Compute the subgraph $H' = H \cup Last(A)$ of $G'$.
10 Compute $(D', \delta', B', R') \leftarrow$ ComputeLowHigh$(H')$
11 return $(G', D', \delta', B', R')$
```

(x, y), the immediate dominator of y is the same in H' and in G'. Let A be the set of affected vertices in G after the insertion of edge (x, y). Since H' is a subgraph of G', any vertex in $V \setminus A$ has the same immediate dominator in H' and in G'. It remains to argue that for each vertex $v \in A \setminus y$, there is a path $\widehat{\pi}_{yv}$ in H' that satisfies Lemma 2.1. Let π_{yv} be the path from y in v in G that was found by the search for affected vertices performed by the algorithm of [25]. We give a corresponding path $\widehat{\pi}_{yv}$ in H'. Recall that every vertex on π_{yv} is scanned and that every scanned vertex is a descendant in D of an affected vertex. We argue that for every two successive affected vertices u and w on π_{yv} there is a path π_{uw} from u to w in H' that consists of vertices of depth at least $depth(w)$. Note that, by properties of the depth-based search, $depth(w) \leq depth(u)$. Indeed, let (p, w) be the edge entering w from π_{yv}. Then $(p, w) \in Last(A)$ and p is a descendant of u in D. Also, since u dominates p in G, u is an ancestor of p in both spanning trees B and R. We let $\widehat{\pi}_{uw} = B[u, p] \cdot (p, w)$. All vertices on $B[u, p]$ are dominated by u, since otherwise there would be a path from s to p avoiding u. So, $\widehat{\pi}_{uw}$ is path from u to w in H' that consists of vertices with depth at least $depth(w)$. ∎

Lemma 2.4. *Algorithm SimpleInsertEdge maintains a low-high order of a flow graph G with n vertices through a sequence of edge insertions in $O(mn)$ total time, where m is the total number of edges in G after all insertions.*

Proof. Consider the insertion of an edge (x, y). If y was unreachable in G then we compute D, two divergent spanning trees B and R, and a low-high order in $O(m)$ time [28]. Throughout the whole sequence of m insertions, such an event can happen $O(n)$ times, so all insertions to unreachable vertices are handled in $O(mn)$ total time.

Now we consider the cost of executing SimpleInsertEdge. when both x and y are reachable in G. Let ν be the number of scanned vertices, and let μ be the number of their adjacent edges. We can update the dominator tree and locate the affected vertices (line 8) in $O(\nu + \mu + n)$ time [25]. At the same time we can compute the edge set $\mathit{Last}(A)$. Computing H' in line 9 takes $O(n)$ time since $B \cup R \cup \mathit{Last}(A)$ contains at most $3(n-1)$ edges. Also, computing the dominator tree, two divergent spanning trees,, and a low-high order of H' in $O(n)$ time [28]. So SimpleInsertEdge runs in $O(\nu+\mu+n)$ time. . The $O(n)$ term gives a total cost of $O(mn)$ for the whole sequence of m insertions. We distribute the remaining $O(\nu + \mu)$ cost to the scanned vertices and edges, that is $O(1)$ per scanned vertex or edge. Since the depth in D of every scanned vertex decreases by at least one, a vertex and an edge can be scanned at most $O(n)$ times. Hence, each vertex and edge can contribute at most $O(n)$ total cost through the whole sequence of m insertions. The $O(mn)$ bound follows. ∎

2.1.2 Efficient algorithm

Here we develop a more practical algorithm that maintains a low-high order δ of a flow graph $G = (V, E, s)$ through a sequence of edge insertions. Our algorithm uses the incremental dominators algorithm of [25] to update the dominator tree D of G after each edge insertion. We describe a process to update δ based on new results on the relation among vertices in D that are affected by the insertion. . These results enable us to identify a subset of vertices for which we can compute a “local” low-high order, that can be extended to a valid low-high order of G after the update. We show that such a “local” low-high order can be computed by a slightly modified version of an algorithm from [28]. We apply this algorithm on a sufficiently small flow graph that is defined by the affected vertices, and is constructed using the concept of edges [52].

Derived edges and derived flow graphs

Derived graphs, first defined in [52], reduce the problem of finding a low-high order to the case of a flat dominator tree [28]. By the parent property of D, if (v, w) is an edge of G, the parent $d(w)$ of w is an ancestor of v in D. Let (v, w) be an edge of G, with w not an ancestor of v in D. Then, the *derived edge* of (v, w) is the edge $(\overline{v}, w)$, where $\overline{v} = v$ if $v = d(w)$, $\overline{v}$ is the sibling of w that is an ancestor of v if $v \neq d(w)$. If w is an ancestor of v in D, then the derived edge of (v, w) is null. Note that a derived edge $(\overline{v}, w)$ may not be an original edge of G. For any vertex $w \in V$ such that $C(w) \neq \emptyset$, we define the *derived flow graph of* w, denoted by $G_w = (V_w, E_w, w)$, as the flow graph with start vertex w, vertex set $V_w = C(w) \cup \{w\}$, and edge set $E_w = \{(\overline{u}, v) \mid v \in V_w$ and $(\overline{u}, v)$ is the non-null derived edge of some edge in $E\}$. By definition, G_w has flat dominator tree, that is, w is the only proper dominator of any vertex $v \in V_w \setminus w$. We can compute a low-high order δ of G by computing a low-high order δ_w in each derived flow graph G_w. Given these low-high orders δ_w, we can compute a low-high order of G in $O(n)$ time by a depth-first traversal of D. During this traversal, we visit the children of each vertex w in their order in δ_w, and number the vertices from 1 to n as they are visited. The resulting preorder of D is low-high on G. Our incremental algorithm identifies, after each edge insertion, a specific derived flow graph G_w for which a low-high order δ_w needs to be updated. Then, it uses δ_w to update the low-high order of the whole flow graph G. Still, computing a low-high order of G_w can be too expensive to give us the desired running time. Fortunately, we can overcome this obstacle by exploiting a relation among the vertices that are affected by the insertion, as specified below. This allows us to compute δ_w in a contracted version of G_w.

Affected vertices

Let (x, y) be the inserted vertex, where both x and y are reachable. Consider the execution of algorithm DBS [25] that updates the dominator tree by applying Lemma 2.1. Suppose vertex v is scanned, and let q be the nearest affected ancestor of v in D. Then, by Lemma 2.1, vertex q is a child of $nca(x, y)$ in D', i.e., $d'(q) = nca(x, y)$, and v remains a descendant of q in D'.

Lemma 2.5. *Let u and v be vertices such that $u \in D(v)$. Then, any simple path from v to u in G contains only vertices in $D(v)$.*

Proof. Since $u \in D(v)$, v dominates u, so all paths from s to u contain v. Let π_{vu} be a simple path from v to u. Suppose, for contradiction, that π_{vu} contains a vertex $w \notin D(v)$. Let π_{wu} be the part of π_{vu} from w to u. Since $w \notin D(v)$, there is a path π_{sw} from s to w that avoids v. But then $\pi_{sw} \cdot \pi_{wu}$ is a path from s to u that avoids v, a contradiction. ∎

Lemma 2.6. *Let v be vertex that is affected by the insertion of (x, y), and let w be a sibling of v in D. If there is an edge (u, w) with u a descendant of v in D then w is also affected.*

Proof. Since v is affected, there is a path π_{yv} from y to v in G that satisfies Lemma 2.1. By Lemma 2.5 and the fact that u is a descendant of v in D, there is a simple path π_{vu} from v to u in G that contains only vertices in $D(v)$. Thus, $\pi_{yv} \cdot \pi_{vu} \cdot (u, w)$ is a path from y to w that also satisfies Lemma 2.1. Hence, w is affected. ∎

Lemma 2.7. *Let v be an ancestor of w in D, and let u be a vertex that is not a descendant of v in D. Then any path from u to w contains v.*

Proof. Let π_{uw} be a path from u to w. Since u is not a descendant of v, there is a path π_{su} from s to u that avoids v. Hence, if π_{uw} does not contain v, then $\pi_{su} \cdot \pi_{uw}$ is path from s to w that avoids v, a contradiction. ∎

Our next lemma provides a key result about the relation of the affected vertices in D.

Lemma 2.8. *All vertices that are affected by the insertion of (x, y) are descendants of a common child c of $nca(x, y)$.*

Proof. Let $z = nca(x, y)$, and let c be the child of z that is an ancestor of y in D. We claim that all affected vertices are descendants of c in D. Suppose, for contradiction, that there is an affected vertex v that is not a descendant of c in D. By Lemma 2.1, v must be a descendant z in D. Also, since the children of z are not affected, v is not a child of z. Hence, v is a proper descendant of another child q of z in D ($q \neq c$). Let π_{yv} be a path from y to v in G that satisfies Lemma 2.1. Since y is not a descendant of q, by Lemma 2.7 path π_{yv} must contain q. But then π_{yv} contains a vertex of depth $depth(d(v))$ or less, which contradicts Lemma 2.1. ∎

We shall apply Lemma 2.8 to construct a flow graph G_A for the affected vertices. Then, we shall use G_A to compute a "local" low-high order that we extend to a valid low-high order of G'.

Low-high order augmentation

Let δ be a low-high order of G, and let δ' be a preorder of the dominator tree D' of G'. We say that δ' *agrees with* δ if the following condition holds for any pair of siblings u, v in D that are not affected by the insertion of (x, y): $u <_{\delta'} v$ if and only if $u <_\delta v$. Our goal is to show that there is a low-high order δ' of G' that agrees with δ.

Lemma 2.9. *Let δ be a low-high order of G before the insertion of (x, y). There is a preorder δ' of D' that agrees with δ.*

Proof. By Lemma 2.1, all affected vertices become children of z in D'. Hence, $C'(z) \supseteq C(z)$, and for any $v \neq z$, $C'(v) \subseteq C(v)$. Then, for each vertex v, we can order the children of v in $C'(v)$ that are not affected according to δ. Finally, we insert the affected vertices in any order in the list of children of z. Let δ' be the preorder of D' that is constructed by a depth-first traversal of D' that visits the children of each vertex w in the order specified above. Then, δ' agrees with δ. ∎

Lemma 2.10. *Let δ' be a preorder of D' that agrees with δ. Let v be a vertex that is not a child of $nca(x, y)$ and is not affected by the insertion of (x, y). Then δ' is a low-high order for v in G'.*

Proof. Since v is not affected, $d(v)$ is still the parent of v in D after the insertion. So, if $(d(v), v) \in E$, then δ' is a low-high order for v in G'. Now suppose that $(d(v), v) \notin E$. Then there are two edges (u, v) and (w, v) in E such that $u <_\delta v <_\delta w$, where w is not a descendant of v in D. Let $(\overline{u}, v)$ and $(\overline{w}, v)$ be the derived edges of (u, v) and (w, v), respectively, in D. Then $\overline{u}$ and $\overline{w}$ are siblings of v in D. Siblings $\overline{u}$ and $\overline{w}$ exist and are distinct by the fact that $(d(v), v) \notin E$ and by the parent property of D. Hence, $\overline{u} <_\delta v <_\delta \overline{w}$. We argue that after the insertion of (x, y), $\overline{u}$ (resp., $\overline{w}$) remains a sibling of v, and an ancestor of u (resp., w). If this is not the case, then there is an affected vertex

q on $D[\overline{u}, u]$. But then, Lemma 2.6 implies that v is also be affected, a contradiction. So, both $\overline{u}$ and $\overline{w}$ remain siblings of v in D', and $(\overline{u}, v)$ and $(\overline{w}, v)$ remain the derived edges of (u, v) and (w, v), respectively, in D'. Then, since δ' agrees with δ, δ' is a low-high order for v in G'. ∎

We shall use Lemmata 2.1 and 2.10 to show that in order to compute a low-high order of G', it suffices to compute a low-high order for the derived flow graph G'_z, where $z = nca(x, y)$. Still, the computation of a low-high order of G'_z is too expensive to give us the desired running time. Fortunately, as we show next, we can limit these computations for a contracted version of G'_z, defined by the affected vertices.

Let δ be a low-high order of G before the insertion of (x, y). Also, let $z = nca(x, y)$, and let δ_z be a corresponding low-high order of the derived flow graph G_z. That is, δ_z is the restriction of δ to z and its children in D. Consider the child c of z that, by Lemma 2.8, is an ancestor of all the affected vertices. Let α and β, respectively, be the predecessor and successor of c in δ_z. Note that α or β may be null. An *augmentation of* δ_z is an order δ'_z of $C'(z) \cup \{z\}$ that results from δ_z by inserting the affected vertices arbitrarily around c, that is, each affected vertex is placed in an arbitrary position between α and c or between c and β.

Lemma 2.11. *Let $z = nca(x, y)$, and let δ_z be a low-high order of the derived flow graph G_z before the insertion of (x, y). Also, let δ'_z be an augmentation of δ_z, and let δ' be a preorder of D' that extends δ'_z. Then, for each child v of z in D, δ' is a low-high order for v in G'.*

Proof. Since v is a child of z in D it is not affected. Hence, $d'(v) = d(v) = z$. Let G'_z be the derived flow graph of z after the insertion of (x, y). It suffices to show that δ'_z is a low-high order for v in G'_z.

