Scalable Algorithms for the Analysis of Massive Networks

DISSERTATION

zur Erlangung des akademischen Grades
Doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von
Eugenio Angriman

Präsidentin/Präsident der Humboldt-Universität zu Berlin
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekanin/Dekan der Mathematisch-Naturwissenschaftlichen Fakultät
Prof. Dr. Elmar Kulke

Gutachter/innen
1. Prof. Dr Henning Meyerhenke
2. Prof. Dr. Giuseppe Francesco Italiano
3. Prof. Dr. Ulrik Brandes

Tag der mündlichen Prüfung: 14 Dezember 2021

Declaration of Independent Work

I declare that I have completed the thesis independently using only the aids and tools specified. I have not applied for a doctor's degree in the doctoral subject elsewhere and do not hold a corresponding doctor's degree. I have taken due note of the Faculty of Mathematics and Natural Sciences PhD Regulations, published in the Official Gazette of Humboldt-Universität zu Berlin no. 42/2018 on 11/07/2018.

Berlin, 2021

. . . . . .

Eugenio Angriman

Acknowledgments

I want to dedicate a few words to everyone who helped me write this thesis.

First and foremost, my supervisor Henning Meyerhenke: he offered me the opportunity to work in his research group and, even in rough times, was never too busy to provide support and guidance during my PhD. I also appreciate his trust, his efforts in keeping the group united and collaborative, and for the time he spent to give me valuable advice. I would also like to thank Giuseppe Francesco (Pino) Italiano and Ulrik Brandes for accepting to act as reviewers of my thesis and taking the time to read it as well as Jan Mendling, Robert Bredereck, and Patrick Schäfer for agreeing to be part of the PhD committee.

A great thank you goes to past and present colleagues I was fortunate to work with. In particular, Elisabetta Bergamini: even though we just exchanged a few emails about NetworKit while I was working on my Master's thesis, her enthusiasm in her work encouraged me to join Henning's group. Also, thank you for the great tips about life in Germany! I am very grateful to Charilaos (Harry) Tzovas for his hospitality during my first days in Karlsruhe, for helping when we moved to Berlin, and for his contagious good mood. Thank you also to Moritz von Looz for helping me since the beginning of my PhD, for the intriguing discussions and the board game evenings. Thanks to Roland Glantz for your help during my months in Karlsruhe and for lending me a nice white shirt which made me look (slightly) more professional during the group photo day – which I forgot about. I also thank Maria Predari, the most competitive foosball player I have ever met – too bad that we did not buy a foosball table after we moved to Berlin, but it probably would have made us miss a few deadlines. A special thank you goes to Alexander van der Grinten for the time he spent listening to my questions and for his guidance that made me a better researcher and programmer. My gratitude also goes to Fabian Brandt-Tumescheit for always being available to solve any kind of technical issues (for some of which I am directly responsible), for the great support he gave us to carry on and promote our research work, and for helping me translate the abstract of this thesis together with Alexander.

I wish to thank all my coauthors and collaborators enjoyed working with: Martin Nöllenburg, Aleksandar Bojchevski, Daniel Zügner, Stephan Günnemann, Christian Schulz, Bora Uçar, Ruben Becker, Gianlorenzo D'Angelo, Hugo Gilbert, Klaus Ahrens, and Michał Boroń. Thank you to all my colleagues from the DFG SPP project "Algorithms for BIG DATA", in particular, Michael Hamann, Manuel Penschuck, Timo Bingmann, Demian Hespe, Lorenz Hübschle-Schneider. Thank you for the interesting discussions and the fun we had at conferences around Germany and in India.

I am also thankful to my friends Davide, Giuseppe, Gianluca, Emanuele, Massimo, Giacomo, Elia, Stefano, Federico, and Nicola for being there. I am glad that, despite many of us are living abroad, we still manage to see each other and have fun together. For their great support, I thank my parents Cecilia and Imerio, my parents-in-law Katya and Alberto, my siblings Sofia and Daniele, and my siblings-in-law Davide and Arianna.

Last but not least, I cannot describe in words my gratitude to my girlfriend Francesca. Since the beginning of this journey, you always supported me with your love, patience, and joy. Despite the distance that separated us during these years, you have always been there for me. Thank you.

