Decremental Dominators and Low-High Orders in Directed Acyclic Graphs

A Thesis

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

by

Konstantinos Giannis

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN COMPUTER SCIENCE THEORY

University of Ioannina

October 2018

Examining committee

- Λουκάς Γεωργιάδης, Αναπληρωτής Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων (Επιβλέπων)
- Λεωνίδας Παληός, Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων
- Σπυρίδων Κοντογιάννης, Αναπληρωτής Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων

Acknowledgements

Foremost, I would like to express my sincere thanks to my thesis advisor, associate professor Loukas Georgiadis for the continuous support of my MSc study and research, for his patience, encouragement and immense knowledge. His guidance and his possitive outlook helped me all these years as his student.

I am profoundly grateful to my co-authors, Giuseppe F. Italiano, Luigi Laura and Aikaterini Karanasiou for their valuable collaboration.

Finally, I must express my very profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Konstantinos Giannis

M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, September 2018

Tilte of Dissertation: Decremental Dominators and Low-High Orders in Directed Acyclic Graphs

Thesis Supervisor: Loukas Georgiadis

Graphs are mathematical objects that model many diverse natural or man-made systems. A graph $G = (V, E)$ consists of a set of vertices V together with a set E of edges. Graphs play an important role in computer science because they can be used to represent essentially any pairwise relationship between objects. For example, graphs can model transportation networks, communication networks, social networks, electronic circuits etc. Graphs are useful not only in computer science but in many academic areas such as chemistry, physics, mathematics and biology. Designing efficient graph algorithms can be a very challenging task.

In this thesis, we consider practical algorithms for maintaining the dominator tree and a low-high order of a directed acyclic graph (DAG) under edge deletions. Let G=(V, E, s) be 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 every path 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 of D that certifies the correctness of D and has further applications in connectivity and path-determination problems.

First, we provide a carefully engineered version of a recent algorithm [ICALP 2017] for maintaining the dominator tree of a directed acyclic graph through a sequence of edge deletions. Then, we show how how to extend this algorithm so that it also maintains a low-high order of the given DAG. Our algorithms, for both tasks, run in O(mn) total time and O(m+n) space, where n is the number of vertices and m is the number of edges before any deletions. These results trivially extend to the case of reducible graphs.

We study the efficiency of our algorithms in practice by conducting an extensive experimental study, using real-world graphs, taken from a variety of application areas, and

artificial graphs. The experimental results show that both algorithms perform very well in practice and are orders of magnitude faster than recomputing from scratch. .

ΕΚΤΕΤΑΜΕΝΗ ΠΕΡΙΛΗΨΗ

Κωνσταντίνος Γιάννης

M.Sc., Τμήμα Μηχανικών Η/Υ & Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Σεπτέμβριος 2018

Τίτλος Διατριβής: Διατήρηση κόμβων κυριαρχίας και λοω-ηιγη διατάξεων σε κατευθυνόμενα άκυκλα γραφήματα μετά από διαγραφές ακμών

Επιβλέπων: Λουκάς Γεωργιάδης

Τα γραφήματα είναι μαθηματικά αντικείμενα που μας βοηθούν στο να μοντελοποιήσουμε πολλά αλγοριθμικά προβλήματα. Ένα γράφημα $G = (V, E)$ αποτελείται από ένα σύνολο κορυφών V και ένα σύνολο ακμών E. Τα γραφήματα είναι μείζονος σημασίας για την επιστήμη των πληροφορικής καθώς χρησιμοποιούνται για την αναπαράσταση της σχέσης μεταξύ των αντικειμένων που μελετάμε. Για παράδειγμα, μέσω των γραφημάτων μπορούμε να μοντελοποιήσουμε οδικά δίκτυα, τηλεπικοινωνιακά δίκτυα, κοινωνικά δίκτυα, ηλεκτρικά κυκλώματα κ.α. Η χρησιμότητα των γραφημάτων δεν περιορίζεται μόνο στην επιστήμη των πληροφορικής αλλά επεκτείνεται και σε πολλούς ακόμα επιστημονικούς τομείς όπως για παράδειγμα τη χημεία, τη φυσική, τα μαθηματικά και τη βιολογία. Η κατασκευή νέων αλγορίθμων για την επεξεργασία γραφημάτων αποτελεί μεγάλη πρόκληση.

Σε αυτή τη διπλωματική εργασία, ασχολούμαστε με πρακτικούς αλγορίθμους για τη διατήρηση ενός δέντρου κυριαρχίας καθώς και μιας λοω-ηιγη διάταξης για κατευθυνόμενα άκυκλα γραφήματα για μια ακολουθία διαγραφών. Έστω $G = (V, E, s)$ ένα κατευθυνόμενο άκυκλο γράφημα με αφετηριακή κορυφή τον κόμβο s. Το δέντρο κυριαρχίας D του γραφήματος G είναι ένα δέντρο με ρίζα τον κόμβο s, τέτοιο ώστε ένας κόμβος v λέμε ότι κυριαρχεί ενός κόμβου w αν και μόνο αν όλα τα μονοπάτια από τη ρίζα s προς τον κόμβο w περνάνε από τον κόμβο v. Τα δέντρα κυριαρχίας έχουν πολλές και σημαντικές εφαρμογές όπως για παράδειγμα στον έλεγχο κυκλωμάτων, στη βιολογία καθώς και σε διάφορα προβλήματα συνεκτικότητας γραφημάτων. Μια λοω-ηιγη διάταξη του G αποτελεί μια προδιάταξη των κόμβων του δέντρου κυριαρχίας D η οποία αποτελεί ένα πιστοποιητικό ορθότητας για το δέντρο κυριαρχίας D και έχει διάφορες εφαρμογές στη συνεκτικότητα των γραφημάτων.

Αρχικά, παρουσιάζουμε μια υλοποίηση ενός πρόσφατου αλγορίθμου [$ICALP2017$] για την ενημέρωση ενός δέντρου κυριαρχίας ενός κατευθυνόμενου άκυκλου γραφήματος για μια ακολουθία διαγραφών και στη συνέχεια μελετάμε το πως μπορούμε να ενημερώσουμε τη λοω-ηιγη διάταξη του άκυκλου γραφήματος παράλληλα με την ενημέρωση του δέντρου κυριαρχίας.

Οι δύο αλγόριθμοι που θα παρουσιάσουμε απαιτούν $O(mn)$ χρόνο εκτέλεσης και $O(m+n)$ χώρο, όπου n είναι το πλήθος των κορυφών του γραφήματος και m το πλήθος των ακμών πριν από τις διαγραφές.

Στο τελευταίο κεφάλαιο, μελετάμε την απόδοση των αλγορίθμων μας διενεργώντας μια εκτενή πειραματική μελέτη. Για τα πειράματά μας χρησιμοποιήσαμε γραφήματα του πραγματικού κόσμου από διάφορες εφαρμογές καθώς και τεχνητά γραφήματα. Τα αποτελέσματα της μελέτης έδειξαν πως και οι δύο αλγόριθμοι είναι αρκετά αποδοτικοί στην πράξη και είναι τάξεις μεγέθους ταχύτεροι απο τους αντίστοιχους που επανυπολογίζουν τη λύση απο την αρχή.

Chapter 1

Introduction

1.1 Thesis Scope

Graphs are mathematical objects that model many diverse natural or man-made systems, by representing pairwise relations between various types of objects. In physics and chemistry, a graph makes a natural model for molecules where vertices represent atoms and edges bonds, in social sciences we use friendship graphs to model social structures based on different kinds of relationships between people or groups and in civil engineering the road system of a city can be represented by a transportation network. More specifically, in computer science graphs are ubiquitous because they are able to describe both the structure and the dynamics of various systems. We can use graphs to represent communication networks, web page connections, data organisation etc.

In several applications, we deal with *dynamic graphs*, i.e., graphs that are subject to updates, such as the insertion or deletion of edges or vertices. Here, we consider only edge updates. Dynamic graph algorithms have been extensively studied for several decades, and many important results have been achieved for fundamental problems, including connectivity, minimum spanning tree, transitive closure, shortest paths (see, e.g., the survey

in [11]). Typically, the goal of a dynamic graph algorithm is to update the solution to a problem efficiently after each update of the graph, that is, much faster than recomputing from scratch (using a static algorithm). Of course, we aim to update the solution as quickly as possible. We can classify dynamic graph problems according to the types of updates allowed. A dynamic graph problem is said to be *fully dynamic* if it is able to process both insertions and deletions of edges, *incremental* if it requires to process edge insertions only and *decremental* if it can process edge deletions only.

Here we consider two decremental problems in directed graphs, namely maintaining the dominator tree and a low-high order of a flow graph. Specifically, we consider how to maintain the dominator tree and a low-high order of an acyclic flow graph through a sequence of edge deletions (see Figure 1.1). These results trivially extend for the class of reducible flow graphs (defined below) that includes acyclic flow graphs. The dynamic dominator problem arises in various applications, such as data flow analysis and compilation [8, 12]. Moreover, dynamic dominators can be used for dynamically testing various connectivity properties in digraphs, such as 2-vertex connectivity, strong bridges and strong articulation points [28]. We remark that the reducible case is interesting for applications in program optimization since one notion of a "structured" program is that its flow graph is reducible. Also, several real-world networks, such as certain types of biological networks, are acyclic [25].

1.2 Preliminaries

Let T be a tree rooted at s with vertex set $V_T \subseteq V$ and D be a dominator tree rooted at s with vertex set $V_D \subseteq V$. We denote by $T(v)$ the subtree of T rooted at v and by $t(v)$ the parent of vertex $v \in V_T$ in T; $t(v) = null$ if v is the root of T. If v is an ancestor of w, $T[v, w]$ is the path in T from v to w. If v is a proper ancestor of w, $T(v, w]$ is the path to w from the child of v that is an ancestor of w. Analogously, $T[v, w)$ denotes the path from v to $t(w)$. Suppose now that the vertex set V_T of T consists of the vertices reachable from s. Equivalently, path $D[s, v]$ consists of the vertices that dominate v. Tree T has the *parent property* if for all $(v, w) \in E$ with $v \in V_T$ and $w \in V_T$, v is a descendant of $t(w)$ in T. If T has the parent property and has a low-high order, then $T = D$ [21]. For every $v \in V$, we denote by $C(v)$ the set of children of vertex v in D. A *flow graph* $G = (V, E, s)$ is a directed graph (digraph) with vertex set V, edge set E, a distinguished start vertex $s \in V$ where s is a vertex without any entering edges. A vertex $v \in V$ of G called *reachable* if there is a path from s to u; if no such path exists, vertex u is called *unreachable*. An edge (u, v) of the forementioned flow graph G is a *bridge* if its deletion makes v unreachable from s. A *reducible* flow graph [26, 39] is one in which every strongly connected subgraph S has a single *entry* vertex v such that every path from s to a vertex in S contains v. A flow graph is reducible if and only if it becomes acyclic when every edge (v, w) such that w dominates v is deleted [39]. We refer to such an edge as a *back edge*. Deleting back edges does not change the dominator tree, since no such edge can be