If $(z, v) \in E$, then (z, v) is an edge in G'_z. So, in this case, δ'_z is a low-high order for v in G'_z. Now suppose that $(z, v) \notin E$. Let δ be a preorder of D that extends δ_z. Then, there are two edges (u, v) and (w, v) in G such that $u <_\delta v <_\delta w$, where w is not a descendant of v in D. The fact that (z, v) is not an edge implies that $u \neq z$ and $w \neq z$. Let $\overline{u}'$ (resp., $\overline{w}'$) be the nearest ancestor of u (resp., w) in D' that is a child of z. We argue that $\overline{u}'$ exists and satisfies $\overline{u}' <_{\delta'_z} v$. Let $\overline{u}$ be the nearest ancestor of u in D. If no vertex on $D(z, u]$ is affected, then $\overline{u}' = \overline{u}$. Also, since $\overline{u} <_{\delta_z} v$ and by the fact that δ'_z is an augmentation of δ_z, we have $\overline{u}' <_{\delta'_z} v$. Suppose now that there is an affected vertex q on $D(z, u]$. By Lemma 2.1, q becomes a child of z in D', hence $\overline{u}' = q$. Also, by Lemma 2.8, q is a proper descendant of c, so $\overline{u} = c$. Then $c <_{\delta_z} v$, and by the construction of δ'_z we have $\overline{u}' <_{\delta'_z} v$.

An analogous argument shows that $\overline{w}'$ exists and satisfies $v <_{\delta'_z} \overline{w}'$. Thus, δ'_z is a low-high order for v in G'_z. ∎

Algorithm

Now we are ready to describe our incremental algorithm for maintaining a low-high order δ of G. For each vertex v that is not a leaf in D, we maintain a list of its children $C(v)$ in D, ordered by δ. Also, for each vertex $v \neq s$, we keep two variables $low(v)$ and $high(v)$. Variable $low(v)$ stores an edge (u, v) such that $u \neq d(v)$ and $u <_\delta v$; $low(v) = null$ if no such edge exists. Similarly, $high(v)$ stores an edge (w, v) such that and $v <_\delta w$ and w is not a descendant of v in D; $high(v) = null$ if no such edge exists. These variables are useful in the applications that we mention in Section 2.2. Finally, we mark each vertex v such that $(d(v), v) \in E$. For simplicity, we assume that the vertices of G are numbered from 1 to n, so we can store the above information in corresponding arrays *low*, *high*, and *mark*. Note that for a reachable vertex v, we can have $low(v) = null$ or $high(v) = null$ (or both) only if $mark(v) = true$. Before any edge insertion, all vertices are unmarked, and all entries in arrays *low* and *high* are null. We initialize the algorithm and the associated data structures by executing a linear-time algorithm to compute the dominator tree D of G [3, 8] and a linear-time algorithm to compute a low-high order δ of G [28]. So, the initialization takes $O(m + n)$ time for a digraph with n vertices and m edges. Next, we describe the main routine to handle an edge insertion. We let (x, y) be the inserted edge. Also, if x and y are reachable before the insertion, we let $z = nca(x, y)$.

```
Algorithm 2: Initialize($G$)
1 Compute the dominator tree $D$ and a low-high order $\delta$ of $G$.
2 foreach reachable vertex $v \in V \setminus s$ do
3     if $(d(v), v) \in E$ then set $mark(v) \leftarrow true$
4     find edges $(u, v)$ and $(w, v)$ such that $u <_\delta v <_\delta w$ and $w \notin D(v)$
5     set $low(v) \leftarrow u$ and $high(v) \leftarrow w$
6 end
7 return $(D, \delta, mark, low, high)$
```

Our main task now is to order the affected vertices according to a low-high order of D'. To do this, we use an auxiliary flow graph $G_A = (V_A, E_A, z)$, with start vertex z, which we refer to as the *derived affected flow graph*. Flow graph G_A is essentially a contracted version of the derived flow graph G'_z (i.e., the derived graph of z after the insertion) as we explain later. The vertices of the derived affected flow graph G_A are z, the affected vertices of G, their common ancestor c in D that is a child of z (from Lemma 2.8), and two auxiliary vertices α^* and β^*. Vertex α^* (resp., β^*) represents vertices in $C(z)$ with lower (resp., higher) order in δ than c. We include in G_A the edges (z, α^*) and (z, β^*). If c is marked then we include the edge (z, c) into G_A, otherwise we add the edges (α^*, c) and (β^*, c) into G_A. Also, for each edge (u, c) such that u is a descendant of an affected vertex v, we add in G_A the edge (v, c). Now we specify the edges that enter an affected vertex w in G_A. We consider each edge $(u, w) \in E$ entering w in G. We have the following cases:

(a) If u is a descendant of an affected vertex v, we add in G_A the edge (v, w).

Algorithm 3: InsertEdge($G, D, \delta, mark, low, high, e$)

```
Input: Flow graph $G = (V, E, s)$, its dominator tree $D$, a low-high order $\delta$ of $G$,
       arrays $mark$, $low$ and $high$, and a new edge $e = (x, y)$.
Output: Flow graph $G' = (V, E \cup (x, y), s)$, its dominator tree $D'$, a low-high
        order $\delta'$ of $G'$, and arrays $mark'$, $low'$ and $high'$.
1  Insert $e$ into $G$ to obtain $G'$.
2  if $x$ is unreachable in $G$ then return $(G', D, \delta, mark, low, high)$
3  else if $y$ is unreachable in $G$ then
4      $(D', \delta', mark', low', high') \leftarrow$ Initialize($G'$)
5      return $(G', D', \delta', mark', low', high')$
6  end
7  Compute the nearest common ancestor $z$ of $x$ and $y$ in $D$.
8  Compute the updated dominator tree $D'$ of $G'$ and return a list $A$ of the affected
   vertices.
9  foreach vertex $v \in A$ do $mark'(y) \leftarrow false$
10 if $z = x$ then $mark'(y) \leftarrow true$
11 Execute DerivedLowHigh($z, A, mark'$).
12 Make a dfs traversal of the subtrees of $D'$ rooted at each vertex $v \in A \cup \{c\}$ to
   compute $\delta'$.
13 foreach vertex $v \in A \cup \{c\}$ do
14     find edges $(u, v)$ and $(w, v)$ such that $u <_{\delta'} v <_{\delta'} w$ and $w \notin D'(v)$
15     set $low'(v) \leftarrow u$ and $high'(v) \leftarrow w$
16 end
17 return $(G', D', \delta', mark', low', high')$
```

(b) If u is a descendant of c but not a descendant of an affected vertex, then we add in G_A the edge (c, w).

(c) If $u \neq z$ is not a descendant of c, then we add the edge (α^*, w) if $u <_\delta c$, or the edge (β^*, w) if $c <_\delta u$.

(d) Finally, if $u = z$, then we add the edge (z, w). (In cases (c) and (d), $u = x$ and $w = y$.)

See Figure 2.1. Recall that α (resp., β) is the siblings of c in D immediately before (resp., after) c in δ, if it exists. Then, we can obtain G_A from G'_z by contracting all vertices v with $v <_\delta c$ into $\alpha = \alpha^*$, and all vertices v with $c <_\delta v$ into $\beta = \beta^*$.

Lemma 2.12. *The derived affected flow graph $G_A = (V_A, E_A, z)$ has flat dominator tree.*

Proof. We claim that for any two distinct vertices $v, w \in V_A \setminus z$, v does not dominate w. The lemma follows immediately from this claim. The claim is obvious for $w \in \{\alpha^*, \beta^*\}$, since G_A contains the edges (z, α^*) and (z, β^*). The same holds for $w = c$, since G_A

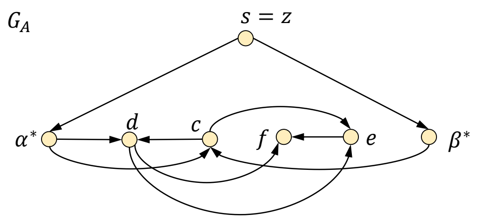

Figure 2.1: The derived affected flow graph G_A that corresponds to the flow graph of Figure 1.2 after the insertion of edge (g, d).

contains the edge (z, c), or both the edges (α^*, c) and (β^*, c). Finally, suppose $w \in V_A \setminus \{z, \alpha^*, \beta^*\}$. Then, by the construction of G_A, vertex w is affected. By Lemma 2.8, $w \in D(c)$, so Lemma 2.5 implies that there is a path in G from c to w that contains only vertices in $D(c)$. Hence, by construction, G_A contains a path from c to w that avoids α^* and β^*, so α^* and β^* do not dominate w. It remains to show that w is not dominated in G_A by c or another affected vertex v. Let (x, y) be the inserted edge. Without loss of generality, assume that $c <_\delta x$. Since w is affected, there is a path π in G from y to w that satisfies Lemma 2.1. Then π does not contain any vertex in $D[c, d(w)]$. Also, by the construction of G_A, π corresponds to a path π_A in G_A from β^* to y that avoids any vertex in $A \cap D[c, d(w)]$. Hence, w is not dominated by any vertex in $A \cap D[c, d(w)]$. It remains to show that w is not dominated by any affected vertex v in $A \setminus D[c, d(w)]$. Since both v and w are in $D(c)$ and v is not an ancestor of w in D, there is a path π' in G from c to w that avoids v. By Lemma 2.5, π' contains only vertices in $D(c)$. Then, by the construction of G_A, π' corresponds to a path π'_A in G_A from c to w that avoids v. Thus, v does not dominate w in G_A. ∎

Lemma 2.13. *Let ν and μ, respectively, be the number of scanned vertices and their adjacent edges. Then, the derived affected flow graph G_A has $\nu + 4$ vertices, at most $\mu + 5$ edges, and can be constructed in $O(\nu + \mu)$ time.*

Proof. The bound on the number of vertices and edges in G_A follows from the definition of the derived affected flow graph. Next, we consider the construction time of G_A. Consider the edges entering the affected vertices. Let w be an affected vertex, and let $(u, w) \neq (x, y)$ be an edge of G'. Let q be nearest ancestor u in $C'(z)$. We distinguish two cases:

- u is not scanned. In this case, we argue that $q = c$. Indeed, it follows from the parent property of D and Lemma 2.8 that both u and w are descendants of c in D. Since u is not scanned, no ancestor of u in D is affected, so u remains a descendant of c in D'. Thus, $q = c$.

- u is scanned. Then, by Lemma 2.2, q is the nearest affected ancestor of u in D.

So we can construct the edges entering the affected vertices in G_A in two phases. In the first phase we traverse the descendants of each affected vertex q in D'. At each descendant u of q, we examine the edges leaving u. When we find an edge (u, w) with w affected, then we insert into G_A the edge (q, w). In the second phase we examine the edges entering each affected vertex w. When we find an edge (u, w) with u not visited during the first phase (i.e., u was not scanned during the update of D), we insert into G_A the edge (c, w). Note that during this construction we may insert the same edge multiple times, but this does not affect the correctness or running time of our overall algorithm. Since the descendants of an affected vertex are scanned, it follows that each phase runs in $O(\nu + \mu)$ time.

Finally, we need to consider the inserted edge (x, y). Let f be the nearest ancestor of x that is in $C(z)$. Since y is affected, $c \neq f$. Hence, we insert into G_A the edge (β^*, y) if $c <_\delta f$, and the edge (α^*, y) if $f <_\delta c$. Note that f is found during the computation of $z = nca(x, y)$, so this test takes constant time. ∎

We use algorithm DerivedLowHigh, shown below, to order the vertices in $C'(z)$ according to a low-high order of ζ of G_A. After computing G_A, we construct two divergent spanning trees B_A and R_A of G_A. For each vertex $v \neq z$, if (z, v) is an edge of G_A, we replace the parent of v in B_A and in R_A, denoted by $b_A(v)$ and $r_A(v)$, respectively, by z. Then we use algorithm AuxiliaryLowHigh to compute a low-high order ζ of G_A. Algorithm AuxiliaryLowHigh is a slightly modified version of a linear-time algorithm of [28, Section 6.1] to compute a low-high order. Our modified version computes a low-high order ζ of G_A that is an augmentation of δ_z. To obtain such a low-high order, we need to assign to α^* the lowest number in ζ and to β^* the highest number in ζ. The algorithm works as follows. While G_A contains at least four vertices, we choose a vertex $v \notin \{\alpha^*, \beta^*\}$ whose in-degree in G_A exceeds its number of children in B_A plus its number of children in R_A and remove it from G_A. (From this choice of v we also have that $v \neq z$.) Then we compute recursively a low-high order for the resulting flow graph, and insert v in an appropriate location, defined by $b_A(v)$ and $r_A(v)$.

Algorithm 4: DerivedLowHigh$(z, A, mark)$

```
1 Compute the derived affected flow graph $G_A = (V_A, E_A, z)$.
2 Compute two divergent spanning trees $B_A$ and $R_A$ of $G_A$.
3 foreach vertex $v \in V_A \setminus \{z, \alpha^*, \beta^*\}$ do
4     if $mark(v) = true$ then set $b_A(v) \leftarrow z$ and $r_A(v) \leftarrow z$
5 end
6 Initialize a list of vertices $\Lambda \leftarrow \emptyset$.
7 Compute $\Lambda \leftarrow$ AuxiliaryLowHigh$(G_A, B_A, R_A, \Lambda)$.
8 Order the set of children $C'(z)$ of $z$ in $D'$ according to $\Lambda$.
```

Lemma 2.14. *Algorithm AuxiliaryLowHigh is correct, that is, it computes a low-high order ζ of G_A, such that for all $v \in V_A \setminus \{z, \alpha^*, \beta^*\}$, $\alpha^* <_\zeta v <_\zeta \beta^*$.*

Algorithm 5: AuxiliaryLowHigh(G_A, B_A, R_A, Λ)

```
if $G_A$ contains only three vertices then
    set $\Lambda \leftarrow \langle \alpha^*, \beta^* \rangle$
    return $\Lambda$
end
Let $v \notin \{\alpha^*, \beta^*\}$ be a vertex whose in-degree in $G_A$ exceeds its number of children
  in $B_A$ plus its number of children in $R_A$.
