Data Structures for 2-Fault-Tolerant Strong Connectivity

A Thesis

submitted to the designated

by the Assembly

of the Department of Computer Science and Engineering

Examination Committee

by

Daniel Tsokaktsis

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION IN DATA SCIENCE AND ENGINEERING

University of Ioannina

School of Engineering

Ioannina 2023

Examining Committee:

- **Loukas Georgiadis**, Assoc. Professor, Department of Computer Science and Engineering, University of Ioannina (Advisor)
- **Leonidas Palios**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Charis Papadopoulos**, Assoc. Professor, Department of Mathematics, University of Ioannina

Dedication

To my family.

Acknowledgements

I would like to express my deep and sincere gratitude to my supervisor Prof. Loukas Georgiadis for all the support and guidance he provided throughout this dissertation making our cooperation impeccable.

Also, I would like to thank PhD candidate Evangelos Kosinas for his contribution in this study.

Last but not least, I would like to express my heartfelt appreciation and gratitude to my parents Christos and Georgia, as well as to my beloved sister Maria for being there for me from the beginning of this journey. Their unconditional love and support encouraged me even more to pursue my goals and dreams.

Table of Contents

List of Figures iii

List of Tables iv

List of Algorithms vii

Abstract viii

Εκτεταμένη Περίληψη ix

1 Introduction 1

1.1 Motivation and objectives 1

1.2 Related work 2

1.3 Our contributions 4

1.4 Thesis outline 6

2 Preliminaries 7

2.1 Basic graph definitions 7

2.2 Auxiliary data structures 8

2.2.1 1-FT-SC-O 8

2.2.2 2-FT-SSR-O 9

2.3 Algorithms and heuristics used in our empirical analysis 9

2.3.1 Selecting split vertices for the *SCC-Tree* $\mathcal{T}$ 9

2.3.2 Trivial ways of answering queries 14

3 Our Contributions 19

3.1 Decomposition Tree 19

3.1.1 Special Graph Classes 24

3.2 Improved Data Structure for General Graphs 26

3.2.1 Choosing a good Δ for a partial-SCC-Tree decomposition 29
3.3 BFS-Based Oracles 31

4 Empirical Analysis **35**
4.1 Datasets 36
4.2 Height of the decomposition tree. 36
4.3 Answering queries 38
4.4 An improved data structure: organizing the CH seeds on a decomposition tree 41

5 Concluding Remarks **45**

Bibliography **46**

A Additional Experimental Results **51**
A.1 Random graphs experiments 51
A.2 Worst-case queries decomposition tree 52
A.3 Relative performance of the BFS-based algorithms 53
A.4 Construction time of the SCC-Tree 54

List of Figures

3.1 Two strongly connected digraphs (left) and corresponding SCC-Trees $\mathcal{T}$ (right). Every node of $\mathcal{T}$ is associated with a subset of $V(G)$ and the underlined vertex is the corresponding split vertex. For example, for the SCC-Tree of (b), the middle child of the root is $N(h)$ and $S_h = \{h, i, j\}$. Also note that $P_f = \langle N(a), N(c), N(d), N(f) \rangle$. 20

4.1 Relative SCC-Tree height of the graphs of Table 4.1 w.r.t. to the number of vertices, resulting from the split vertex selection algorithms of Table 2.1. 39

A.1 Relative SCC-Tree height of the graphs of Table A.2 w.r.t. to the number of vertices, resulting from the split vertex selection algorithms of Table 2.1. 54

List of Tables

2.1 An overview of the algorithms considered for selecting split vertices of the decomposition tree. The bounds refer to a digraph with n vertices and m edges. The stated bounds for *LabelPropagation* and *PageRank* assume that they run for a constant number of iterations. 10

4.1 Graph instances used in the experiments, taken from [1], [2] and [3]. n and m are the numbers of vertices and edges, respectively, n_a is the number of strong articulation points (SAPs), and n_{sp} is the number of vertices that are SAPs or belong to a proper separation pair. 36

4.2 SCC-Tree height of the graphs of Table 4.1, resulting from the split vertex selection algorithms of Table 2.1. The symbols † and ‡ refer to decompositions that were not completed due to exceeding the RAM memory of our system ($>$ 128GB) or due to requiring more than 48 hours. 38

4.3 Characteristics of *partial*-SCC-Trees achieved by the algorithm of Section 3.2. 40

4.4 Relative performance of the BFS-based algorithms on Rome99 (left) and Google_small (right). 40

4.5 Results for 1M random queries using the SCC-Tree with split vertices selected by MCN. 41

4.6 Comparing running times (avg. (s) per query after 1M random queries) for various algorithms. 42

4.7 Results for 1M random queries using *simpleBFS* and *biBFS*. Here is shown the number of edges that we had to access on average per query, as well as the total time for answering all queries on every graph. The third column for every algorithm shows the time in nanoseconds that is charged to every edge access. 43
4.8 The total time for answering 100M 2-FT-SSR queries using our implementation of Choudhary's data structure. By comparing the times/query with the times per edge access in Table 4.7, we can see that the time per 2-FT-SSR query is comparable to a few edge accesses. We report the average over 10 different random choices of CH-seeds. We note that the variance per graph is negligent. 43
4.9 Simulation for answering 10K queries with `ChBFS` and *ChTree* using 10 seeds that have high chance to give rise to a bad instance. This experiment was repeated for 100 different selections of seeds. We see that *ChTree* can answer at least 90% of those instances without resorting to BFS. 44

A.1 Randomly generated strongly connected graphs. n and m are the numbers of vertices and edges, respectively, n_a is the number of strong articulation points (SAPs), and n_{sp} is the number of vertices that are SAPs or belong to a proper separation pair 53
A.2 SCC-Tree height of the graphs of Table A.1, resulting from the split vertex selection algorithms of Table 2.1. 53
A.3 Results for 1M random queries using the SCC-Tree with split vertices selected by MCN. 54
A.4 Comparing running times (avg. (s) per query after 1M random queries) for various algorithms 55
A.5 Results for worst-case queries using the SCC-Tree with split vertices selected by MCN. 55
A.6 Relative performance of the BFS-based algorithms on Twitter (left) and Gnutella25 (right). 56
A.7 Relative performance of the BFS-based algorithms on Lastfm-asia (left) and NotreDame (right). 56

A.8 Relative performance of the BFS-based algorithms on Stanford (left) and Epinions1 (right). 57
A.9 Time in seconds for constructing SCC-Tree $\mathcal{T}$ using the heuristics from Table 2.1. The symbols † and ‡ refer to decompositions that were not completed due to exceeding the RAM memory of our system ($>$ 128GB) or due to requiring more than 48 hours. 57

List of Algorithms

2.1 LabelPropagation 11
2.2 PageRank 12
2.3 MostCriticalNode 14
2.4 qSep 15
2.5 FindSeparator 16
2.6 DFS 16
2.7 BFS 17
2.8 biBFS 18
3.1 SCC-TreeDecomposition(G) 21
3.2 2FTSC(x,y,f_1,f_2,T) 25
3.3 2FTSC-*partial*-SCC-Tree 30
3.4 2FTSC-sBFS 33
3.5 2FTSC-ChBFS 34

Abstract

Daniel Tsokaktsis, M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, 2023.
Data Structures for 2-Fault-Tolerant Strong Connectivity.
Advisor: Loukas Georgiadis, Associate Professor.

In this thesis, we study the problem of efficiently answering strong connectivity queries under two vertex or edge failures. Given a directed graph G with n vertices, we provide a data structure with $O(nh)$ space and $O(h)$ query time, where h is the height of a decomposition tree of G into strongly connected subgraphs. This immediately implies data structures with $O(n \log n)$ space and $O(\log n)$ query time for graphs of constant treewidth and $O(n^{3/2})$ space and $O(\sqrt{n})$ query time for planar graphs. For general directed graphs, we introduce a refined version of our data structure that achieves $O(n\sqrt{m})$ space and $O(\sqrt{m})$ query time, where m is the number of edges. In our experimental study, we first evaluate various methods to construct a decomposition tree with small height h in practice. Then, we provide efficient implementations of our data structures and evaluate their empirical performance by conducting an extensive experimental study on real-world and artificial graphs. The results presented in this thesis are partially included in [4].

Εκτεταμενη Περιληψη

Δανιήλ Τσοκακτσής, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, 2023.
Δομές Δεδομένων για Ισχυρή Συνεκτικότητα με Ανοχή 2 Σφαλμάτων.
Επιβλέπων: Λουκάς Γεωργιάδης, Αναπληρωτής Καθηγητής.

Οι θεμελιώδεις ιδιότητες της προσβασιμότητας και της ισχυρής συνεκτικότητας έχουν μελετηθεί εκτενώς από τους επιστήμονες τόσο για τα κατευθυνόμενα όσο και για τα μη κατευθυνόμενα γραφήματα. Η ανάγκη και η σπουδαιότητα μελέτης αυτών πηγάζει από το γεγονός πως εμφανίζονται σε πληθώρα θεωρητικών και πρακτικών προβλημάτων.

Στην παρούσα μεταπτυχιακή εργασία θα ασχοληθούμε με ερωτήματα ισχυρής συνεκτικότητας κορυφών στο fault-tolerant (ή αλλιώς sensitivity) μοντέλο. Στο εν λόγω μοντέλο πραγματοποιούμε ένα σταθερό πλήθος ενημερώσεων στο αρχικό γράφημα κι έπειτα απαντούμε τα ερωτήματα που μας ενδιαφέρουν. Οι ενημερώσεις είναι παροδικές και αφορούν διαγραφές κορυφών (ή ακμών) και συνήθως υποθέτουμε ότι το πλήθος τους είναι μικρό. Εμείς θα επικεντρωθούμε στην περίπτωση όπου έχουμε δύο διαγραφές κορυφών. Συνεπώς, τα ερωτήματα θα είναι της μορφής: "Είναι οι κορυφές x και y ισχυρά συνδεδεμένες στο γράφημα G δίχως τις f_1 και f_2;"

Προκειμένου κανείς να απαντήσει αποδοτικά ερωτήματα της άνωθεν μορφής θα πρέπει να κατασκευάσει ιδιαίτερες δομές δεδομένων (oracles) οι οποίες, ιδανικά, θα απαιτούν γραμμικό χώρο και θα απαντούν τα ερωτήματα σε σταθερό χρόνο. Μέχρι στιγμής στη βιβλιογραφία, για γενικά κατευθυνόμενα γραφήματα και για τουλάχιστον δύο σφάλματα οι δομές οι οποίες σχετίζονται με το Fault-Tolerant μοντέλο και μπορούν να χρησιμοποιηθούν για ερωτήματα ισχυρής συνεκτικότητας

απαιτούν $\Omega(n^2)$ χώρο, με συνέπεια η χρήση τους να είναι σχεδόν απαγορευτική για μεγάλα γραφήματα.

Εμείς, δοθέντος ενός γραφήματος G, παρουσιάζουμε μία δομή δεδομένων που απαιτεί $O(nh)$ χώρο και $O(h)$ χρόνο, όπου h το ύψος του δένδρου διάσπασης (decomposition tree) του G σε ισχυρά συνεκτικά υπογραφήματα. Άμεση απόρροια αυτού είναι η κατασκευή δομών με $O(n \log n)$ χώρο και $O(\log n)$ χρόνο για γραφήματα σταθερού treewidth, ενώ $O(n^{3/2})$ χώρο και $O(\sqrt{n})$ χρόνο για επίπεδα γραφήματα. Επιπλέον, για γενικά κατευθυνόμενα γραφήματα παρουσιάζουμε μία πιο προσεκτική εκδοχή του δένδρου διάσπασης μέσω της οποίας κατασκευάζουμε μία δομή δεδομένων με $O(n\sqrt{m})$ χώρο και $O(\sqrt{m})$ χρόνο, όπου m είναι το πλήθος των ακμών.

Τέλος παρουσιάζουμε πειραματικά αποτελέσματα που αφορούν την κατασκευή δένδρου διάσπασης χαμηλού ύψους και αξιολογούμε την δομή μας πραγματοποιώντας εκτενείς αναλύσεις σε πραγματικά και τεχνητά γραφήματα.

Ορισμένα αποτελέσματα αυτής της μεταπτυχιακής εργασίας περιλαμβάνονται στην εργασία [4].

Chapter 1

Introduction

1.1 Motivation and objectives

1.2 Related work

1.3 Our contributions

1.4 Thesis outline

1.1 Motivation and objectives

Fundamental graph properties such as (strong) connectivity and reachability have been extensively studied for both undirected and directed graphs. As real world networks are prone to failures, which can be unpredictable, the fault-tolerant (or sensitivity) model has drawn the attention of several researchers in the recent past [5, 6, 7, 8, 9, 10, 11]. Instead of allowing for an arbitrary sequence of updates, the fault-tolerant model only allows to apply batch updates of small size to the original input data. In this work we focus on constructing a data structure (oracle) that can answer strong connectivity queries between two vertices of given directed graph (digraph) under any two vertex (or edge) failures.

A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a maximal subgraph of G in which there is a directed path from each vertex to every other vertex. The strongly connected components of G partition the vertices of G such that two vertices $x, y \in V$ are strongly connected (denoted by $x \leftrightarrow y$) if they belong to the same strongly connected component of G. Computing the strongly connected

components of a directed graph is one of the most fundamental graph problems that finds numerous applications in many diverse areas. As real-world networks are prone to failures, we would like to be able to assess the effect of vertex or edge failures on the connectivity of the network. Towards such a direction, we wish to compute a small-size data structure for reporting efficiently whether two vertices are strongly connected under the possibility of vertex (or edge) failures. Usually, the task is to keep a data structure (oracle) that supports queries of the following form: for any two vertices x, y and any set F of k vertices (or edges) determine whether x and y are strongly connected in $G - F$. More formally, we aim to construct an efficient fault-tolerant strong-connectivity oracle under possible (bounded) failures.

Definition 1.1 (Fault-Tolerant Strong Connectivity Oracle). Given a graph $G = (V, E)$, a k-fault-tolerant strong-connectivity oracle (k-FT-SC-O) is a data structure that, for any two vertices $x, y \in V$ and for any k failed vertices $f_1, \ldots, f_k \in V$ (or failed edges $f_1, \ldots, f_k \in E$), can determine (fast) whether x and y are strongly connected in $G - \{f_1, \ldots, f_k\}$.

To measure the efficiency of an oracle, two main aspects are concerned: the size of the computed data structure and the running time for answering any requested query. Ideally, we would aim for linear-size oracles with constant query time, but this seems out of reach for many problems [9]. For instance, it is known that for a single vertex/edge failure (i.e., $k = 1$) an oracle with $O(n)$ space and $O(1)$ query time is achievable [12]. However, for a larger number of failures (i.e., $k > 1$) the situation changes considerably. Even for $k = 2$, straightforward approaches would lead to an $O(n^2)$-size oracle with constant query time.

1.2 Related work

Maintaining the strongly connected components under edge updates has received much of attention, both in the dynamic setting, where the updates are permanent, and in the fault-tolerant model, where edge failures are part of the query.