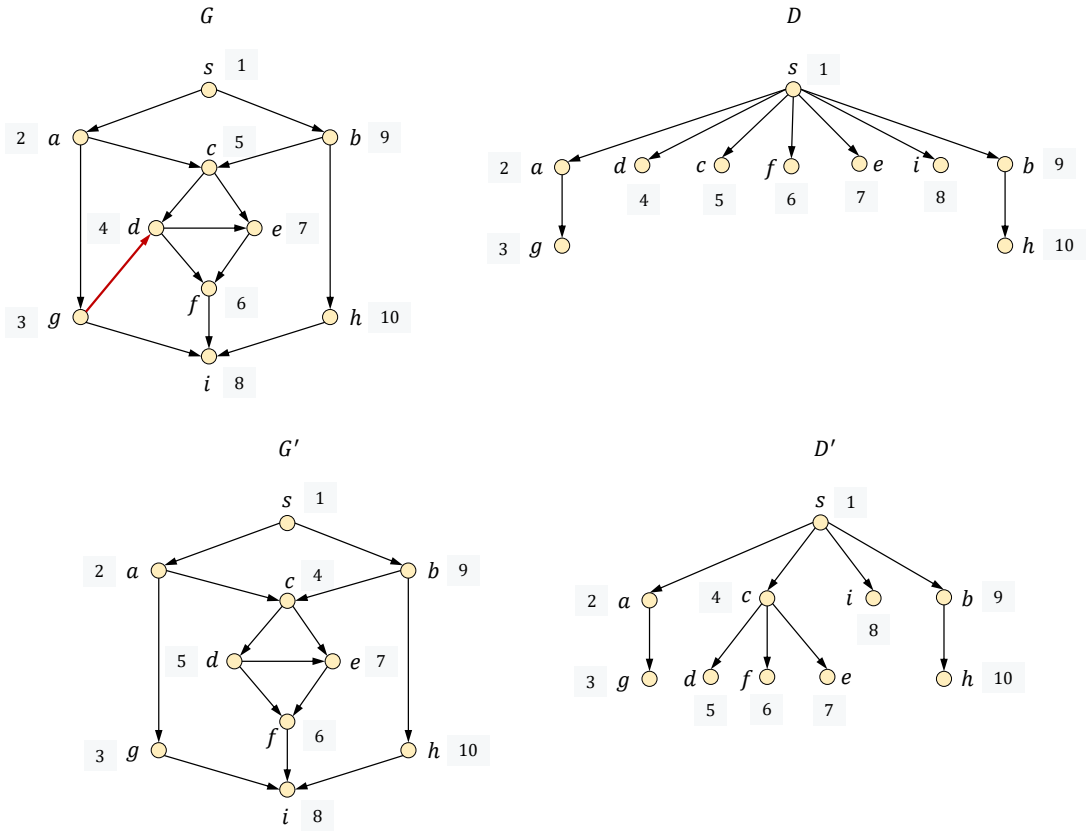

Figure 1.1: (Top) A flow graph G and its dominator tree D. The numbers correspond to a preorder numbering of D that is a low-high order of G. (Bottom) The flow graph G' and its dominator tree D' after the deletion of edge (g, d).

on a simple path from s. Deletion of such edges thus reduces the problem of computing dominators on a reducible flow graph to the same problem on an acyclic graph. Every reducible flow graph has a *topological order*, which is a total order of its vertices such that if (x, y) is an edge, x is ordered before y.

Dominators For the set of reachable vertices of flow graph G, we can define a dominator relation as follows. A vertex v is a *dominator* of a vertex w (v *dominates* w) if every path from s to w contains v; v is a *proper dominator* of w if v dominates w and $v \neq w$. The previously defined dominator relation in flow graph G can be represented by a tree rooted at s. We call such a tree a *dominator tree* D, where u dominates w if and only if u is an ancestor of w in dominator tree D (see Figure 1.1). For every reachable vertex w of D, except the root s, we denote by $d(w)$ the parent of w in D. The dominator tree is a central tool in program optimization and code generation [9], and it has applications in other diverse areas including constraint programming [35], circuit testing [3], biology [1, 25], memory profiling [33], the analysis of diffusion network [24], and in connectivity problems [13, 14, 17, 18, 20, 27, 28, 29, 30].

Low-High order A *preorder* of a flow graph G is a total order of the vertices of G, such that for every vertex v, the descendants of v in the dominator tree of G are ordered consecutively after vertex v. A *low-high order* δ, of a flow graph G, is a preorder of the vertices of the dominator tree D such for all reachable vertices $v \neq s$, either $(d(v), v) \in E$ or there are two edges $(u, v) \in E$, $(w, v) \in E$. Both distinguished vertices u and w are reachable in G with the property that 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 1.1). Every flow graph G has a low-high order that is computable in linear time. Low−high orders provide a correctness certificate for dominator trees that is entirely straightforward to verify in linear time. 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 not only computes the solution of a problem but also provides a correctness certificate with the property that one can use the provided certificate in order to verify that the given solution is correct. Low-high orders also have applications in path-determination problems [40] and fault-tolerant network design [4, 5, 22].

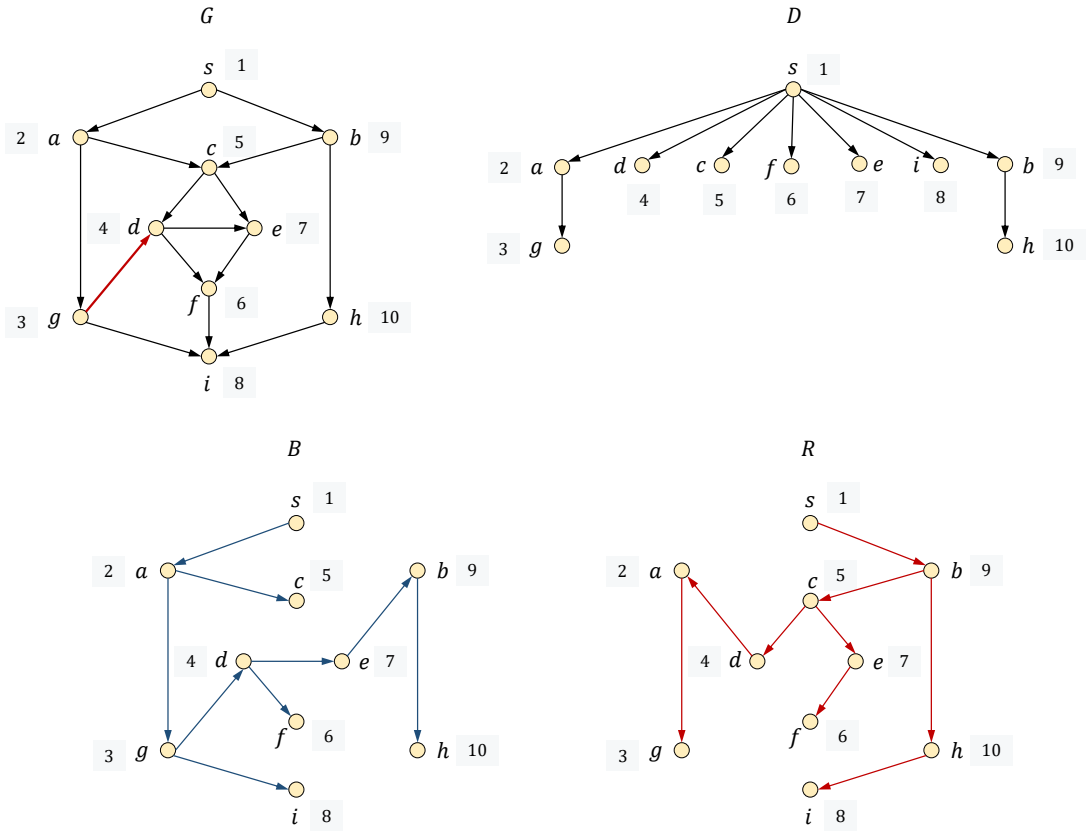

Figure 1.2: The flow graph of Figure 1.1 and two strongly divergent spanning trees B and R.

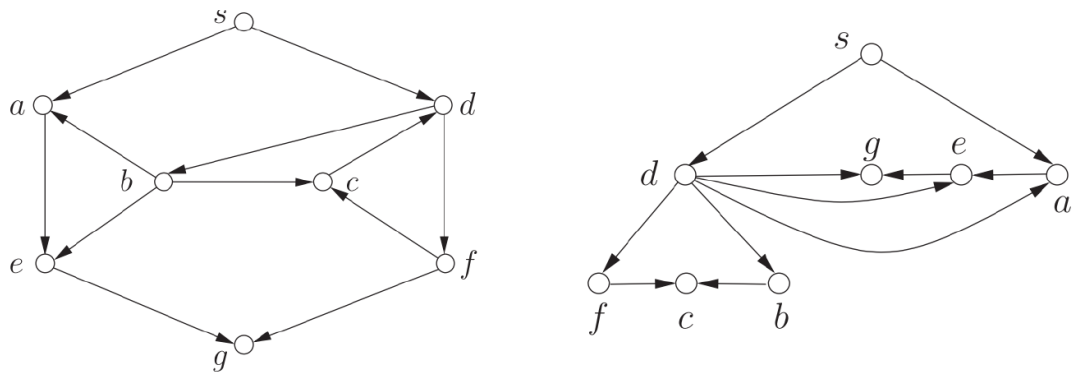

Figure 1.3: A flow graph and its corresponding derived graph.

Divergent spanning trees *Divergent spanning trees* are closely related to low-high orders [21]. Let $G[V_r]$ be a flow graph with start vertex s that is induced by vertex set V_r, where V_r is the set of all reachable vertices. Two spanning trees of $G[V_r]$ rooted at s, namely B and R, are *divergent* if for every vertex $u \in V_r$, the paths from s to u in both B and R share only the dominators of u. We call B and R *strongly divergent* if for every pair of vertices u and w, either the path in B from s to u and the path in R from s to w, share only the common dominators of u and w, or the path in R from s to u and the path in B from s to w, share only the the common dominators of u and w. From now on, in order to simplify our notation, we will refer to B and R as *strongly divergent spanning trees* of G. Every flow graph has a pair of strongly divergent spanning trees, which are easy to compute in $O(m)$ time from a given low-high order of G. 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) [13], in fast algorithms for approximating the smallest 2-vertex-connected spanning subgraph of a digraph [14], and in constructing sparse subgraphs of a given digraph that maintain certain connectivity requirements [17, 29, 30].

Derived Edges A key concept of the decremental algorithm of Georgiadis et al. [16] for maintaining the dominator tree D of a DAG, is the concept of *derived edges*. Recall that from the previously denoted parent property of D, if (v, w) is an edge of G, the parent of w, namely $d(w)$, is an ancestor of v in D. Let (v, w) be an edge of G, with w not an ancestor of v in D (Such edges do not exist if G is acyclic). Then, the *derived edge* of (v, w) is the edge $(\overline{v}, w)$; if $v = d(w)$ then $\overline{v} = v$, otherwise if $v \neq d(w)$ $\overline{v}$ is the sibling of w that is an ancestor of v. 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 edge in the edge set of G (see Figure 1.3). Given the dominator tree D of a flow graph $G = (V, E, s)$ and a list of edges $S \subseteq E$, we can compute the derived edges of S in $O(|V| + |S|)$ time [21].

1.3 Our Contribution

The problem of updating the domination relation has been extensively studied for few decades (see, e.g., [2, 7, 8, 16, 19, 36, 37]). Simple algorithms have been proposed to update the dominators after a sequence of edge insertions (incremental dominators problem). Those algorithms achieve a total of $O(mn)$ running time, where n is the number of vertices of the flow graph and m is the number of edges after all insertions [2, 8, 19]. The decremental version of the problem seems much harder to solve. Cicerone et al. [8] achieve a total $O(mn)$ update bound using $O(n^2)$ space for reducible flow graphs, where m is the initial number of edges. For general directed graphs, Georgiadis et al. [16] presented an algorithm that can process a sequence of edge deletions in a flow graph and achieves $O(mn \log n)$ total running time using $O(n^2 \log n)$ space, and can answer dominance queries, i.e., does vertex u dominate vertex v, in constant time. In the same paper, Georgiadis et al. [16] presented an algorithm for reducible flow graphs that achieves $O(mn)$ total running time using $O(m+n)$ space. Implementing this algorithm for reducible flow graphs turns out to be a challenging task. Nevertheless, here we present an efficient implementation that performs very well in practice and requires careful engineering and choice of data structures. In particular, we propose a data structure for an extension of the dynamic list order maintenance problem [6, 10] and a data structure for maintaining and updating derived edges [21]. We assess the merits of our algorithm in practical scenarios by conducting a thorough experimental study, with a variety of test graphs taken from defferent application areas. We note that a conditional lower bound in [16] suggests that it might be hard to substantially improve the $O(mn)$ update bounds in the partial dynamic (incremental or decremental) problem of maintaining the dominator tree, even for acyclic flow graphs.