Delete $v$ and its incoming edges from $G_A$, $B_A$, and $R_A$.
if $v$ was not a leaf in $B_A$ then
    let $w$ be the child of $v$ in $B_A$; replace $b_A(w)$ by $b_A(v)$
end
else if $v$ was not a leaf in $R_A$ then
    let $w$ be the child of $v$ in $R_A$; and replace $r_A(w)$ by $r_A(v)$
end
Call AuxiliaryLowHigh($G_A, B_A, R_A, \Lambda$) recursively for the new graph $G_A$.
if $b_A(v) = z$ then
    insert $v$ anywhere between $\alpha^*$ and $\beta^*$ in $\Lambda$
end
else
    insert $v$ just before $b_A(v)$ in $\Lambda$ if $r_A(v)$ is before $b_A(v)$ in $\Lambda$, just after $b_A(v)$
      otherwise
end
return $\Lambda$
```

Proof. We first show that algorithm AuxiliaryLowHigh runs to completion, i.e., it selects every vertex $v \in V_A \setminus \{z, \alpha^*, \beta^*\}$ at some execution of line 5. The recursive call in line 13 invokes algorithm AuxiliaryLowHigh on a sequence of smaller flow graphs G_A. We claim that the following invariants hold for each such flow graph G_A:

(i) the dominator tree D_A of G_A is flat;

(ii) the subgraphs B_A and R_A corresponding to G_A are divergent spanning trees of G_A rooted at z;

(iii) for every $v \neq z$, either $b_A(v) = r_A(v) = z$ or $b_A(v)$, $r_A(v)$, and z are all distinct.

For the initial graph G_A the invariants hold by construction. Assume that the invariants hold on entry to line 5. Suppose, now, that line 5 chooses a vertex $v \notin \{\alpha^*, \beta^*\}$. Since v has in-degree at most 2 in G_A, the choice of v implies that it has at most one outgoing edge. Hence v is a leaf in either B_A or R_A. If it is a leaf in both, deleting v and its incoming edges preserves all the invariants. Suppose v is a leaf in R_A but not B_A. Then v has in-degree 2 in G_A; that is, $b_A(v) \neq r_A(v)$, which implies by (iii) that $b_A(v)$, $r_A(v)$, and v are distinct siblings in D_A. Let w be the child of v in B_A. Since $r_A(w) \neq v$, v,

$r_A(w)$, and z are distinct by (iii). Also $r_A(w) \neq b_A(v)$, since $r_A(w) = b_A(v)$ would imply that $r_A(w)$ dominates w by (ii). Finally, $b_A(v) \neq z$, since $b_A(v)$ is a sibling of v and hence of w in D_A. We conclude that replacing $b_A(w)$ by $b_A(v)$ in line 8 preserves (iii). This replacement preserves (i) since v does not dominate w, it preserves (ii) since it removes v from the path in B_A from s to w. Replacing $b_A(w)$ makes v a leaf in B_A, after which its deletion preserves (i)-(iii).

Now we show that the invariants imply that line 5 can always choose a vertex v. All vertices in $V_A \setminus z$ are leaves in D_A. Let X be the subset of V_A that consists of the vertices x such that $b_A(x) \neq r_A(x)$. Each vertex in X has in-degree 2 in G_A, so there are $2|X|$ edges that enter a vertex in X. By invariant (iii), each edge leaving a vertex in X enters a vertex in X. Invariant (iii) also implies that at least two edges enter X from $V_A \setminus X$. Hence, there are at most $2(|X|-1)$ edges that leave a vertex in X, so there must be a vertex v in X with out-degree at most 1. We claim that v can be selected in line 5. First note that the in-degree of v in G_A exceeds its out-degree in G_A. If v is a leaf in both B_A and R_A then it can be selected. If not, then v must be a leaf in either B_A or R_A, since otherwise its common child w in B_A and R_A would violate (ii). Hence v can be selected in this case also.

Finally, we claim that the computed order is low-high for G_A, such that α^* is first and β^* is last in this order. The latter follows by the assignment in line 2. So the claim is immediate if G_A has three vertices. Suppose, by induction, that this is true if G_A has $k \geq 3$ vertices. Let G_A have $k+1$ vertices and let v be the vertex chosen for deletion. The insertion position of v guarantees that v has the low-high property. All vertices in G_A after the deletion of v have the low-high property in the new $G_A \setminus z$ by the induction hypothesis, so they have the low-high property in the old G_A with the possible exception of w, one of whose incoming edges differs in the old and the new G_A. Suppose $b_A(w)$ differs; the argument is symmetric if $r_A(w)$ differs. Now we have that v, w, $b_A(v)$, and $r_A(w)$ are distinct children of z in D_A. Since w has the low-high property in the new G_A, it occurs in Λ between $r_A(w)$ and $b_A(v)$. Insertion of v next to $b_A(v)$ leaves w between $r_A(w)$ and v, so it has the low-high property in the old G_A as well. ∎

The correctness of algorithm InsertEdge follows from Lemmata 2.10, 2.11 and 2.14.

Lemma 2.15. *Algorithm InsertEdge is correct.*

Proof. Let $(G', D', \delta', \mathit{mark}', \mathit{low}', \mathit{high}')$ be the output of InsertEdge$(G, D, \delta, \mathit{mark}, \mathit{low}, \mathit{high}, e)$. We only need to consider the case where both endpoints of the inserted edge $e = (x, y)$ are reachable in G. Let A be the set of affected vertices, and let $z = \mathit{nca}(x, y)$. Also, let c be the child of z in D that is a common ancestor of all vertices in A. We will show that the computed order δ' is a low-high order of G' that agrees with δ. This fact implies that the arrays mark', low', high' were updated correctly, since their entries did not change for the vertices in $V \setminus \big(A \cup \{c\}\big)$.

By construction, δ' agrees with δ. Let δ_z (resp., δ'_z) be the restriction of δ (resp., δ') to $C(z)$ (resp., $C'(z)$). Then, by Lemma 2.14, δ'_z is an augmentation of δ_z. So, by Lemmata

2.10 and 2.11, δ' is a low-high order in G' for any vertex $v \notin A \cup \{c\}$. Finally, Lemma 2.14 implies that δ' is also a low-high order in G' for the vertices in $A \cup \{c\}$. ∎

Theorem 2.1. *Algorithm* InsertEdge *maintains a low-high order of a flow graph G with n vertices through a sequence of edge insertions in $O(mn)$ total time, where m is the total number of edges in G after all insertions.*

Proof. Consider the insertion of an edge (x, y). If y was unreachable in G then we compute D and a low-high order in $O(m)$ time. Throughout the whole sequence of m insertions, such an event can happen $O(n)$ times, so all insertions to unreachable vertices are handled in $O(mn)$ total time.

Now we consider the cost of executing InsertEdge when both x and y are reachable in G. Let ν be the number of scanned vertices, and let μ be the number of their adjacent edges. We can update the dominator tree and locate the affected vertices (line 8) in $O(\nu + \mu + n)$ time [25]. Computing $z = nca(x, y)$ in line 7 takes $O(n)$ time just by using the parent function d of D. Lines 9–10 and 12 are also executed in $O(n)$ time. The for loop in lines 13–16 takes $O(\nu + \mu)$ since we only need to examine the scanned edges. (Variables $low(c)$ and $high(c)$ need to be updated only if there is a scanned edge entering c.) It remains to account for time to compute G_A and a low-high order of it. From Lemma 2.13, the derived affected flow graph can be constructed in $O(\nu + \mu)$ time. In algorithm AuxiliaryLowHigh, we represent the list Λ with the off-line dynamic list maintenance data structure of [28], which supports insertions (in a given location) and order queries in constant time. With this implementation, AuxiliaryLowHigh runs in linear-time, that is $O(\nu + \mu)$. So InsertEdge runs in $O(\nu + \mu + n)$ time. The $O(n)$ term gives a total cost of $O(mn)$ for the whole sequence of m insertions. We distribute the remaining $O(\nu + \mu)$ cost to the scanned vertices and edges, that is $O(1)$ per scanned vertex or edge. Since the depth in D of every scanned vertex decreases by at least one, a vertex and an edge can be scanned at most $O(n)$ times. Hence, each vertex and edge can contribute at most $O(n)$ total cost through the whole sequence of m insertions. The $O(mn)$ bound follows. ∎

2.2 Applications

2.2.1 Strongly divergent spanning trees and path queries

We can use the arrays *mark*, *low*, and *high* to maintain a pair of strongly divergent spanning trees, B and R, of G after each update. Recall that B and R are *strongly divergent* if for every pair of vertices v and w, we have $B[s, v] \cap R[s, w] = D[s, v] \cap D[s, w]$ or $R[s, v] \cap B[s, w] = D[s, v] \cap D[s, w]$. Moreover, we can construct B and R so that they are also edge-disjoint except for the bridges of G. A *bridge* of G is an edge (u, v) that is contained in every path from s to v. Let $b(v)$ (resp., $r(v)$) denote the parent of a vertex v in B (resp., R). To update B and R after the insertion of an edge (x, y), we only need to update $b(v)$ and $r(v)$ for the affected vertices v, and possibly for their common ancestor

c that is a child of $z = nca(x, y)$ from Lemma 2.8. We can update $b(v)$ and $r(v)$ of each vertex $v \in A \cup \{c\}$ as follows: set $b(v) \leftarrow d(v)$ if $low(v) = null$, $b(v) \leftarrow low(v)$ otherwise; set $r(v) \leftarrow d(v)$ if $high(v) = null$, $r(v) \leftarrow high(v)$ otherwise. If the insertion of (x, y) does not affect y, then $A = \emptyset$ but we may still need to update $b(y)$ and $r(y)$ if $x \notin D(y)$ in order to make B and R maximally edge-disjoint. Note that in this case $z = d(y)$, so we only need to check if both $low(y)$ and $high(y)$ are null. If they are, then we set $low(y) \leftarrow x$ if $x <_\delta y$, and set $high(y) \leftarrow x$ otherwise. Then, we can update $b(y)$ and $r(y)$ as above.

Now consider a query that, given two vertices v and w, asks for two maximally vertex-disjoint paths, π_{sv} and π_{sw}, from s to v and from s to w, respectively. Such queries were used in [53] to give a linear-time algorithm for the 2-disjoint paths problem on a directed acyclic graph. If $v <_\delta w$, then we select $\pi_{sv} \leftarrow B[s, v]$ and $\pi_{sw} \leftarrow R[s, w]$; otherwise, we select $\pi_{sv} \leftarrow R[s, v]$ and $\pi_{sw} \leftarrow B[s, w]$. Therefore, we can find such paths in constant time, and output them in $O(|\pi_{sv}| + |\pi_{sw}|)$ time. Similarly, for any two query vertices v and w, we can report a path π_{sv} from s to v that avoids w. Such a path exists if and only if w does not dominate v, which we can test in constant time using the ancestor-descendant relation in D [49]. If w does not dominate v, then we select $\pi_{sv} \leftarrow B[s, v]$ if $v <_\delta w$, and select $\pi_{sv} \leftarrow R[s, v]$ if $w <_\delta v$.

2.2.2 Fault tolerant reachability

Baswana et al. [] study the following reachability problem. We are given a flow graph $G = (V, E, s)$ and a spanning tree $T = (V, E_T)$ rooted at s. We call a set of edges E' *valid* if the subgraph $G' = (V, E_T \cup E', s)$ of G has the same dominators as G. The goal is to find a valid set of minimum cardinality. As shown in [31], we can compute a minimum-size valid set in $O(m)$ time, given the dominator tree D and a low-high order of δ of it. We can combine the above construction with our incremental low-high algorithm to solve the incremental version of the fault tolerant reachability problem, where G is modified by edge insertions and we wish to compute efficiently a valid set for any query spanning tree T. Let $t(v)$ be the parent of v in T. Our algorithm maintains, after each edge insertion, a low-high order δ of G, together with the *mark*, *low*, and *high* arrays. Given a query spanning tree $T = (V, E_T)$, we can compute a valid set of minimum cardinality E' as follows. For each vertex $v \neq s$, we apply the appropriate one of the following cases: (a) If $t(v) = d(v)$ then we do not insert into E' any edge entering v. (b) If $t(v) \neq d(v)$ and v is marked then we insert $(d(v), v)$ into E'. (c) If v is not marked then we consider the following subcases: If $t(v) >_\delta v$, then we insert into E' the edge (x, v) with $x = low(v)$. Otherwise, if $t(v) <_\delta v$, then we insert into E' the edge (x, v) with $x = high(v)$. Hence, can update the minimum valid set in $O(mn)$ total time.

We note that the above construction can be easily generalized for the case where T is forest, i.e., when E_T is a subset of the edges of some spanning tree of G. In this case, $t(v)$ can be null for some vertices $v \neq s$. To answer a query for such a T, we apply the previous construction with the following modification when $t(v)$ is null. If v is marked then we insert $(d(v), v)$ into E', as in case (b). Otherwise, we insert both edges entering v from

$low(v)$ and $high(v)$. In particular, when $E_T = \emptyset$, we compute a subgraph $G' = (V, E', s)$ of G with minimum number of edges that has the same dominators as G. This corresponds to the case $k = 1$ in [7].

Chapter 3

Sparse Subgraphs for 2-Connectivity in Directed Grpahs

In this chapter, we consider the problem of computing the smallest strongly connected spanning subgraph of G that maintains the pairwise 2-vertex-connectivity of G. We consider two cases. In the first case G is 2-vertex connected and we provide a 2 approximation ratio algorithm in order to find the smallest 2-vertex-connected spanning subgraph (2VCSS) of G. In the second case G is not 2-vertex connected and we provide linear-time approximation algorithms that achieve an approximation ratio of 6t and maintain the pairwise 2-vertex connectivity of G. So, we show how to approximate, in linear time, within a factor of 6 the smallest strongly connected spanning subgraph of G that maintains respectively: both the 2-vertex-connected blocks and the 2-vertex-connected components of G (2VC-B-C); all the 2-connectivity relations of G (2C) and provide heuristics that improve the size of the computed subgraphs in practice.