Zusammenfassung

Die Netzwerkanalyse ist eine Sammlung von Techniken, die darauf abzielen, nicht-triviale Erkenntnisse aus vernetzten Daten durch die Untersuchung von Beziehungsmustern zwischen den Entitäten eines Netzwerks zu gewinnen. Zu den Erkenntnissen, die aus einem Netzwerk gewonnen werden können, gehört die Bestimmung der Wichtigkeit von Entitäten an Hand von definierten Kriterien – in sozialen Netzwerken gibt es beispielsweise in der Regel einige Teilnehmer, die einflussreicher sind als andere oder deren Entfernung aus dem Netzwerk eine erhebliche Veränderung des Kommunikationsflusses bedeuten würde. Eine andere Möglichkeit besteht darin, für jeden Teilnehmer eines Netzwerks den am besten geeigneten Partner zu finden, wenn man die paarweisen Präferenzen (oder die Kompatibilität) der Teilnehmer kennt, die miteinander verbunden werden sollen – auch bekannt als das Maximum Weighted Matching-Problem (MWM). Beispiele hierfür sind Plattformen für Matchmaking oder die Paarung von Spielern in einem Schachturnier.

Die Wichtigkeit ist hierbei stark an die jeweilige Anwendung gebunden. Daher wurden in den letzten Jahren mehrere *Zentralitätsmaße* eingeführt. Diese Maße stammen hierbei aus Jahrzehnten, in denen die Rechenleistung sehr begrenzt war und die Netzwerke im Vergleich zu heute viel kleiner waren – Skalierbarkeit auf große Datenmengen wurde daher nicht berücksichtigt. Heutzutage sind jedoch Netzwerke mit Millionen von Kanten allgegenwärtig und eine vollständige exakte Berechnung vieler traditioneller Zentralitätsmaße – die immer noch weit verbreitet sind – ist zu zeitaufwendig. Dieses Problem wird noch verstärkt, wenn das Ziel darin besteht, die *Gruppe* der k-Knoten mit der höchsten gemeinsamen Zentralität zu finden; dieses Problem hat nützliche Anwendungen bei der Optimierung von Standorten von Lagern und der Einflussmaximierung. Skalierbare Algorithmen zur Identifizierung hochzentraler (Gruppen von) Knoten in großen Graphen sind von zentraler Bedeutung für eine umfassende Netzwerkanalyse. Die heutigen Netzwerke sind nicht nur groß, sondern verändern sich zusätzlich im zeitlichen Verlauf. Daraus ergibt sich die Herausforderung, die Erkenntnisse, die aus dem Netzwerk gewonnen wurden, nach einer Änderung effizient zu aktualisieren. Effiziente *dynamische* Algorithmen sind daher ein weiterer wesentlicher Bestandteil moderner Analyse-Pipelines.

Hauptziel dieser Arbeit ist es, skalierbare algorithmische Lösungen für die zwei bereits genannten Herausforderungen zu liefern: die Identifizierung wichtiger Knotenpunkte in einem Netzwerk und deren effiziente Aktualisierung in sich verändernden Netzwerken. Die meisten unserer Algorithmen benötigen Sekunden bis einige Minuten, um diese Aufgaben in realen Netzwerken mit bis zu Hunderten Millionen von Kanten zu Lösen, was eine deutliche Verbesserung gegenüber dem Stand der Technik darstellt.

Die Berechnung von MWMs in großen Netzwerke ist rechenintensiv. Aus diesem Grund gibt es in der Literatur zahlreiche schnelle inexakte Algorithmen für MWM, sowie dynamische Algorithmen zur Aufrechterhaltung eines (approximativen) MWMs in dynamischen Graphen. Es wurden jedoch nur wenige Anstrengungen unternommen, um diese Algorithmen in der Praxis zu implementieren, insbesondere im dynamischen Fall, so dass ihre tatsächliche Leistung unbekannt ist. Daher besteht ein weiteres Ziel dieser Arbeit darin, die Lücke zwischen Theorie und Praxis im Zusammenhang mit dynamischen MWM zu schließen. Insbesondere entwickeln wir einen Algorithmus, der ein approximatives MWM nach mehreren Kantenaktualisierungen von Graphen mit Milliarden von Kanten in nur einem Bruchteil einer Sekunde aktualisiert.

Abstract

Network analysis is a collection of techniques aimed to unveil non-trivial insights from networked data by studying relationship patterns between the entities of a network. Among the insights that can be extracted from a network, a popular one is to quantify how important an entity is with respect to the others according to some importance criteria – e.g., social networks often have participants that are more influential than others or whose removal from the network would imply a major disruption of the communication flow. Another one is to find the most suitable matching partner for each participant of a network knowing the pairwise preferences (or the compatibility) of the participants to be matched with each other – which can be formalized as the well-known maximum weighted matching problem, or MWM. Think of matchmaking platforms or pairing players in a chess tournament.