Our second contribution is to show that we can maintain decrementally a low-high order of a reducible flow graph in $O(mn)$ total time. As previously mentioned, by providing an algorithm that updates both the dominator tree for a sequence of edge deletions and a low-high order, implies the first decremental certifying algorithm [34] for computing dominators in $O(mn)$ total time for reducible flow graphs. It also immediately provides $O(mn)$-time algorithms for the following problems:

- A data structure that maintains an acyclic flow graph G decrementally, 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 that are maximally vertex-disjoint, i.e., 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. Such a data structure (in the static case) was used by Tholey [40] in a linear-time algorithm for the 2-disjoint paths problem on a directed acyclic graph (DAG).

- A decremental version of the fault-tolerant reachability problem [4, 5] in DAGs. We

maintain an acyclic flow graph $G = (V, E, s)$ through a sequence of edge deletions, 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 $E' \subseteq E \backslash E_F$ of minimum cardinality, such that the subgraph $G' = (V, E_F \cup E', s)$ of G has the same dominators as G.

An incremental low-high order algorithm with $O(mn)$ total update time was presented in [15]. As in the dynamic dominators problem, the decremental version seems more difficult than the incremental. To highlight this aspect, note that a single edge deletion can cause $O(n)$ changes in a given low-high order even if the dominator tree remains unaltered (See Figure 3.1). On the other hand, in the incremental setting, it suffices to update the low-high order only for the vertices that change parent in the dominator tree.

1.4 Applications

Here we provide a couple immediate applications of our decremental low-high order algorithm.

1.4.1 Strongly divergent spanning trees and path queries

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 $G[V_r]$, 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 [21].

We augment our decremental algorithm so that for each vertex $v \neq s$ we keep two variables $low(v)$ and $high(v)$. Variable $low(v)$ stores an edge $(u, v) \in E$ such that $u \neq d(v)$ and $u < v$ in low-high; $low(v) = null$ if no such edge exists. Similarly, $high(v)$ stores an edge $(w, v) \in E$ such that and $v < w$ in low-high and w is not a descendant of v in D; $high(v) = null$ if no such edge exists. Note that these are just original edges of the sparse subgraph H that correspond to the derived edges in $\overline{H}$. Finally, we mark each vertex v such that $(d(v), v) \in E$. Note that for a reachable vertex v, we can have $low(v) = null$ or $high(v) = null$ (or both) only if $mark(v) = true$.

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. 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 deletion of an edge (x, y), we only need to update $b(v)$ and $r(v)$ for the vertices v that are relocated in the updated low-high order. Specifically, we set $b(v) \leftarrow d(v)$ if $low(v) = null$, $b(v) \leftarrow low(v)$ otherwise. Then, we set $r(v) \leftarrow d(v)$ if $high(v) = null$, $r(v) \leftarrow high(v)$ otherwise.

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 [40] 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 [38]. 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 $w <_\delta v$.

1.4.2 Fault tolerant reachability

Baswana et al. [4] 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 [22], 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 decremental low-high algorithm to solve the decremental version of the fault tolerant reachability problem on DAGs, where G is modified by edge deletions 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 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 $v < t(v)$ in low-high, then we insert into E' the edge (x, v) with $x = low(v)$. Otherwise, if $t(v) < v$ in low-high, 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 [5].

1.5 Road map

The rest of this thesis is organized as follows.In Chapter 2, we describe how the decremental dominators algorithm of [16] works. Then, we describe how a carefully engineered version of this algorithm, by incorporating efficient solutions for the following tasks that we encountered during the implementation: (i) answering ancestor-descendant queries in the dynamically changing dominator trees, (ii) maintaining dynamically the derived edges of a graph, and (iii) handling the deletion of bridges.

In Chapter 3, we present an algorithm for updating a low-high order of an acyclic flow graph after edge deletions. On of the key ideas in this algorithm is to maintain a sparse subgraph of the acyclic graph G that has the same dominator tree as G.

In Chapter 4, we conduct an experimental study by implementing the algorithms from Chapters 2 and 3. Our study was conducted on real-world graphs taken from a variety of application areas. We examine the efficiency of our algorithms with three dynamized versions of SNCA (two for the decremental dominators problem and one for the low-high order problem).

Chapter 2

Decremental Dominators

In this chapter, we provide a specialised solution for maintaining the dominator tree, under a sequence of edge deletions in reducible flow graphs. A well-known algorithm to compute the dominator tree D of an acyclic flow graph G is the following from M. S. Hecht and J. D. Ullman [26] which builds D incrementally. First we compute a topological order for vertices in G that are reachable from root s. Then we process all reachable vertices in topological order, and for each vertex v we compute the nearest common ancestor u for all incoming edges of v and we set $d(v) \leftarrow u$. The provided solution for the update of the dominator tree has a total update time of $O(mn)$ and uses $O(m+n)$ space.

Let (x, y) be the deleted edge, we call the deletion of (x, y) *regular* if (x, y) in not a bridge in G, i.e. y remains reachable from root s after the deletion of (x, y). By G' and D' we denote the flow graph and its dominator tree after the update ($G' = G \setminus (x, y)$). In general, for any given function f on V, we let f' be the function after the update. In particular, $d'(v)$ denotes the parent of v in D'. By definition, $D' \neq D$ only if x is reachable before the update.

2.1 Affected vertices

Now we consider how the dominator tree D is affected after the deletion of a single edge. We say that a vertex v is *affected* by the deletion if v has a new parent in D' ($d'(v) \neq d(v)$), and *unaffected* otherwise. In the case where vertex v is affected, $d'(v)$ does not dominate v in G. Since the effect of an edge deletion is the reverse of an edge insertion, [19, Lemma 1] and [21, Lemma 4.1] imply the following:

Lemma 2.1. *Suppose x is reachable and the deletion of edge (x, y) is regular, i.e., y does not become unreachable after the deletion. Then the following statements hold:*

(a) All affected vertices become descendants in D' of a child c of $d(y)$.

(b) A vertex v is affected if and only if $(d(v), v)$ is not an edge of G' and all edges $(u, v) \in E \setminus (x, y)$ correspond to the same derived edge $(\overline{u}, v) = (c, v)$ of G.

(c) After the deletion, each affected vertex v becomes a child of a vertex on the critical path $D'[c, d'(y)]$.

(d) No vertex on $D'[c, d'(y)]$ is affected. Hence, $D'[c, d'(y)] = D[c, d'(y)]$.

We note that statements (a) and (c) hold for arbitrary flow graphs, while (b) and (d) are true only for acyclic (and reducible) flow graphs. The algorithm of [16] applies Lemma 2.1 in order to locate the affected vertices in some topological order of G as follows. For every vertex v we maintain a count $\mathit{InSiblings}(v)$ and a list $\mathit{DerivedOut}(v)$. $\mathit{InSiblings}(v)$ corresponds to the number of distinct siblings w of v such that (w, v) is a derived edge. $\mathit{DerivedOut}(v)$ is a list of derived edges (v, u) leaving each vertex v. As we locate each affected vertex, we find its new parent in the dominator tree and we update the counts *InSiblings* for every sibling of v. The first step of the algorithm is to check if vertex y is affected after the deletion of the edge (x, y), as suggested by Lemma 3.1(b). Specifically, we update the count $\mathit{InSiblings}(y)$, and if $\mathit{InSiblings}(y) = 1$ after the update, we compute the nearest common ancestor z of all vertices in $\mathit{In}(y)$, where $\mathit{In}(y)$ is the set of vertices with a leaving edge towards y in G'. Nearest common ancestor z will be the new parent of y ($d'(y) = z$) in D' and by Lemma 2.1(c), z is a descendant of a sibling c of y in D. Next, we update the $\mathit{InSiblings}(v)$ counts for all $v \in \mathit{DerivedOut}(y)$. Specifically, we decrement $\mathit{InSiblings}(v)$ if $v \in \mathit{DerivedOut}(c)$; if $\mathit{InSiblings}(v) = 1$ then we identify v as affected and inserted into a FIFO queue Q. For each vertex w extracted from Q, we repeat the same process by updating the *InSiblings* counts for every sibling of w in D. Since we discover the affected vertices in topological order, none of these siblings of w has been inserted into Q yet.

Now we describe how we can find the new parent of each affected vertex. We can locate the new parent $d'(w)$ of each affected vertex w extracted from queue Q, similarly as for y, i.e. by computing the nearest common ancestor in D' of all vertices in $\mathit{In}(w)$. This solution, however, does not guarantee the desired $O(mn)$ total update time. To achieve the desired update time, we locate $d'(w)$ by traversing the critical path $D[c, d'(y)]$ in top-down order, until we find a vertex u such that $\mathit{In}(w)$ contains a vertex that is not a descendant of u in D'. When we locate u we set $d'(w) \leftarrow u$. Finally, we can compute the updated *InSiblings* counts and *DerivedOut* lists in a postprocessing step. The analysis in [16] is based on the fact that the affected vertices that remain reachable increase their depth in D. Notice that a vertex w can be processed at most once per deletion.

Algorithm 1: DeleteEdge$(G, \mathit{preorder}, \mathit{size}, e)$

Input: Flow graph $G = (V, E, s)$, its dominator tree D, arrays *preorder* and *size*, and an edge $e = (x, y)$.

Output: Flow graph $G' = (V, E \setminus (x, y), s)$, its dominator tree D', and arrays *preorder*$'$ and *size*$'$.