Fault-tolerant data structures. Baswana, Choudhary, and Roditty [13] presented a data structure of size $O(2^k n^2)$ that is computed in $O(2^k n^2 m)$ time, and outputs all strongly connected components in $O(2^k n \log^2 n)$ time under at most k failures. For $k = 1$, Georgiadis, Italiano, and Parotsidis [12] gave an $O(n)$-space single-fault strong

connectivity oracle (1-FT-SC-O) that can report all strongly connected components in $O(n)$ time, and test strong connectivity for any two vertices in $O(1)$ time, under a single vertex/edge failure. A closely related problem is to be able to maintain reachability information under failures, either with respect to a fixed source vertex s (single-source reachability) or with respect to a set of vertex pairs $\mathcal{P} \subseteq V \times V$ (pairwise reachability). Choudhary [6] presented a 2-fault-tolerant single-source reachability oracle (2-FT-SSR-O) with $O(n)$ space that answers in $O(1)$ time whether a vertex v is reachable from the source vertex s in $G - \{f_1, f_2\}$, where f_1, f_2 are two failed vertices. Later, Chakraborty, Chatterjee, and Choudhary [14], gave a 2-fault-tolerant pairwise reachability oracle with $O(n\sqrt{|\mathcal{P}|})$ size that answers in $O(1)$ time whether a vertex u reaches a vertex v in $G - \{f_1, f_2\}$, for any pair $(u, v) \in \mathcal{P}$. The above results imply 2-FT-SC oracles of $O(n^2)$ size and $O(1)$ query time, either by storing a 1-FT-SC-O [12] of $G - v$ for all $v \in V$, or by storing a 2-FT-SSR-O for all $v \in V$ as sources, or by setting $\mathcal{P} = V \times V$ in [14]. Recently, Van den Brand and Saranurak [11] presented a Monte Carlo sensitive reachability oracle that preprocess a digraph with n vertices in $O(n^\omega)$ time and stores $O(n^2 \log n)$ bits. Given a set of k edge insertions/deletions and vertex deletions, the data structure is updated in $O(k^\omega)$ time and stores additional $O(k^2 \log n)$ bits. Then, given two query vertices u and v, the oracle reports if there is directed path from u to v in $O(k^2)$ time. For planar graphs, Italiano, Karczmarz, and Parotsidis [15] show how to construct a 1-fault-tolerant all-pairs reachability oracle of $O(n \log n)$-space that answers in $O(\log n)$ time whether a vertex u reaches a vertex v in $G - f$, where f is a failed vertex or edge. So, using this result in a straightforward way, by constructing such a data structure for every $G - v, v \in V$, would yield a 2-FT-SC oracle for planar graphs with $O(n^2 \log n)$ space and $O(\log n)$ time.

All the previous approaches yield data structures that require $\Omega(n^2)$ space, which is prohibitive for large networks. Thus, it is natural to explore the direction of trading-off space with query time. Furthermore, within the fault-tolerant model, one may seek to compute a sparse subgraph H of G (called preserver) that enables to answer (strong connectivity or reachability) queries under failures in H instead of G, which can be done more efficiently since H is sparse. Chakraborty and Choudhary [5] provided the first sub-quadratic (i.e., $O(n^{2-\epsilon})$ for $\epsilon > 0$) subgraph that preserves the strongly connected components of G under $k \geq 2$ edge failures, by showing the existence of a preserver of size $\widetilde{O}(k2^k n^{2-1/k})$ that is computed by a polynomial (randomized) algorithm.

Dynamic data structures. An alternative approach for answering queries under failures is via dynamic data structures. In our case, we can use data structures that support vertex/edge updates (deletions and insertions) and can answer strong connectivity queries. To answer a query of the form: "*Are x and y strongly connected in $G-\{f_1, f_2\}$?*", for two failed edges f_1 and f_2, we can first delete f_1 and f_2, by updating the data structure, and then answer the query. To get ready to answer the next query we have to reinsert the deleted edges. Typically, the situation is more complicated when we have vertex failures, since we also have to take care of the edges adjacent to the failed vertices. The main problem with this approach is that the update operation is often too time-consuming and leads to bad query time. Furthermore, there is a conditional lower bound of $\Omega(m)$ update time for a single vertex (or edge) deletion for general digraphs [16, 17]. For a planar digraph G, Charalampopoulos and Karczmarz [18] gave an $O(n \log n)$-space data structure maintaining G under edge insertions and deletions with $O(n^{4/5} \log^2 n)$ worst-case update time that can compute the identifier of the strongly connected component of any $v \in V(G)$ in $O(\log^2 n)$ time. The initialization time is $O(n \log^2 n)$. Hence, this implies an $O(n \log n)$-space data structure that can answer strong connectivity queries between two vertices under two edge failures in planar digraphs in $O(n^{4/5} \log^2 n)$ time.

1.3 Our contributions

We provide a general framework for computing dual fault-tolerant strong connectivity oracles based on a decomposition tree $\mathcal{T}$ of a digraph G into strongly connected subgraphs. Following Łącki [19], we refer to $\mathcal{T}$ as an *SCC-Tree of G*. Informally, the SCC-Tree is obtained from G by iteratively removing vertices in a specified order and assigning on each node of the tree the strongly connected components of the remaining graph. We analyze our oracle with respect to the height h of $\mathcal{T}$, which depends on the number of strongly connected components obtained in each level of the tree and, thus, on the chosen order of the removed vertices. Then, by storing some auxiliary data structures [6, 12] at each node of $\mathcal{T}$, we obtain the following result:

Theorem 1.1. *Let $G = (V, E)$ be a digraph on n vertices and let h be the height of an SCC-Tree of G. There is a polynomial-time algorithm that computes a 2-FT-SC oracle for G of size $O(nh)$ that answers strong connectivity queries between two vertices of G under*

two vertex (or edge) failures in $O(h)$ time.

Despite the fact that there are graphs for which $h = \Omega(n)$, our experimental study reveals that the height of $\mathcal{T}$ is much smaller in practice. To that end, we evaluate various methods to construct a decomposition tree with small height h in practice. We note that such SCC-Trees are useful in various decremental connectivity algorithms. See, e.g., [20, 21, 19]. We also note that a corresponding notion in undirected graphs, referred to as elimination trees, also have numerous applications. See e.g. [22, 23]. It is known that finding an elimination tree of minimum height is NP-hard for general undirected graphs [24], hence the same holds for SCC-Trees in general directed graphs. Therefore, our experimental study may be of independent interest.

Theorem 1.1 immediately implies the following results for special graph classes.

Corollary 1.1. *Let $G = (V, E)$ be a directed planar graph with n vertices. There is a polynomial-time algorithm that computes a 2-FT-SC oracle of $O(n\sqrt{n})$ size with $O(\sqrt{n})$ query time.*

Corollary 1.2. *Let $G = (V, E)$ be a directed graph, whose underlying undirected graph has treewidth bounded by a constant. There is a polynomial-time algorithm that computes a 2-FT-SC oracle of $O(n \log n)$ size with $O(\log n)$ query time.*

For general directed graphs, we also provide a refined version of our data structure that builds a *partial-SCC-Tree*, and achieves the following bounds.

Theorem 1.2. *Let $G = (V, E)$ be a digraph on n vertices and m edges, and let Δ be an integer parameter in $\{1, \ldots, m\}$. There is a polynomial-time algorithm that computes a 2-FT-SC oracle of $O(mn/\Delta)$ size that answers strong connectivity queries between two vertices of G under two vertex (or edge) failures in $O(m/\Delta + \Delta)$ time.*

Theorem 1.2 provides a trade-off between space and query time. To minimize the query time, we set $\Delta = \sqrt{m}$ which gives the following result.

Corollary 1.3. *Let $G = (V, E)$ be a digraph on n vertices and m edges, and let Δ be an integer parameter in $\{1, \ldots, m\}$. There is a polynomial-time algorithm that computes a 2-FT-SC oracle of $O(n\sqrt{m})$ size with $O(\sqrt{m})$ query time.*

Thus, when $m = o(n^2)$, the oracle of Corollary 1.3 achieves $o(n^2)$ space and $o(n)$ query time. Furthermore, for sparse graphs, where $m = O(n)$, we have an oracle of $O(n^{3/2})$ space and $O(\sqrt{n})$ query time.

Finally, we provide efficient implementations of our data structures and evaluate their empirical performance by conducting an extensive experimental study on graphs taken from real-world applications. We state our results in terms of vertex failures but we note that they also hold for edge failures, as one can easily reduce edge failures to vertex failures by splitting each edge using a new vertex.

1.4 Thesis outline

The rest of the thesis is structured as follows: Chapter 2 contains the necessary background information. Chapter 3 presents in-detail analysis of our contributions. Chapter 4 demonstrates our empirical analysis and experimental results whereas Chapter 5 concludes our work.

Chapter 2

Preliminaries

2.1 Basic graph definitions

2.2 Auxiliary data structures

2.3 Algorithms and heuristics used in our empirical analysis

2.1 Basic graph definitions

Let $G = (V, E)$ be a directed graph (digraph). For any subgraph H of G, we denote by $V(H) \subseteq V$ the vertex set of H, and by $E(H) \subseteq E$ the edge set of H. For $S \subseteq V$, we denote by $G[S]$ the subgraph of G induced by the vertices in S and by $G - S$ its subgraph that results after the removal of the vertices in S from G.

Given a path P in G and two vertices $u, v \in V(P)$, we denote by $P[u, v]$ the subpath of P starting from u and ending at v. If P starts from s and ends at t we say that P is a $s \to t$ path. Two vertices $u, v \in V$ are *strongly connected* in G, denoted by $u \leftrightarrow v$, if there exist a $u \to v$ path and a $v \to u$ path in G. The *strongly connected components* (SCCs) of G are its maximal strongly connected subgraphs. Thus, two vertices $u, v \in V$ are strongly connected if and only if they belong to the same strongly connected component of G. The *size* of a strongly connected component is given by the number of its edges. It is well-known that the SCCs of G form a partition of its vertices.

The *reverse digraph* of G, denoted by G^R, is obtained from G by reversing the direction of all edges.

The predecessors (resp., successors) of a vertex v in G, denoted by $Pred_G(v)$ (resp., $Succ_G(v)$), is the set of vertices that reach v (resp., are reached from v) in G.

A vertex of G is a *strong articulation point* (SAP) if its removal increases the number of strongly connected components. A strongly connected digraph G is 2-*vertex-connected* if it has at least three vertices and no strong articulation points. Similarly, two vertices $f_1, f_2 \in V$ form a *separation pair* if their removal increases the number of strongly connected components. A strongly connected digraph G is 3-*vertex-connected* if it has at least four vertices and no separation pairs. Note that a SAP x of G forms a separation pair with any other vertex, so we make the following distinction. We say that a separation pair $\{f_1, f_2\}$ is *proper* if f_2 is a SAP of $G - f_1$ or f_1 is a SAP of $G - f_2$ (or both).

A graph is called planar if there exists an embedding of the vertices and a mapping of the edges to simple curves in the plane, such that no two curves intersect except possibly at their endpoints.

In [25] Robertson and Seymour gave a definition of a decomposition tree and treewidth. According to them, the *width* of a tree decomposition is the number of vertices in the largest subgraph (node of the tree) and the *treewidth* of a graph is the minimum of the widths of its tree decompositions.

2.2 Auxiliary data structures

Consider a digraph G with n vertices, and let s be a designated start vertex. Our oracles make use of the following auxiliary data structures for G.

2.2.1 1-FT-SC-O

Georgiadis, Italiano and Parotsidis [12] presented a linear-time algorithm that computes a single-fault-tolerant strong-connectivity oracle (1-FT-SC-O) of $O(n)$ size that answers in $O(1)$ time queries of the form "*are vertices x and y strongly connected in $G - f$?*", where the vertices $x, y \in V(G)$ and the failed vertex $f \in V(G)$ are parts of the query. We denote by $1FTSC(x, y, f)$ the answer to such a query.

2.2.2 2-FT-SSR-O

Choudhary [6] showed that there is a polynomial-time algorithm that computes a dual-fault-tolerant single-source reachability oracle (2-FT-SSR-O) of $O(n)$ size that answers in $O(1)$ time reachability queries of the form "*is vertex v reachable from s in $G - \{f_1, f_2\}$?*", where the vertex $v \in V(G)$ and the failed vertices $f_1, f_2 \in V(G)$ are parts of the query. We denote by $2FTR_s(v, f_1, f_2)$ the answer to such a query. Moreover, we use a similar 2-FT-SSR oracle for G^R, i.e., an oracle of $O(n)$ size that answers in $O(1)$ time reachability queries of the form "*is vertex s reachable from v in $G - \{f_1, f_2\}$?*". We denote by $2FTR_s^R(v, f_1, f_2)$ the answer to such a query.

We state a simple fact that will be useful in our query algorithms.

Observation 2.1. *For any vertices $x, y \in V(G) - s$, we have $x \leftrightarrow y$ in $G - \{f_1, f_2\}$ only if $2FTR_s(x, f_1, f_2) = 2FTR_s(y, f_1, f_2)$ and $2FTR_s^R(x, f_1, f_2) = 2FTR_s^R(y, f_1, f_2)$.*

2.3 Algorithms and heuristics used in our empirical analysis

In this section, we first consider various fast heuristics that aim at computing an SCC-Tree of small height. Then, we also provide some simple-minded approaches for answering strong connectivity queries under two failures.

2.3.1 Selecting split vertices for the *SCC-Tree $\mathcal{T}$*

As stated in Section 1.3 in order to construct $\mathcal{T}$ we have to iteratively remove vertices in a specified order. The selection of the vertices is crucial since once a vertex it is removed the underlying structure of the graph may change significantly and as a result the height of $\mathcal{T}$ will vary. In Łącki's work [19] the selection of the vertex is arbitrary since there are graphs for which any sequence of splitting vertices would give an SCC-Tree of $\Omega(n)$ height. In practice, however, different methods for selecting split vertices may result to vastly different SCC-Tree heights. Thus, we implemented and tested some well-known algorithms used in finding the "*important vertices*" in graphs. Table 2.1 briefly describes the algorithms and heuristics used for the construction of the decomposition tree.

Label Propagation (LP) is an algorithm used in the community detection problem and it is derived from the work of Raghavan, Albert, Kumara [27]. The main idea

Table 2.1: An overview of the algorithms considered for selecting split vertices of the decomposition tree. The bounds refer to a digraph with n vertices and m edges. The stated bounds for *LabelPropagation* and *PageRank* assume that they run for a constant number of iterations.

Algorithm	Technique	Complexity	Reference
Random	Choose the split vertex uniformly at random	$O(1)$	
LabelPropagation (LP)	Partition vertex set into communities and select vertex with maximum number of neighbours in other communities	$O(m)$	[26, 27]
PageRank (PR)	Compute the Page Rank of all vertices and return the one with maximum value	$O(m)$	[28]
MostCriticalNode (MCN)	Return the vertex whose deletion minimizes the number of strongly connected pairs	$O(m)$	[12, 29]
q-Separator (qSep)	Compute a high-quality separator for a graph with a high diameter ($\geq \sqrt{n}$)	$O(m)$	[20]
q-Separator and MostCriticalNode (qSep+MCN)	If the graph has high diameter then compute a high-quality separator, otherwise compute the MCN	$O(m)$	[20, 12, 29]
Loop nesting tree (LNT)	Use LNT as the decomposition tree	$O(m)$[1]	[30]

of LP is to "spread" the labels across the network until either an equilibrium or a maximum number of iterations has been reached. At the beginning of this algorithm every vertex is associated with a unique label (community). During an iteration, every vertex of the graph is processed in a random order and a new label is assigned to it according to its neighbours' labels. More specifically, it gets the label with the most appearances between its neighbours. The previous procedure is repeated until there is no changes in the labels of the graph or a maximum number of iterations has been reached. The running time per iteration is $O(m)$, where m is the number of the edges. A simple pseudocode of this algorithm is provided in Algorithm 2.1.

In our case, after computing the vertex labels, we partition the vertices into communities, and select as a split vertex the vertex that has the maximum number of neighbors in other communities.

[1] LNT can compute all split vertices in $O(m)$ time.