3.1 Approximation algorithm of 2VCSS

Let $G = (V, E)$ be a strongly connected digraph. A vertex x of G is a *strong articulation point* if $G \setminus x$ is not strongly connected. A strongly connected digraph G is *2-vertex-connected* if it has at least three vertices and no strong articulation points [20, 37]. Here we consider the problem of approximating a smallest 2-vertex-connected spanning subgraph (2VCSS) of G. This problem is NP-hard [18]. We show that algorithm LH-Z

Algorithm 6: LH-Z(G)

```
Input: 2-vertex-connected digraph $G = (V, E)$
Output: 2-approximation of a smallest 2-vertex-connected spanning subgraph
        $H = (V, E_H)$ of $G$
1  Choose an arbitrary vertex $s$ of $G$ as start vertex.
2  Compute a strongly connected spanning subgraph $H = (V \setminus s, E_H)$ of $G \setminus s$.
3  Set $H \leftarrow (V, E_H)$.
4  Compute a low-high order $\delta$ of flow graph $G$ with start vertex $s$.
5  foreach vertex $v \neq s$ do
6      if there are two edges $(u, v)$ and $(w, v)$ in $E_H$ such that $u <_\delta v$ and $v <_\delta w$ then
7          do nothing
8      end
9      else if there is no edge $(u, v) \in E_H$ such that $u <_\delta v$ then
10         find an edge $e = (u, v) \in E$ with $u <_\delta v$
11         set $E_H \leftarrow E_H \cup \{e\}$
12     end
13     else if there is no edge $(w, v) \in E_H$ such that $v <_\delta w$ then
14         find an edge $e = (w, v) \in E$ with $v <_\delta w$ or $w = s$
15         set $E_H \leftarrow E_H \cup \{e\}$
16     end
17 end
18 Execute the analogous steps of lines 4–17 for the reverse flow graph $G^R$ with start
   vertex $s$.
19 return $H = (V, E_H)$
```

(given below), which uses low-high orders, achieves a linear-time 2-approximation for this problem. The best previous approximation ratio achievable in linear-time was 3 [21], so we obtain a substantial improvement. The best approximation ratio for 2VCSS is 3/2, and is achieved by the algorithm of Cheriyan and Thurimella [9] in $O(m^2)$ time, or in $O(m\sqrt{n} + n^2)$ by a combination of [9] and [21]. Computing small spanning subgraphs is of particular importance when dealing with large-scale graphs, e.g., with hundreds of million to billion edges. In this framework, one big challenge is to design linear-time algorithms, since algorithms with higher running times might be practically infeasible on today's architectures. Let $G = (V, E)$ be a strongly connected digraph. In the following, we denote by $G^R = (V, E^R)$ the *reverse digraph* of G that results from G after reversing all edge directions.

Lemma 3.1. *Algorithm LH-Z computes a 2-vertex-connected spanning subgraph of G.*

Proof. We need to show that the computed subgraph H is 2-vertex-connected. From [37], we have that a digraph H is 2-vertex connected if and only if it satisfies the following property: For an arbitrary start vertex $s \in V$, flow graphs $H = (V, E, s)$ and $H^R =$

(V, E^R, s) have flat dominator trees, and $H \setminus s$ is strongly connected. The digraph H computed by algorithm LH-Z satisfies the latter condition because of line 2. It remains to show that H has flat dominator tree. The same argument applies for H^R, thus completing the proof. Let δ be the low-high order δ of G, computed in line 3. We argue that after the execution of the for loop in lines 5–17, δ is also a low-high order for all vertices in H. Consider an arbitrary vertex $v \neq s$. Let (x, v) be an edge entering v in the strongly connected spanning subgraph of G computed in line 2. If $x >_\delta v$, then, by the definition of δ, there is at least one edge $(y, v) \in E$ such that $y <_\delta v$. Hence, after the execution of the for loop for v, the edge set E_H will contain at least two edges (u, v) and (w, v) such that $u <_\delta v <_\delta w$. On the other hand, if $x <_\delta v$, then the definition of δ implies that there an edge $(y, v) \in E$ such that $y >_\delta v$ or $y = s$. Notice that in either case $y \neq x$. So, again, after the execution of the for loop for v, the edge set E_H will contain at least two edges (u, v) and (w, v) such that either $u <_\delta v <_\delta w$, or $u <_\delta v$ and $w = s$. It follows that δ is a low-high order for all vertices $v \neq s$ in H. By [28], this means that H contains two strongly divergent spanning trees B and R of G. Since G has flat dominator tree, we have that $B[s, v] \cap R[s, v] = \{s, v\}$ for all $v \in V \setminus s$. Hence, since H contains B and R, the dominator tree of H is flat. ∎

We remark that the construction of H in algorithm LH-Z guarantees that s will have in-degree and out-degree at least 2 in H. (This fact is implicit in the proof of Lemma 3.1.) Indeed, H will contain the edges from s to the vertices in $V \setminus s$ with minimum and maximum order in δ, and the edges entering s from the vertices in $V \setminus s$ with minimum and maximum order in δ^R.

Theorem 3.1. *Algorithm LH-Z computes a 2-approximation for 2VCSS in linear time.*

Proof. We establish the approximation ratio of LH-Z by showing that $|E_H| \leq 4n$. The approximation ratio of 2 follows from the fact that any vertex in a 2-vertex-connected digraph must have in-degree at least two. In line 2 we can compute an approximate smallest strongly connected spanning subgraph of $G \setminus s$ [40]. For this, we can use the linear-time algorithm of Zhao et al. [55], which selects at most $2(n-1)$ edges. Now consider the edges selected in the for loop of lines 5–17. Since after line 2, $H \setminus s$ is strongly connected, each vertex $v \in V \setminus s$ has at least one entering edge (x, v). If $x <_\delta v$ then lines 10–11 will not be executed; otherwise, $v <_\delta x$ and lines 14–15 will not be executed. Thus, the for loop of lines 5–17 adds at most one edge entering each vertex $v \neq s$. The same argument implies that the analogous steps executed for G^R add at most one edge leaving each vertex $v \neq s$. Hence, E_H contains at most $4(n-1)$ at the end of the execution. ∎

3.2 Approximation algorithms and heuristics for 2VC-B

Let $G = (V, E)$ be the input strongly connected digraph. In problem 2VC-B, we wish to compute a strongly connected spanning subgraph G' of G that has the same 2-vertex-connected blocks of G, with as few edges as possible. We consider the following approach. Start with the empty graph $G' = (V, \emptyset)$, and add as few edges as possible until G' is guaranteed to have the same 2-vertex-connected blocks as G. We consider three linear-time algorithms that apply this approach. The first two are based on the sparse certificates for 2-vertex-connected blocks from [24, 27], which use divergent spanning trees. The third is a new algorithm that selects the edges of G' with the help of low-high orders.

Divergent Spanning Trees. We can compute a sparse certificate $C(G)$ for the 2-vertex-connected blocks of a strongly connected digraph G using the algorithm of [24], which is based on a linear-time construction of two divergent spanning trees of a flow graph [28]. We refer to this algorithm as DST-B. 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, and by $D(s)$ and $D^R(s)$ the dominator trees of $G(s)$ and $G^R(s)$ respectively. Also, let $C(v)$ and $C^R(v)$ be 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, where $C^0(r) = \{r\}$, $C^1(r) = C(r)$, and so on. 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. 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. In order to bound the size of all auxiliary graphs, we eliminate parallel edges during those contractions. We call an edge $e \in E_r \setminus E$ a *shortcut* edge of G_r. That is, a shortcut edge is formed by the contraction of a part of G into an auxiliary vertex of G_r. Thus, a shortcut edge is not an original edge of G but corresponds to at least one original edge, and is adjacent to at least one auxiliary vertex.

Algorithm DST-B selects the edges that are inserted into $C(G)$ in three phases. During the construction, the algorithm may choose a shortcut edge or a reverse edge to be inserted into $C(G)$. In this case we insert the associated original edge instead. Also, an edge may be selected multiple times, so we remove multiple occurrences of such edges in a postprocessing step. In the first phase, we insert into $C(G)$ the edges of two maximally edge-disjoint divergent spanning trees, $T_1(G(s))$ and $T_2(G(s))$ of $G(s)$. In the second phase we process the auxiliary graphs of $G(s)$ that we refer to as the *first-level auxiliary graphs*. For each such auxiliary graph $H = G_r$, we compute two maximally edge-disjoint divergent spanning trees $T_1(H^R(r))$ and $T_2(H^R(r))$ of the corresponding reverse flow graph $H^R(r)$ with start vertex r. We insert into $C(G)$ the edges of these two spanning trees. It can be proved that, at the end of this phase, $C(G)$ induces a strongly connected spanning subgraph of G. Finally, in the last phase we process the *second-level auxiliary graphs*, which are the auxiliary graphs of H^R for all first-level auxiliary graphs H. Let H_q^R be a second-level auxiliary graph of H^R. For every strongly connected component S of $H_q^R \setminus q$, we choose an arbitrary vertex $v \in S$ and compute a spanning tree of S and a spanning tree of S^R, and insert their edges into $C(G)$.

This construction inserts $O(n)$ edges into $C(G)$, and therefore achieves a constant approximation ratio for 2VC-B. However, due to the use of auxiliary vertices and two levels of auxiliary graphs, we do not have a good bound for this constant. (The first-level auxiliary graphs have at most $4n$ vertices and $4m + n$ edges in total [24].) We propose a modification of DST-B, that we call DST-B modified: For each auxiliary graph, we do not select in $C(G)$ the edges of its two divergent spanning trees that have only auxiliary descendants. Also, for every second-level auxiliary graph, during the computation of its strongly connected components we include the chosen edges that already form a strongly connected component.

Divergent Spanning Trees and Loop Nesting Trees. An alternative linear-time algorithm to compute a sparse certificate $C(G)$ for the 2-vertex-connected blocks can be obtained via loop nesting trees, as described in [27]. As in algorithm DST-B, we compute two maximally edge-disjoint divergent spanning trees T_1 and T_2 of $G(s)$, and insert their edges into $C(G)$. But instead of computing auxiliary graphs, we compute a loop nesting tree L of $G(s)$ and insert into $C(G)$ the edges that define L. These are the edges of a dfs tree of $G(s)$, and at most $n - 1$ additional edges that are required to define the loops of $G(s)$. (See [28, 51] for the details.) Then, we repeat the same process in the reverse direction, i.e., for $G^R(s)$. As shown in [27], a spanning subgraph having the same dominator trees and loop nesting trees (in both directions) as the digraph G, has the same 2-edge- and 2-vertex-connected blocks as G. We refer to this algorithm as DLN-B.

Theorem 3.2. *Algorithm DLN-B achieves an approximation ratio of* 6*, in linear time, for problem 2VC-B.*

Proof. Consider first the "forward" pass of the algorithm. It adds at most $2(n - 1)$ edges for the two divergent spanning trees, and at most $2(n-1)$ edges that define a loop nesting tree of $G(s)$. By [28, 51], both these constructions use the edges of a dfs tree of $G(s)$ and some additional edges. Hence, we can use the same dfs tree to compute the divergent spanning trees and the loop nesting tree. This gives a total of at most $3(n - 1)$ edges. Similarly, the "reverse" pass computes at most $3(n-1)$ edges, so algorithm DLN-B selects at most $6(n-1)$ edges. Since the resulting subgraph must be strongly connected, any valid solution to problem 2VC-B has at least n edges, so DLN-B achieves a 6-approximation. By [28, 51], both the computation of a pair of divergent spanning trees and of a loop nesting tree can be done in linear time, hence DLN-B also runs in linear time. ▮▮

Low-High Orders and Loop Nesting Trees. Now we introduce a new linear-time construction of a sparse certificate, via low-high orders, that we refer to as LHL-B. The algorithm consists of two phases. In the first phase, we insert into $C(G)$ the edges that define the loop nesting trees L and L^R of $G(s)$ and $G^R(s)$, respectively, as in algorithm DLN-B. In the second phase, we insert enough edges so that $C(G)$ (resp., $C^R(G)$) maintains a low-high order of $G(s)$ (resp., $G^R((s))$. Let δ be a low-high order on $G(s)$. Subgraph

$C(G)$ satisfies the low-high order δ if, for each vertex $v \neq s$, one of the following holds: (a) there are two edges (u, v) and (w, v) in $C(G)$ such that $u <_\delta v$, $v <_\delta w$, and w is not a descendant of v in $D(s)$; (b) $(d(v), v)$ is a strong bridge of G and is contained in $C(G)$; or (c) $(d(v), v)$ is an edge of G that is contained in $C(G)$, and there is another edge (u, v) in $C(G)$ such that $u <_\delta v$ and $u \neq d(v)$.

Theorem 3.3. *Algorithm LHL-B is correct and achieves an approximation ratio of* 6 *for problem 2VC-B, in linear time.*

Proof. By construction, the sparse certificate $C(G)$ computed by LHL-B satisfies a low-high order δ of $G(s)$. This implies that $C(G)$ contains two divergent spanning trees T_1 and T_2 of $G(s)$ [28]. Moreover, cases (b) and (c) of the construction ensure that T_1 and T_2 are maximally edge-disjoint. This is because when case (a) does not apply for a vertex v, then $C(G)$ contains $(d(v), v)$. Also, $d(v)$ is the only vertex u that satisfies $u <_\delta v$ if and only if $(d(v), v)$ is a strong bridge. Hence, $C(G)$ indeed contains two maximally edge-disjoint divergent spanning trees of $G(s)$. Similarly, $C(G)$ also contains two maximally edge-disjoint divergent spanning trees of $G^R(s)$. So the correctness of LHL-B follows from the fact that DLN-B is correct.