Delete e from G to obtain $G' = (V, E', s)$.
if *x was unreachable in G* **then return** $(G', D, \mathit{preorder}, \mathit{size})$
else if *y is becomes unreachable in G'* **then**
    $(D', \mathit{preorder}', \mathit{size}') \leftarrow$ Initialize(G')
    return $(G', D', \mathit{preorder}', \mathit{size}')$
end
Let $\mathit{In}(y)$ be the set of vertices v such that (v, y) is an edge in G'.
Let f be the child of $d(y)$ that is an ancestor of x.
if *there is no vertex $v \in \mathit{In}(y)$ such that $v \in D(f)$* **then**
    Set $\mathit{InSiblings}(y) \leftarrow \mathit{InSiblings}(y) - 1$.
    Set $\mathit{DerivedOut}(f) \leftarrow \mathit{DerivedOut}(f) \setminus y$.
end
if $(d(y), y) \in E'$ *or* $\mathit{InSiblings}(y) \geq 2$ **then return** $(G', D, \mathit{preorder}, \mathit{size})$
Compute the nearest common ancestor z of $\mathit{In}(y)$ in D'.
if $z = d(y)$ **then return** $(G, D, \mathit{preorder}, \mathit{size})$
Let c be the child of $d(y)$ that is an ancestor of y in D'.
Set $d'(y) \leftarrow z$.
Execute UpdateInSiblings(y).
while *Q is not empty* **do**
    Extract a vertex w from Q.
    forall $v \in D(w)$ **do** set $\mathit{AffectedAncestor}(v) \leftarrow w$.
    Execute LocateNewParent(w) and UpdateInSiblings(w).
end
Delete the affected vertices from $\mathit{DerivedOut}(c)$.
Let S be the set of all edges entering affected vertices. Compute the derived edges $\overline{S}$ of S.
Compute $\mathit{InSiblings}(w)$ for all affected vertices w.
Compute $\mathit{DerivedOut}(v)$ for all vertices v such that $(v, w) \in \overline{S}$.
Make a dfs traversal of D' to compute the updated arrays *preorder*$'$ and *size*$'$.
return $(G', D', \mathit{preorder}', \mathit{size}')$

Procedure UpdateInSiblings(w)

```
1  foreach vertex q ∈ DerivedOut(w) do
2      if q ∈ DerivedOut(c) then
3          set InSiblings(q) ← InSiblings(q) − 1
4          if InSiblings(q) = 1 and d(q) ∉ In(w) then insert q into Q
5      end
6      else
7          DerivedOut(c) ← DerivedOut(c) ∪ q
8      end
9  end
10 Set DerivedOut(w) ← ∅.
```

Procedure LocateNewParent(w)

```
1  foreach vertex u ∈ D'(c, d'(y)] in top-down order do
2      if there is an edge (v, w) ∈ E' such that v ∉ D'(w) then
3          set d'(w) ← d(u) and return
4      end
5  end
```

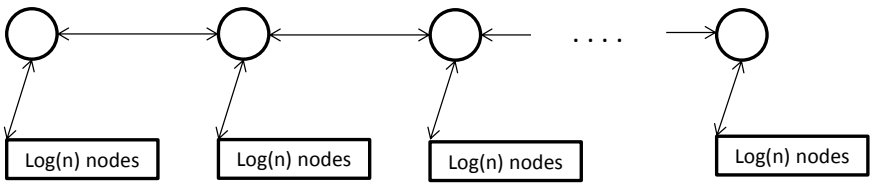

Figure 2.1: Representatives list data structure

2.2 Efficient Implementation

Providing an efficient implementation of the above algorithm turns out to be a very challenging task. In particular, we need to incorporate efficient solutions for the following tasks of the algorithm: (i) answering ancestor-descendant queries in the dynamically changing dominator tree D, (ii) maintaining dynamically the derived edges of G, and (iii) handling the deletion of bridges. We note that (i) and (ii) are not needed when we update D incrementally.

2.2.1 Ancestor-descendant queries

A crucial task for the update of the dominator tree is to answer ancestor-descendant queries in constant time. Throughout the execution of the algorithm, we need to test the ancestor-descendant pairwise relation between the vertices of D. These kind of queries are significant to the update process because they help us locate the new parent for every affected vertex $v \neq y$. To that end, it suffices to recompute a preorder and a postorder numbering for the vertices of D after each update. We can easily compute both preorder and postorder by simply performing a dfs traversal on D in $O(n)$ time. We say that a vertex v is descendant of a vertex u, if and only if $u \leq v$ in preorder and $v \leq u$ in postorder [38]. Another option is to represent preorder and postorder with a data structure for the dynamic list order problem [6, 10]. Both methods guarantee the desired $O(mn)$ total update bound, but the use of a dynamic list order data structure gives a much faster implementation in practice. From [38], we know that for every vertex w of a tree T, its subtree $T(w)$ follows vertex w in preorder and precedes w in postorder. By taking advantage of the fact that for each affected vertex v we can move the entire subtree of $D(v)$ in the new location in the dynamic lists, rather than inserting the vertices in $D(v)$ one by one, we can speed up the update process. Specifically, we remove the subtree $D(v)$ from its current locations in the two dynamic lists and insert them immediately after $d'(v)$ in the preorder list and immediately before the first descendant of $d'(v)$ in the postorder list. For this purpose, we adapted the dynamic list order data structure of Bender et al. [6] that uses a two-level structure (implementing a numbering scheme) and supports insertions, deletions and order queries in constant amortized time. The top-level of the two-level structure mentioned above is a doubly connected list which we call representatives list. Each vertex in the representatives' list is linked with a bottom level doubly connected list of $\log n$) elements, where n is the number of vertices in the graph (2.1). We modify this structure so that it can also support the following operation:

$move(u, v, w)$: Move the items between u and v (inclusive) from their current location in the dynamic list and insert them right after w.

We implement the above operation as follows. The first step is to find the representative nodes for u and v in representatives list (top-level structure); we call those representatives left-representative and right-representative, respectively. The second step is to check if the left-representative mentioned above (right-representative, respectively) has nodes in its bottom level list that do not belong to the moving set of items; If there are such items then we split the bottom-level list, and we create a new representative node. After this step, both representative nodes, and every other representative node between them has bottom level items that belong to the moving set. Therefore, we can quickly move the entire set of items by linking the left-representative and the right-representative to their new position in the dynamic list, right after item w. Finally, we check if we can merge the representative nodes that we move or split with their neighbours.

2.2.2 Derived edges

Recall that affected vertices are these who change their parent in the dominator tree after an update, and for every affected vertex we need to update the *InSiblings* counts and the *DerivedOut* lists. (i) edges entering affected vertices, and (ii) edges that enter a former sibling of y from a descendant of an affected vertex. Let S be the set of these edges. As mentioned in the previous chapter, we can compute the derived edges of set S in $O(n + |S|)$ time [21], which suffices for our $O(mn)$ bound since every edge in S is adjacent to at least one vertex that changes depth in D. This method is based on bucket sorting using a preorder numbering of D and it is not suitable for our framework, since we do not maintain a preorder numbering of the vertices, but use a dynamic list order data structure instead. Here we propose a more practical method. First we note that for each edge (u, v) of type (ii), i.e., u is a descendant of an affected vertex and $d(v) = d'(v) = d(y)$, we have $\overline{u} = c$. Now let (u, v) be of type (i), i.e., v is affected so $d'(v) \in D'[c, y)$ and u is a descendant of $d'(v)$. If $u = d'(v)$ then $\overline{u} = u$, so suppose u is a proper descendant of $d'(v)$. Let w_v be the next vertex on $D'[c, y]$ following $d'(v)$ ($w_v = d'(d'(v))$), and let z_u be the nearest ancestor of u such that $d'(z_u) \in D'[c, y]$. Then, $\overline{u} = w_v$ if $d'(z_u) \neq d'(v)$, and $\overline{u} = z_u$ if $d'(z_u) = d'(v)$. Note that we have already computed w_v, for each affected vertex v, when we locate its new parent in D'. Hence, it suffices to compute z_u for all edges (u, v) where u is a proper descendant of $d'(v)$. We do that by visiting the ancestors of u until we reach z_u. First we mark all vertices on $D'[c, y]$, so we stop our search when reaching a vertex that has a marked parent. To avoid multiple visits to the same vertices, we maintain at each vertex w a label $l(w)$, initially null. After we locate z_u, we set $l(w) = z_u$ for each visited vertex w. Thus, the next search stops at a vertex w such that $d'(w)$ is marked or $l(w)$ is not null. Therefore, we can compute all the new derived edges in $O(n + |S|)$ time as desired.

2.2.3 Unreachable vertices

When we remove an edge (x, y), some vertices may become unreachable if the deleted edge is a bridge in G. Since we deal with acyclic graphs, this means that (x, y) is the only edge of the flow graph entering y from a reachable vertex. Hence, we can easily detect if the deleted edge (x, y) is a bridge since we have $\mathit{InSiblings}(y) = 0$ and $d(y) = x$. From a theoretical point of view, we can achieve $O(mn)$ total running time by recomputing the dominator tree from scratch after each such deletion, since the total number of bridges that can appear is at most $n - 1$. In practice, this approach is not good because it causes a significant slowdown in our algorithm. A better idea to improve the performance of our algorithm is to handle the deletion of a bridge (x, y) as follows:

1. Compute the set of edges Y from vertices in $D(y)$ to vertices in $D \setminus D(y)$. Note that no edge $e \in Y$ is a bridge in $G \setminus (Y \setminus e)$, since for any vertex $v \in D \setminus D(y)$, all edges in $(w, v) \in Y$ correspond to the same derived edge $(\overline{w}, v)$.

2. Process each edge $e \in Y$ as a regular deletion.

3. Delete $D(y)$ from the dominator tree D' of G', and update accordingly the data structures.

Note that Steps 1 and 3 take $O(m)$ time. Also, since in Step 2 we have regular deletions, the total running time remains $O(mn)$.

Chapter 3

Decremental low-high order

In this chapter, we consider the problem of updating a low-high order of an acyclic flow graph $G = (V, E, s)$ after the deletion of an edge (x, y). First, we show how to achieve an $O(mn)$ total update bound using a sparsification technique, similar to the one used for the incremental problem in [15]. The idea is to maintain a sparse subgraph $H = (V, E_H)$ of G with the same vertex set, and $O(n)$ edges. Subgraph H has the same dominator tree as G. Recall that by 2.1(c), each vertex v with $(d(v), v) \notin E$ has two entering edges (u, v) and (w, v) such that $\overline{u} \neq \overline{w}$; then, it suffices to add two such edges in H.

Corollary 3.1. *Let $H = (V, E_H)$ be subgraph of an acyclic flow graph G such that E_H contains:*

(a) All edges $(u, v) \in E$ such that $u = d(v)$.

(b) Two edges (u, v) and (w, v) such that $\overline{u} \neq \overline{w}$ for each vertex v with $(d(v), v) \notin E$.

Then H has the same dominator tree as G. Moreover, a low-high order of H is also a valid low-high order of G.

Note that the two edges in Corollary 3.1(b) exist by Lemma 2.1(c). Clearly $H = (V, E_H)$ has $O(n)$ edges as required. We can compute a low-high order for H in $O(|E_H|) = O(n)$ time using the static algorithm of [21]. For every non-leaf vertex x of D, the algorithm arranges the children $C(x)$ in a local low-high order δ_x. Vertices in $C(x)$ are

separated into two categories depending on whether they have a directed edge from x or not. We place all vertices $v \in C(x)$ that have a directed edge from x, $(x, v) \in E$, in arbitrary order in δ_x. Then, we process the remaining children of x in topological order as follows. For each vertex v where $(x, v) \notin E$, graph H contains edges (u, v) and (w, v) such that $\overline{u} \neq \overline{w}$, so $\overline{u}$ and $\overline{w}$ precede v in the topological order and are already located in δ_x. Hence, it suffices to insert v in any location in δ_x between $\overline{u}$ and $\overline{w}$. When we have computed all local low-high ordered lists of children, we obtain a complete low-high order of G by arranging each subtree $D(v)$ of D immediately after v. After the deletion of (x, y) we need to update H in order to ensure that it still satisfies Corollary 3.1. We can do this during the update of the derived edges, after we have located all their affected vertices and their new parents in D'. Therefore, we get the following result.

Theorem 3.1. *We can maintain a low-high order of a reducible flow graph G with n vertices through a sequence of edge deletions in $O(mn)$ total time, where m is number of edges in G before all deletions.*

3.1 Bounded search algorithm

Here we present an efficient algorithm that updates a low-high order faster in practice. To that end, we also need to maintain *DerivedIn*(v) lists. Each one of them has the derived edges (u, v) entering vertex v. The algorithm has to process two different set of vertices. The first set includes the affected vertices, recall that affected is all vertices that change their dominator in D'. It is quite easy to update the set of affected vertices. The problematic case is when we have to update the low-high order for the set of unaffected vertices because each one of them may cause many changes in the given low-high order. For the latter case, we propose a bounded search process that identifies the unaffected vertices that may need to be relocated in their current low-high order.