Because the notion of importance is tied to the application under consideration, numerous *centrality measures* have been introduced. However, many of these measures were conceived in a time when computing power was very limited and networks were much smaller compared to today's, and thus scalability to large datasets was not considered. Today, massive networks with millions of edges are ubiquitous and a complete exact computation for traditional centrality measures – still widely used to extract meaningful information from large datasets – often requires an excessive amount of time. This issue is amplified if our objective is to find the *group* of k vertices that is the most central *as a group*, which is useful to applications such as optimizing warehouse locations or influence maximization. Scalable algorithms to identify highly central (groups of) vertices on massive graphs are thus of pivotal importance for large-scale network analysis. In addition to their size, today's networks often evolve over time and this poses the challenge of efficiently updating the findings we gathered on the network after a change occurs. Hence, efficient *dynamic* algorithms are essential for modern network analysis pipelines.

In this work, we propose scalable algorithms for the two aforementioned challenges, namely: identifying important vertices in a network and efficiently updating them in evolving networks. In real-world graphs with up to hundreds of millions of edges, most of our algorithms require seconds to a few minutes to perform these tasks, improving significantly over the state of the art.

Concerning MWM, solving this problem optimally in large graphs is computationally too expensive, and fast inexact algorithms for MWM are abundant in the literature as well as dynamic algorithms to maintain an (approximate) MWM. In the dynamic case, however, little effort was devoted to actually implementing these algorithms in practice, leaving their actual performance unknown. In this work, we extend a state-of-the-art approximation algorithm for MWM to dynamic graphs; experiments show that our dynamic MWM algorithm handles multiple edge updates in graphs with billion edges in only a fraction of a second.

Contents

List of Figures

List of Tables

Part I

Introduction

1 Introduction

1.1 Context: Network Analysis

Most complex systems and phenomena are amenable to be modeled as graphs as they consist of entities interacting with each other. Think of the network of computers exchanging data between them that form the Internet, the (virtual) friendships among the users of a social network, the power grids connecting generating facilities to electrical substations, or the biochemical interactions happening in a biological cell. Often, the analysis of the patterns of relationships of a network (also known as network analysis) reveals interesting insights about the underlying system [25, 177, 277, 295]. Today, network analysis is an established scientific field aiming to unveil non-trivial properties of complex systems by analyzing the structure of their interconnections.

Network analysis emerged from graph theory, whose roots date back in the early 18th Century when the Seven Bridges of Königsberg problem[1] was modeled as a graph and resolved (negatively) by the Swiss mathematician Leonhard Euler. Graph theory continued to develop and started to be applied to other areas such as chemistry or social sciences. In particular, thanks to the great interest that graph theory sparked among social scientists, the 20th Century witnessed the development of Social Network Analysis (SNA), a discipline that investigates social ties between individuals through the use of graphs.

The idea of looking at society in terms of interconnections among social actors was first conceived by the French philosopher Auguste Comte [43, p. 14]. Only in the 1930s, however, the group led by the Romanian-American psychologist Jacob Levy Moreno developed the first *sociograms* [220] (i.e., graphic representations of social ties between people) as well as an approach that includes all the defining properties of social network analysis. As argued by Freeman: "It was based on structural intuitions, it involved the collection of systematic empirical data, graphic imagery was an integral part of its tools and it embodied an explicit mathematical model." [43, p. 39].

In this thesis, we address two popular problems that, among others, immediately found applications in the context of social networks and were later applied to many other fields beyond SNA [233]: (i) identifying the individuals that are the most "important" according to some criterion and (ii) matching each individual (or as many as possible) with the best suitable candidate – also known as the maximum (weighted) matching problem.

Centrality measures – i.e., functions that assign to each vertex (or edge) of a network an importance score – turned out to be a successful technique to identify important individuals in a social network. Centrality stems from the concept of *centralization*, devised for the first time in the late 1940s thanks to experiments aimed to study communication patterns and group collaboration conducted at MIT by Alex Bavelas's

[1] A recreational activity for the residents of Königsberg was to determine whether one could cross all the seven bridges of the city exactly once and, if possible, return to their starting point.

group [33]. Results suggested that centralization is related to a group's efficiency in terms of problem-solving capabilities [279, p. 45] and agreed with the perception of influence of each individual actor [43, p. 68]. These experiments led to the definition of *closeness centrality*[2] in 1950 [34] and, in the following decades, to the development of many other centrality indices that have been employed in a multitude of applications far beyond SNA. Examples include the control of epidemic spreading [93], the study of the political integration of Indian social life [81] or even the analysis of the centralization of political parties and elite networks in the 15th Century Florence to explain the power of the Medici family [3].