Algorithm 2.1 LabelPropagation

Require: max number of iteration K

1: Initialize the labels for all nodes in the graph. $C_x(0) = x$; /* For vertex x at time 0 assign $label(x) = x$ */
2: $i \leftarrow 1$;
3: **while** $i \leq K$ **and** $[\exists x \in V(G) \; : \; C_x(i-1) \neq C_x(i), \; i > 1]$ **do**
4: `shuffle(V(G));`
5: **for** $x \in V(G)$ **do**
6: $C_x(i) =$ `mostFrequentLabel();` /* Get most frequent label according to x's neighbours */
7: **end for**
8: $i \leftarrow i + 1$;
9: **end while**

PageRank (PR) is a well-known algorithm used to rank websites. However, it is also capable to find communities in a graph. It derived from the work of Page, Brin, Motwani, Winograd [28] and the main idea is the following. A page (vertex x) has a high page rank score if the sum of the scores of the pages (vertices) that link (with out-edge) to the current page is also high. The algorithm starts by assigning a random-value vector as the initial scores. Then until it achieves convergence or reaches a predefined maximum number of iterations, the vector is updated based on the following equation.

$$\mathbf{pr}^{k+1} = (1 - \alpha) \cdot \mathbf{1}/n + \alpha \cdot \mathbf{pr}^{k} \cdot M,$$

where α is the *teleporting* constant, $\mathbf{1}$ the row vector with *1's* and M the random transition matrix i.e. $M = D^{-1}A$, where D the diagonal matrix of out-degrees and A the adjacency matrix. The running time per iteration is $O(m)$, where m is the number of the edges. This algorithm is presented in Algorithm 2.2.

It is obvious to conclude that as split vertex we select the vertex with the highest PageRank score, breaking ties arbitrary.

Loop Nesting Tree (LNT) derives from the work of Tarjan [30]. LNT is a hierarchical representation of strongly connected subgraphs of G and is defined with respect to some source vertex r and its corresponding DFS tree, T_r. If the graph is not strongly connected then it is called *Loop Nesting Forest* (LNF). LNT can be constructed as follows. For any vertex u, the *loop* of u, denoted by $loop(u)$, is the set of

Algorithm 2.2 PageRank

Require: max number of iteration K, teleporting constant α, threshold ϵ

1: $\mathbf{pr}(1)$ =`randomInitialVector();` /* Assign random initial vector that sums to 1 */
2: $i \leftarrow 1$;
3: $\mathbf{pr}(i+1) = (1-\alpha) \cdot \mathbf{1}/n + \alpha \cdot \mathbf{pr}(i) \cdot M$;
4: **while** $i \leq K$ **and** $\|\mathbf{pr}(i+1) - \mathbf{pr}(i)\|_2 > \epsilon$ **do**
5: $\quad\mathbf{pr}(i+1) = (1-\alpha) \cdot \mathbf{1}/n + \alpha \cdot \mathbf{pr}(i) \cdot M$;
6: $\quad i \leftarrow i+1$;
7: **end while**

all descendants x of u in T_r such that there is a path from x to u in G containing only descendants of u in T_r . Any two vertices in $loop(u)$ are mutually reachable, thus $loop(u)$ induces a strongly connected subgraph of G. It is concluded that for any two vertices u and v, $loop(u)$ and $loop(v)$ are either disjoint or nested. The *loop nesting tree* H of G, with respect to T_r, is defined as the tree in which the parent of any vertex v, denoted by $h(v)$, is the nearest proper ancestor u of v in T_r such that $v \in loop(u)$ if there exists such a vertex u and null otherwise. LNT can be computed in linear time $O(m)$ [30, 31].

Based on the definition of LNT, it can be used as the decomposition tree $\mathcal{T}$. In contrast to the rest methods used in producing an SCC-Tree, LNT, can be computed in a single pass of $O(m)$ time (the other methods require $O(hm)$ time, where h is the height of the tree).

Most Critical Node (MCN) derives from the work of Georgiadis, Italiano, Paudel [29] and is used for the critical node detection problem (CNDP). The main goal of this problem is to find a subset S of at most k vertices such that the residual graph $G \setminus S$ has minimum pairwise strong connectivity. Given a digraph G and let $C_1, C_2, \ldots, C_z$ be its strongly connected components they define the *connectivity value of* **G** as

$$f(G) = \sum_{i=1}^{z} \binom{|C_i|}{2}.$$

Note that $f(G)$ equals to the pairwise strong connectivity value, thus by minimizing the above function ($\arg\min_{S \subseteq V} f(G \setminus S)$)) the task is complete. [29] is restricted to the case of finding a single most critical node, i.e., $k = 1$. They provide a linear time algorithm finding the *most critical node* in $O(m)$ time.

The output of the MCN algorithm is used as split vertex since we are sure that if

such a vertex exists (the graph is not 2-vertex-connected) then this vertex will cause the greatest reduction in connectivity, hoping that the graph will be decomposed in many SCCs, which may result to a decomposition tree $\mathcal{T}$ of small height.

Before we present a pseudocode for computing the MCN (Algorithm 2.3), we provide some auxiliary definitions about it. Apart form strong articulation points (SAPs) and loop nesting trees (LNT), they make heavy use of dominators and the dominator trees of a flow graph. Given a flow graph G_r, a vertex v is a *dominator* of a vertex w (v *dominates* w) in G_r if every path from r to w contains v. The dominator relation in G_r can be represented by a tree rooted at r, called *dominator tree*. In this tree, every vertex v dominates a vertex w if and only if v is an ancestor of w. A vertex $v \neq r$ is called *nontrivial dominator* of G_r if v is the parent of some vertex w. Similarly, a vertex v is a *nontrivial dominator* of w in G_r if v dominates w and $v \notin \{r, w\}$. If v is a *nontrivial dominator* of w in both flow graphs G_r, G_r^R then v is called *common nontrivial dominator* of w. The dominator tree can be computed in linear time, that is, $O(m)$ [31, 32].

Lastly, they present a way of efficiently computing the connectivity value of $f(G - v)$, for each SAP v of G. In order to achieve this they proved the following equation.

$$f(G - v) = f(\tilde{D}(v)) + f(\tilde{D}^R(v)) - f(PCD(v)) + f(PCA(v))$$

Where $\tilde{D}(v)$ (resp. $\tilde{D}^R(v)$) is the set of proper descendants of vertex v in the dominator tree D (resp. D^R), $PCD(v) = \tilde{D}(v) \cap \tilde{D}^R(v)$ and $PCA(v) = V \setminus (\tilde{D}(v) \cup \tilde{D}^R(v))$.

Q-Separator (qSep) method is based on the following definition:

Definition 2.1. (q-separator [20]) Let $G = (V, E)$ be a graph with n vertices, and let $q \geq 1$ be an integer. A q-separator for G is a non-empty set of vertices $S \subseteq V$, such that each SCC of $G \setminus S$ contains at most $n - q \cdot |S|$ vertices.

Chechik et al. [20] showed that a strongly connected graph G with n vertices and m edges of diameter $\delta \geq \sqrt{n}$ has a q-separator with quality $q = \sqrt{n}/(2 \log n)$ that can be computed in $O(m)$ time. Hence, we apply *qSep* only if the current graph has diameter at least $\sqrt{n}$. If this is the case, then we remove the $|S|$ vertices of the q-separator one at a time. Otherwise, we need to choose a split vertex by applying some other method.

In our experiments, we combined *qSep* with the *MostCriticalNode (MCN)* algorithm from [12, 29]. Note that we do not wish to compute the exact diameter of the graph,

Algorithm 2.3 MostCriticalNode

```
1: Compute the reverse digraph G^R;
2: Select an arbitrary root vertex s ∈ V(G);
3: Compute the dominator trees D and D^R w.r.t. s;
4: Compute the sets of non-trivial dominators N and N^R;
5: SAPs ← N ∪ N^R;
6: if G − s is not strongly connected then
7:    SAPs ← SAPs ∪ {s};
8: end if
9: if SAPs = ∅ then
10:    return randomVertex(V);
11: end if
12: Compute the loop nesting trees H and H^R;
13: cnode ← 0, cvalue ← f(G), value ← 0;
14: for all strong articulation point v ∈ SAPs do
15:    Compute f(D̃(v)), f(D̃^R(v)), f(PCD(v)), f(PCA(v));
16:    value ← f(D̃(v)) + f(D̃^R(v)) − f(PCD(v)) + f(PCA(v));
17:    if cvalue > value then
18:       cnode ← v;
19:       cvalue ← value;
20:    end if
21: end for
22: return cnode;
```

as this will take $O(mn)$ time. Hence, we relaxed this condition and thus we apply qSep method if for a randomly selected vertex, say v, the longest bfs path either in G or in G^R is at least $\sqrt{n}$. During our experimentation we noticed that it didn't affect the overall height of the SCC-Tree. A pseudocode for this method is described in Algorithms 2.4 and 2.5

2.3.2 Trivial ways of answering queries

The simplest way to answer strong connectivity queries of the form "*Are x and y strongly connected in $G - \{f_1, f_2\}$?*" is by traversing the graph two times (paths of the

Algorithm 2.4 qSep

Require: digraph G

1: $s \leftarrow randomVertex(V(G))$; /* Select arbitrary vertex as root */
2: $sep = \emptyset$;
3: **if** `FindSeparator(G,s)` **or** `FindSeparator(G`^{R}`,s)` **then**
4: **return** sep; /* Graph has diameter at-least $\sqrt{n}$ */
5: **end if**
6: **return** $\emptyset$; /* Couldn't find a separator */

form $x \to y$ and $y \to x$) while avoiding the failed vertices. This can be accomplished by either using DFS, BFS or bidirectional-BFS. All referred methods require linear space and time, that is, $O(n)$ space and $O(n+m)$ query time.

Depth first search **DFS** and breadth first search **BFS** are well-known algorithms used for graph traversals. Both begin from a designated source vertex s and proceed to some unvisited neighbour. In DFS we jump to the first unvisited neighbour from where we repeat the procedure. If there is no-one left to explore, we backtrack to its parent and continue the search as previously. On the other hand in BFS, initially, we add all unvisited neighbours, of the source vertex, to a queue data structure and continue the exploration from the first extracted vertex (from where we repeat the same technique). This procedure continues until the queue is empty.

Bidirectional-BFS A well-established improvement over the simple BFS is the *bidirectional*-BFS [33]. This works by alternating the search from x to y in G with a search from y to x in G^R (in order to determine whether x reaches y). If either traversal reaches a vertex that was discovered by the other, then both terminate, and the answer is positive. If either search gets stuck and is unable to make progress, we conclude that the answer is negative. We implemented the variant where the searches alternate immediately after discovering a new edge.

In Algorithms 2.6, 2.7 and 2.8 we present some simple pseudocodes for the mentioned algorithms. In case of DFS and BFS we assume that we explore the whole graph, if possible. If we wanted to avoid some (failed) vertices, simply, we could add the condition "**and** u (resp. w) is **not** a failed vertex" in lines 3 and 6 respectively. The same condition can be applied for biBFS in lines 11 and 21.

Algorithm 2.5 FindSeparator

Require: digraph G, root s

```
1: BFS(G, s);                                   /* Calculate BFS tree w.r.t root s */
2: if bfs_tree_depth < √n then
3:     return false;
4: end if
5: for integer i ∈ {2, ..., bfs_tree_depth − 1} do
6:     if layer_size[i] ≤ √n then
7:         aboveLayers ← SUM(layer_size, 1, i − 1);
8:         belowLayers ← SUM(layer_size, i + 1, bfs_tree_depth);
9:         larger ← MAX(aboveLayers, belowLayers);
10:        smaller ← MIN(aboveLayers, belowLayers);      /* We want the above and
           below layers have similar size */
11:        if smaller/larger ≥ 0.75 then
12:            for all vertices, v, ∈ layer i do
13:                sep = sep ∪ {v};
14:            end for
15:            return true;
16:        end if
17:    end if
18: end for
19: return false;
```

Algorithm 2.6 DFS

Require: source vertex $s \in V(G)$

```
1: mark s as visited;
2: for all out-going edges, e = (s, u), of s do
3:     if v is not visited then
4:         DFS(v);                /* recursively call DFS with source the vertex v */
5:     end if
6: end for
```

Algorithm 2.7 BFS

Require: source vertex $s \in V(G)$, queue data structure Q

1: mark s as *visited*;
2: `Q.insert(s);`
3: **while** Q is **not** empty **do**
4: $v \leftarrow$ `Q.pop();`
5: **for all** *out-going* edges, $e = (v, w)$, of v **do**
6: **if** w is **not** *visited* **then**
7: $visited[w] \leftarrow$ **true**;
8: `Q.insert(w);`
9: **end if**
10: **end for**
11: **end while**

Algorithm 2.8 biBFS

Require: source vertex x, destination vertex y, queue data structures Q_x, Q_y, arrays to store information $visited_x, visited_y$

1: mark x as *visited* for *visited*$_x$;
2: mark y as *visited* for *visited*$_y$;
3: Q_x`.insert(x)`;
4: Q_y`.insert(y)`
5: **while** Q_x **and** Q_y are **not** empty **do**
6: $v \leftarrow Q_x$.pop();
7: **if** $visited_y[v]$ =**true then**
8: **return true**;
9: **end if**
10: **for all** *out-going* edges, $e = (v, z)$, of v **do**
11: **if** z is **not** *visited*$_x$ **then**
12: $visited_x[z] \leftarrow$ **true**;
13: Q_x.insert(z);
14: **end if**
15: **end for**
16: $w \leftarrow Q_y$.pop();
17: **if** $visited_x[w]$ =**true then**
18: **return true**;
19: **end if**
20: **for all** *out-going* edges, $e = (w, z)$, of w **do**
21: **if** z is **not** *visited*$_y$ **then**
22: $visited_y[z] \leftarrow$ **true**;
23: Q_y.insert(z);
24: **end if**
25: **end for**
26: **end while**
27: **return false**;

Chapter 3

Our Contributions

3.1 Decomposition Tree

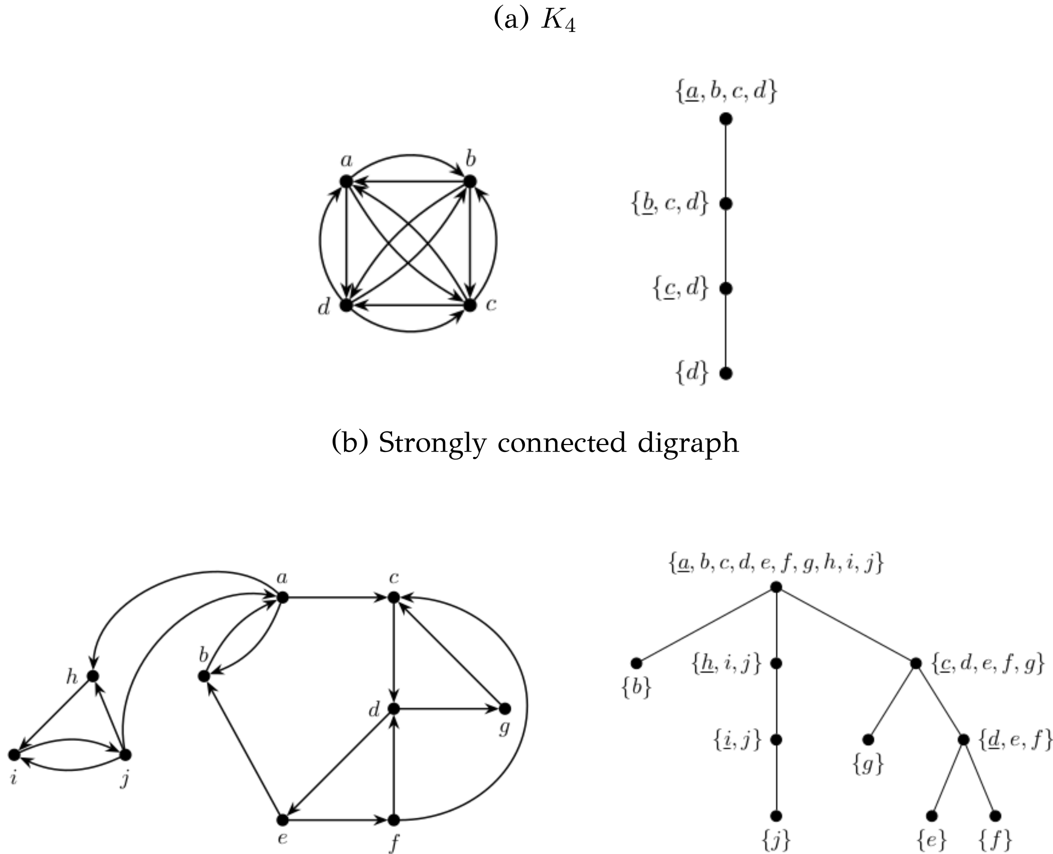
3.2 Improved Data Structure for General Graphs

3.3 BFS-Based Oracles

3.1 Decomposition Tree

We construct an SCC-Tree $\mathcal{T}$ of a strongly connected digraph $G = (V, E)$ based on the idea introduced by Łącki [19]. If the input digraph G is not strongly connected, then we construct a separate SCC-Tree for each strongly connected component. Each node $N(t)$ of $\mathcal{T}$ corresponds to a vertex t of G, referred to as the *split vertex* of $N(t)$. Also, a node $N(t)$ of $\mathcal{T}$ is associated with a subset of vertices $S_t \subseteq V$ that contains t. For every vertex $x \in V$, we define P_x to be the set of nodes $N(t)$ in $\mathcal{T}$ such that $x \in S_t$. We also let V_x be the set of split vertices that correspond to P_x, i.e., $V_x = \{t \in V : N(t) \in P_x\}$. Figure 3.1 gives two examples of SCC-Trees. The first one is for the complete directed graph K_4, while the second one is for a small planar digraph.