Next we bound the approximation ratio of LHL-B. The edges selected to maintain a loop nesting tree L of $G(s)$ contain at least one entering edge for each vertex $v \neq s$. This means that it remains to include at most one edge for each vertex $v \neq s$ in order to satisfy a low-high order of $G(s)$. The symmetric arguments holds for the reverse direction as well, so $C(G)$ contains at most $6(n-1)$ edges, which gives an approximation ratio of 6. ∎

We note that both DLN-B and LHN-B also maintain the 2-edge-connected blocks of the input digraph. We use this fact in Section 3.3, where we compute a sparse subgraph that maintains all 2-connectivity relations. We can improve the solution computed by the above algorithms by using the following filter.

Two Vertex-Disjoint Paths Test. We test if $G' \setminus (x, y)$ contains two vertex-disjoint paths from x to y. If this is the case, then we remove edge (x, y); otherwise, we keep the edge (x, y) in G' and proceed with the next edge. For doing so, we define the modified graph G'' of G' after vertex-splitting (see, e.g., [1]): for each vertex v, replace v by two vertices v^+ and v^-, and add the edge (v^-, v^+). Then, we replace each edge (u, w) in G' by (u^+, v^-) in G'', so v^- has the edges entering v and v^+ has the edges leaving v. Now we can test if G' still has two vertex-disjoint paths from x to y after deleting (x, y) by running two iterations of the Ford-Fulkerson augmenting paths algorithm [15] for finding two edge-disjoint paths on G'' by treating x^+ as the source and y^- as the sink. Note that we need to compute G'' once for all such tests. If an edge (x, y) is deleted from G', then we also delete (x^+, y^-) from G''. Since G' has $O(n)$ edges, this test takes $O(n)$ time per edge, so the total running time is $O(n^2)$. We refer to this filter as 2VDP. In our implementations we applied 2VDP on the outcome of DLN-B in order to assess our algorithms with a solution close to minimum. For the 2VC-B problem the algorithm obtained after applying such a filter is called 2VDP-B. In order to improve the running time of 2VDP in practice, we apply a speed-up heuristic for *trivial edges* (x, y): if x belongs to a 2-vertex-connected block and has outdegree two or y belongs to a 2-vertex-connected block and has indegree two, then (x, y) must be included in the solution.

3.3 Approximation algorithms and heuristics for 2C

To get an approximate solution for problem 2C, we combine our algorithms for 2VC-B with algorithms that approximate 2VCSS [9, 21]. We also take advantage of the fact that every 2-vertex-connected component is contained in a 2-edge-connected component. This property suggests the following approach for 2C. First, we compute the 2-vertex-connected components of G and solve the 2VCSS problem independently for each such component. Then, we apply one of the algorithms DLN-B or LHL-B for 2VC-B on G. Since the sparse certificate from DLN-B or LHL-B also maintain the 2-edge-connected blocks, it remains to include edges that maintain the 2-edge-connected components of G. We can find these edges in a *condensed graph* $\breve{G}$ defined as follows. Digraph $\breve{G}$ is formed from G by contracting each 2-vertex-connected component of G into a single supervertex. Note that any two 2-vertex-connected components may have at most one vertex in common: if two such components share a vertex, they are contracted into the same supervertex. The resulting digraph $\breve{G}$ is a multigraph since the contractions can create loops and parallel edges. For any vertex v of G, we denote by $\breve{v}$ the supervertex of $\breve{G}$ that contains v. Every edge $(\breve{u}, \breve{v})$ of $\breve{G}$ is associated with the corresponding original edge (u, v) of G. Now we describe the main steps of our algorithm for 2C:

1. Compute the 2-vertex-connected components. Solve independently the 2VCSS problem for each such component, using the linear-time algorithm of [21].

2. Form the condensed multigraph $\breve{G}$, and compute its 2-edge-connected components. Solve independently the 2ECSS problem for each such component, using edge-disjoint spanning trees [26].

3. Execute the DNL-B or LHL-B algorithms on the original graph G and compute a sparse certificate for the 2-edge- and the 2-vertex-connected blocks.

The solution to the 2C problem consists of the edges selected in each step of the algorithm. Note that in Step 2, we should allow 2-edge-connected components of size two because such a component may correspond to the union of 2-vertex-connected components of the original graph. We consider two versions of our algorithm, DLN-2C and LHL-2C, depending on the algorithm for the 2VC-B problem used in Step 3.

Theorem 3.4. *Algorithms DLN-2C and LHL-2C compute a 6-approximation for problem 2C. Moreover, if the 2-edge- and the 2-vertex- connected components of G are available, then the algorithms run in linear time.*

Proof. Let n_v be the number of vertices of G that belong to some 2-vertex-connected component of G. Also, let $\breve{n}$ be the number of vertices in $\breve{G}$, and let $\breve{n}_e$ be the number of vertices of $\breve{G}$ that belong to some 2-edge-connected component of $\breve{G}$. By the analysis in the proof of Theorem 3.5, the algorithm for 2VC-B-C selects less than $6(n + n_v)$ edges. For the 2ECSS problems, we can compute a 2-approximate solution in linear-time as in [26], using edge-disjoint spanning trees [13, 51]. Let $\breve{C}$ be a 2-edge-connected component of $\breve{G}$. We select an arbitrary vertex $\breve{v} \in \breve{C}$ as a root and compute two edge-disjoint spanning trees in the flow graph $\breve{C}(\breve{v})$ and two edge-disjoint spanning trees in the reverse flow graph $\breve{C}^R(\breve{v})$. Thus, we select less than $4\breve{n}_e$ edges. Hence, the subgraph computed by the algorithm has less than $6(n + n_c + \breve{n}_e)$ edges.

Now consider any solution to 2C. It has to include $2n_c + 2\breve{n}_e$ edges in order to maintain the 2-vertex and the 2-edge-connected components of G. Moreover, since the resulting subgraph must be strongly connected, there must be at least one edge entering each of the $\breve{n} - \breve{n}_e$ vertices of $\breve{G}$ that do not belong in a 2-edge-connected component of $\breve{G}$. Thus, the optimal solution has at least $2n_c + \breve{n}_e + \breve{n}$ edges. Note that $\breve{n}_c + \breve{n} \geq n$, so the the optimal solution has at least $n + n_c + \breve{n}_e$ edges and the approximation ratio of 6 follows.

Finally, we show that all three steps of the algorithms DLN-2C and LHL-2C run in linear time given the 2-edge- and the 2-vertex- connected components of G. This is immediate for Steps 1 and 3. In Step 2, we do not need to compute the 2-edge-connected components of $\breve{G}$ from scratch, but we can form them from the 2-edge-connected components of G using contractions. Let C be a 2-edge-connected component of G. We contract each 2-vertex-connected component of G contained in C into a single supervertex. Then, the resulting digraph $\breve{C}$ is a 2-edge-connected component of $\breve{G}$. ∎

If we wish to improve the quality of the computed solution G', we can apply the 2VDP filter, and the analogous 2-edge-disjoint paths filter 2EDP, as follows. In Step 1, we run the 2VDP filter for the edges computed by the linear-time algorithm of [21]. This produces a minimal solution for 2VCSS in each 2-vertex-connected component of G. Similarly, in Step 2, we run the 2EDP filter for the edges of the edge-disjoint spanning trees computed in each 2-edge-connected component of $\breve{G}$. This produces a minimal solution for 2ECSS in each 2-edge-connected component of $\breve{G}$. Finally, we run the 2VDP filter on the whole G', but only consider the edges added in Step 3 of our algorithm, since the edges from Steps 1 and 2 are needed to maintain the 2-vertex- and the 2-edge-connected components. We implemented this algorithm, using DLN-B for Step 3, and refer to it as 2VDP-2C.

Theorem 3.5. *There is a polynomial-time algorithm for 2VC-B-C that achieves an approximation ratio of* 6. *Moreover, if the* 2*-vertex-connected components of* G *are available, then the algorithm runs in linear time.*

Proof. A result in [21] shows that, given a 2-vertex-connected digraph with ν vertices, we can compute in linear time a 2-vertex-connected spanning subgraph that has less than 6ν edges. Hence, if n_c is the number of vertices that belong in a 2-vertex-connected component of G, then applying this algorithm to each 2-vertex-connected component selects less than $6n_c$ edges. Finally, we apply the construction of a sparse certificate for the 2-vertex-connected blocks which selects at most $6(n-1)$ edges by Theorems 3.2 or 3.3. Hence, the subgraph computed by the algorithm has less than $6(n+n_c)$. One the other hand, any solution to 2VC-B-C has to include at least $2n_c$ edges for the 2-vertex-connected components of G, and at least $n - n_c$ edges in order to obtain a strongly connected subgraph. Thus, the optimal solution has at least $n + n_c$ edges, so the approximation ratio of 6 follows. ▮▮

Approximation algorithms and heuristics for 2VC-B-C. Executing Steps 1 and 3 of the above algorithm described for 2C, is enough to produce a certificate for the 2VC-B-C problem. If we use DLN-B or LHL-B for Step 3, then we obtain a 6-approximate solution for 2VC-B-C. We call the corresponding algorithms DLN-B-C and LHL-B-C, respectively. As in the 2VC-B and 2C problems, we can improve the quality of the computed solution by applying the 2VDP filter for the edges that connect different 2-vertex-connected components. We implemented this algorithm, using DLN-B for Step 3, and refer to it as 2VDP-B-C.

Chapter 4

Experimental Study

In this chapter we conduct an experimental study, implementing efficiently our algorithms that have been induced to this work, on real world graphs taken from a variety of application areas. We present an experimental study on our incremental low-high order algorithms and on 2-vertex-connected spanning subgraphs which succeed the best sparse with low-high orders. We also present an experimental study on computing strongly connected spanning subgraphs that maintain certain 2-connectivity requirements.

4.1 Experimental analysis of incremental low-high order algorithms

For the experimental evaluation we use the graph datasets shown in Table 4.1. 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 core. We report CPU times measured with the `getrusage` function, averaged over ten different runs. In Table 4.1 we can see some statistics about the real-world graphs we used in our experimental evaluation.

Graph	**Largest SCC**			**2VCCs**			**Type**
	n	m	avg. δ	n	m	avg. δ	
rome99	3352	8855	2.64	2249	6467	2.88	road network
twitter-higgs-retweet	13086	63537	4.86	1099	9290	8.45	twitter
enron	8271	147353	17.82	4441	123527	27.82	enron mails
web-NotreDame	48715	267647	5.49	1409	6856	4.87	web
				1462	7279	4.98	
				1416	13226	9.34	
soc-Epinions1	32220	442768	13.74	17117	395183	23.09	trust network
Amazon-302	241761	1131217	4.68	55414	241663	4.36	co-purchase
WikiTalk	111878	1477665	13.21	49430	1254898	25.39	Wiki communications
web-Stanford	150475	1576157	10.47	5179	129897	25.08	web
				10893	162295	14.90	
web-Google	434818	3419124	7.86	77480	840829	10.85	web
Amazon-601	395230	3301051	8.35	276049	2461072	8.92	co-purchase
web-BerkStan	334857	4523232	13.51	1106	8206	7.42	web
				4927	28142	5.71	
				12795	347465	27.16	
				29145	439148	15.07	

Table 4.1: Real-world graphs used in the experiments, sorted by the file size of their largest SCC. We used both the largest SCC and the some of the 2VCCs (inside the largest SCC) in our experiments.

We compare the performance of four algorithms. As a baseline, we use a static low-high order algorithm from [28] based on an efficient implementation of the Lengauer-Tarjan algorithm for computing dominators [41] from [29]. Our baseline algorithm, SLT, constructs, as intermediary, two divergent spanning trees. After each insertion of an edge (x, y), SLT recomputes a low-high order if x is reachable. An improved version of this algorithm, that we refer to as SLT-NCA, tests if the insertion of (x, y) affects the dominator tree by computing the nearest common ancestor of x and y. If this is the case, then SLT-NCA recomputes a low-high order as SLT. The other two algorithms are the ones we presented in Section 2. For our simple algorithm, DBS-DST, we extend the incremental dominators algorithm DBS of [25] with the computation of two divergent spanning trees and a low-high order, as in SLT. Algorithm DBS-DST applies these computations on a sparse subgraph of the input digraph that maintains the same dominators. Finally, we tested an implementation of our more efficient algorithm, DBS-AUX, that updates the low-high order by computing a local low-high order of an auxiliary graph.

We compared the above incremental low-high order algorithms in two different field tests. In the first one, we considered 2-vertex connected graphs, and we dynamized them in the following manner: we removed a percentage of edges (i.e., 5%, 10%, and 20% respectively), selected uniformly at random, that were incrementally added to the graph. Note that during the execution of the algorithms some vertices may be unreachable at first. Also, at the end of all insertions, the final graph has flat dominator tree. In Figure 4.1 (top) we can see that the algorithms are well distinguished: our DBS-AUX performs consistently better than the other ones (with the exception of two NotreDame instances). The total running times are given in Table 4.2. On average, DBS-DST is about 2.84 times faster than SLT-NCA, with their relative performance depending on the

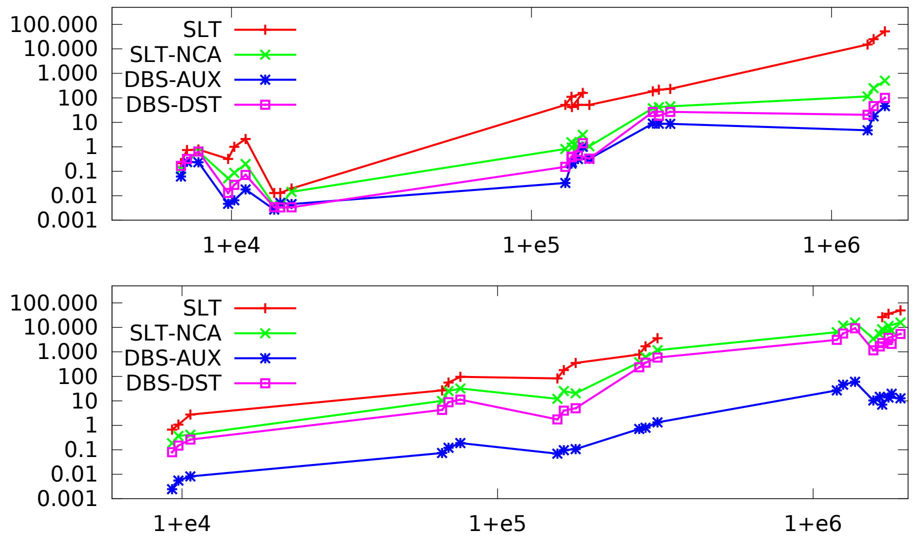

Figure 4.1: Incremental low-high order: dynamized 2VC graphs (top) and edge insertion in strongly connected graphs (bottom). Running times, in seconds, and number of edges both shown in logarithmic scale.