3.1.1 Affected vertices

As previously mentioned, the set of affected vertices is quite easy to update. The crucial observation is that the algorithm for updating the dominator tree in chapter 2 discovers the affected vertices in topological order. Therefore, when we move the affected vertices in their new position in D' and update their incoming derived edges, we can position them in low-high order. For each affected vertex v, if $(d(v), v) \notin E$, then *DerivedIn*(v) contains two vertices u and w such that $u < w$ in low-high order, so we can insert v between these two vertices.

3.1.2 Unaffected vertices

Now we deal with the more challenging case of updating the low-high order of unaffected vertices. As we observed, a single edge deletion may cause many changes in a given low-

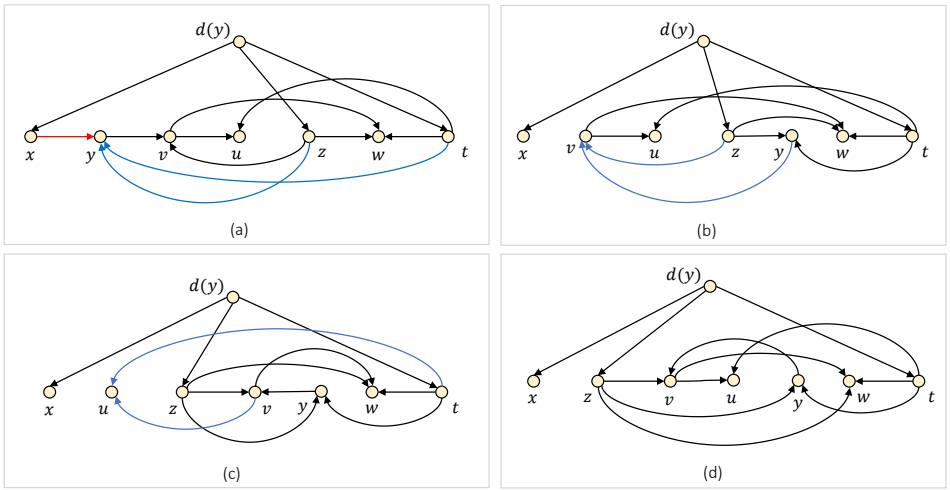

Figure 3.1: An example of propagation of changes in the low-high order after the deletion of an edge. Vertices are arranged from left to right in low-high order. (a) After the deletion of (x, y), y violates the given low-high order. (b)-(c) Moving y between z and t causes a new violation at vertex v, which in turn causes another violation at vertex u after v is placed between z and y. (d) The low-high order is finally restored when we place u between v and t.

high order, even if there are no affected vertices (See Figure 3.1). After we update the dominator tree and the low-high order of the affected vertices, the first step is to initialise a vertex set I, which contains all unaffected vertices that have at least one entering derived edge from an affected vertex. The next step is to fix the low-high order for every vertex in I. However by fixing low-high order for the set I; we may invalidate the low-high order of other vertices that are reachable from vertices in I. Thus, we compute a set X $(I \subseteq X)$ of vertices that may need to be relocated in low-high order due to the changes in the low-high order caused from I. The next lemma determines the location of the vertices in I.

Lemma 3.1. *Let v be an unaffected vertex that violates the given low-high order after updating the dominator tree in response to an edge deletion (i.e., $v \in I$). Then $d'(v) = d(y)$.*

Proof. A vertex v may violate the low-high order only if it has an entering edge (u, v) such that u is a descendant of an affected vertex and the derived edge of (u, v) changes. From the parent property of the dominator tree we have that for all $(v, w) \in E$ with v and w reachable, v is a descendant of $d(w)$ in D. Since, by Lemma 2.1(c), all affected vertices become descendants of a child c of $d(y)$, the derived edge of (u, v) changes only if v is a child of $d(y)$. Since v is unaffected, $d'(v) = d(v) = d(y)$. □

The above lemma also helps us narrow our search down for candidate vertices that we

may need to relocate in the given low-high order in response to updating of the position in the low-high order of the vertices in I. Since I consists only of children of $d(y)$, we only need to search among the unaffected children of $d(y)$ that are reachable from I. As we relocate vertices in low-high order, this process may cascade. (See Figure 3.1).

Initially, we set $X = I$ and for every vertex in I we execute a search in order to discover all vertices that may violate the given low-high order due to the replacement of the vertices in I. During this search, it is crucial to avoid any unnecessary propagation of changes in the low-high order. To achieve this, when we process a vertex $v \in X$, we examine its outgoing derived edges. In order to bound the total running time of our algorithm by $O(mn)$, we maintain a sparse spanning subgraph $H = (V, E_H)$ of G with $O(n)$ edges that satisfies Corollary 3.1, together with the derived edges $\overline{E}_H$ of E_H. We also maintain the invariant that for each vertex v such that $(d(v), v) \notin E$, the two derived edges $(u, v), (w, v) \in \overline{E}_H$ are such that $u < v < w$ in low-high order.

Our algorithm, FixLH(y), computes a vertex set $X \subseteq C'(d(y))$ that we need to process in order to ensure that vertices in X satisfy the low-high order of G'. Initially, we set $X = I$ and for every vertex in I we examine its outgoing derived in $\overline{E}_H$ in order to discover all vertices that may violate the given low-high order due to the replacement of the vertices in I. During this search, it is crucial to avoid any unnecessary propagation of changes in the low-high order, because it has a huge impact on the execution time of our algorithm (see Figure 3.1). To achieve this, when we examine an outgoing derived edge $(u, v) \in \overline{E}_H$ of $u \in X$, we test if v has two derived edges $(x, v), (y, v) \in \overline{E}_H$ such that $x < v < y$ where $x, y \notin X$. If we can find two such edges, then v keeps on satisfying the current low-high order, and there is no need to add v in X. If this is not the case, we insert v in X. This bounded search is outlined by Procedure scan. Note that we can only afford to check a constant number k of entries in $DerivedIn(v)$ in order to have $O(n)$ running time per deletion. (In our experiments we set $k \leq 3$).

Algorithm 2: FixLH(y)

1 I = children of $d(y)$ that violate the low-high order of G after the deletion
  /*$I \subseteq \{y\}$ `if` y `is not affected; otherwise,` I `contains unaffected children of` $d(y)$ `that have an entering edge from a descendant of an affected vertex` */
2 initialize $X = I$ /*X `will contain the unaffected children of` $d(y)$ `that need to be relocated in low-high order` */
3 **foreach** *vertex* $u \in I$ **do**
4 | **if** u *not scanned* **then** *scan*(u)
5 **end**
6 Process vertices in X in topological order to place them in low-high order using the edges in $\overline{E_H}$

Procedure scan(u)

```
1  foreach derived edge (u, v) ∈ Ē_H do
2      if v ∉ X and (d(v), v) ∉ E then
3          if u < w in low-high order then
4              examine the first k = O(1) edges in DerivedIn(v) to find a replacement
                derived edge e = (w, v) with w ∉ X and w < v in low-high order
5          end
6          else
7              examine the first k = O(1) edges in DerivedIn(v) to find a replacement
                derived edge e = (z, v) with z ∉ X and v < z in low-high order
8          end
9          if a replacement derived edge e was found then
10             replace (u, v) with e in Ē_H
11         end
12         else
13             insert v into X
14             scan(v)
15         end
16     end
17 end
```

Lemma 3.2. *Algorithm FixLH correctly updates the low-high order of the children of $d(y)$ in D' in $O(n)$ time.*

Proof. To prove the correctness of algorithm FixLH, first note that it correctly updates the low-high order of all vertices in X. Now we need to argue that the remaining vertices satisfy the updated low-high order. Observe that any vertex v that is visited during the search for X, is not inserted into X only if $(d(v), v) \in \overline{E}_H$ or if both derived edges in $\overline{E}_H$ entering v are not in X. Clearly, the same holds for all vertices that are not visited during this process. Hence, any vertex $v \notin X$ does not violate the computed low-high order before and after relocating the vertices in X.

Now we argue that the algorithm runs in $O(n)$ time. Each vertex v may change its two entering edges in $\overline{E}_H$ at most $k = O(1)$ times, since we look for replacement edges only in the first $k = O(1)$ edges in *DerivedIn*(v). Thus, *DerivedIn*(v) will be examined in lines 4 and 7 of Procedure scan a constant number of times in total for each v, so we spend constant time for each vertex. Finally, we need to process the vertices of X in topological order. Note that the vertices may be inserted in X in arbitrary order. We can sort them topologically by computing a topological order of the of subgraph of $\overline{H} = (V, \overline{E}_H)$ that is induced by the vertices of X. Since $\overline{E}_H$ has $O(n)$ edges, this steps also takes $O(n)$ time. □

∎

3.2 Implementation Issues

We can extend the decremental dominators algorithm from the previous chapter so that it also maintains a low-high order as described above. The following implementation issues affect the efficiency of our algorithm in practice.

Representation of a low-high order. Since a low-high order is a preorder of D, we could use the same dynamic list order data structure as the one we use in the previous chapter to store the preorder and postorder numbering of the vertices in D. This choice, however, has a negative impact to the execution time of our algorithm. A low-high order may need to update many times during an edge deletion, and every time such a low-high update happens we need to update the data structures for both the preorder and postorder of D, even though we do not need a postorder structure to store a low-high order. For this reason, we use a separate dynamic list order data structure for the low-high order, which is updated independently of the preorder and postorder data structures of D.

Unreachable vertices. As in the decremental dominators algorithm, we have to take special care of how the deletion of a bridge (x, y) is handled. To that end, we first tested the two methods mentioned in Section 2.2: (a) Run a static algorithm to recompute the dominator tree of D and a low-high order from scratch, and (b) Process each edge $e = (u, v)$ with u a descendant of y in D and v not a descendant of y in D as a regular deletion (e cannot be a bridge) and update the low-high order after each such deletion. Then delete (x, y), making all descendants of y in D unreachable from s. Unlike the decremental dominators algorithm where choice (b) was always superior compared to (a), things are a bit different for the update of the low-high order, and that's because during the sequence of regular deletions a vertex may be scanned several times when the FixLH process is executed. Hence, we also implemented the following improvement, which updates the low-high order of unaffected vertices only once, after we have processed all regular deletions. Specifically, we first update the dominator tree as in (b) but do not compute the complete low-high order after each regular deletion of an edge $e = (u, v)$. As we process each regular deletion (u, v), we also fix the low-high order of each affected vertex. Let $A^\star$ denote the set of all affected vertices found during all regular deletions. For each edge (w, t) such that w is a descendant of an affected vertex in $A^\star$ we insert t in a list $I^\star$. We compute a set $X^\star$ of vertices which may need their low-high to be updated by executing $scan(v)$, starting from all vertices v in $I^\star$ that have not been scanned yet. Finally, we sort $X^\star$ topologically and update the low-high order of all vertices in $X^\star$.

All of the above three methods are executed in $O(m)$ time per bridge deletion, so they all guarantee the $O(mn)$ total running time. In our experiments, however, the last method turned out to be an order of magnitude faster than (a) and (b).

Chapter 4

Experimental Study

In order to assess the efficiency of our decremental dominators and decremental low-high order algorithms in practice, we conduct an extensive experimental study. Our study was conducted on real-world graphs taken from a variety of application areas. 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 Dell Precision Tower 7820 CTO Base machine running Ubuntu (16.04 LTS), equipped with an Intel Xeon Gold 5118 2.3 GHz processor with 16 MB L3 cache and 192GB DDR4-2400 RAM at 2,666 MHz. We did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the `getrusage` function.