Figure 3.1: Two strongly connected digraphs (left) and corresponding SCC-Trees $\mathcal{T}$ (right). Every node of $\mathcal{T}$ is associated with a subset of $V(G)$ and the underlined vertex is the corresponding split vertex. For example, for the SCC-Tree of (b), the middle child of the root is $N(h)$ and $S_h = \{h, i, j\}$. Also note that $P_f = \langle N(a), N(c), N(d), N(f) \rangle$.



(a) K_4

(b) Strongly connected digraph

An SCC-Tree of G is constructed as follows:

- We choose a split vertex r of G, and let $N(r)$ be the root of $\mathcal{T}$. We associate $N(r)$ with $S_r = V(G)$.
- For a node $N(t) \in \mathcal{T}$ such that $|S_t| \geq 2$, let $H_1, \ldots, H_k$ be the SCCs of $G - V_t$. For every $i \in \{1, \ldots, k\}$ we choose a split vertex $t_i \in V(H_i)$ and make the corresponding node $N(t_i)$ a child of t. We set $S_{t_i} = V(H_i)$, and recursively compute an SCC-Tree for $G[S_{t_i}]$ rooted at $N(t_i)$.

Algorithm 3.1 describes in-detail a straightforward way for computing an SCC-Tree of a strongly connected graph G. Subroutine `SelectSplitNode(G)` is used to select

Algorithm 3.1 SCC-TreeDecomposition(G)

Require: G strongly connected graph /* Otherwise, apply same algorithm for every SCC */

1: $v \leftarrow$ SelectSplitNode(G); /* e.g., One of the algorithms of Section 2.3.1 */
2: Make v the root of T, set $S_v = V(G)$ and mark v in S_v;
3: Compute the SCCs $H_1, \ldots, H_k$ of $G - v$;
4: **for** $i \in \{1, \ldots, k\}$ **do**
5: Recursively compute T_i = SCC-TreeDecomposition(H_i);
6: Make the subtree T_i a child of v in T;
7: **end for**
8: **return** T;

a split node for the decomposition as we implemented and tested various algorithms and heuristics described in Section 2.3.1.

Observe that the number of nodes of $\mathcal{T}$ is exactly $|V|$ and every $v \in V$ appears exactly once as a split vertex in a node $N(v) \in \mathcal{T}$. Thus there is a one-to-one correspondence between the vertices of G and the nodes of $\mathcal{T}$. For every node $N(t) \in \mathcal{T}$, we define G_t to be the strongly connected subgraph of G induced by S_t, i.e., $G_t = G[S_t]$. By construction, it follows that for every $x \in V$ the nodes of P_x form a path (starting from the root) in $\mathcal{T}$. Thus we can think of P_x as an ordered set (where its elements are ordered from the root to the node $N(r)$ of $\mathcal{T}$) and we denote by $P_x(i)$ its ith element (if $|P_x| < i$ then $P_x(i) = \textit{null}$). For $x, y \in V$, we define their *nearest common ancestor*, $\textit{nca}(x, y)$, in $\mathcal{T}$ to be the last common element of P_x and P_y. At each node $N(t) \in \mathcal{T}$, we store the auxiliary data structures of Section 2.2, which we use to answer a query, as we describe next.

Answering a query. Now we describe an algorithm, which given an SCC-Tree $\mathcal{T}$ of a strongly connected graph $G = (V, E)$, two query vertices $x, y \in V$ and two failed vertices $f_1, f_2 \in V$, answers the query $2\textit{FTSC}(x, y, f_1, f_2)$ that asks whether x and y are strongly connected in $G - \{f_1, f_2\}$. The algorithm begins at the root $N(r)$ of $\mathcal{T}$ and descends the path $P_{\textit{nca}(x,y)} = P_x \cap P_y$. When we visit a node $N(t)$ we perform the following steps:

1. If t is a failed vertex, say $t = f_1$, then we check if $N(t) = \textit{nca}(x, y)$. If this is the case then we return `FALSE`. Otherwise, we return the result of the query $1\textit{FTSC}(x, y, f_2)$ for G_w, where $N(w)$ is the child of $N(t)$ containing x and y.

($N(w)$ is the next node on $P_{nca(x,y)}$.)

2. If t is not a failed vertex, we test the condition (C): $(2FTR_t(x, f_1, f_2) \neq 2FTR_t(y, f_1, f_2)) \vee (2FTR^R_t(x, f_1, f_2) \neq 2FTR^R_t(y, f_1, f_2))$ in G_t. If it is true, then we return FALSE.

3. If (C) is false and both $2FTR_t(x, f_1, f_2)$ and $2FTR^R_t(x, f_1, f_2)$ are true in G_t, then we return TRUE. Otherwise, we proceed to the next node on $P_{nca(x,y)}$.

The above procedure is presented more detailed in Algorithm 3.2.

Lemma 3.1. *The query algorithm is correct.*

Proof. First, we consider the correctness of the query algorithm. Consider a query $2FTSC(x, y, f_1, f_2)$ that asks if x and y are strongly connected in $G - \{f_1, f_2\}$. We first argue that if our procedure returns TRUE, then x and y are strongly connected in $G - \{f_1, f_2\}$. This happens in one of the following cases:

- One of the failing vertices, say f_1, is the split vertex t in the currently visited node $N(t)$ of $\mathcal{T}$. Let $N(w)$ be the child of $N(t)$ containing x and y. (This node exists because otherwise, the query algorithm would return FALSE.) We have two cases:

  – The other failing vertex, $f_2 \in S_w$. Then, we return the answer $1FTSC(x, y, f_2)$ for G_w, which is TRUE if and only if $x \leftrightarrow y$ in $G_t - \{f_1, f_2\}$. Hence, x and y are strongly connected in $G - \{f_1, f_2\}$.

  – Vertex $f_2 \notin S_w$. Then, we return TRUE since x and y are in the same SCC of $G_t - t$, induced by the vertices of S_w, which does not contain any failed vertices.

- The split vertex t of $N(t)$ is not a failed vertex. The query algorithm returns TRUE when we have $2FTR_t(x, f_1, f_2) = 2FTR_t(y, f_1, f_2)$ and $2FTR^R_t(x, f_1, f_2) = 2FTR^R_t(y, f_1, f_2)$, and also both $2FTR_t(x, f_1, f_2)$ and $2FTR^R_t(x, f_1, f_2)$ are true in G_t. Then, by Observation 2.1, x and y are both strongly connected with t in $G_t - \{f_1, f_2\}$. Thus, x and y are strongly connected in $G - \{f_1, f_2\}$. In case that neither x nor y are strongly connected with t and $2FTR_t(x, f_1, f_2) = 2FTR_t(y, f_1, f_2) = 2FTR^R_t(x, f_1, f_2) = 2FTR^R_t(y, f_1, f_2) =$ FALSE we proceed with the next node of the path P because the existence of t does not affect the connectivity of x and y. See Lemma 3.2.

For the opposite direction, suppose that x and y are strongly connected in $G - \{f_1, f_2\}$. Let C be the SCC of $G - \{f_1, f_2\}$ that contains both x and y. We argue that our procedure will return a positive answer. The vertices of C are contained in S_r, where r is the split vertex of the root node $N(r)$ of the SCC-Tree $\mathcal{T}$ (we remind that $S_r = V(G)$), meaning that $P_x(1) = P_y(1)$. Let k be the positive integer such that $nca(x, y) = P_x(k) = P_y(k)$. Assume that the query algorithm has visited the first $i \leq k$ nodes of $P_{nca(x,y)}$ without returning a positive answer. Then, C does not contain any of the split vertices of the first i nodes of $P_{nca(x,y)}$. Hence, if no positive answer has been returned until step $k-1$, then for every $i \in \{1, \dots k-1\}$, C was entirely contained in all sets S_{t_i}, where t_i is the split vertex of $N(t_i) = P_x(i) = P_y(i)$. By the definition of k, the next node $t = t_k$ considered by the algorithm has at least two distinct children that contain x and y, respectively. This implies that t is a vertex of C. Then, we have $2FTR_t(x, f_1, f_2) = 2FTR_t(y, f_1, f_2)$ and $2FTR_t^R(x, f_1, f_2) = 2FTR_t^R(y, f_1, f_2)$, and also both $2FTR_t(x, f_1, f_2)$ and $2FTR_t^R(x, f_1, f_2)$ are true in G_t. So the algorithm returns TRUE. □

Lemma 3.2. *Let $G = (V, E)$ be a digraph and let $x, y, s \in V$ such that $x \leftrightarrow y$ and $x \not\leftrightarrow s$ in G. Then $x \leftrightarrow y$ in $G - s$.*

Proof. Suppose, for the sake of contradiction, that $x \not\leftrightarrow y$ in $G - s$. As $x \leftrightarrow y$ in G, it must be that there is no $x \to y$ path avoiding s or $y \to x$ path avoiding s (or both) in G. Without loss of generality, we assume that every $x \to y$ path in G contains s and let P be such a path (it exists because $x \leftrightarrow y$ in G). Let also Q be a $y \to x$ path in G (it exists because $x \leftrightarrow y$ in G). Then, $P[x, s]$ is a $x \to s$ path in G and $P[s, y] \cup Q$ is a $s \to x$ path in G. Therefore, $x \leftrightarrow s$ in G, a contradiction. □

Space and running time. Regarding the running time of the query, we observe that the query algorithm makes at most $O(h)$ queries to the auxiliary data structures, where h is the height of the SCC-Tree $\mathcal{T}$. As each auxiliary structure has constant query time, the oracle provides the answer in $O(h)$ time. Regarding space, note that at each node $N(t) \in \mathcal{T}$, we store the auxiliary data structures of Section 2.2, which require $O(|S_t|)$ space. Hence, our oracle occupies $\sum_{t \in V(G)} O(|S_t|) = O(nh)$ space. This concludes the proof of Theorem 1.1.

Implementation details. The oracle of Choudhary [6] computes detour paths with respect to two divergent spanning trees [34] T_1 and T_2 of G. The spanning trees T_1 and T_2 are rooted at s and have the property that for any vertex $v \neq s$, the only

common vertices on the two tree s-v paths are the dominators of v. Moreover, T_1 and T_2 can be computed in $O(m)$ time [34]. To answer a query $2FTR_s(x, f_1, f_2)$, [6] uses a data structure for reporting minima on tree paths [35]. Specifically, Demaine et al. [35] show that a tree T on n vertices and edge weights can be preprocessed in $O(n \log n)$ time to build a data structure of $O(n)$ size so that given any $u, v \in T$, the edge of smallest weight on the tree path from u to v can be reported in $O(1)$ time. The data structure of [35] is rather complicated, as it applies a micro-macro decomposition of T which uses word-level parallelism. Here, we applied two simpler methods. One is based on the work of Demaine [35] but it has worse preprocessing time and the other is based on a heavy-path decomposition [36] of a tree. However, from our experiments we noticed that both methods performed similarly.

3.1.1 Special Graph Classes

The space and query time of the oracle of Section 3.1 depends on the value of the parameter h (the height of the SCC-Tree), which can be $O(n)$. For restricted graph classes, we can choose the split vertices in a way that guarantees better bounds for h. Such classes are planar graphs and bounded treewidth graphs.

Definition 3.1. A *vertex separator* of an undirected graph $G = (V, E)$ is a subset of vertices, whose removal decomposes the graph into components of size at most $\alpha|V|$, for some constant $0 < \alpha < 1$. A family of graphs $\mathcal{F}$ is called $f(n)$-separable if

- for every $F \in \mathcal{F}$, and every subgraph $H \subseteq F$, $H \in \mathcal{F}$,
- for every $F \in \mathcal{F}$, such that $n = |V(F)|$, F has a vertex separator of size $f(n)$.

Lemma 3.3 ([19]). *Let $G = (V, E)$ be a directed strongly connected graph, such that $G \in \mathcal{F}$ is Cn^s-separable ($s \geq 0$). Moreover, assume that the separators for every graph $\mathcal{F}$ can be found in linear time. Then, we can build an SCC-decomposition tree for G of height $O(h(n))$ in $O(|E|h(n))$ time, where $h(n) = O(n^s)$ for $s > 0$ and $h(n) = O(\log n)$ for $s = 0$.*

We next show the applicability of Lemma 3.3 by providing efficient 2-FT-SC oracles on well-known graph classes with structured underlying properties.

Planar graphs. Here we assume that the underlying undirected graph is planar. The following size of separators in planar graphs is well-known.

Algorithm 3.2 2FTSC(x,y,f_1,f_2,T)

Require: $x, y, f_1, f_2 \in V(G)$, SCC-Tree $\mathcal{T}$

1: $i \leftarrow 1$;
2: **while** $P_x(i) = P_y(i)$ **do**
3: $t \leftarrow$ split vertex of $P_x(i)$;
4: **if** $t = f_1$ *or* $t = f_2$ **then**
5: **if** $N(t) = nca(x, y)$ **then**
6: **return false**; /* x, y ended up in different SCCs */
7: **else**
8: $f \leftarrow \{f_1, f_2\} - t$;
9: $N(w) \leftarrow P(i + 1)$;
10: **if** $f \notin N(w)$ **then**
11: **return true**;
12: **end if**
13: **return** `1FTSC(x,y,f,`G_w`)`; /* G_w corresponds to the subgraph of $N(w)$ */
14: **end if**
15: **end if**
16: **if** $(2FTR_t(x, f_1, f_2) \neq 2FTR_t(y, f_1, f_2)) \vee (2FTR^R_t(x, f_1, f_2) \neq 2FTR^R_t(y, f_1, f_2))$ **then**
17: **return false**; /* Check condition (C) */
18: **else if** $2FTR_t(x, f_1, f_2) = 2FTR^R_t(x, f_1, f_2) =$ **true then**
19: **return true**;
20: **else**
21: $i \leftarrow i + 1$; /* Proceed with the next node */
22: **end if**
23: **end while**
24: **return false**; /* x, y ended up in different SCCs */

Theorem 3.1 ([37]). *Planar graphs are $\sqrt{8n}$-separable and the separators can be found in linear time.*

Combined with Lemma 3.3, the previous result when G is planar yields the following:

Lemma 3.4. *Let $G = (V, E)$ be a directed strongly connected planar graph. Then, we can build an SCC-decomposition tree for G of height $O(\sqrt{n})$ and $O(n^{3/2})$ space.*

Graphs of bounded treewidth. Here we consider graphs for which their underlying undirected graph has bounded treewidth.