The maximum weighted matching problem (or MWM), in turn, derives from the well-studied Stable Roommates Problem (SRP). SRP was first introduced in the seminal work by Gale and Shapley in 1962 as a variation of the famous Stable Marriage Problem [115] and appears in numerous practical applications [149]. In its most basic form, SRP takes as input a set of participants, with each one having ranked the others in order of preference. The desired output is a *stable* matching (i.e., a separation of the set into disjoint pairs of "roommates"); a matching is stable if there are no two participants that prefer to be matched together rather than with their current matching partner. Among all the possible stable matchings in a network, however, some applications require to find the one with maximum weight,[3] hence the maximum weighted matching problem. Examples include information retrieval [176], pattern recognition [82], multilevel graph partitioning [219], and many others [58, 138, 237, 283]. As of today, MWM is one of the most popular combinatorial optimization problems [49, 154, 201].

1.2 Motivation

Centrality in Large-Scale Networks. Traditionally, centrality measures have been applied to graphs with relatively small size. In the last decades, however, factors such as the increasing computing power of modern machines, the rapid growth of the Internet and of online social networks, automatic data collection and many others catalyzed the rapid growth (both in size and quantity) of graph datasets. Hence, massive networks with millions to billions of vertices and edges are now ubiquitous. Computing (even in approximation) traditional centrality measures in such large graphs is challenging as many of them inherently have a high computational complexity or are averse to approximation schemes or require operations that are not amenable to parallelization [156].

A goal of this work is to develop *scalable* algorithms for centrality computation in large real-world networks. The term "scalability" refers here to the capability of an algorithm to process massive data sets in reasonable time. Usually, this implies that the time complexity is (nearly-)linear in the input size. We do not refer in this context to *parallel* scalability (i.e., the capability of an algorithm to use multiple processors to execute faster), although in some cases we exploit shared-memory or even distributed-memory parallelism to speed our algorithms up. Our algorithms achieve scalability through three strategies: *approximation* (i.e., yielding inexact results with bounded errors), *heuristics* (i.e., yielding inexact solutions without error bounds), and by devising and computing new centrality measures that are more amenable to scalability than traditional ones.

[2] Closeness is one of the oldest centrality measures; it is defined as the reciprocal of farness, i.e., the sum of shortest-path distances from an actor to all the others.

[3] The weight of a matching M is defined as the sum of the edge weights in M.

Chapter 5 provides an example of approximation: we present a fast algorithm to approximate electrical closeness as well as other electrical centrality measures. Because computing electrical closeness exactly is prohibitive on large graphs, we introduce a new sampling-based approach that yields results with higher quality (in terms of maximum absolute error) and is faster compared to state-of-the-art algorithms. This enables for the first time the possibility to approximate electrical closeness on graphs with hundreds of millions of vertices in minutes.

Concerning heuristics, Chapter 6 introduces new local search algorithms for group-closeness maximization, i.e., the problem of finding a set of k vertices with highest group-closeness centrality.[4] This problem is $\mathcal{NP}$-hard and there is no algorithm known to solve it exactly in reasonable time on graphs with more than a few thousands of vertices. In a matter of seconds, our heuristics compute high-quality solutions on networks with hundreds of millions of vertices – instead of several hours with existing algorithms.

Because shortest-path based measures are inherently expensive to compute, we propose, in Chapter 8, an alternative (algebraic) group centrality measure inspired by Katz centrality (see Eq. (2.10)) that is based on *walks* of any length instead of shortest paths. Our experiments suggest that not only our new measure can be maximized (in approximation) faster than existing group centralities, but it also improves the precision of popular graph mining tasks to a greater extent than existing measures.

Dynamic Graphs. Networks often evolve over time, creating new connections or deleting existing ones. Changes can happen very frequently: consider the frenetic activity on social media platforms or the emails that are sent every second.[5] For these *dynamic* networks, analyzing each snapshot independently (e.g., by running a static algorithm after every single update) is clearly an inefficient choice, especially for rapidly-changing networks. This raises the need for more effective approaches tailored to dynamic graphs.

Thus, another objective of this work is to design scalable dynamic algorithms capable of efficiently recomputing the required information after one or multiple changes occur. In particular, we develop algorithms to efficiently maintain in a dynamic graph: (i) the ranking of the top-k vertices with highest closeness centrality (Chapter 3) and a $(1/2)$-approximation of the MWM (Chapter 9).