Table 4.1 shows some statistics about the graphs used in our experimental evaluation. In all test instances, we select the first vertex of the graph as the start vertex. Similar results were produced when we choose a random vertex of the graph as the start vertex. Given an acyclic flow graph, we create our decremental instances as follows. For every test graph, we consider a portion of its edges as the edges that will be deleted. The portion size is controlled by a parameter $p \in [0, 1]$. Let m be the initial number of edges in the graph. We create a sequence of deletions by choosing $\lfloor p \cdot m \rfloor$ random edges in the original graph uniformly at random. For each graph and each choice of p, we create 10 such random instances using different seeds for the initialization of the random functions, and report the average running times. (For a given input graph, two values $p_1 < p_2$ of p, and a fixed seed, the deletion sequence for p_1 is a subsequence of the deletion sequence for p_2). The algorithms compute (in static mode) the dominator tree and the low-high order, in case of the decremental low-high order algorithms, of the given acyclic graph and then they run in decremental mode, processing the sequence of deletions. Note that during

the execution of the algorithms some vertices may become unreachable (after a deletion of a bridge), and thus some subsequent deletions may involve unreachable vertices. All algorithms can detect and ignore these deletions. For computing dominators in static mode we use the SNCA algorithm from [23], which is a simplified variant of the classic Lengauer-Tarjan algorithm [31]. As an intermediary, this algorithm computes a sparse subgraph H of the input graph G that has the same dominators as G. The indegree of each vertex in H is at most 2, so H has at most $2(n-1)$ edges (the start vertex has zero indegree). For computing a low-high order, we augment this algorithm with the low-high order algorithm for acyclic graphs from [21]. We speedup the computation of a low-high order by using only the edges in H (instead of all the edges of G).

Graph Details			
Graph	**Type**	n	m
bitcoin	WN	6005	9648
advogato	SN	2320	17809
amazon-302	WG	55414	126663
soc-epinions	SN	17117	158754
web-Berkstan	WG	29145	169870
web-google	WG	77480	372859
wikitalk	SN	49430	664139
amazon-601	WG	276049	1259198

Table 4.1: Graph instances used in the experiments. The original graphs are taken from [32] and converted to DAGs by including vertices and edges reachable from the start vertex and deleting depth-first search back edges. The number of vertices n and edges m refer to the produced instances.

4.1 Decremental Dominators

We compare the performance of three algorithms that update the dominator tree. The first one is our efficient algorithm Decr presented in Section 2, and two dynamised versions of SNCA. We did not consider the algorithm of Cicerone et al. [8] since it requires $O(n^2)$ space, and therefore is impractical for large graphs. The first dynamised version of SNCA named DSNCA1 tests if the deleted edge (x, y) belongs in the sparse subgraph H. If not, then the dominator tree is not affected, and therefore the algorithm does nothing. In case the deleted edge belongs in the sparse subgraph H, DSNCA1 runs SNCA from scratch for the whole graph. The second version of SNCA called DSNCA2 performs the same test, but if $(x, y) \in H$, it computes the nearest common ancestor z of x and y in D and runs SNCA only for the subgraph of G induced by $D(z)$. In Table 4.2 we present the average running times of the three algorithms mentioned above over ten random deletion sequences. From the obtained results it is clear that our efficient algorithm is superior

compared with DSNCA1 and DSNCA2. Indeed, except for one input graph (amazon-302), in all other instances, Decr is one or two orders of magnitude faster than the dynamised versions of SNCA. Also, we note that in most instances, a significant fraction of the input graph gets disconnected from that start vertex after the deletion of 50% of the edges, and therefore many subsequent edge deletions are ignored. Thus, as we can see in Table 4.2, in most instances the running times of the algorithms remain almost the same for $p \geq 0.5$. Even though someone would expect DSNCA2 to be faster than DSNCA1 because DSNCA2 only recomputes the dominator tree only for the subgraph induced by $D(z)$ (z is the nearest common ancestor of x and y), in practice, DSNCA2 does not provide a significant improvement in the running times and in some instances it even causes slowdown due to the overhead of computing the nearest common ancestor.

4.2 Low-High order

Here we examine the efficiency of our algorithm, Decr-LH, with a dynamized version of SNCA that also computes a low-high order of an acyclic flow graph. This algorithm, that we refer to as DSNCA-LH, works as follows. It maintains a sparse subgraph $H = (V, E_H)$ of G such that for each $v \neq s$, $(d(v), v) \in E_H$, or E_H contains edges (u, v) and (w, v) with $u < v < w$. When we delete an edge (x, y) we test if this edge belongs to H. If not, then the dominator tree and the low-high order are not affected we do nothing. Otherwise, we look into the entering edges of v and try to find a replacement edge for (x, y) so that y satisfies the current low-high order. If this fails, then we compute the dominator tree and the low-high from scratch. The corresponding average running times reported in the last two columns of Table 4.2. Similarly to dominators, DSNCA-LH is not competitive with our efficient algorithm. We observe that maintaining a low-high order along with our efficient decremental algorithm incurs a very low overhead, which is on average less than 7% of the running time of Decr. Hence, both algorithms Decr and Decr-LH, perform very well in practice. In Figure 4.1 we examine how the running time of the decremental low-high algorithm is affected by the method we use to handle the deletion of a bridge. To that point, we compare the three different versions of Decr-LH that implement the three different methods described in chapter 3.2. The first method (Decr-LH-v1) runs the static algorithm that computes the low-high order for the whole graph, the second one (Decr-LH-v2) handles each edge (u, v) with $u \in D(y)$ and $v \in D \setminus D(y)$ as a regular deletion. We compare these two methods with our improved algorithm for updating the low high order (Decr-LH-v3).

In the first place, we observe that the way we handle the deletion of a bridge is very crucial to the effectiveness of our algorithms. In particular, for every test graph, our improved algorithm is always faster by an order of magnitude, as shown in Table 4.3. The Decr-LH-v3 algorithm is up to thirteen times faster compared to the Decr-LH-v2 algorithm and up to twenty times faster compared to Decr-LH-v1. Although we expect that Decr-LH-v1 would be slower in every test graph compared to Decr-LH-v2, that's not

the case for graphs advogato, bitcoin and amazon-302 because after the deletion of a bridge Decr-LH-v2 may have to update many times the low-high order for the children of a given node that is descendant of node y. Furthermore, another factor that affects the performance of Decr-LH-v2 compared to the performance of Decr-LH-v1 is the number of nodes that become unreachable after the deletion of a bridge as we can see by combining the information we get from Tables 4.3 and 4.4.

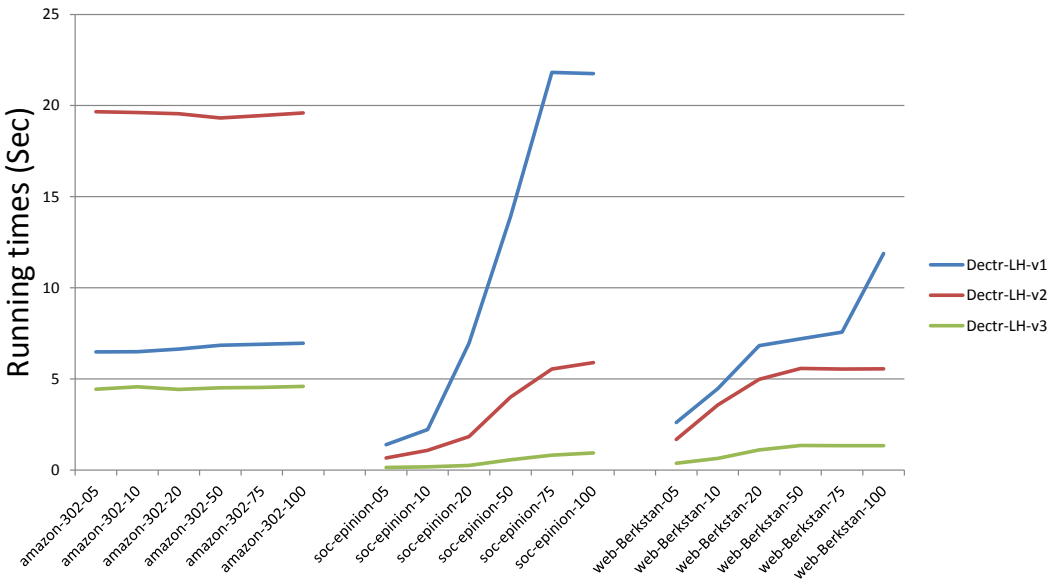

Figure 4.1: Average running times (in seconds) of three versions of Decr-LH that handle the deletion of a bridge (x, y) as described in Section 3.2. Decr-LH-v1 applies method (a) (running a static algorithm from scratch), Decr-LH-v2 applies method (b) (handling each edge (u, v) with $u \in D(y)$ and $v \in D \setminus D(y)$ as a regular deletion), while Decr-LH-v3 applies the improved method of Section 3.2.

Graphs	Decremental Dominators			Decremental Low-High	
	DSNCA1	DSNCA2	Decr	DSNCA-LH	Decr-LH
bitcoin_05	0.05	0.05	**0.01**	0.17	**0.01**
bitcoin_10	0.09	0.10	**0.03**	0.35	**0.03**
bitcoin_20	0.17	0.18	**0.03**	0.71	**0.04**
bitcoin_50	0.34	0.37	**0.08**	1.59	**0.09**
bitcoin_75	0.39	0.46	**0.12**	1.92	**0.13**
bitcoin_100	0.42	0.46	**0.17**	1.94	**0.17**
advogato_05	0.06	0.05	**0.02**	0.16	**0.03**
advogato_10	0.12	0.09	**0.02**	0.31	**0.03**
advogato_20	0.20	0.18	**0.04**	0.63	**0.04**
advogato_50	0.47	0.41	**0.10**	1.37	**0.11**
advogato_75	0.54	0.51	**0.11**	1.68	**0.12**
advogato_100	0.54	0.50	**0.12**	1.71	**0.12**
amazon-302_05	7.60	7.68	**4.22**	26.08	**4.44**
amazon-302_10	8.13	7.69	**4.35**	26.28	**4.57**
amazon-302_20	8.09	7.68	**4.36**	26.30	**4.43**
amazon-302_50	8.22	7.77	**4.22**	26.22	**4.52**
amazon-302_75	8.14	7.69	**4.32**	26.38	**4.54**
amazon-302_100	8.23	7.71	**4.34**	25.87	**4.59**
soc-epinions_05	5.59	3.72	**0.08**	10.36	**0.14**
soc-epinions_10	11.38	7.49	**0.12**	21.38	**0.18**
soc-epinions_20	22.31	15.39	**0.18**	45.66	**0.26**
soc-epinions_50	56.04	40.25	**0.42**	128.82	**0.56**
soc-epinions_75	80.85	61.29	**0.67**	194.26	**0.82**
soc-epinions_100	89.02	68.02	**0.80**	217.47	**0.94**
web-Berkstan_05	4.64	5.23	**0.33**	16.45	**0.38**
web-Berkstan_10	8.68	9.94	**0.60**	31.28	**0.64**
web-Berkstan_20	14.98	16.28	**1.05**	52.84	**1.11**
web-Berkstan_50	21.78	21.35	**1.37**	71.79	**1.35**
web-Berkstan_75	22.42	21.42	**1.32**	71.96	**1.34**
web-Berkstan_100	22.48	21.49	**1.32**	71.88	**1.34**
web-google_05	38.31	43.22	**2.70**	123.13	**3.76**
web-google_10	71.69	85.94	**3.93**	248.14	**5.14**
web-google_20	104.55	157.08	**7.37**	458.20	**8.53**
web-google_50	110.56	186.86	**10.54**	550.62	**11.13**
web-google_75	110.09	186.70	**10.56**	553.24	**11.38**
web-google_100	110.44	186.83	**10.54**	552.15	**11.30**
WikiTalk_05	97.82	71.12	**1.72**	139.41	**3.20**
WikiTalk_10	195.87	143.51	**3.12**	291.18	**5.92**
WikiTalk_20	392.72	289.26	**5.48**	615.37	**8.95**
WikiTalk_50	948.96	695.69	**30.89**	1600.38	**32.19**
WikiTalk_75	1296.04	1018.67	**60.04**	2361.56	**60.30**
WikiTalk_100	1410.21	1014.21	**61.08**	2808.81	**61.98**
amazon-601_05	871.09	790.92	**75.65**	2879.21	**83.09**
amazon-601_10	1564.34	1417.06	**99.86**	4723.03	**107.75**
amazon-601_20	2202.82	2118.52	**128.90**	4878.07	**130.05**
amazon-601_50	2388.70	2068.55	**141.77**	7674.28	**142.25**
amazon-601_75	2505.03	2395.96	**144.10**	7706.98	**144.19**
amazon-601_100	2700.07	2769.54	**140.60**	7686.45	**142.48**

Table 4.2: Average running times in seconds over 10 random deletion sequences. The suffixes in the graph names correspond to the percentage of deleted edges $p = 5\%, 10\%, 20\%, 50\%, 75\%$, and 100%.