density of the graph (the higher the average degree the better DBS-DST performs w.r.t. SLT-NCA.) As we mentioned, the naive SLT is the worst performer. The above observed behavior of the algorithms is similar also in the second test. Here, we consider the strongly connected graphs, and we incrementally insert random edges up to a certain percentage of the original number of edges (i.e., as before, 5%, 10%, and 20% respectively). We use strongly connected graphs only in order to guarantee that all vertices are reachable from the selected source. (Strong connectivity has no other effect in these tests.) The endpoints of each new edge are selected uniformly at random, and the edge is inserted if it is not a loop and is not already present in the current graph. The ranking of the algorithms does not change, as we can see in Figure 4.1 (bottom), but the difference is bigger: we note a bigger gap of more than two orders of magnitude, in particular, between DBS-AUX and the couple SLT-NCA and DBS-DST. This is expected because, unlike the first test, here all edges connect already reachable vertices. This means that DBS-DST and DBS-AUX do not execute a full restart for any of these insertions. The total running times are given in Table 4.3.

4.2 Experimental analysis of 2-vertex-connected spanning subgraphs

In this experimental evaluation we compared four algorithms for computing the (approximated) smallest 2-vertex-connected spanning subgraph. The dataset and the machine that we use to conduct the experiments are the same with these in the first section. Specifically, we tested two algorithms from [21], FAST which computes a 3-approximation in

Graph	**nodes**	**starting edges**	**final edges**	**SLT**	**SLT-NCA**	**DBS-AUX**	**DBS-DST**
rome05	2249	6144	6467	0.216091	0.120026	0.060632	0.16457
rome10	2249	5820	6467	0.734963	0.231678	0.242326	0.319025
rome20	2249	5174	6467	0.772791	0.646389	0.231463	0.639627
twitter05	1099	8826	9290	0.320313	0.051123	0.004682	0.012953
twitter10	1099	8361	9290	0.996879	0.085982	0.006498	0.027945
twitter20	1099	7432	9290	2.06744	0.198226	0.018012	0.070981
NotreDame05	1416	12565	13226	0.012942	0.003981	0.002727	0.003421
NotreDame10	1416	11903	13226	0.012958	0.003997	0.005094	0.003341
NotreDame20	1416	10581	13226	0.019733	0.01446	0.004571	0.003374
enron05	4441	117351	123527	51.5453	0.811483	0.033019	0.152272
enron10	4441	111174	123527	109.719	1.5252	0.204307	0.388753
enron20	4441	98822	123527	158.83	3.08813	0.999617	1.40979
webStanford05	5179	123402	129897	42.1674	0.936905	0.236135	0.370119
webStanford10	5179	116907	129897	51.2838	1.02925	0.316648	0.439147
webStanford20	5179	103918	129897	51.6162	1.04364	0.323679	0.329067
Amazon05	55414	229580	241663	185.868	37.4155	8.91418	26.5169
Amazon10	55414	217497	241663	214.185	41.4565	8.80395	18.4656
Amazon20	55414	193330	241663	230.7	44.8627	8.66914	26.7402
WikiTalk05	49430	1192153	1254898	15026.2	113.946	4.7007	20.2353
WikiTalk10	49430	1129408	1254898	24846.2	247.601	17.5997	45.7164
WikiTalk20	49430	1003918	1254898	51682	500.581	45.9101	99.6058

Table 4.2: Running times of the plot shown in Figure 4.1 (top) .

Graph	**nodes**	**starting edges**	**final edges**	**SLT**	**SLT-NCA**	**DBS-AUX**	**DBS-DST**
rome05	3352	8855	9298	0.662001	0.185072	0.002457	0.07962
rome10	3352	8855	9741	1.06533	0.366822	0.005531	0.147052
rome20	3352	8855	10626	2.74201	0.410448	0.008154	0.259299
twitter05	13086	63537	66714	26.5965	9.94862	0.073755	4.2933
twitter10	13086	63537	69891	55.4924	25.189	0.120719	8.73372
twitter20	13086	63537	76244	96.3205	31.7239	0.186917	11.1431
enron05	8271	147353	154721	82.6084	11.9889	0.068994	1.72764
enron10	8271	147353	162088	180.222	25.0999	0.09557	3.97011
enron20	8271	147353	176824	353.174	20.1978	0.106017	4.94514
NotreDame05	48715	267647	281029	785.012	375.356	0.70628	234.757
NotreDame10	48715	267647	294412	1691.05	610.29	0.79135	359.028
NotreDame20	48715	267647	321176	3593.09	1168.5	1.31932	585.807
Amazon05	241761	1131217	1187778	>24h	6386.97	26.5493	3094.69
Amazon10	241761	1131217	1244339	>24h	11905.7	45.1881	5628.51
Amazon20	241761	1131217	1357460	>24h	15871	60.197	9157.5
WikiTalk05	111878	1477665	1551548	>24h	3414.28	10.3364	1157.84
WikiTalk10	111878	1477665	1625432	>24h	5301.51	14.5151	1666.28
WikiTalk20	111878	1477665	1773198	>24h	7296.72	19.5778	2124.6
webStanford05	150475	1576157	1654965	>24h	8403.03	7.028	2295.55
webStanford10	150475	1576157	1733773	>24h	11503.4	13.7287	3749.12
webStanford20	150475	1576157	1891388	>24h	15792.1	12.7093	5381.12

Table 4.3: Running times of the plot shown in Figure 4.1 (bottom) .

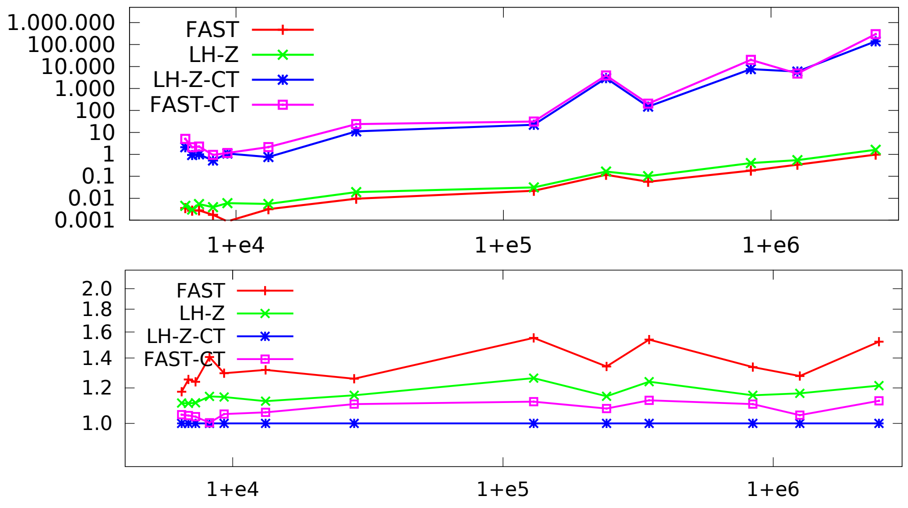

Figure 4.2: Smallest 2-vertex-connected spanning subgraph. Top: running times, in seconds, and number of edges both shown in logarithmic scale. Bottom: relative size of the resulting 2VCSS.

linear-time by using divergent spanning trees, and FAST-CT which combines FAST with the 3/2-approximation algorithm of Cheriyan and Thurimella [9]. In the experiments reported in [21], the former algorithm achieved the fastest running times, while the latter the best solution quality. We compare these algorithms against our new algorithm LH-Z of Section 3.1, and a new hybrid algorithm LH-Z-CT, that combines LH-Z with the algorithm of Cheriyan and Thurimella [9].

Algorithm LH-Z-CT works as follows. First, it computes a 1-matching M in the input graph G [17], using bipartite matching as in [9]. Let H be the subgraph of $G \setminus s$, for arbitrary start vertex s, that contains only the edges in M. We compute the strongly connected components $C_1, \ldots, C_k$ in H, and form a contracted version G' of G as follows. For each strongly connected component C_i of H, we contract all vertices in C_i into a representative vertex $u_i \in C_i$. (Contractions are performed by union-find [50] and merging lists of out-edges of G.) Then, we execute the linear-time algorithm of Zhao et al. [55] to compute a strongly connected spanning subgraph of G', and store the original edges of G that correspond to the selected edges by the Zhao et al. algorithm. Let Z be this set of edges. We compute a low-high order of G with root s, and use it in order to compute a 2-vertex-connected spanning subgraph W of G using as many edges from Z and M as possible, as in LH-Z. Then, we run the filtering phase of Cheriyan and Thurimella. For each edge (x, y) of W that is not in M, we test if x has two vertex-disjoint paths to y in $W \setminus (x, y)$. If it does, then we set $W \leftarrow W \setminus (x, y)$. We remark that, similarly to FAST-CT, LH-Z preserves the 3/2 approximation guarantee of the Cheriyan-Thurimella algorithm for $k = 2$ and improves its running time from $O(m^2)$ to $O(m\sqrt{n}+n^2)$, for a digraph with n vertices and m arcs. In our implementation, the bipartite matching is computed via max-flow, using an implementation of the Goldberg-Tarjan push-relabel algorithm [33] from [12], which is very fast in practice. (This implementation was provided by the authors of

Graph	**Size**		**FAST**		**LH-Z**		**LH-Z-CT**		**FAST-CT**	
	nodes	edges	time	edges	time	edges	time	edges	time	edges
rome99	2249	6467	0.003581	5691	0.004513	5370	2.099891	4837	5.042213	5057
web-NotreDame$_1$	1409	6856	0.0027216	3796	0.0030316	3356	0.927683	3029	2.119423	3153
web-NotreDame$_2$	1462	7279	0.002784	3949	0.005436	3545	0.999031	3189	2.262008	3300
web-BerkStan$_1$	1106	8206	0.001753	3423	0.00393	2795	0.525704	2433	0.949573	2440
twitter-higgs-retweet	1099	9290	0.000885	3553	0.005976	3143	1.087169	2745	1.153608	2879
web-NotreDame$_3$	1416	13226	0.003182	4687	0.005515	3990	0.751014	3560	2.143551	3768
web-BerkStan$_2$	4927	28142	0.009555	13391	0.018985	12296	11.223287	10646	23.887951	11750
web-Stanford	5179	129897	0.022056	17940	0.031001	14583	22.141856	11556	31.346059	12920
Amazon-302	55414	241663	0.11804	164979	0.164081	141467	2986.022813	123095	3935.135495	132847
web-BerkStan$_3$	12795	347465	0.056428	45111	0.1021736	36328	149.561649	29307	203.913794	32989
web-Google	77480	840829	0.182113	256055	0.401427	221327	7668.066338	191616	20207.08225	211529
WikiTalk	49430	1254898	0.338172	176081	0.548573	161128	5883.002974	138030	4770.705853	143958
Amazon-601	276049	2461072	0.977274	932989	1.607812	744345	140894.0119	612760	298775.2092	688159

Table 4.4: Running times and number of edges in the resulting 2-vertex-connected spanning subgraph; plots shown in Figure 4.2 .

[12].)

In Figure 4.2 (top) we can see the running times of the four algorithms. (See also Table 4.4.) It is easy to observe that the algorithms belong to two distinct classes, with FAST and LH-Z being faster than the other two by approximately five orders of magnitude. In the bottom part of Figure 4.2 we can see the relative size of the smallest spanning subgraph computed by the four algorithms. In all of our experiments, the smallest subgraph was the one computed by our new hybrid algorithm LH-Z-CT. One the other hand, on average LH-Z is only twice as slow as FAST but improves the solution quality by more than 13%. Summing up, if one wants a fast and good solution LH-Z is the right choice.

4.3 Experimental analysis of sparse subgraphs

We implemented the algorithms previously described: 5 for 2VC-B, 3 for 2VC-B-C, and 3 for 2C, as summarized in Table 4.5. All implementations were written in `C++` and compiled with `g++ v.4.4.7` with flag `-O3`. We performed our experiments on a GNU/Linux machine, with Red Hat Enterprise Server v6.6: a PowerEdge T420 server 64-bit NUMA with two Intel Xeon E5-2430 v2 processors and 16GB of RAM RDIMM memory. Each processor has 6 cores sharing a 15MB L3 cache, and each core has a 2MB private L2 cache and 2.50GHz speed. In our experiments we did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the `getrusage` function. All our running times were averaged over ten different runs.