1.3 Methodology

Algorithm Engineering. The algorithmic contributions presented in this thesis were developed following the *algorithm engineering* paradigm. Algorithm engineering can be summarized as a feedback loop of five iterative phases: (i) modeling the problem, which usually stems from practical applications, (ii) designing an algorithm, (iii) analyzing it theoretically, (iv) implementing it, and (v) evaluating it via systematic experiments (i.e., experimental algorithmics), see Figure 1.1 for an overview. The process is cyclic rather than sequential because experimental results shall unveil insights about that problem that lead to further (theoretical) improvements of the algorithms. According to this paradigm, the implementation and the experimental evaluation of an algorithm are not left to practitioners but are part of the whole development process. Hence, all the algorithms presented in this thesis are implemented in practice and evaluated on

[4] Group-closeness is a set function that measures the centrality of a group $S \subset V$ of vertices according to the average shortest-path distance from S to the vertices outside S. See Eq. (2.1) for the distance between a vertex and a set of vertices.

[5] See `https://www.internetlivestats.com`.

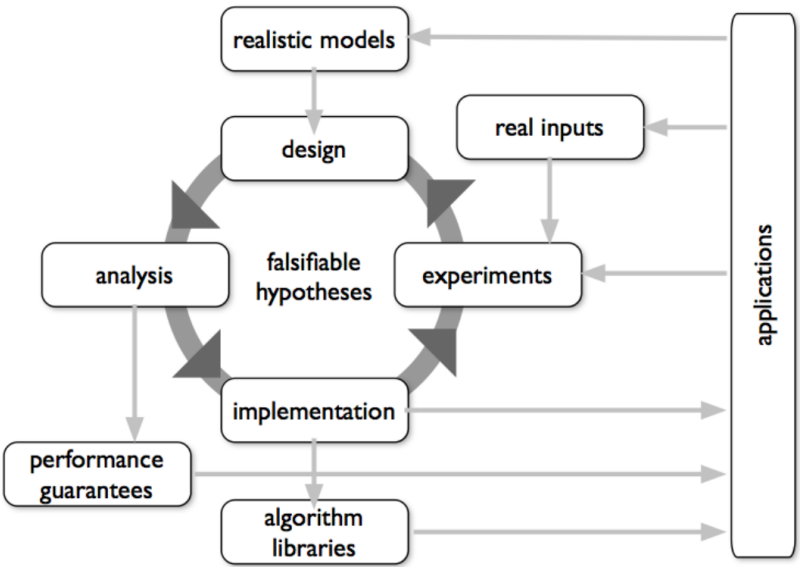


Figure 1.1: Algorithm Engineering schema [255].

real-world instances; in particular, they are implemented in C++ as part of the open-source NetworKit [273] library.

Experimental Algorithmics. As described above, experimental evaluation is a crucial step of the algorithm engineering process. In particular, to effectively support our conclusions, it is essential to implement a systematic and reproducible experimental pipeline. To this end, our experiments follow, when applicable, the guidelines for experimental algorithmics presented in Ref. [14] – which I coauthored. In particular, we carefully select appropriate sets of instances (both real-world and synthetic) belonging to different classes.[6] We always obtain real-world instances from public repositories (e.g., KONECT [173], SNAP [181], and others) whereas for synthetic instances we provide details about the graph generators we used. Also, we make our code publicly available as part of the NetworKit toolkit. The reproducibility of our experiments is guaranteed by managing them with SimexPal [14], a software that automates the experimental pipeline and simplifies gathering and analyzing the results according to the guidelines [14].

1.4 Outline and Contribution

This work is structured into five parts. The first one provides a broad overview of the addressed network-analytic challenges and fundamental notation and definitions, Parts II–IV present in detail our algorithmic contributions, and Part V is devoted to concluding remarks. More precisely, Part II focuses on computing and approximating popular single-vertex centrality measures. In Part III, we consider *group-centrality*, i.e., the concept of centrality extended to groups of vertices as described by Everett and Borgatti [104]; in particular, we introduce approximation algorithms and fast heuristics for group centrality maximization.

[6] Although there are several ways to categorize a network [14], in this work, we often refer to the class of *complex* networks. As the name suggests, complex networks present non-trivial (complex) topological features, most notably: a small diameter (*small-world* effect [227]), clustering (i.e., the tendency of vertices to form densely connected clusters), and a skewed degree distribution (many vertices with low degree and a few vertices with high degree) [5]. Simple networks such as road networks or random graphs (e.g., graphs generated by the Erdős Rényi model) do not present such features.