Theorem 3.2 (Reed [38]). *Graphs of treewidth at most k are k-separable. Assuming that k is constant, the separators can be found in linear time.*

It is known that graphs with constant treewidth have $O(n)$ edges (see Reed [38]). This fact combined with Theorem 3.2 and Lemma 3.3 implies that when $G = (V, E)$ is a directed graph, whose treewidth (of its underlying undirected graph) is bounded by a constant k, then we can build an SCC-decomposition tree for G of height $O(\log n)$ in $O(n \log n)$ time. Thus, we obtain the following result.

Theorem 3.3. *Let $G = (V, E)$ be a directed graph, whose treewidth of its underlying undirected graph is bounded by a constant k. We can construct in polynomial time a data structure of size $O(n \log n)$ that answers strong connectivity queries between two vertices of G under two vertex (or edge) failures in $O(\log n)$ time.*

3.2 Improved Data Structure for General Graphs

In this section, we present an improved data structure for general graphs. Our data structure uses $O(n\sqrt{m})$ space and answers strong connectivity queries in $O(\sqrt{m})$ time. The main idea of the improved oracle is that when we build the SCC-Tree $\mathcal{T}$, we can stop the decomposition of a subgraph G_t early if some appropriate conditions are satisfied (e.g. when G_t is 3-vertex connected). We refer to such a decomposition tree $\mathcal{T}$ of G as a *partial-SCC-Tree*. Let Δ be an integer parameter in $[1, m]$. A subgraph G' of G is "large" if it contains at least $\Delta + 1$ edges, and "small" otherwise. Next, we define $\Delta-$good graphs.

Definition 3.2 ($\Delta-$good). A strongly connected graph G is "$\Delta-$good" if it has the following property: For every separation pair $\{f_1, f_2\}$ of G, the graph $G' = G - \{f_1, f_2\}$ satisfies the following:

1. It has at most one "large" strongly connected component C that contains at least $\Delta + 1$ edges.

2. All the remaining strongly connected components have size (i.e., number of edges) at most Δ.

3. For every node x of G' not belonging to C (*large* SCC) it holds that either $G[\mathit{Pred}_{G'}(x) \cup \{f_1, f_2\}]$ or $G[\mathit{Succ}_{G'}(x) \cup \{f_1, f_2\}]$ contains at most Δ edges.

Note that condition 2 in Definition 3.2 is actually implied by condition 3, but we state it explicitly for clarity.

The following lemma shows that all 2-FT-SC queries in a $\Delta-$good graph can be answered in $O(\Delta)$ time, by performing four local searches with threshold Δ.

Lemma 3.5. *Let G be a $\Delta-$good graph. Then, any 2-fault strong connectivity query can be answered in $O(\Delta)$ time.*

Proof. Consider a query that asks if vertices x and y are strongly connected in $G' = G - \{f_1, f_2\}$. Note that $x \leftrightarrow y$ in G' if and only if $y \in \mathit{Succ}_{G'}(x)$ and $x \in \mathit{Succ}_{G'}(y)$, which implies $\mathit{Succ}_{G'}(x) = \mathit{Succ}_{G'}(y)$ (equivalently, $\mathit{Pred}_{G'}(x) = \mathit{Pred}_{G'}(y)$). To answer the query, for $z \in \{x, y\}$, we run simultaneously searches from and to z (by executing a BFS or DFS from z in G' and $(G')^R$, respectively) in order to discover the sets $\mathit{Pred}_{G'}(z)$ and $\mathit{Succ}_{G'}(z)$. (To perform a search in G', we execute a search in G but without expanding the search from f_1 or f_2 if we happen to meet them.) We stop such a search early, as soon as the number of traversed edges reaches $\Delta + 1$. If this happens both during the search for $\mathit{Pred}_{G'}(z)$ and for $\mathit{Succ}_{G'}(z)$, then we conclude that $z \in C$ (*large* SCC). Thus, if all the four searches for $\mathit{Pred}_{G'}(z)$ and $\mathit{Succ}_{G'}(z)$, for $z \in \{x, y\}$, are stopped early, we know that both x and y belong to C and so there are strongly connected.

Now, suppose that the search for $\mathit{Succ}_{G'}(x)$ traversed at most Δ edges. Then, $x \leftrightarrow y$ in G' only if the search for $\mathit{Succ}_{G'}(y)$ also traversed at most Δ edges. Hence, if the search for $\mathit{Succ}_{G'}(y)$ stopped early, we know that x and y are not strongly connected in G'. Otherwise, we just need to check if $y \in \mathit{Succ}_{G'}(x)$ and $x \in \mathit{Succ}_{G'}(y)$. The case where the search for $\mathit{Pred}_{G'}(x)$ traversed at most Δ edges is analogous.

So, in every case, we can test if x and y are strongly connected in G' in $O(\Delta)$ time. □

We call a separation pair $\{f_1, f_2\}$ of G “good” if every strongly connected component of $G - \{f_1, f_2\}$ contains at most Δ edges. Now, to build a *partial*-SCC-Tree $\mathcal{T}$, we distinguish the following cases.

1. G **is “small”**. We stop the decomposition here, because all queries in G can be answered in $O(\Delta)$ time.

2. G **is** 3**-vertex-connected**. Then, G does not contain any separation pairs, so all queries are answered (in the affirmative) in $O(1)$ time.

3. G **contains a good separation pair** $\{f_1, f_2\}$. Here, we choose $\{f_1, f_2\}$ as the next two split vertices of G, because all children of G in the decomposition tree correspond to "small" graphs.

4. G **is** $\Delta-$**good.** Then we stop the decomposition here, because all queries can be answered in $O(\Delta)$ time by Lemma 3.5.

5. **None of the above applies.** For this case, we prove that G has the following property:

Lemma 3.6. *(Case 5) There is at least one separation pair $\{f_1, f_2\}$ of G such that every SCC of $G' = G - \{f_1, f_2\}$ is either small, or contains fewer than $m - \Delta$ edges.*

Proof. Since G is not 3-vertex-connected, there is at least one separation pair. Also, since G is not $\Delta-$good, there exists at least one separation pair $\{f_1, f_2\}$ with the property that either: (a) $G - \{f_1, f_2\}$ contains more than one large SCC, or (b) $G - \{f_1, f_2\}$ contains only one large SCC, C, and for at least one vertex v of G' that does not belong to C, we have that both $G'[\mathit{Pred}_{G'}(v) \cup \{f_1, f_2\}]$ and $G'[\mathit{Succ}_{G'}(v) \cup \{f_1, f_2\}]$ contain at least Δ edges.

If (a) is true, then the Lemma holds since all SCCs of G' contain fewer than $m - \Delta$ edges. Now suppose that (b) is true. Since C is a SCC of G', we either have $\mathit{Succ}_{G'}(v) \cap C = \emptyset$ or $C \subset \mathit{Succ}_{G'}(v)$. If $\mathit{Succ}_{G'}(v)$ does not contain C, then C has fewer than $m - \Delta$ edges, and all the other strongly connected components have size at most Δ. Hence, the Lemma holds. Otherwise, $C \subset \mathit{Succ}_{G'}(v)$, and since $v \notin C$, we have $\mathit{Pred}_{G'}(v) \cap C = \emptyset$. Since $G'[\mathit{Pred}_{G'}(v) \cup \{f_1, f_2\}]$ contains at least Δ edges, C has fewer than $m - \Delta$ edges. Hence, the Lemma holds in this case as well. □

Obviously, only the last case may lead to repeated decompositions of G, but due to Lemma 3.6 this occurs at most m/Δ times. Thus, the decomposition tree has height $O(m/\Delta)$, and so it requires $O(mn/\Delta)$ space. Moreover, queries can be answered in $O(m/\Delta + \Delta)$ time. This proves Theorem 1.2. The running time is minimized for $\Delta = \sqrt{m}$, which gives Corollary 1.3.

Algorithm 3.3 is used for answering 2-fault tolerant queries given a *partial*-SCC-Tree where a 2-FT-SSR and 1-FT-SC oracle are initialized for every node of the tree

which fell into case 3 or 5. The idea is similar with that on Algorithm 3.2 except here we also check for the extra cases. Subroutine `areStronglyConnected(x,y,`f_1`,`f_2`,` Δ`)` performs the four biBFS traversals, with threshold $\Delta + 1$, in order to find the sets of $Pred_{G_w}(x)$, $Succ_{G_w}(x)$ (resp. for y) and answer the strong connectivity query as described earlier.

3.2.1 Choosing a good Δ for a partial-SCC-Tree decomposition

Although we may always set $\Delta = \sqrt{m}$ in order to minimize both $O(\Delta)$ and $O(m/\Delta)$ (the height of the decomposition tree), in practice we may have that a small enough Δ may be able to provide a *partial*-SCC-Tree with height $O(\Delta)$. For example, this is definitely the case when G itself is $\Delta-$good. Otherwise, it may be that after deleting a few pairs of vertices, we arrive at subgraphs that are $\Delta-$good.

Table 4.3 shows some examples of real graphs where we have computed a value for Δ such that the *partial*-SCC-Tree has height at most Δ. Thus, we get data structures for those graphs that can answer 2-FT-SC queries in $O(\Delta)$ time. We arrived at those values for Δ by essentially performing binary search, in order to find a Δ that is as small as possible and such that either the graph is $\Delta-$good, or it has a *partial*-SCC-Tree decomposition with height at most Δ.

The computationally demanding part here is to determine whether a graph is $\Delta-$good, for a specific Δ. The straightforward method that is implied by the definition takes $O(n^2(m+n\Delta))$ time. (I.e., this simply checks the SCCs after removing every pair of vertices, and it performs local searches with threshold $\Delta + 1$ starting from every vertex.) Instead, we use a method that takes $O(nm + \Sigma_{v \in V(G)} SCC_v(G)\Delta)$ time, where $SSC_v(G)$ denotes the total number of strongly connected components of $G \setminus \{v, u\}$, for every vertex $u \in G \setminus v$. In practice, this works much better than the stated bound, because $SSC_v(G)$ is approximately $\Theta(n)$, for every $v \in G$.

The idea is to check the SCCs after the removal of every vertex $v \in G$ (this explains the $O(nm)$ part). If $G \setminus v$ has at least three large SCCs, then we can immediately determine that G is not $\Delta-$good. Otherwise, we distinguish three cases, depending of whether $G \setminus v$ has 0, 1 or 2 large SCCs. In the first case, we can terminate the computation, because all SCCs of $G \setminus v$ are small. In the other two cases, we essentially rely on the work [39], with which we can compute in $O(SCC_v(G))$ time all the strongly connected components of $G\setminus\{v, u\}$, for every vertex $u \in G\setminus v$, by exploiting information

Algorithm 3.3 2FTSC-*partial*-SCC-Tree

Require: $x, y, f_1, f_2 \in V(G)$, *partial* SCC-Tree

```
1:  i ← 1;
2:  while P_x(i) = P_y(i) do
3:     N(t) ← P_x(i);                                   /* Current node of the path. */
4:     G_t ← induced subgraph of S_t                    /* S_t is associated with the node N(t) */
5:     if G_t is small or Δ−good then
6:        return areStronglyConnected(x,y,f_1,f_2);
7:     else if G_t is 3-vertex-connected then
8:        return true;
9:     end if
10:    t ← split vertex of P_x(i);     /* G_t contains a good separation pair or satisfies
       Lemma 3.6 */
11:    if t = f_1 or t = f_2 then
12:       if N(t) = nca(x,y) then
13:          return false;                              /* x, y ended up in different SCCs */
14:       else
15:          f ← {f_1, f_2} − t;
16:          N(w) ← P(i+1);
17:          return 1FTSC(x,y,f,G_w);   /* G_w corresponds to the subgraph of N(w) */
18:       end if
19:    end if
20:    if (2FTR_t(x,f_1,f_2) ≠ 2FTR_t(y,f_1,f_2)) ∨ (2FTR_t^R(x,f_1,f_2) ≠ 2FTR_t^R(y,f_1,f_2))
       then
21:       return false;                                 /* Check condition (C) */
22:    else if 2FTR_t(x,f_1,f_2) = 2FTR_t^R(x,f_1,f_2) = true then
23:       return true;
24:    else
25:       i ← i + 1;                                    /* Proceed with the next node */
26:    end if
27: end while
28: return false;                                       /* x, y ended up in different SCCs */
```

from the dominator trees and the loop nesting forests of the SCCs of $G \setminus v$. For every such small component, it suffices to select a representative vertex x, and perform the two local searches from x in G and G^R with threshold $\Delta + 1$ (and blocking vertices v and u), in order to determine whether x either reaches at most Δ edges, or is reached by at most Δ edges.

Still, our result on the *partial*-SCC-Tree decomposition (Corollary 1.3) is mainly of theoretical interest, since the procedure for determining whether a graph is $\Delta-$good becomes very slow even in moderately large graphs. Thus, in Section 4 we suggest much more efficient heuristics, that work remarkably well in practice.

3.3 BFS-Based Oracles

As mentioned in Section 2.3.2 the straightforward way to determine whether two vertices x and y remain strongly connected after the removal of f_1, f_2, is to perform a graph traversal (e.g., BFS) in order to check whether x reaches y in $G - \{f_1, f_2\}$, and conversely. We call BFS algorithm as *simpleBFS* and *bidirectional*-BFS as *biBFS*. For either algorithm we measure the work done by keeping track of the edges that we had to access in order to arrive at the answer. As expected, *biBFS* has to do on average less work than *simpleBFS*, and thus we use *biBFS* as the baseline.

One of our most important contributions is a heuristic that we call *seeded* BFS. This precomputes some data structures on a few (random) vertices which we call seeds, and we use during the BFS if we meet them.[1] Specifically, we use every seed r in order to expand a DFS tree DFS_r of G with root r, and we maintain the preordering of all vertices w.r.t. DFS_r, as well as the number of descendants $ND_r(v)$ on DFS_r for every vertex $v \in G$. This information can be computed in linear time, and it can be used in order to answer ancestry queries w.r.t. DFS_r in $O(1)$ time [41]. We do the same on G^R with the same seed vertices.

Now, in order to answer a SC-query for x and y in $G - \{f_1, f_2\}$, we first perform a bidirectional BFS from x to y with the following twist: if we meet a seed r, then we check whether either of f_1, f_2 is an ancestor of y on DFS_r. If neither of f_1, f_2 is an ancestor of y, then we can immediately conclude that x reaches y in $G - \{f_1, f_2\}$.

[1] In the literature, the vertices that support such functionality are commonly called supportive vertices, or landmarks [33, 40].

Otherwise, we just continue the search. Then we use the same method in order to determine the reachability from y to x in $G - \{f_1, f_2\}$. Furthermore, we can improve over this idea a little more: even before starting the BFS, we perform this simple check that we have described, in order to see if x reaches a seed, and then a seed reaches y. If the number of seeds is very small (e.g., 10), then this initial scanning of the seeds takes a negligible amount of time. What is remarkable, is that even with a single random seed, there is a very high probability that a random 2-FT-SC query will be answered even before the BFS begins! We call our implementation of this idea *sBFS*. Here, the two measures of efficiency are, first, whether the answer was given by a seed (before starting the BFS), and second, what is the total number of edges that we had to access (in case that none of the seeds could provide immediately the answer). We expect the average number of edges accessed to be much lower than in *biBFS*, because we use the seeds to speed up the search in the process. If *sBFS* uses k seeds, we denote the algorithm as *sBFS*(k).