We measure the quality of the solution computed by algorithm A on problem $\mathcal{P}$ by a *quality ratio* defined as $q(A,\mathcal{P}) = \delta^A_{avg}/\delta^{\mathcal{P}}_{avg}$, where δ^A_{avg} is the average vertex indegree of the subgraph computed by A and $\delta^{\mathcal{P}}_{avg}$ is a lower bound on the average vertex indegree of the optimal solution for $\mathcal{P}$. Specifically, for 2VC-B and 2VC-B-C we define $\delta^B_{avg} = (n+k)/n$, where n is the total number of vertices of the input digraph and k is the number of vertices

Algorithm	Problem	Technique	Time
DST-B	2VC-B	Original sparse certificate from [24] based on divergent spanning trees	$O(m+n)$
DST-B modified	2VC-B	Modified sparse certificate from [24]	$O(m+n)$
DLN-B	2VC-B	Sparse certificate from [27] based on divergent spanning trees and loop nesting trees	$O(m+n)$
LHL-B	2VC-B	New sparse certificate based on low-high orders and loop nesting trees	$O(m+n)$
2VDP-B	2VC-B	2VDP filter applied on the digraph produced by DLN-B	$O(n^2)$
DLN-B-C	2VC-B-C	DST-B combined with the linear-time 2VCSS algorithm of [21]	$O(m+n)^\dagger$
LHL-B-C	2VC-B-C	LHL-B combined with the linear-time 2VCSS algorithm of [21]	$O(m+n)^\dagger$
2VDP-B-C	2VC-B-C	2VDP filter applied on the digraph produced by DLN-B-C	$O(n^2)$
DLN-2C	2C	DLN-B-C combined with the linear-time 2ECSS algorithm using edge-disjoint spanning trees	$O(m+n)^\ddagger$
LHL-2C	2C	LHL-B-C combined with the linear-time 2ECSS algorithm using edge-disjoint spanning trees	$O(m+n)^\ddagger$
2VDP-2C	2C	2VDP and 2EDP filters applied on the digraph produced by DLN-2C	$O(n^2)$

Table 4.5: The algorithms considered in our experimental study. The worst-case bounds refer to a digraph with n vertices and m edges. Running times indicated by † assume that the 2-vertex-connected components of the input digraph are available; running times indicated by ‡ assume that also the 2-edge-connected components are available.

Dataset	n	m	file size	δ_{avg}	s^*	δ^B_{avg}	δ^C_{avg}	type
Rome99	3353	8859	100KB	2.64	789	1.76	1.76	road network
P2p-Gnutella25	5153	17695	203KB	3.43	1840	1.60	1.60	peer2peer
P2p-Gnutella31	14149	50916	621KB	3.59	5357	1.56	1.56	peer2peer
Web-NotreDame	53968	296228	3,9MB	5.48	9629	1.50	1.50	web graph
Soc-Epinions1	32223	443506	5,3MB	13.76	8194	1.56	1.56	social network
USA-road-NY	264346	733846	11MB	2.77	46476	1.80	1.80	road network
USA-road-BAY	321270	800172	12MB	2.49	84627	1.69	1.69	road network
USA-road-COL	435666	1057066	16MB	2.42	120142	1.68	1.68	road network
Amazon0302	241761	1131217	16MB	4.67	69616	1.74	1.74	prod. co-purchase
WikiTalk	111881	1477893	18MB	13.20	14801	1.45	1.45	social network
Web-Stanford	150532	1576314	22MB	10.47	14801	1.62	1.58	web graph
Amazon0601	395234	3301092	49MB	8.35	69387	1.82	1.82	prod. co-purchase
Web-Google	434818	3419124	50MB	7.86	89838	1.59	1.58	web graph
Web-Berkstan	334857	4523232	68MB	13.50	53666	1.56	1.51	web graph

Table 4.6: Real-world graphs sorted by file size of their largest SCC; n is the number of vertices, m the number of edges, and δ_{avg} is the average vertex indegree; s^* is the number of strong articulation points; δ^B_{avg} and δ^C_{avg} are lower bounds on the average vertex indegree of an optimal solution to 2VC-B and 2C, respectively.

that belong in (nontrivial) 2-vertex-connected blocks [1]. We set a similar lower bound δ^C_{avg}

[1]This follows from the fact that in the sparse subgraph the k vertices in blocks must have indegree at least two, while the remaining $n-k$ vertices must have indegree at least one, since we seek for a strongly

Dataset	DST-B	DST-B modified	DLN-B	LHL-B	2VDP-B	DLN-B-C	LHL-B-C	2VDP-B-C	DLN-2C	LHL-2C	2VDP-2C
Rome99	1.384	1.363	1.432	1.388	1.170	1.462	1.459	1.199	1.462	1.459	1.198
P2p-Gnutella25	1.726	1.602	1.713	1.568	1.234	1.712	1.568	1.234	1.712	1.568	1.234
P2p-Gnutella31	1.717	1.647	1.732	1.602	1.273	1.732	1.573	1.273	1.732	1.573	1.273
Web-NotreDame	2.072	2.067	2.108	2.085	1.588	2.232	2.149	1.628	2.250	2.180	1.638
Soc-Epinions1	2.082	1.964	2.213	2.027	1.475	2.474	2.411	1.572	2.474	2.411	1.573
USA-road-NY	1.255	1.251	1.371	1.357	1.168	1.376	1.374	1.175	1.376	1.374	1.175
USA-road-BAY	1.315	1.311	1.374	1.365	1.242	1.375	1.379	1.246	1.375	1.379	1.246
USA-road-COL	1.308	1.307	1.354	1.348	1.249	1.357	1.357	1.252	1.357	1.357	1.252
Amazon0302	1.918	1.791	1.849	1.719	1.245	2.020	1.928	1.386	2.032	1.944	1.399
WikiTalk	2.145	2.126	2.281	2.190	1.796	2.454	2.441	1.863	2.454	2.441	1.863
Web-Stanford	2.115	2.019	2.130	2.078	1.572	2.287	2.257	1.622	2.238	2.209	1.584
Amazon0601	1.926	1.793	1.959	1.747	1.196	2.241	2.155	1.278	2.242	2.157	1.279
Web-Google	2.052	2.004	2.083	2.051	1.485	2.306	2.335	1.585	2.338	2.372	1.602
Web-Berkstan	2.302	2.233	2.290	2.275	1.692	2.472	2.492	1.767	2.410	2.431	1.717

Table 4.7: Quality ratio $q(A, \mathcal{P})$ of the solutions computed for 2VC-B, 2VC-B-C and 2C.

for 2C, with the only difference that k is the number of vertices that belong in (nontrivial) 2-edge-connected blocks, since every 2-vertex-connected component or block is contained in a 2-edge-connected block. Note that the quality ratio is an upper bound of the actual approximation ratio. The smaller the values of $q(A, \mathcal{P})$ (i.e., the closer to 1), the better is the approximation obtained by algorithm A for problem $\mathcal{P}$.

We now report the results of our experiments with all the algorithms considered for problems 2VC-B and 2C. For the 2VC-B problem, the quality ratio of the spanning subgraphs computed by the different algorithms is shown in Table 4.7 (left), while their running times are plotted in Figure 4.3 (left). Similarly, for the 2VC-B-C and 2C problems, the quality ratio of the spanning subgraphs computed by the different algorithms is shown in Table 4.7 (right), while their running times are plotted in Figure 4.3 (right).

We observe that all our algorithms perform well in terms of the quality of the solution they compute. Indeed, the quality ratio is less than 2.5 for all algorithms and inputs. Our modified version of DST-B performs consistently better than the original version. Also in all cases, LHL-B computed a higher quality solution than DLN-B. For most inputs, DST-B modified computes a sparser graph than LHL-B, which is somewhat surprising given the fact that we do not have a good bound for the (constant) approximation ratio of DST-B modified. On the other hand, LHL-B is faster than DST-B modified by a factor of 4.15 on average and has the additional benefit of maintaining both the 2-vertex and the 2-edge-connected blocks. The 2VDP filter provides substantial improvements of the solution, since all algorithms that apply this heuristic have consistently better quality ratios (1.38 on average and always less than 1.87). However, this is paid with much higher running times, as those algorithms can be even 5 orders of magnitude slower than the other algorithms.

From the analysis of our experimental data, all algorithms achieve consistently better approximations for road networks than for most of the other graphs in our data set. This can be explained by taking into account the macroscopic structure of road networks,

connected spanning subgraph.

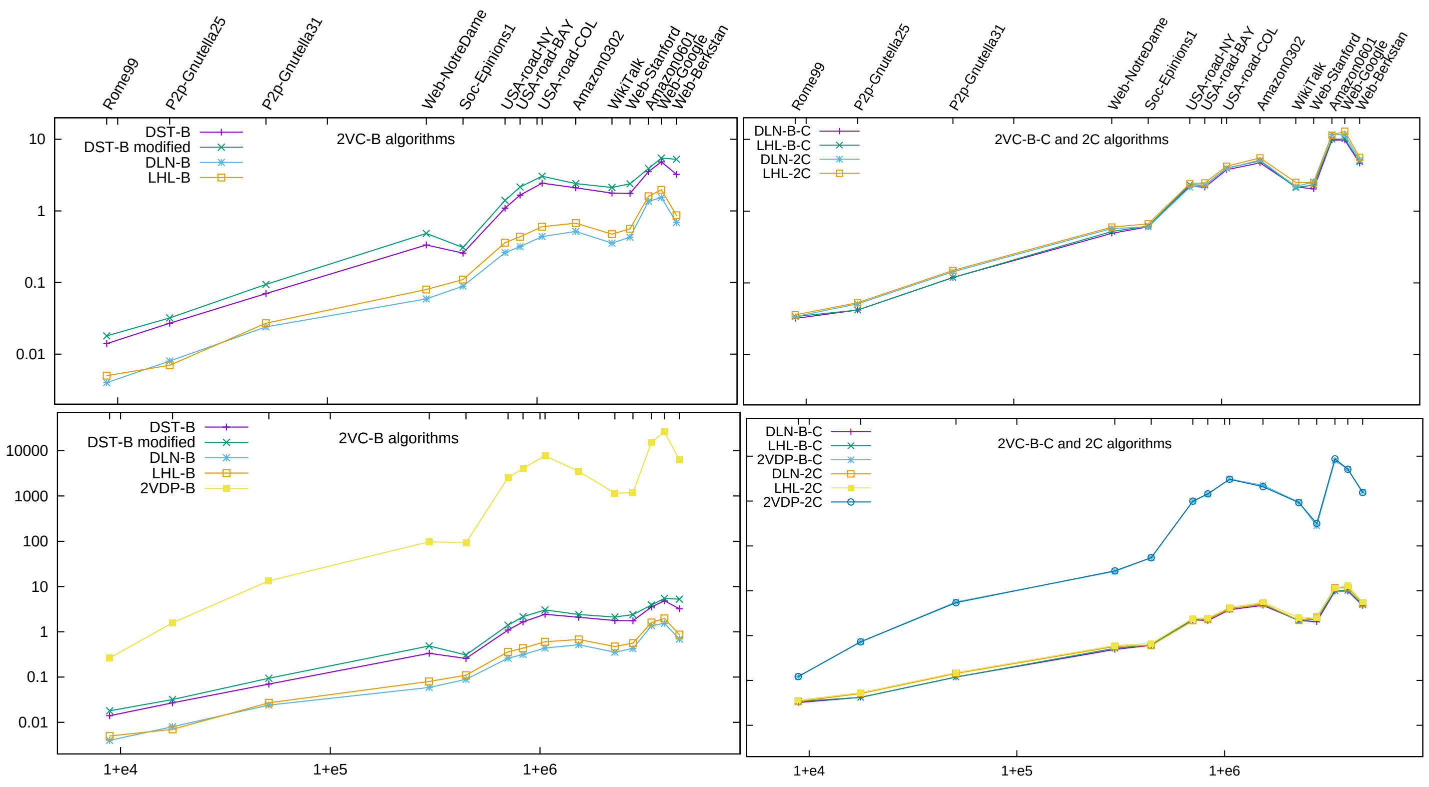

Figure 4.3: Running times in seconds with respect to the number of edges (in log-log scale). The upper plots get a close-up view of the fastest algorithms by not considering 2VDP-B, 2VDP-B-C and 2VDP-2C.

which is rather different from other networks. Indeed, road networks are very close to be "undirected": i.e., whenever there is an edge (x, y), there is also the reverse edge (y, x) (except for one-way roads). Roughly speaking, road networks mainly consist of the union of 2-vertex-connected components, joined together by strong bridges, and their 2-vertex-connected blocks coincide with their 2-vertex-connected components. In this setting, a sparse strongly connected subgraph of the condensed graph will preserve both blocks and components. On the other hand, such a gain on the solution for the road networks is balanced at the cost of their additional running time.

In addition, our experiments highlight interesting tradeoffs between practical performance and quality of the obtained solutions. In particular, the fastest algorithms for the 2VC-B problem are the ones based on loop-nesting trees (DLN-B and LHL-B), with LHL-B achieving consistently better solutions than DLN-B.

Finally we show more details with the quality ratio achieved by our algorithms and the corresponding running times. Table 4.7 is split in two tables depending on the problem. In Figure D1 we show the corresponding plotted values of the quality ratio for the solutions computed by our algorithms for all considered problems, whereas Tables D10 and D11 report the running times of our algorithms in seconds.

Dataset	DST-B	DST-B modified	DLN-B	LHL-B	2VDP-B
Rome99	1.384	1.363	1.432	1.388	1.170
P2p-Gnutella25	1.726	1.602	1.713	1.568	1.234
P2p-Gnutella31	1.717	1.647	1.732	1.602	1.273
Web-NotreDame	2.072	2.067	2.108	2.085	1.588
Soc-Epinions1	2.082	1.964	2.213	2.027	1.475
USA-road-NY	1.255	1.251	1.371	1.357	1.168
USA-road-BAY	1.315	1.311	1.374	1.365	1.242
USA-road-COL	1.308	1.307	1.354	1.348	1.249
Amazon0302	1.918	1.791	1.849	1.719	1.245
WikiTalk	2.145	2.126	2.281	2.190	1.796
Web-Stanford	2.115	2.019	2.130	2.078	1.572
Amazon0601	1.926	1.793	1.959	1.747	1.196
Web-Google	2.052	2.004	2.083	2.051	1.485
Web-Berkstan	2.302	2.233	2.290	2.275	1.692

Table D8: Quality ratio $q(A, \mathcal{P})$ of the solutions computed for 2VC-B.