Finally, Part IV presents a batch-dynamic algorithm that maintains a $(1/2)$-approximation of an MWM in fully-dynamic graphs.

The contributions presented in this thesis appeared in the publications reported in Appendix A; in the following, we provide a detailed overview.

Part II: Algorithms for Single-Vertex Centrality Measures. Centrality measures are widely used to quantify the importance of single vertices on the basis of their structural position in a network. Concerning closeness, Bergamini et al. [37] introduced an efficient algorithm to identify the top-k vertices with highest closeness centrality that, in practice, is much faster than computing the closeness of all vertices and extracting the top-k ranking later. On dynamic graphs, however, it would anyway be too costly to run this algorithm after each edge update (or batch of updates). Hence, in Chapter 3, we propose a batch-dynamic algorithm for updating the top-k ranking after multiple edge updates. Our algorithm is developed upon the existing static strategy by Bergamini et al. [37]. Experiments show that, for single edge updates and for batches with up to 100 edge updates, our dynamic algorithm is one to four orders of magnitude faster than the static one.

In Chapter 4, we consider the problem of implementing an effective parallelization strategy for adaptive sampling and we apply it to betweenness centrality approximation as a case study. Adaptive sampling algorithms draw random samples according to an algorithm-specific distribution and aggregate them. A *stopping condition* determines whether enough samples have been drawn – thus the name "adaptive". Because the stopping condition requires to evaluate all the data generated so far, parallelization strategies for adaptive sampling algorithms are challenging to devise. In addition, evaluating the stopping condition is often not cheap (e.g., for our betweenness case study, it is linear in the number of vertices of the graph); hence, there is a trade-off between (i) frequently checking the stopping condition, which implies additional time overhead (but avoids drawing many samples in excess) and (ii) checking the stopping condition less frequently (with a higher risk of drawing many samples in excess). We introduce a new epoch-based parallelization framework for adaptive sampling that avoids expensive synchronization costs. The main idea is to split the execution of each thread into discrete *epochs*. During an epoch, each thread draws samples; the stopping condition is checked only at the end of each epoch and always by the same thread, while the others keep drawing samples for the next epoch. In this way, threads do not idle while the stopping condition is being checked. We adapt this framework to KADABRA, the state-of-the-art algorithm for betweenness approximation. Our interest in KADABRA stems from the fact that this algorithm implements an adaptive sampling technique but it fails to scale to large numbers of threads due to high synchronization costs. We propose three algorithms that achieve different trade-offs in terms of memory footprint and determinism of the results. Furthermore, we use parameter tuning [14] to optimize the frequency of checking the stopping condition. Our experimental study shows that our framework achieves much better parallel speedups compared to a straightforward OpenMP-based parallelization strategy for KADABRA.

Finally, Chapter 5 targets the problem of approximating *electrical* centrality measures, especially electrical closeness and forest closeness. These measures interpret the underlying graph as an electrical network (with edge weights representing the resistance between two vertices) and the distance between two vertices u and v is computed as the resulting resistance between them (i.e., the *effective resistance* or resistance distance, see Section 2.3.2). Resistance distance and forest distance consider *all* the paths between u and v, not only the

shortest ones. Unfortunately, existing methods to compute electrical closeness and forest closeness exactly rely on computing the pseudoinverse $\mathbf{L}^\dagger$ of the Laplacian matrix $\mathbf{L}$ (see Section 2.1), which is prohibitive in terms of both time and memory – typically, $\mathbf{L}^\dagger$ is dense. We present a new efficient sampling-based strategy that only requires a linear amount of additional memory and exploits two main properties: (i) the resistance distance between any two vertices only requires an arbitrary column and the diagonal of $\mathbf{L}^\dagger$ (not the whole matrix) and (ii) the resistance distance of an edge e is proportional to the fraction of the spanning trees that contain e. Our algorithm provides a probabilistic $\pm\varepsilon$-approximation of $\mathrm{diag}(\mathbf{L}^\dagger)$ by solving just one Laplacian system and by sampling a fixed amount of uniform spanning trees. Experimental results show that, compared to state-of-the-art strategies, our algorithm not only is much faster and more memory-efficient, but also yields much more accurate approximation of $\mathrm{diag}(\mathbf{L}^\dagger)$ in terms of absolute error, which results in more precise complete rankings of the elements of $\mathrm{diag}(\mathbf{L}^\dagger)$.