Observe that the checks at the seeds may provide an inconclusive answer (i.e., if either f_1 or f_2 is an ancestor of the target vertex on the tree-path starting from the seed). Thus, we may instead initialize a 2-FT-SSR data structure on every seed, so that every reachability query provides immediately the real answer. In this case, we can extract all the information that the seeds can provide before the BFS begins. We do this by using the four reachability queries $\mathit{2FTR}_r(x, f_1, f_2)$, $\mathit{2FTR}_r(y, f_1, f_2)$, $\mathit{2FTR}_r^R(x, f_1, f_2)$ and $\mathit{2FTR}_r^R(y, f_1, f_2)$, as we did in Section 3.1. We call our implementation of this idea *ChBFS* (or *ChBFS*(k), to emphasize the use of k seeds). As in *sBFS*, the two measures of efficiency here are whether one of the seeds provided the answer, or, if not, what is the number of edges that we had to traverse with the bidirectional BFS.

We consider the queries that force *ChBFS* to perform BFS in order to get the answer, worst-case instances for this algorithm. In order to reduce the probability of such events, we suggest organizing the seeds on a decomposition tree. This reduces the possibility of the worst-case instances, and it may also provide some extra "free" seeds on the intermediary levels of the decomposition tree. We elaborate on this idea in Section 4.4.

Algorithms 3.4 and 3.5 describe how to answer a strong connectivity query for the BFS-Based oracles.

Algorithm 3.4 2FTSC-sBFS

Require: x, y, f_1, f_2 from set $V(G)$, DFS trees DFS_r, DFS_r^R for every seed r

1: **for** $r \in Seeds$ **do**
2: flag_1 ← **false**;
3: flag_2 ← **false**;
4: **if not**($is_ancestor(f_1, x, DFS_r^R)$) **and not**($is_ancestor(f_2, x, DFS_r^R)$) **then**
5: **if not**($is_ancestor(f_1, y, DFS_r)$) **and not**($is_ancestor(f_2, y, DFS_r)$) **then**
6: flag_1 ← **true**; /* $x \to y$, through r */
7: **end if**
8: **end if**
9: **if not**($is_ancestor(f_1, y, DFS_r^R)$) **and not**($is_ancestor(f_2, y, DFS_r^R)$) **then**
10: **if not**($is_ancestor(f_1, x, DFS_r)$) **and not**($is_ancestor(f_2, x, DFS_r)$) **then**
11: flag_2 ← **true**; /* $y \to x$, through r */
12: **end if**
13: **end if**
14: **if** flag_1=flag_2=**true then**
15: **return true**;
16: **else if** flag_1≠flag_2 **then**
17: **return false**;
18: **else**
19: continue;
20: **end if**
21: **end for**
22: **return** `biBFS(x,y,f1,f2)` **and** `biBFS(y,x,f1,f2)` /* If the seeds failed to provide an answer, perform two biBFS traversals */

Algorithm 3.5 2FTSC-ChBFS

Require: x, y, f_1, f_2 from set $V(G)$, for every seed, r, initialized a 2-FT-SSR w.r.t r

1: **for** $r \in Seeds$ **do**
2:    **if** $(2FTR_r(x, f_1, f_2) \neq 2FTR_r(y, f_1, f_2)) \vee (2FTR_r^R(x, f_1, f_2) \neq 2FTR_r^R(y, f_1, f_2))$ **then**
3:       **return false;**
4:    **else if** $(2FTR_t(x, f_1, f_2) = 2FTR_t^R(x, f_1, f_2)) =$ **true then**
5:       **return true;**
6:    **else**
7:       continue;
8:    **end if**
9: **end for**
10: **return** `biBFS(x,y,f1,f2)` **and** `biBFS(y,x,f1,f2)` /* If the seeds failed to provide an answer, perform two biBFS traversals */

Chapter 4

Empirical Analysis

4.1 Datasets

4.2 Height of the decomposition tree.

4.3 Answering queries

4.4 An improved data structure: organizing the CH seeds on a decomposition tree

We implemented our algorithms in C++, using g++ v.7.4.0 with full optimization (flag -O3) to compile the code.[1] The reported running times were measured on a GNU/Linux machine, with Ubuntu (18.04.5 LTS): a Dell PowerEdge R715 server 64-bit NUMA machine with four AMD Opteron 6376 processors and 128GB of RAM memory. Each processor has 8 cores sharing a 16MB L3 cache, and each core has a 2MB private L2 cache and 2300MHz speed. In our experiments we did not use any parallelization, and each algorithm ran on a single core. We report CPU times measured with the `high_resolution_clock` function of the standard library `chrono`, averaged over ten different runs.

[1] Our code, together with some sample input instances is available at `https://github.com/dtsok/2-FT-SC-O`.

Table 4.1: Graph instances used in the experiments, taken from [1], [2] and [3]. n and m are the numbers of vertices and edges, respectively, n_a is the number of strong articulation points (SAPs), and n_{sp} is the number of vertices that are SAPs or belong to a proper separation pair.

Graph	Type	n	m	n_a	n_{sp}	Reference
Google_small	web graph	950	1,969	179	182	[1]
Twitter	communication network	1,726	6,910	615	1,005	[1]
Rome	road network	3,353	8,870	789	1,978	[2]
Gnutella25	p2p network	5,152	17,691	1,840	3,578	[3]
Lastfm-Asia	social network	7,624	55,612	1,338	2,455	[3]
Epinions1	social network	32,220	442,768	8,194	11,460	[3]
NotreDame	web graph	48,715	267,647	9,026	15,389	[3]
Stanford	web graph	150,475	1,576,157	20,244	56,404	[3]
Amazon0302	co-purchase graph	241,761	1,131,217	69,616	131,120	[3]
USA-road-NY	road network	264,346	733,846	46,476	120,823	[2]

4.1 Datasets

The real-world graphs we used in our experiments are reported in Table 4.1. From each original graph, we extracted its largest SCC, except for Google_small for which we use its second-largest SCC, hence the reported statistics refer to those SCC. Additional results concerning the artificial graphs can be found in the Appendix A.1.

From Table 4.1 we observe that a significant fraction of the vertices belong to at least one proper separation pair (value n_{sp} in the table). Indeed, at least 19% of the vertices, and 44% on average, belong to a proper separation pair.

4.2 Height of the decomposition tree.

As stated in Section 2.3, we consider various methods for constructing a decomposition tree $\mathcal{T}$ of G with small height h in practice. We note that such decomposition trees are useful in various decremental connectivity algorithms (see, e.g., [20, 21, 19]), so this experimental study may be of independent interest. We consider only fast methods for selecting split vertices, detailed in Table 2.1. Note that all methods require $O(m)$ time to select a split node x of G, except *Random* which selects a vertex in

constant time. Still, we need $O(m+n)$ time to compute the SCCs of $G-x$. Also note that, *LNT* can find all split vertices in $O(m)$ time.

The experimental results for the graphs of Table 4.1 are presented in Table 4.2, and are plotted in Figure 4.1.We observe that *MCN* and *qSep+MCN* achieved overall significantly smaller decomposition height compared to the other methods. In fact, *Random*, *LP*, and *PR* did not manage to produce the decomposition tree for the largest graphs (Amazon0302 and USA-road-NY) in our collection, due to memory or running time restrictions. For the remaining (smaller) graphs in our collection, *Random* performed very poorly, giving a decomposition height that was larger by a factor of 11.9 on average compared to *MCN*. Also, on average, *LP* performed better than *PR*. We see that *MCN* produced a tree height that, on average, was smaller by a factor of 2.1 compared to *LP* and by a factor of 4.1 compared to *PR*. Finally, *MCN* and *qSep+MCN* have similar performance in most graphs, but for the two road networks (Rome and USA-road-NY), *qSep+MCN* produce a significantly smaller decomposition height. Comparing the results of the above algorithms with these from the loop nesting tree (LNT) we observe that on average for the small graphs (excluding Amazon0302 and USA-road-NY), *LP* outperforms *LNT* by a factor of 1.4. *PR* has similar performance and *LNT* outperforms *Random* by a factor of 5. Including the graphs Amazon0302 and USA-road-NY, *MCN* and *qSep+MCN* outperforms *LNT* by a factor of 2.5 and 3.2, respectively.

Table 4.3 reports the characteristics of *partial*-SCC-Trees achieved by the algorithm of Section 3.2 for some input graphs. Specifically, we report the height of the *partial*-SCC-Tree and the value of the parameter Δ which gives the maximum number of edges that need to be explored in order to answer a query. We were not able to include results for the larger graphs in our collection due to high running times to compute the *partial*-SCC-Tree. We observe that the results are very encouraging. For example, we see that Epinions1 is a 60-good graph. Thus, we can answer every 2-FT-SC query on Epinions1 after scanning at most $4 \times 60 + 4$ edges. Note that Epinions1 has 442.768 edges. In NotreDame, we may have to reach a tree-node of depth 168 in order to arrive at a 170-good subgraph, in which we can answer the 2-FT-SC query after scanning at most $4 \times 170 + 4$ edges. Before that, we will have to perform 4×168 2-FT-SSR queries on the tree-nodes that we traverse. This is still much faster than the scanning of ~ 15000 edges that we have to perform on average with a single bidirectional BFS on NotreDame, as shown in Table 4.7.

Table 4.2: SCC-Tree height of the graphs of Table 4.1, resulting from the split vertex selection algorithms of Table 2.1. The symbols † and ‡ refer to decompositions that were not completed due to exceeding the RAM memory of our system (> 128GB) or due to requiring more than 48 hours.

Graph	*Random*	*MCN*	*LP*	*PR*	*qSep+MCN*	*LNT*
Google_small	214	9	14	128	9	16
Twitter	732	232	352	351	235	430
Rome	843	542	478	1125	380	1,417
Gnutella25	2,118	838	2,105	1,440	858	1,920
Lastfm-Asia	4,958	2,202	1,443	3,752	2,198	2,981
Epinions1	19,764	5,602	6,826	8,367	5,857	7,317
NotreDame	11,688	365	1,704	672	390	1,346
Stanford	42,383	1,617	5,738	12,694	1,638	4,161
Amazon0302	†	16,615	‡	‡	16,606	47,484
USA-road-NY	†	19,073	‡	‡	7,829	82,098

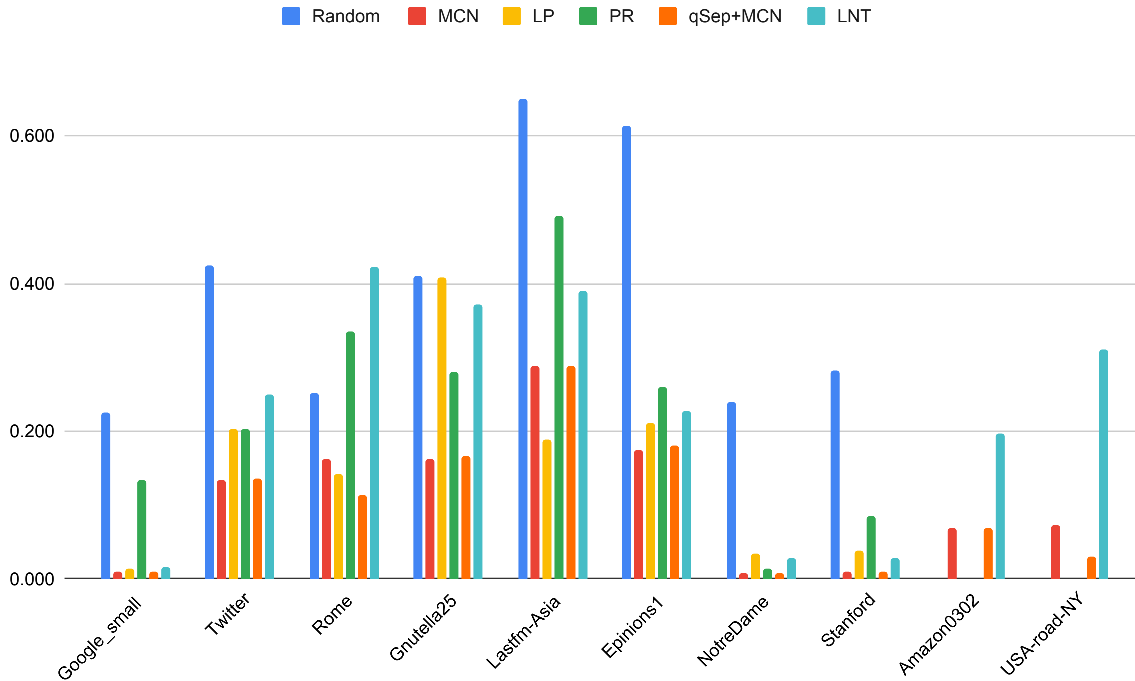
4.3 Answering queries

First, we consider random queries. We create each query $2FTSC(x, y, f_1, f_2)$ by selecting the vertices x, y and the failed vertices f_1, f_2 uniformly at random. The corresponding results for the (basic) SCC-Tree are reported in Table 4.5. Evidently, the SCC-Tree is very effective, as almost all queries are answered at the root node of the tree.

In Table 4.7 we see the time that is needed in order to answer 1M queries using *simpleBFS* and *biBFS*. We can also see the average edge accesses per query. We note that edge accesses is an accurate indicator of the total time of those algorithms. *biBFS* charges every edge access a little higher, because every new edge discovery is succeeded by an alteration to the direction of the BFS. From Table 4.7 we can see that *biBFS* performs much better than *simpleBFS*, and thus we use *biBFS* as the baseline, and as the last resort when all other heuristics fail to provide the answer.

Table 4.4 demonstrates the superiority of *sBFS* and *ChBFS* for the graphs Rome99 and Google_small. The tables for the rest graphs are in the Appendix A.3. We tested those algorithms with a few number k of seeds, $k \in \{1, 2, 5, 10\}$. In those tables we measure the two indicators of efficiency of the seeded BFS algorithms. That is, we

Figure 4.1: Relative SCC-Tree height of the graphs of Table 4.1 w.r.t. to the number of vertices, resulting from the split vertex selection algorithms of Table 2.1.



compute the percentage of the queries that are answered simply by querying the seeds (let us call these "good" instances), and the average number of edges explored per query, when we have to resort to BFS (in the "bad" instances).[2]

What is remarkable, is that most queries are answered by simply querying the seeds, and very rarely do we have to resort to BFS. Observe that the higher the number of seeds, the higher the probability that they will provide the answer. However, even with a single seed we get very good chances in obtaining the answer. As expected, the seeds in which we have initialized a 2-FT-SSR oracle (CH-seeds) have better chances for providing the answer. (In some cases, we get 100% of the answers from the CH seeds.) We note that these results essentially explain the very good times that we observe in Table 4.5, since even a single seed can provide the answer to most queries

[2]To clarify, these tables do not show the average number of edges explored per bad instance, but the average number of edges explored for all instances. Thus, the average number of edges reported is a good indicator of the relative running times.

Table 4.3: Characteristics of *partial*-SCC-Trees achieved by the algorithm of Section 3.2.

Graph	height	Δ	m	$\lfloor\sqrt{m}\rfloor$
Google_small	3	5	1,969	44
Twitter	0	35	6,910	83
Rome99	0	46	8,870	94
Gnutella25	0	11	17,691	133
Lastfm_Asia	0	21	55,612	235
Epinions1	0	60	442,768	665
NotreDame	168	170	267,647	517

Table 4.4: Relative performance of the BFS-based algorithms on Rome99 (left) and Google_small (right).

Rome99			Google_small		
Algorithm	avg #edges explored	% of answer by seed	Algorithm	avg #edges explored	% of answer by seed
simpleBFS	8,438.23		simpleBFS	1,527.02	
biBFS	3,672.39		biBFS	148.72	
sBFS(1)	100.81	96.09	sBFS(1)	3.95	97.98
sBFS(2)	11.25	99.47	sBFS(2)	3.11	98.58
sBFS(5)	1.87	99.86	sBFS(5)	3.02	98.66
sBFS(10)	0.61	99.92	sBFS(10)	2.98	98.69
ChBFS(1)	2.66	99.93	ChBFS(1)	1.67	99.07
ChBFS(2)	0.00	100.00	ChBFS(2)	0.56	99.80
ChBFS(5)	0.00	100.00	ChBFS(5)	0.05	99.95
ChBFS(10)	0.00	100.00	ChBFS(10)	0.02	99.97

(and thus, only rarely do we have to descend to deeper level of the decomposition tree). As we can see in Table 4.8, performing the 2-FT-SSR queries on the CH-seeds is a very affordable operation, comparable to accessing a few edges.