Graphs	Decremental Low-High		
	Decr-LH-v1	**Decr-LH-v2**	**Decr-LH-v3**
bitcoin_05	0.02	0.12	**0.01**
bitcoin_10	0.04	0.16	**0.03**
bitcoin_20	0.08	0.20	**0.04**
bitcoin_50	0.19	0.30	**0.03**
bitcoin_75	0.28	0.36	**0.13**
bitcoin_100	0.33	0.40	**0.17**
advogato_05	0.04	0.06	**0.03**
advogato_10	0.06	0.12	**0.03**
advogato_20	0.12	0.20	**0.04**
advogato_50	0.25	0.35	**0.11**
advogato_75	0.32	0.37	**0.12**
advogato_100	0.35	0.36	**0.12**
amazon-302_05	6.48	19.66	**4.44**
amazon-302_10	6.49	19.62	**4.57**
amazon-302_20	6.64	19.55	**4.43**
amazon-302_50	6.85	19.32	**4.52**
amazon-302_75	6.90	19.45	**4.54**
amazon-302_100	6.96	19.60	**4.59**
soc-epinions_05	1.10	0.67	**0.14**
soc-epinions_10	2.22	1.09	**0.18**
soc-epinions_20	6.96	1.84	**0.26**
soc-epinions_50	13.91	4.01	**0.56**
soc-epinions_75	21.82	5.54	**0.82**
soc-epinions_100	21.75	5.90	**0.94**
web-Berkstan_05	2.61	1.68	**0.38**
web-Berkstan_10	4.47	3.57	**0.64**
web-Berkstan_20	6.83	4.98	**1.11**
web-Berkstan_50	7.20	5.58	**1.35**
web-Berkstan_75	7.57	5.54	**1.34**
web-Berkstan_100	11.88	5.56	**1.34**
web-google_05	27.32	18.58	**3.76**
web-google_10	45.69	76.98	**5.14**
web-google_20	59.94	140.17	**8.53**
web-google_50	61.28	144.58	**11.13**
web-google_75	63.84	146.71	**11.38**
web-google_100	63.45	144.20	**11.30**
WikiTalk_05	15.31	6.44	**3.20**
WikiTalk_10	36.00	13.27	**5.92**
WikiTalk_20	192.94	32.66	**8.95**
WikiTalk_50	362.70	297.88	**32.19**
WikiTalk_75	388.40	520.64	**60.30**
WikiTalk_100	308.60	542.88	**61.98**

Table 4.3: Average running times (in seconds) of three versions of Decr-LH that handle the deletion of a bridge (x, y) as described in Section 3.2. Decr-LH-v1 applies method (a) (running a static algorithm from scratch), Decr-LH-v2 applies method (b) (handling each edge (u, v) with $u \in D(y)$ and $v \in D \setminus D(y)$ as a regular deletion), while Decr-LH-v3 applies the improved method of Section 3.2.

Graphs	Statistics			
	Deletions	**Regular**	**Bridges**	**Skipped**
bitcoin_05	482	409.3	66.3	6.4
bitcoin_10	964	812.4	133.7	17.9
bitcoin_20	1929	1571.1	314.8	43.1
bitcoin_50	4824	3520.5	923.5	380
bitcoin_75	7236	4426.4	1499.2	1310.4
bitcoin_100	9648	4623.2	1881.1	3143.7
advogato_05	890	844.6	20.8	24.6
advogato_10	1780	1644.8	43.0	92.2
advogato_20	3561	3151.8	98.2	311
advogato_50	8904	6358.6	306.8	2238.6
advogato_75	13356	6989.2	446.6	5920.2
advogato_100	17809	7004.4	464.4	10340.2
amazon-302_05	6333	3391.2	1011.2	1930.6
amazon-302_10	12666	3986.4	1264.2	7415.4
amazon-302_20	25332	4107.8	1340.8	19883.4
amazon-302_50	63331	4123.2	1357.8	57850
amazon-302_75	94997	4123.8	1362.0	89511.2
amazon-302_100	126663	4123.8	1363.2	121176
soc-epinions_05	7937	7860.0	64.8	12.2
soc-epinions_10	15875	15667.0	153.4	54.6
soc-epinions_20	31750	31075.6	411.4	263
soc-epinions_50	79377	75085.2	1983.8	2308
soc-epinions_75	119065	106583.4	5136.8	7344.8
soc-epinions_100	158754	119556.0	9818.0	29380
web-Berkstan_05	8493	8068.6	181.2	243.2
web-Berkstan_10	16987	15588.2	424.6	974.2
web-Berkstan_20	33974	28536.6	1073.4	4364
web-Berkstan_50	84935	45208.6	2810.4	36916
web-Berkstan_75	127402	47087.2	3395.4	76919.4
web-Berkstan_100	169870	47173.4	3478.4	119218.2
web-google_05	18642	17528.2	598.8	515
web-google_10	37285	33669.2	1379.6	2236.2
web-google_20	74571	55486.6	2942.4	16142
web-google_50	186429	67637.8	4695.2	114096
web-google_75	279644	67739.6	4728.4	207176
web-google_100	372859	67742.6	4733.8	300382.6
WikiTalk_05	33206	33071.8	115.6	18.6
WikiTalk_10	66413	66039.6	279.6	93.8
WikiTalk_20	132827	131562.8	766.0	498.2
WikiTalk_50	332069	301943.0	4497.6	25628.4
WikiTalk_75	498104	403477.2	12688.0	81938.8
WikiTalk_100	664139	428014.0	19058.6	217066.4
amazon-601_05	62959	53690.8	4167.0	5101.2
amazon-601_10	125919	97230.8	8276.4	20411.8
amazon-601_20	251839	147971.2	15044.4	88823.4
amazon-601_50	629599	161620.0	18554.6	449424.4
amazon-601_75	944398	161636.6	18564.2	764197.2
amazon-601_100	1259198	161637.4	18568.2	1178992.4

Table 4.4: Average statistics for the deletion sequences used in our experiments. Column 2 (deletions) gives the total number of deletions in the sequence, column 2 gives the number of deletions that involve reachable vertices (regular), and column 3 gives the number of bridges that are deleted during the deletion sequence.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant reachability subgraph: Generic and optimal. In *Proc. 48th ACM Symp. on Theory of Computing*, pages 509–518, 2016.

[6] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th European Symposium on Algorithms*, pages 152–164, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[7] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute update. In *Proc. 15th ACM POPL*, pages 274–284, 1988.

[8] 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.

[9] 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.

[10] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 365–372, 1987.

[11] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook, 2nd Edition, Vol. 1*, pages 9.1–9.28. CRC Press, 2009.

[12] K. Gargi. A sparse algorithm for predicated global value numbering. *SIGPLAN Not.*, 37(5):45–56, May 2002.

[13] 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.

[14] 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.

[15] L. Georgiadis, K. Giannis, A. Karanasiou, and L. Laura. Incremental Low-High Orders of Directed Graphs and Applications. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:21, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[16] L. Georgiadis, T. D. Hansen, G. F. Italiano, S. Krinninger, and N. Parotsidis. Decremental data structures for connectivity and dominators in directed graphs. In *ICALP*, pages 42:1–42:15, 2017.

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

[18] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. In *Proc. 42nd Int'l. Coll. on Automata, Languages, and Programming*, pages 605–616, 2015.

[19] 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. Full version: *CoRR*, abs/1604.02711.

[20] L. Georgiadis, G. F. Italiano, and N. Parotsidis. Incremental 2-edge-connectivity in directed graphs. In *ICALP*, pages 49:1–49:15, 2016.

[21] 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.

[22] L. Georgiadis and R. E. Tarjan. Addendum to "dominator tree certification and divergent spanning trees". *ACM Transactions on Algorithms*, 12(4):56:1–56:3, August 2016.

[23] 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.

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

[25] Andreas D.M. Gunawan, Bhaskar DasGupta, and Louxin Zhang. A decomposition theorem and two algorithms for reticulation-visible networks. *Information and Computation*, 252:161 – 175, 2017.

[26] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.

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

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

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

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

[31] 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.

[32] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. June 2014.

[33] 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.

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

[35] 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.

[36] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *POPL*, pages 287–296, 1994.

[37] 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.

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

[39] R. E. Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci.*, 9(3):355–365, 1974.

[40] T. Tholey. Linear time algorithms for two disjoint paths problems on directed acyclic graphs. *Theoretical Computer Science*, 465:35–48, 2012.