Part III: Algorithms for Group Centrality Measures. Group centrality is useful for applications seeking sets of vertices that are central *as a group* rather than the top most individually central vertices. Examples include finding the k most influential actors in a social network to promote some product or idea [162] or placing resources among k peers in a large P2P network so that they are easily accessible by others [121].

Unfortunately, finding the most central group of k vertices is an $\mathcal{NP}$-hard problem for most group centrality measures. This leaves approximation algorithms and heuristics the only viable options for instances with more than a few thousand vertices. Yet, existing algorithms for group centrality maximization have limitations: (i) they fail to scale to large networks and, in the group-closeness case, (ii) they do not provide any approximation guarantee. Moreover, existing measures are not suitable for disconnected graphs. In this part, we propose solutions to these issues.

In Chapters 6 and 8, we target the lack of truly scalable algorithms for group centrality maximization from two different directions. More precisely, Chapter 6 focuses exclusively on group-closeness: The fastest existing algorithm for this problem is the greedy ascent heuristic by Bergamini et al. [38], which needs several hours to handle graphs with hundreds of millions of edges. We introduce a family of novel local search heuristics that require no more than a few minutes to handle even larger graphs while yielding solutions with nearly the same quality. The fact that group-closeness is based on shortest paths poses complexity-theoretic limitations to the development of scalable approximation algorithms to maximize this measure. This motivates us to approach this problem from a different angle, that is, introducing an alternative measure. Chapter 8 introduces GED-Walk (for *Group Exponentially Decaying Walk*), a novel group centrality measure inspired by Katz centrality. Similarly to Katz, it takes into account *walks* of any length – with shorter walks being more relevant than longer ones. We present algorithms to compute and maximize (in approximation) GED-Walk. On real-world networks and for groups with up to 100 vertices, our maximization algorithm for GED-Walk is up to two orders of magnitude faster than state-of-the-art greedy maximization algorithms for group-betweenness, group-closeness, and group-harmonic. Chapter 8 also targets the lack of electrical group centrality measures for disconnected graphs by introducing group forest closeness, i.e., forest closeness[7] extended to sets of vertices, and by adapting the greedy maximization algorithm by Li et al. [183] to group forest closeness. Our experiments also show that, in connected graphs, the precision of

[7] Forest closeness (see Section 2.4) is an electrical centrality measure designed to handle disconnected graphs.

popular graph mining applications can be improved by GED-Walk to a greater extent compared to other measures. Analogous results are achieved in disconnected graphs by group forest closeness.

Chapter 7 deals with approximating group-closeness maximization. Building on top of the theoretical results achieved in Ref. [12], we present the first approximation algorithm for this problem. A clear trade-off between solution quality and running time emerges from our experiments: our approximation algorithms consistently find higher quality solutions compared to existing approaches at the cost of additional running time.

Part IV: Maximum Weighted Matching in Fully-Dynamic Graphs. On dynamic graphs, updating a previously computed matching after each update (or a batch of updates) is a more efficient approach than re-running a static algorithm from scratch – similarly to what we observe in Chapter 3 for the top-k closeness centrality ranking. Despite the wide range of algorithms for dynamic maximum (weighted) matching proposed in the literature, little effort was invested into implementing these algorithms in practice and evaluating their performance on real-world instances. Only recently, implementations and experimental analyses were done by Henzinger et al. [140] for dynamic maximum cardinality matching algorithms and in Ref. [16] for dynamic MWM – which I coauthored. These algorithms efficiently update an approximate maximum (weighted) matching after single edge updates.

In Chapter 9, we consider the problem of maintaining a $(1/2)$-approximate MWM after *multiple* edge updates, as applications dealing with rapidly-evolving networks may not require to compute a solution for every single snapshot of the graph – e.g., self-organizing LTE networks [143]. We take inspiration from the state-of-the-art Suitor algorithm by Manne and Halappanavar [203]. Our dynamic strategy is conceptually similar to the one implemented for top-k closeness in Chapter 3: we use Suitor to compute an approximate MWM on an initial snapshot of the graph; then, after one or multiple edge updates, we identify the affected vertices (i.e., the ones whose matching partner needs to be updated) and update their matching partner accordingly. Although in a worst-case scenario every vertex is affected by a single update (which implies that our dynamic algorithm has the same worst-case time complexity than the static Suitor), in our experiments, we see that the actual number of affected vertices is very small – it grows at most linearly with the batch size. For single edge updates, our dynamic approach is on average faster than the state of the art [16], whereas for batches with up to 10^4 edge updates it is two to six orders of magnitude faster than a static recomputation (Ref. [16] only supports single edge updates).