Lastly, in Table 4.6 we present the average query time (in seconds) for various methods. As we can see the methods *2-FT-SC-O*, *sBFS* and *ChBFS* perform quite similarly and, as expected, are far superior than the simple graph traversals *DFS*, *simpleBFS*, *biBFS*.

Table 4.5: Results for 1M random queries using the SCC-Tree with split vertices selected by MCN.

Graph	tree	query depth			query time		query result		avg. calls	
	depth	min	max	avg.	total (s)	avg. (s)	+	−	2-FT-SSR-O	1-FT-SC-O
Google_small	9	0	5	0.0030	5.00e-2	5.00e-8	987,220	12,780	3.976	0.000181
Twitter	232	0	3	0.0010	6.18e-2	6.18e-8	998,083	1,917	3.990	0.001000
Rome99	542	0	1	0.0005	8.70e-2	8.70e-8	999,623	377	3.997	0.000575
Gnutella25	838	0	1	0.0004	6.70e-2	6.70e-8	999,501	499	3.998	0.000410
Lastfm-Asia	2,202	0	1	0.0002	5.40e-2	5.40e-8	999,851	149	3.999	0.000243
NotreDame	365	0	9	0.0001	7.24e-2	7.24e-8	999,760	240	4.000	0.000034
Stanford	1,617	0	1	0.0000	8.99e-2	8.99e-8	999,953	47	4.000	0.000016
Epinions1	5,602	0	1	0.0000	6.60e-2	6.60e-8	999,920	80	4.000	0.000047
Amazon0302	16,615	0	1	0.0000	9.90e-2	9.90e-8	999,979	21	4.000	0.000008
USA-NY	19,073	0	1	0.0000	3.23e-1	3.23e-7	999,993	7	4.000	0.000008

4.4 An improved data structure: organizing the CH seeds on a decomposition tree

Although the algorithm *ChBFS* has the best performance, it has mainly two drawbacks. First, initializing the 2-FT-SSR data structures on the seeds is very costly, and thus we cannot afford to use a lot of seeds. And second, as noted in Section 3.3, there are instances of queries where the seeds cannot provide the answer, and therefore we have to resort to BFS.

We can make a more intelligent use of the CH-seeds by organizing them on a decomposition tree. More precisely, we use the CH-seeds as split vertices in order to produce an SCC-decomposition tree. This confers two advantages. (1) We may get some extra "free" CH-seeds on the intermediary levels of the decomposition tree. (2) We essentially maintain all the reachability information that can provided from the seeds, as if we had initialized them on the whole graph.

Let us elaborate on points (1) and (2). First, initializing Choudhary's data structure for a single vertex takes $O(mn)$ time. However, every level of the decomposition tree has $O(m)$ edges in total. Thus, we can afford to initialize as many Choudhary's data structures on every level as are the nodes in it, at the total cost of initializing a single data structure. This explains (1). Furthermore, at the deepest level of the decomposition tree we can afford to initialize a $sBFS(1)$ data structure on every node (as this is constructed in linear time and takes $O(n)$ space). We call the resulting data structure *ChTree*. The proof for (2) is essentially given by induction on the level of

Table 4.6: Comparing running times (avg. (s) per query after 1M random queries) for various algorithms.

Graph	*2-FT-SCO*	*DFS*	*simpleBFS*	*biBFS*	*sBFS(10)*	*ChBFS(3)*
Google_small	5.00e-08	5.62e-06	3.90e-06	8.31e-07	6.08e-08	5.18e-08
Twitter	6.18e-08	5.30e-05	2.77e-05	1.09e-06	4.57e-08	7.37e-08
Rome99	8.70e-08	7.04e-05	6.02e-05	3.32e-05	5.46e-08	1.11e-07
Gnutella25	6.70e-08	1.55e-04	7.67e-05	1.72e-06	4.64e-08	7.74e-08
Lastfm-Asia	5.40e-08	3.36e-04	1.46e-04	5.51e-06	4.35e-08	1.01e-07
NotreDame	7.24e-08	2.23e-03	5.91e-04	5.18e-05	7.25e-08	8.53e-08
Stanford	8.99e-08	1.10e-03	5.76e-03	4.32e-04	1.02e-07	9.93e-08
Espinions1	6.60e-08	9.90e-03	9.23e-04	1.35e-06	4.97e-08	1.53e-08
Amazon0302	9.90e-08	1.37e-02	7.06e-03	2.23e-04	7.71e-08	8.90e-08
USA-road-NY	3.23e-07	6.11e-03	5.58e-03	3.97e-03	2.81e-07	9.50e-08

the decomposition tree.

We have conducted an experiment in order to demonstrate the superiority of this idea. Specifically, we first observe that the problem of the bad instances is caused by separation pairs whose removal leaves all the seeds concentrated into small SCCs, whereas the query vertices lie in larger components that are unreachable from the seeds. In our experiments, we used 10 CH-seeds. However, from Table 4.4, we can see that it is very rare to get bad instances from 10 random seeds. Thus, we have to contrive a way to get seeds that have a high probability to give rise to a lot of bad instances. To do this, we compute the SAPs of the graph, and we process some of them randomly. If for a sap s the total number of vertices in the SCCs of $G \setminus s$, except the largest one, is at least 10, then we select randomly 10 seeds from those components. Then, we generate 10K random queries, where one of the failed vertices is s, and the query vertices lie in SCCs that do not contain seeds.

On the one hand, we use *ChBFS* to answer the queries. As expected, the seeds almost always fail to provide the answer, and so *ChBFS* can do no better than resort to *biBFS*. (However, sometimes we manage to squeeze out a negative answer from the seeds, due to Observation 2.1.) On the other hand, we use the SCC-decomposition tree *ChTree* to answer the queries.

Since the data structures of Choudhary take a lot of time to be initialized, we could perform a large number of those experiments by simulating the process of answering

Table 4.7: Results for 1M random queries using *simpleBFS* and *biBFS*. Here is shown the number of edges that we had to access on average per query, as well as the total time for answering all queries on every graph. The third column for every algorithm shows the time in nanoseconds that is charged to every edge access.

Graph	simpleBFS			biBFS		
	#edges/query	time (s)	edge access (ns)	#edges/query	time (s)	edge access (ns)
Google_small	1,527.15	3.52	2.30	148.65	0.80	5.40
Twitter	4,777.29	24.88	5.21	179.02	1.13	6.31
Rome99	8,443.75	56.71	6.72	3,665.99	36.56	9.97
Gnutella25	9,749.95	66.58	6.83	302.92	1.82	6.01
Lastfm-Asia	40,162.93	142.29	3.54	997.71	6.03	6.04
NotreDame	220,197.49	590.06	2.68	15,003.47	59.36	3.96
Stanford	1,273,970.76	3,160.50	2.48	89,705.73	449.72	5.01
Epinions1	319,384.06	660.44	2.07	273.72	1.10	4.03

Table 4.8: The total time for answering 100M 2-FT-SSR queries using our implementation of Choudhary's data structure. By comparing the times/query with the times per edge access in Table 4.7, we can see that the time per 2-FT-SSR query is comparable to a few edge accesses. We report the average over 10 different random choices of CH-seeds. We note that the variance per graph is negligent.

100M CH QUERIES	Google_small	Twitter	Rome99	Gnutella25	Lastfm-Asia	NotreDame	Stanford	Epinions1
time (s)	1.170	1.412	1.988	1.535	2.277	1.477	1.766	1.532
time/query (ns)	11.699	14.116	19.883	15.345	22.768	14.770	17.663	15.325

the queries using a 2-FT-SSR oracle. That is, at every node of the decomposition tree, we simply used biBFS in order to determine whether the split vertex that corresponds to it reaches the query vertices. In this way, we can report the percentage of the queries that can be answered without resorting to BFS. As we can see in Table 4.9, more than 90% of those bad instances can be answered without performing BFS, by using only the data structures on the decomposition tree (that has height at most 10).

Table 4.9: Simulation for answering 10K queries with `ChBFS` and *ChTree* using 10 seeds that have high chance to give rise to a bad instance. This experiment was repeated for 100 different selections of seeds. We see that *ChTree* can answer at least 90% of those instances without resorting to BFS.

Graph	avg # edges ChBFS	avg # edges ChTree	% of answer by seed in ChBFS	% of answer by seed in ChTree
Google_small	145.06	5.88	0.07	95.37
Twitter	179.26	4.62	0.07	96.06
Gnutella25	302.91	0.86	0.02	99.47
Lastfm-Asia	1,007.88	62.81	0.00	91.32
NotreDame	14,989.20	51.06	0.01	99.39
Stanford	89,723.23	19.38	0.01	99.96
Epinions1	273.93	1.42	0.00	99.09
Amazon0302	19,462.16	2.19	0.00	99.98

Chapter 5

Concluding Remarks

Our experiments demonstrate that *sBFS* is a remarkably good heuristic for efficiently answering 2-fault tolerant strong connectivity queries in practice and by relying on the 2-FT-SSR oracle of Choudhary [6], we can improve the accuracy of this heuristic for non-pathological queries (*ChBFS*). Moreover, the organization of the CH-seeds into a decomposition tree minimizes the likelihood of bad instances (*ChTree*). It seems that choosing the most critical nodes (MCN) [29] of a graph as split vertices, leads to an SCC-Tree decomposition with few levels since these decompose the graph quickly into SCCs and thus this is the best choice for applications as we increase the likelihood of answering the queries fast, using a few $O(1)$-time calls to the auxiliary data structures. Lastly, *partial*-SCC-Tree is mainly of theoretical interest, since the procedure for determining whether a graph is $\Delta-$good is very slow.

Bibliography

[1] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

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

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

[4] L. Georgiadis, E. Kosinas, and D. Tsokaktsis, "2-fault-tolerant strong connectivity oracles," in *Proceedings of the SIAM Symposium on Algorithm Engineering and Experiments (ALENEX 2024)*, 2024, to appear.

[5] D. Chakraborty and K. Choudhary, "New extremal bounds for reachability and strong-connectivity preservers under failures," in *ICALP*, 2020, pp. 25:1–25:20.

[6] K. Choudhary, "An Optimal Dual Fault Tolerant Reachability Oracle," in *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 55, 2016, pp. 130:1–130:13. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6265

[7] C. Demetrescu, M. Thorup, R. A. Chowdhury, and V. Ramachandran, "Oracles for distances avoiding a failed node or link," *SIAM J. on Computing*, vol. 37, no. 5, pp. 1299–1318, Jan. 2008. [Online]. Available: http://dx.doi.org/10.1137/S0097539705429847

[8] R. Duan and S. Pettie, "Dual-failure distance and connectivity oracles," in *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*,

SODA 2009, New York, NY, USA, January 4-6, 2009, C. Mathieu, Ed. SIAM, 2009, pp. 506–515.

[9] M. Henzinger, A. Lincoln, S. Neumann, and V. V. Williams, “Conditional Hardness for Sensitivity Problems,” in *8th Innovations in Theoretical Computer Science Conference (ITCS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), C. H. Papadimitriou, Ed., vol. 67. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 26:1–26:31. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/8178

[10] M. Parter and D. Peleg, “Sparse fault-tolerant BFS trees,” in *ESA*, 2013, pp. 779–790.

[11] J. van den Brand and T. Saranurak, “Sensitive distance and reachability oracles for large batch updates,” in *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, 2019, pp. 424–435.

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

[13] S. Baswana, K. Choudhary, and L. Roditty, “An efficient strongly connected components algorithm in the fault tolerant model,” *Algorithmica*, vol. 81, no. 3, pp. 967–985, 2019.

[14] D. Chakraborty, K. Chatterjee, and K. Choudhary, “Pairwise Reachability Oracles and Preservers Under Failures,” in *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bojańczyk, E. Merelli, and D. P. Woodruff, Eds., vol. 229. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 35:1–35:16. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/16376

[15] G. F. Italiano, A. Karczmarz, and N. Parotsidis, “Planar reachability under single vertex or edge failures,” in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2021, pp. 2739–2758. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611976465.163

[16] A. Abboud and V. V. Williams, "Popular conjectures imply strong lower bounds for dynamic problems," in *FOCS*, 2014, pp. 434–443.

[17] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak, "Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture," in *STOC*, 2015, pp. 21–30.

[18] P. Charalampopoulos and A. Karczmarz, "Single-source shortest paths and strong connectivity in dynamic planar graphs," in *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, ser. LIPIcs, vol. 173, 2020, pp. 31:1–31:23. [Online]. Available: https://doi.org/10.4230/LIPIcs.ESA.2020.31

[19] J. Łącki, "Improved deterministic algorithms for decremental reachability and strongly connected components," *ACM Transactions on Algorithms*, vol. 9, no. 3, p. 27, 2013.

[20] S. Chechik, T. D. Hansen, G. F. Italiano, J. Lacki, and N. Parotsidis, "Decremental single-source reachability and strongly connected components in $\tilde{o}(m\sqrt{n})$ total update time," in *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, 2016, pp. 315–324.

[21] 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*, 2017, pp. 42:1–42:15.

[22] B. A. Berendsohn and L. Kozma, "Splay trees on trees," *CoRR*, vol. abs/2010.00931, 2020. [Online]. Available: https://arxiv.org/abs/2010.00931

[23] P. Bose, J. Cardinal, J. Iacono, G. Koumoutsos, and S. Langerman, "Competitive online search trees on trees," 2019.

[24] A. Pothen, *The Complexity of Optimal Elimination Trees*, ser. Technical report. Pennsylvania State University, Department of Computer Science, 1988. [Online]. Available: https://books.google.gr/books?id=PfyjtgAACAAJ

[25] N. Robertson and P. Seymour, "Graph minors. ii. algorithmic aspects of tree-width," *Journal of Algorithms*, vol. 7, no. 3, pp. 309–322, 1986. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0196677486900234

[26] P. Boldi, M. Rosa, and S. Vigna, "Robustness of social and web graphs to node removal," *Social Network Analysis and Mining*, vol. 3, no. 4, pp. 829–842, 2013.

[27] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, p. 036106, 2007.

[28] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[29] N. Paudel, L. Georgiadis, and G. F. Italiano, "Computing critical nodes in directed graphs," *ACM Journal of Experimental Algorithmics*, vol. 23, no. 2, pp. 2.2:1–2.2:24, Jul. 2018. [Online]. Available: http://doi.acm.org/10.1145/3228332

[30] R. E. Tarjan, "Edge-disjoint spanning trees and depth-first search," *Acta Informatica*, vol. 6, no. 2, pp. 171–85, 1976.

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

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

[33] K. Hanauer, M. Henzinger, and C. Schulz, "Faster Fully Dynamic Transitive Closure in Practice," in *18th International Symposium on Experimental Algorithms (SEA 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Faro and D. Cantone, Eds., vol. 160. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 14:1–14:14. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12088

[34] L. Georgiadis and R. E. Tarjan, "Dominator tree certification and divergent spanning trees," *ACM Transactions on Algorithms*, vol. 12, no. 1, pp. 11:1–11:42, Nov. 2015. [Online]. Available: http://doi.acm.org/10.1145/2764913

[35] E. D. Demaine, G. M. Landau, and O. Weimann, "On cartesian trees and range minimum queries," *Algorithmica*, vol. 68, p. 610–625, 2014.

[36] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *Journal of Computer and System Sciences*, vol. 26, pp. 362–391, 1983.

[37] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs," *SIAM Journal on Applied Mathematics*, vol. 36, no. 2, pp. 177–189, 1979.