Dataset	DLN-B-C	LHL-B-C	2VDP-B-C	DLN-2C	LHL-2C	2VDP-2C
Rome99	1.462	1.459	1.199	1.462	1.459	1.198
P2p-Gnutella25	1.712	1.568	1.234	1.712	1.568	1.234
P2p-Gnutella31	1.732	1.573	1.273	1.732	1.573	1.273
Web-NotreDame	2.232	2.149	1.628	2.250	2.180	1.638
Soc-Epinions1	2.474	2.411	1.572	2.474	2.411	1.573
USA-road-NY	1.376	1.374	1.175	1.376	1.374	1.175
USA-road-BAY	1.375	1.379	1.246	1.375	1.379	1.246
USA-road-COL	1.357	1.357	1.252	1.357	1.357	1.252
Amazon0302	2.020	1.928	1.386	2.032	1.944	1.399
WikiTalk	2.454	2.441	1.863	2.454	2.441	1.863
Web-Stanford	2.287	2.257	1.622	2.238	2.209	1.584
Amazon0601	2.241	2.155	1.278	2.242	2.157	1.279
Web-Google	2.306	2.335	1.585	2.338	2.372	1.602
Web-Berkstan	2.472	2.492	1.767	2.410	2.431	1.717

Table D9: Quality ratio $q(A, \mathcal{P})$ of the solutions computed for 2VC-B-C and 2C.

Dataset	DST-B	DST-B modified	DLN-B	LHL-B	2VDP-B
Rome99	0.014	0.018	0.004	0.005	0.264
P2p-Gnutella25	0.027	0.032	0.008	0.007	1.587
P2p-Gnutella31	0.070	0.094	0.024	0.027	13.325
Web-NotreDame	0.335	0.486	0.059	0.080	97.355
Soc-Epinions1	0.258	0.309	0.089	0.110	92.812
USA-road-NY	1.095	1.402	0.261	0.360	2546.484
USA-road-BAY	1.659	2.152	0.316	0.435	4089.389
USA-road-COL	2.439	3.050	0.438	0.603	7739.256
Amazon0302	2.101	2.410	0.517	0.675	3503.910
WikiTalk	1.777	2.125	0.355	0.473	1158.855
Web-Stanford	1.756	2.395	0.429	0.564	1174.984
Amazon0601	3.532	3.924	1.363	1.605	15349.126
Web-Google	4.837	5.467	1.533	1.968	26299.714
Web-Berkstan	3.239	5.261	0.690	0.869	6301.410

Table D10: Running times in seconds of the algorithms for the 2VC-B problem.

Dataset	DLN-B-C	LHL-B-C	2VDP-B-C	DLN-2C	LHL-2C	2VDP-2C
Rome99	0.032	0.034	0.122	0.034	0.036	0.122
P2p-Gnutella25	0.042	0.042	0.729	0.051	0.053	0.725
P2p-Gnutella31	0.119	0.119	5.613	0.143	0.149	5.422
Web-NotreDame	0.491	0.521	27.091	0.573	0.600	27.746
Soc-Epinions1	0.606	0.621	54.559	0.602	0.664	54.548
USA-road-NY	2.227	2.337	991.092	2.153	2.415	995.913
USA-road-BAY	2.153	2.298	1429.443	2.296	2.476	1447.318
USA-road-COL	3.770	3.969	3093.258	3.938	4.228	3064.297
Amazon0302	4.708	5.017	2244.856	5.135	5.509	2094.263
WikiTalk	2.179	2.133	943.690	2.203	2.513	924.810
Web-Stanford	2.037	2.313	279.236	2.561	2.487	317.115
Amazon0601	9.793	10.038	8065.680	11.669	11.397	8696.212
Web-Google	9.789	10.172	5095.600	11.535	12.979	5128.337
Web-Berkstan	4.670	4.872	1595.033	5.178	5.601	1546.041

Table D11: Running times in seconds of the algorithms for the 2VC-B-C and 2C problems.

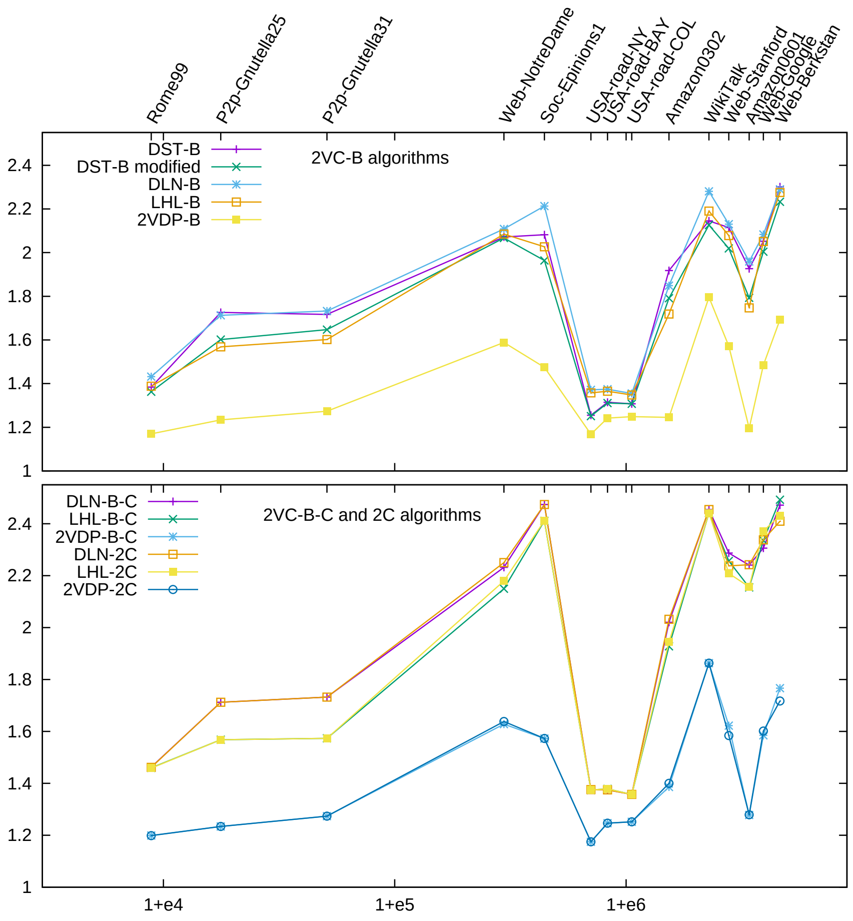

Figure D1: The plotted quality ratios taken by Tables D8 and D9, respectively.

Bibliography

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[2] S. Allesina and A. Bodini. Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions. *Journal of Theoretical Biology*, 230(3):351–358, 2004.

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

[4] S. Alstrup and P. W. Lauridsen. A simple dynamic algorithm for maintaining a dominator tree. Technical Report 96-3, Department of Computer Science, University of Copenhagen, 1996.

[5] M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana. Fault equivalence identification using redundancy information and static and dynamic extraction. In *Proceedings of the 19th IEEE VLSI Test Symposium*, March 2001.

[6] S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability for directed graphs. In Yoram Moses, editor, *Distributed Computing*, volume 9363 of *Lecture Notes in Computer Science*, pages 528–543. Springer Berlin Heidelberg, 2015.

[7] Surender Baswana, Keerti Choudhary, and Liam Roditty. Fault tolerant subgraph for single source reachability: generic and optimal. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, pages 509–518. ACM, 2016.

[8] 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.

[9] J. Cheriyan and R. Thurimella. Approximating minimum-size k-connected spanning subgraphs via matching. *SIAM J. Comput.*, 30(2):528–560, 2000.

[10] S. Cicerone, D. Frigioni, U. Nanni, and F. Pugliese. A uniform approach to semi-dynamic problems on digraphs. *Theor. Comput. Sci.*, 203:69–90, August 1998.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[12] Daniel Delling, Andrew V Goldberg, Ilya Razenshteyn, and Renato F Werneck. Graph partitioning with natural cuts. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1135–1146. IEEE, 2011.

[13] J. Edmonds. Edge-disjoint branchings. *Combinat. Algorithms*, pages 91–96, 1972.

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

[15] L. R. Ford; D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[16] Harold N Gabow. The minset-poset approach to representations of graph connectivity. *ACM Transactions on Algorithms (TALG)*, 12(2):24, 2016.

[17] Harold N Gabow and Robert E Tarjan. Faster scaling algorithms for general graph matching problems. *Journal of the ACM (JACM)*, 38(4):815–853, 1991.

[18] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[19] Karthik Gargi. A sparse algorithm for predicated global value numbering. In *ACM SIGPLAN Notices*, volume 37, pages 45–56. ACM, 2002.

[20] L. Georgiadis. Testing 2-vertex connectivity and computing pairs of vertex-disjoint s-t paths in digraphs. In *Proc. 37th Int'l. Coll. on Automata, Languages, and Programming*, pages 738–749, 2010.

[21] L. Georgiadis. Approximating the smallest 2-vertex connected spanning subgraph of a directed graph. In *Proc. 19th European Symposium on Algorithms*, pages 13–24, 2011.

[22] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. *CoRR*, abs/1409.6277, 2014.

[23] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. In *SODA 2015*, pages 1988–2005, 2015.

[24] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *ICALP 2015*, pages 605–616, 2015.

[25] L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni. An experimental study of dynamic dominators. In *Proc. 20th European Symposium on Algorithms*, pages 491–502, 2012.

[26] L. Georgiadis, G. F. Italiano, C. Papadopoulos, and N. Parotsidis. Approximating the smallest spanning subgraph for 2-edge-connectivity in directed graphs. In *ESA 2015*, pages 582–594, 2015.

[27] L. Georgiadis, G. F. Italiano, and N. Parotsidis. A new framework for strong connectivity and 2-connectivity in directed graphs. *CoRR*, arXiv:1511.02913, November 2015.

[28] L. Georgiadis and R. E. Tarjan. Dominator tree certification and divergent spanning trees. *ACM Transactions on Algorithms*, 12(1):11:1–11:42, November 2015.

[29] 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.

[30] Loukas Georgiadis, Giuseppe F Italiano, and Nikos Parotsidis. Incremental 2-edge-connectivity in directed graphs. *arXiv preprint arXiv:1607.07073*, 2016.

[31] Loukas Georgiadis and Robert E Tarjan. Addendum to "dominator tree certification and divergent spanning trees". *ACM Transactions on Algorithms (TALG)*, 12(4):56, 2016.

[32] Loukas Georgiadis and Robert E Tarjan. Dominator tree certification and divergent spanning trees. *ACM Transactions on Algorithms (TALG)*, 12(1):11, 2016.

[33] Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[34] M. Gomez-Rodriguez and B. Schölkopf. Influence maximization in continuous time diffusion networks. In *29th International Conference on Machine Learning (ICML)*, 2012.

[35] M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *ICALP 2015*, pages 713–724, 2015.

[36] Monika Henzinger, Sebastian Krinninger, and Veronika Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *International Colloquium on Automata, Languages, and Programming*, pages 713–724. Springer, 2015.

[37] G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theor. Comput. Sci.*, 447(0):74–84, 2012.

[38] R. Jaberi. Computing the 2-blocks of directed graphs. *RAIRO-Theor. Inf. Appl.*, 49(2):93–119, 2015.

[39] Raed Jaberi. On computing the 2-vertex-connected components of directed graphs. *Discrete Applied Mathematics*, 204:164–172, 2016.

[40] S. Khuller, B. Raghavachari, and N. E. Young. Approximating the minimum equivalent digraph. *SIAM J. Comput.*, 24(4):859–872, 1995. Announced at *SODA 1994*, 177-186.

[41] 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.

[42] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 115–124, 2010.

[43] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[44] Miranda Mowbray and Antonio Lain. Dominator-tree analysis for distributed authorization. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 101–112. ACM, 2008.

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

[46] L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Proc. 8th International Conference on Practical Aspects of Declarative Languages*, pages 73–87, 2006.

[47] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proc. 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 287–296, 1994.

[48] V. C. Sreedhar, G. R. Gao, and Y. Lee. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems*, 19:239–252, 1997.

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

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

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

[52] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

[53] T. Tholey. Linear time algorithms for two disjoint paths problems on directed acyclic graphs. *Theoretical Computer Science*, 2012. In press.

[54] J. Zhao and S. Zdancewic. Mechanized verification of computing dominators for formalizing compilers. In *Proc. 2nd International Conference on Certified Programs and Proofs*, pages 27–42. Springer, 2012.

[55] L. Zhao, H. Nagamochi, and T. Ibaraki. A linear time 5/3-approximation for the minimum strongly-connected spanning subgraph problem. *Information Processing Letters*, 86(2):63–70, 2003.

Short Vita

Aikaterini Karanasiou holds a Diploma degree in Electrical & Computer Engineering from the Polytechnic School of Electrical & Computer Engineering, Aristotle University of Thessaloniki, Greece. Her research interests are focused to the design and analysis of algorithms, algorithms engineering, approximation algorithms, graph theory, power energy and power electronics. Aikaterini has been an assistant in the laboratories of the undergraduate course on Computer Architecture and has also been an assistant in the laboratories of the undergraduate course on Data Structures in the Department of Computer Science & Engineering, University of Ioannina.