Appendix A

Figures coresponding to Table 4.4

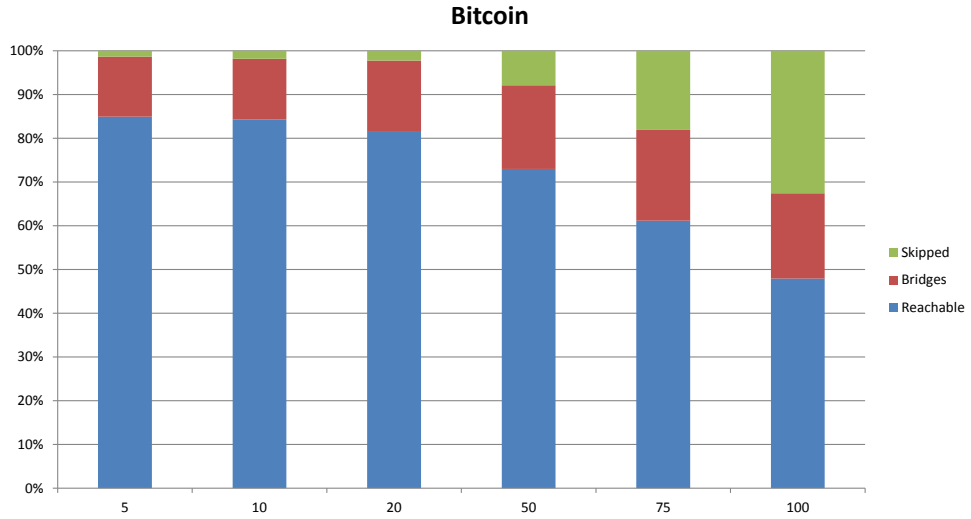

Figure A.1: Bitcoin statistics

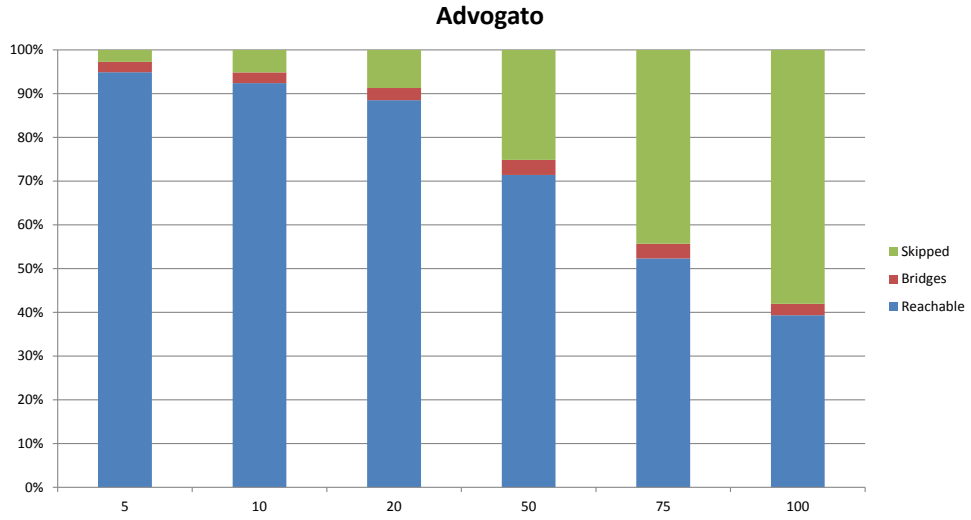

Figure A.2: Advogato statistics

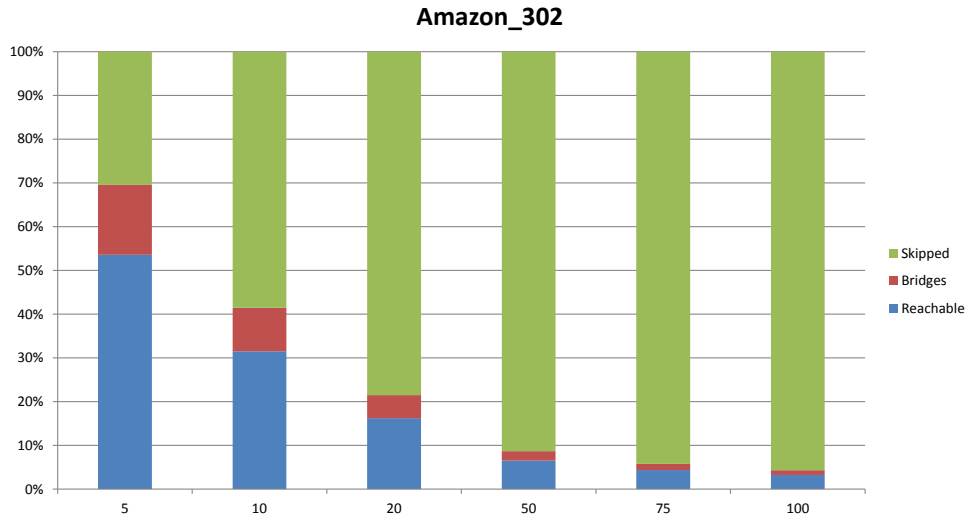

Figure A.3: Amazon-302 statistics

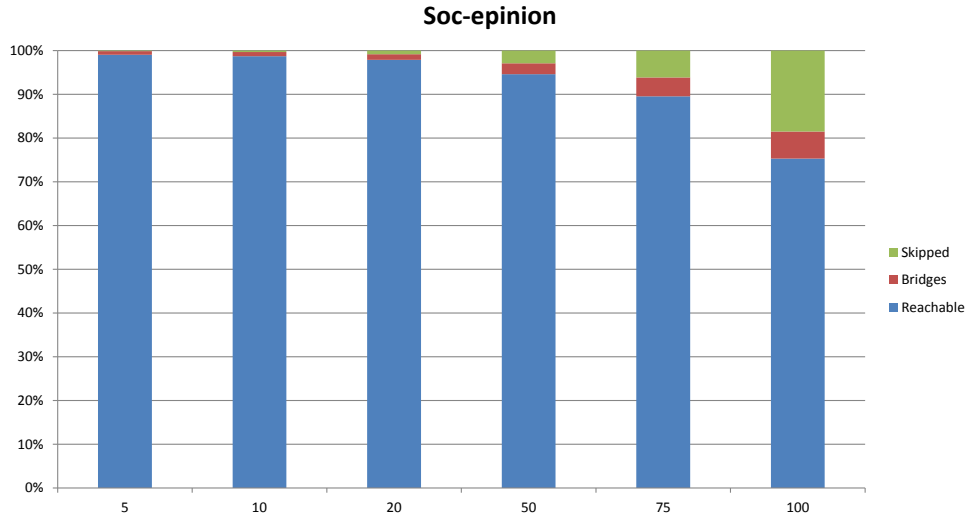

Figure A.4: soc-epinion statistics

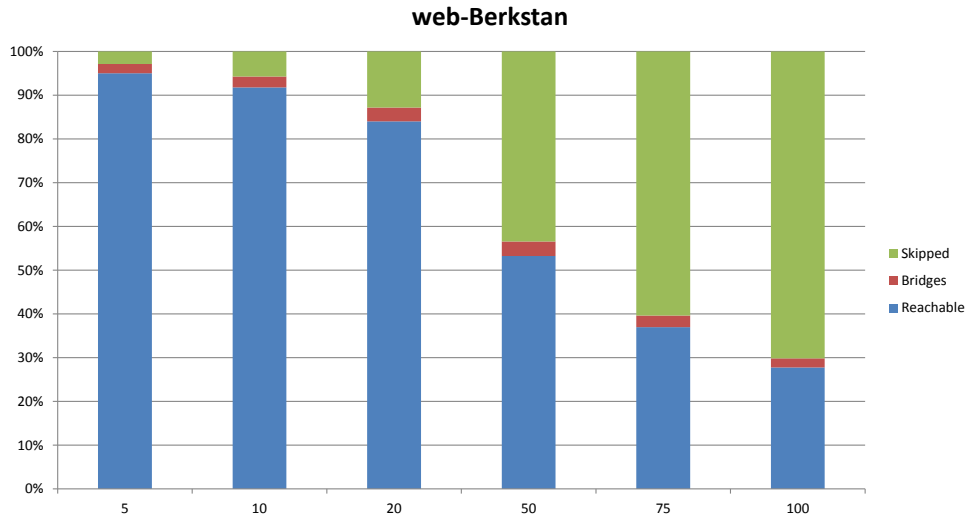

Figure A.5: web-Berkstan statistics

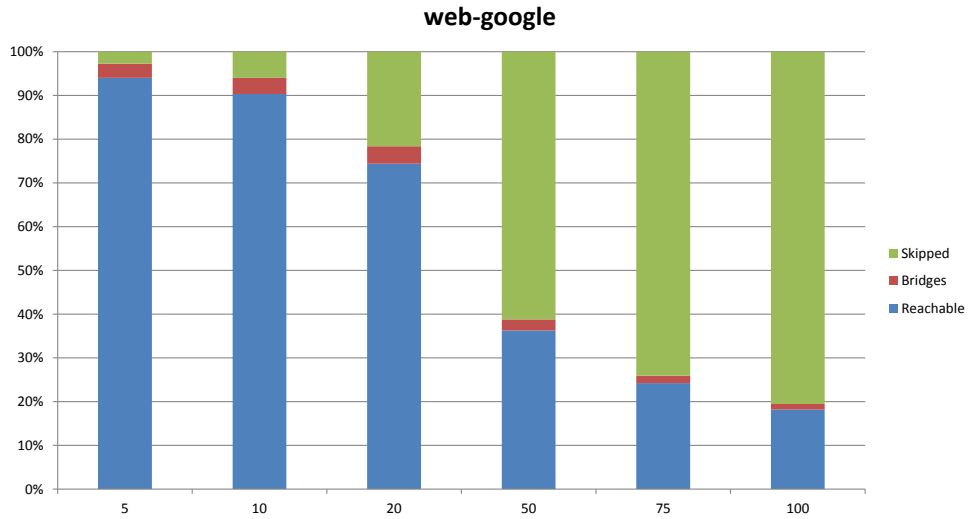

Figure A.6: web-google statistics

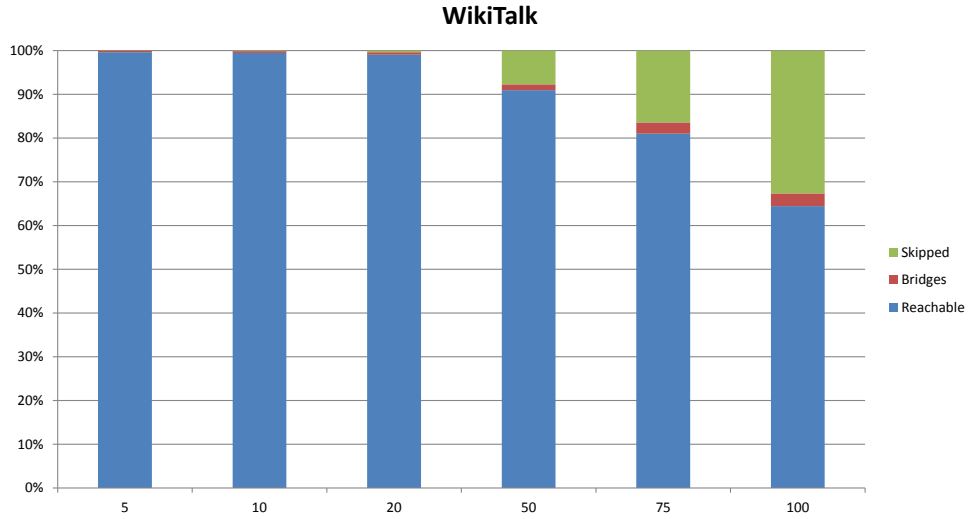

Figure A.7: WikiTalk statistics

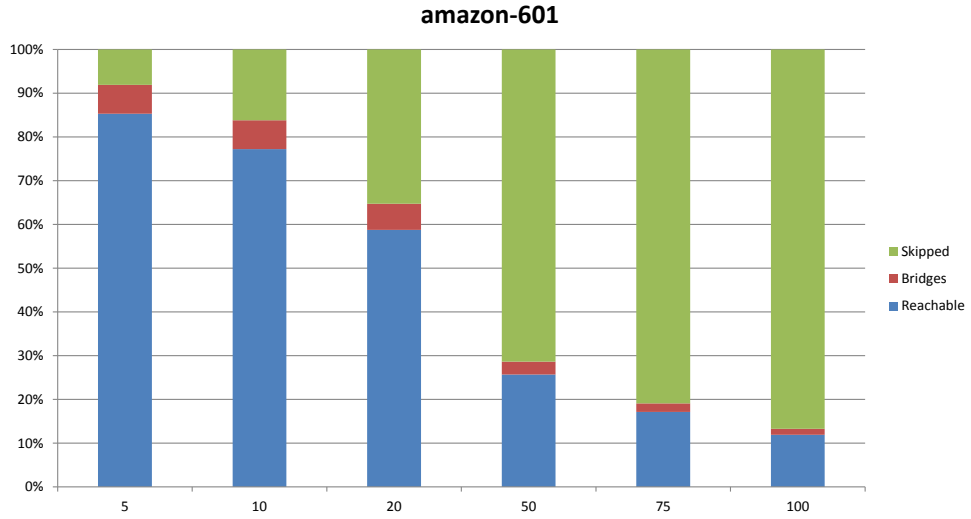

Figure A.8: Amazon-601 statistics

Short Vitae

Konstantinos Giannis received his diploma in Computer Science Engineering (2016) from the Department of Computer Science & Engineering, University of Ioannina, Greece. His research interests are the design and analysis of algorithms, algorithms engineering, algorithmic graph theory and artificial intelligence. Konstantinos worked as a teaching assistant of the undergraduate courses "Data Structures" and "Oparating Systems", in the Department of Computer Science & Engineering, University of Ioannina. He also worked as a junior researcher in the european project PRIDE.