[38] B. A. Reed, "Finding approximate separators and computing tree width quickly," in *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, S. R. Kosaraju, M. Fellows, A. Wigderson, and J. A. Ellis, Eds. ACM, 1992, pp. 221–228. [Online]. Available: https://doi.org/10.1145/129712.129734

[39] L. Georgiadis, G. F. Italiano, and N. Parotsidis, "Strong connectivity in directed graphs under failures, with applications," in *SODA*, 2017, pp. 1880–1899.

[40] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '05. USA: Society for Industrial and Applied Mathematics, 2005, p. 156–165.

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

[42] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," 2009.

Appendix A

Additional Experimental Results

A.1 Random graphs experiments

A.2 Worst-case queries decomposition tree

A.3 Relative performance of the BFS-based algorithms

A.4 Construction time of the SCC-Tree

A.1 Random graphs experiments

Here, we present some additional results regarding the decomposition tree height for some randomly generated strongly connected graphs described in Table A.1.

The kronecker graph belongs to the family of Kronecker graphs. From the work of Leskovec et al. [42] it is shown that these artificial graphs obey common real-network properties and that they can efficiently represent one. Additionally, they provide a fast method for generating such graphs.

The graphs random_1 and random_2 were generated as follows. Firstly, we predefined the number of the vertices of the graph and a fixed probability p. Then until the graph was strongly connected we performed Bernoulli trials by comparing a randomly generated value with the fixed probability p and added an arbitrary edge, (u, v), if the trial succeeded.

Lastly, for the generation of the two planar graphs planar_1 and planar_2 we performed the following. Firstly, we predefined the number, n, of the vertices of the graph. Then we generated n random, non-intersecting points in the plane and

constructed a Voronoi diagram with these points as input. In this diagram, every point lies in a region which does not intersect with other regions, except only with its borders. We used the n points as the vertices of the graph and the common neighbouring border between two regions as the edge that connects the corresponding vertices that lie in those regions.

For the graphs random_1, random_2, planar_1 and planar_2, we observe that, in contrast to the real graphs of Table 4.1, few vertices participates in a proper separation pair or are a strong articulation point. For planar_1 and planar_2 this is somewhat expected since they are undirected graphs. For the Kronecker graph, we see that a significant amount of its vertices is a SAP or belong to a proper separation pair.

In Table A.2 the heights of the corresponding SCC-Trees are presented and they are plotted in Figure A.1.

As in Section 4.2 we observe that for all the graphs *MCN* and *qSep+MCN* outperform all the other methods. In particular, on average, *MCN* outperforms *Random*, *LP* and *PR* by a factor of 1.6, 1.4 and 1.6 respectively. Moreover, *qSep+MCN* method, outperforms *MCN* by a factor of 3.1. Regarding the results of *LNT*, we observe that every other method performed better, on average, than it. Specifically, *LNT* produced a decomposition tree height that was larger by a factor of 1.8 compared to *MCN*, 3.1 compared to *qSep+MCN*, 1.5 compared to *LP* and 1.1 compare to *PR*.

We note that in contrast to the real graphs of Table 4.1, here, the methods perform quite similar to each-other. However, the one that stands-out is *qSep+MCN*, which produced the best results for the graphs planar_1 and planar_2.

Lastly, in Tables A.3 and A.4 we provide the results of answering 1M random queries for the oracle of Section 3.1 and we compare it with various algorithms such as *DFS*, *simpleBFS*, *biBFS*, *sBFS(10)*, *ChBFS(3)*.

A.2 Worst-case queries decomposition tree

Table A.5 presents some results for queries that elicit worst-case response times for the SCC-Tree oracle. We create each query $2\mathit{FTSC}(x, y, f_1, f_2)$ by selecting the vertices x, y such that the depth of $N(t) = \mathit{nca}(x, y)$ is large and the failed vertices f_1, f_2 are located in S_t. In the last column of Table A.5 we also give the average query times achieved by biBFS, which are remarkably good. Indeed, in most of these pathological

Table A.1: Randomly generated strongly connected graphs. n and m are the numbers of vertices and edges, respectively, n_a is the number of strong articulation points (SAPs), and n_{sp} is the number of vertices that are SAPs or belong to a proper separation pair

Graph	Type	n	m	n_a	n_{sp}
kronecker	Directed	729	2,398	285	561
random_1	Directed	1,000	6,476	17	131
random_2	Directed	4,998	46,479	8	65
planar_1	Undirected	1,896	5,588	4	159
planar_2	Undirected	19,646	58,586	12	561

Table A.2: SCC-Tree height of the graphs of Table A.1, resulting from the split vertex selection algorithms of Table 2.1.

Graph	*Random*	*MCN*	*LP*	*PR*	*qSep+MCN*	*LNT*
kronecker	294	99	241	216	102	260
random_1	698	461	634	607	471	695
random_2	3,930	2,949	3,823	3,525	2,989	3,915
planar_1	565	540	404	721	131	1,040
planar_2	5,570	4,396	4,469	8,915	506	7,331

queries for SCC-Tree, biBFS needs to explore very few edges to provide an answer.

A.3 Relative performance of the BFS-based algorithms

Tables A.6, A.7 and A.8 presents the relative performance of the BFS-based algorithms for the rest graphs excluding Amazon0302 and USA-road-NY, since due to their size, the initialization of the 2-FT-SSR-O's were too time consuming. However, based on the following results, we strongly believe that even for these two graphs the results will be analogous.

As in Section 4.3, here, the results are similar. *biBFS* outperforms *simpleBFS* while the seeded variants *sBFS* and *ChBFS* are far superior. By increasing the number of the seeds, the probability of answering the query without traversing any edges also increases dramatically.

Figure A.1: Relative SCC-Tree height of the graphs of Table A.2 w.r.t. to the number of vertices, resulting from the split vertex selection algorithms of Table 2.1.

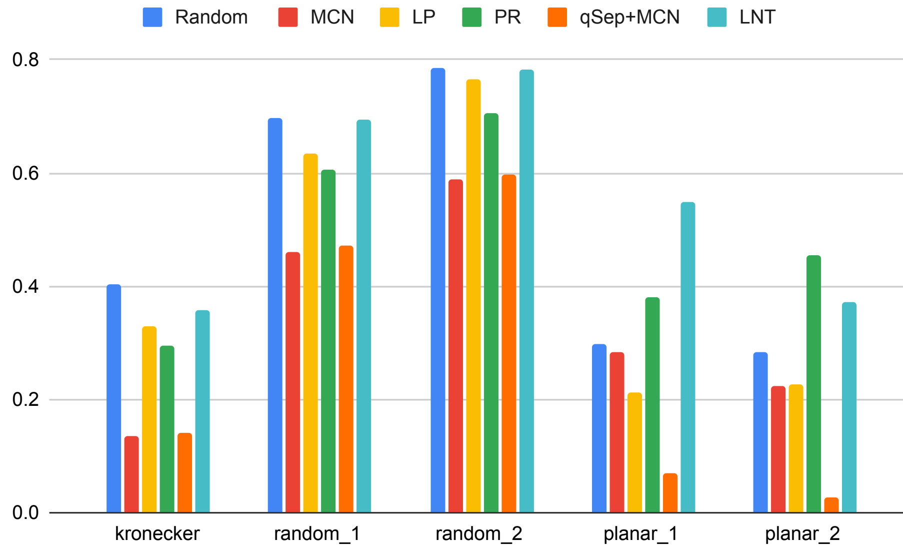


Table A.3: Results for 1M random queries using the SCC-Tree with split vertices selected by MCN.

Graph	tree depth	query depth			query time		query result		avg. calls	
		min	max	avg.	total (s)	avg. (s)	+	−	2-FT-SSR-O	1-FT-SC-O
kronecker	99	0	44	0.0027	6.40e-02	6.40e-08	996,293	3,707	3.978	0.0025
random_1	461	0	1	0.002	6.12e-02	6.12e-08	999,927	73	3.992	0.002025
random_2	2,949	0	1	0.0004	8.97e-02	8.97e-08	999,999	1	3.998	0.000427
planar_1	540	0	1	0.001	1.20e-01	1.20e-07	999,994	6	3.995	0.001016
planar_2	4,396	0	1	0.0001	1.24e-01	1.24e-07	1,000,000	0	3.999	0.000117

A.4 Construction time of the SCC-Tree

In Table A.9 are shown the time in seconds for constructing the SCC-Tree for the different heuristics used. During the decomposition we did not initialize the auxiliary data structures from 2.2.

It is obvious that the *MCN* heuristic used selecting the split vertices not only produces the best results regarding tree-height but also in most experiments is the fastest. For the *qSep+MCN* heuristic the results are similar to *MCN* except for some instances (Rome99, USA-road-NY, planar_1, planar_2) in which is remarkable faster. Note that for these exactly graphs, *qSep+MCN* provided the best decomposition tree as previously mention in Tables 4.2 and A.2.

Table A.4: Comparing running times (avg. (s) per query after 1M random queries) for various algorithms

Graph	*2-FT-SCO*	*DFS*	*simpleBFS*	*biBFS*	*sBFS(10)*	*ChBFS(3)*
kronecker	6.40e-08	2.06e-05	8.53e-06	5.51e-07	4.57e-08	7.08e-08
random_1	6.12e-08	3.42e-05	1.01e-05	5.24e-07	4.37e-08	7.60e-08
random_2	8.97e-08	2.07e-04	4.94e-05	1.01e-06	4.54e-08	6.60e-08
planar_1	1.20e-07	3.57e-05	2.82e-05	2.10e-05	4.06e-08	1.45e-07
planar_2	1.24e-07	3.94e-04	3.07e-04	2.33e-04	5.90e-08	1.22e-07

Table A.5: Results for worst-case queries using the SCC-Tree with split vertices selected by MCN.

Graph	number of	tree	query depth			query time		query result		avg. calls		biBFS avg.
	queries	depth	min	max	avg.	total (s)	avg. (s)	+	−	2FT-SSR-O	1-FT-SC-O	query time (s)
Google_small	107	9	5	6	5	2.80e-5	2.62e-7	35	72	16.67	0.018	4.67e-8
Twitter	5,168	232	158	231	178	1.91e-1	3.70e-5	950	4,218	709.34	0.003	2.32e-8
Rome99	3,641	542	401	542	464	8.98e-1	2.47e-4	89	3,552	1,855.16	0.002	1.59e-8
Gnutella25	11,688	838	609	838	712	3.49e+0	2.99e-4	0	11,688	2,848.92	0.000	2.52e-8
Lastfm-Asia	1,116	2,202	1,570	2,184	1,824	1.36e+0	1.22e-3	5	1,111	7,290.80	0.001	3.32e-8
NotreDame	100,000	365	327	361	337	1.04e+1	1.04e-4	27,716	72,284	1,345.98	0.000	2.42e-8
Stanford	100,000	1,617	1,325	1,565	1,400	6.88e+1	6.88e-4	5,694	94,306	5,598.76	0.000	2.24e-8

Lastly, as expected, the *loop nesting tree* can be constructed in the shortest time however, the resulting tree may have excessive height.

Table A.6: Relative performance of the BFS-based algorithms on Twitter (left) and Gnutella25 (right).

Twitter		
Algorithm	avg # edges explored	% of answer by seed
simpleBFS	4,777.38	
biBFS	179.05	
sBFS(1)	2.39	97.89
sBFS(2)	0.48	99.39
sBFS(5)	0.17	99.68
sBFS(10)	0.10	99.74
ChBFS(1)	0.25	99.86
ChBFS(2)	0.00	100.00
ChBFS(5)	0.00	100.00
ChBFS(10)	0.00	100.00

Gnutella25		
Algorithm	avg # edges explored	% of answer by seed
simpleBFS	9,747.45	
biBFS	302.88	
sBFS(1)	1.52	99.07
sBFS(2)	0.18	99.84
sBFS(5)	0.04	99.93
sBFS(10)	0.02	99.94
ChBFS(1)	0.24	99.93
ChBFS(2)	0.00	100.00
ChBFS(5)	0.00	100.00
ChBFS(10)	0.00	100.00

Table A.7: Relative performance of the BFS-based algorithms on Lastfm-asia (left) and NotreDame (right).

Lasfm-Asia		
Algorithm	avg # edges explored	% of answer by seed
simpleBFS	40,131.40	
biBFS	997.50	
sBFS(1)	2.09	99.75
sBFS(2)	0.29	99.95
sBFS(5)	0.12	99.97
sBFS(10)	0.06	99.98
ChBFS(1)	0.26	99.97
ChBFS(2)	0.00	100.00
ChBFS(5)	0.00	100.00
ChBFS(10)	0.00	100.00

NotreDame		
Algorithm	avg # edges explored	% of answer by seed
simpleBFS	220,350.71	
biBFS	14,929.37	
sBFS(1)	13.13	99.87
sBFS(2)	6.16	99.94
sBFS(5)	4.85	99.95
sBFS(10)	3.71	99.96
ChBFS(1)	2.17	99.99
ChBFS(2)	0.00	100.00
ChBFS(5)	0.00	100.00
ChBFS(10)	0.00	100.00

Table A.8: Relative performance of the BFS-based algorithms on Stanford (left) and Epinions1 (right).

Stanford		
Algorithm	avg # edges explored	% of answer by seed
simpleBFS	1,267,171.57	
biBFS	89,474.38	
sBFS(1)	28.14	99.96
sBFS(2)	13.67	99.98
sBFS(5)	10.87	99.98
sBFS(10)	9.31	99.98
ChBFS(1)	3.67	100.00
ChBFS(2)	0.01	100.00
ChBFS(5)	0.00	100.00
ChBFS(10)	0.00	100.00

Epinions1		
Algorithm	avg # edges explored	% of answer by seed
simpleBFS	320,355.93	
biBFS	273.78	
sBFS(1)	0.13	99.93
sBFS(2)	0.02	99.98
sBFS(5)	0.01	99.99
sBFS(10)	0.01	99.99
ChBFS(1)	0.03	99.99
ChBFS(2)	0.00	100.00
ChBFS(5)	0.00	100.00
ChBFS(10)	0.00	100.00

Table A.9: Time in seconds for constructing SCC-Tree $\mathcal{T}$ using the heuristics from Table 2.1. The symbols † and ‡ refer to decompositions that were not completed due to exceeding the RAM memory of our system (> 128GB) or due to requiring more than 48 hours.

Graph	*Random*	*MCN*	*LP*	*PR*	*qSep+MCN*	*LNT*
Google_small	0.015	0.007	0.007	0.070	0.007	0.0035
Twitter	0.007	0.154	0.408	0.209	0.160	0.0018
Rome99	0.152	0.621	0.825	1.305	0.343	0.0020
Gnutella25	0.610	1.732	14.950	2.499	1.820	0.0035
Lastfm-asia	2.654	6.582	8.539	12.430	7.093	0.0043
Epinions1	204.290	102.872	239.00	113.00	106.920	0.0235
NotreDame	31.110	3.201	113.750	8.470	3.286	0.0068
Stanford	721.030	48.00	1,577.00	1,137.00	50.020	0.0635
Amazon0302	†	1,697.00	‡	‡	1,787.810	0.1092
USA-road-NY	†	1,190.00	‡	‡	19.300	0.0392
kronecker	0.023	0.047	0.141	0.063	0.057	0.0006
random_1	0.065	0.248	0.442	0.230	0.241	0.0013
random_2	1.682	9.344	14.311	7.495	9.900	0.0047
planar_1	0.070	0.330	0.283	0.412	0.021	0.0011
planar_2	6.617	29.830	43.690	43.393	0.306	0.0048

Short Biography

Daniel Tsokaktsis was born in Arta, Greece, in 1998. In October 2016 he enrolled in the undergraduate program of Mathematics of the University of Ioannina and graduated in February 2021. In October 2021 he enrolled in the post-graduate program of the Department of Computer Science and Engineering of University of Ioannina. His research interests focus on Graph Theory, Algorithms and Data Structures as well as Data Mining.