2 Preliminaries

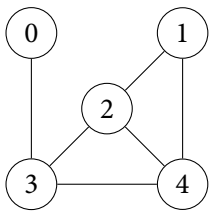
2.1 Graphs

A graph[8] is an ordered pair (V, E) where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, $n = |V|$ is the number of vertices, and $m = |E|$ is the number of edges. A graph $G = (V, E)$ is a *subgraph* of another graph $G' = (V', E')$ if G contains all the vertices and edges of G', i.e., $V \subseteq V'$ and $E \subseteq E' \cap (V \times V)$.

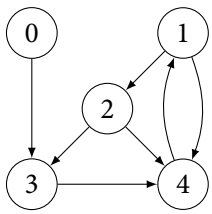
An edge $e = \{u, v\} \in E$ is *incident* to u and v and both vertices are *adjacent* to each other. The *neighborhood* of a vertex u is the set of vertices adjacent to u, i.e., $N(u) := \{v \in V \mid \{u, v\} \in E\}$. The vertices in $N(u)$ are also called the *neighbors* of u. The *degree* of a vertex u, denoted $\deg(u)$, is the cardinality of its neighborhood.

If edges are ordered sets of pairs then the graph is called *directed* and we denote by (u, v) an edge *from* a vertex u *to* a vertex v; otherwise, the graph is called *undirected* – Figure 2.1a and Figure 2.1b show an example of an undirected graph and a directed graph, respectively. In directed graphs, a vertex u has *out*-neighbors, i.e., the set of vertices $N_{\text{out}}(u) := \{v \in V \mid (u, v) \in E\}$, and *in*-neighbors, i.e., $N_{\text{in}}(u) := \{v \in V \mid (v, u) \in E\}$; the neighbors of u is the union of u's out-neighbors and in-neighbors: $N(u) := N_{\text{out}}(u) \cup N_{\text{in}}(u)$. The cardinalities $\deg_{\text{out}}(u) := |N_{\text{out}}(u)|$, $\deg_{\text{in}}(u) := |N_{\text{in}}(u)|$, and $\deg(u) := |N(u)|$ are called *out*-degree, *in*-degree, and degree of u, respectively.

A *weighted* graph is an ordered triplet (V, E, w) where (V, E) is an unweighted graph and w is the *weight* function, i.e., $w : E \to \mathbb{R}$. In weighted graphs, the degree can be extended to the sum of the weights of the (in/out) neighbors, namely the *weighted degree* and the *weighted in-* and *out-degree*. In this thesis, we only consider strictly positive edge weight functions. An unweighted graph can also be interpreted as a weighted graph with weight function $w : E \to 1$, i.e., every edge has weight 1.



(a) Example of undirected graph with $n = 5$ vertices and $m = 6$ edges.



(b) Example of directed graph with $n = 5$ vertices and $m = 7$ edges.

Figure 2.1: Examples of undirected and directed graphs.

[8] In the literature, graphs representing real-world phenomena are often called "networks" – e.g., social networks, road networks, etc. We use these two names interchangeably. Further, in the context of graphs, the names "node" and "vertex" are often used as synonyms. We use "vertex" for an element in V and we reserve "node" for computing units in a distributed memory system.

Matrix Representations of a Graph. A graph $G = (V, E, w)$ can be represented using matrices. In this work, matrices are denoted by bold capital letters such as $\mathbf{M}$ and the element at the i-th row and j-th column of $\mathbf{M}$ by $\mathbf{M}[i, j]$. Similarly, we type vectors as bold lowercase letters such as $\mathbf{v}$ and we denote the element at index i in $\mathbf{v}$ by $\mathbf{v}[i]$. The *adjacency matrix* $\mathbf{A}$ of G is a square $n \times n$ matrix such that $\mathbf{A}[u, v] = w(u, v)$ if there is an edge from u to v, zero otherwise. Clearly, if G is undirected, then $\mathbf{A}$ is symmetric. The *Laplacian matrix* $\mathbf{L}$ of G is defined as:

$$\mathbf{L} := \mathbf{D} - \mathbf{A},$$

where $\mathbf{D}$ is the *degree matrix* of G, i.e., the diagonal matrix such that $\mathbf{D}[u, u]$ is the (possibly weighted) degree of u. If G is undirected, then $\mathbf{L}$ is symmetric and positive-semidefinite. For example, the adjacency and the Laplacian matrices of the graph in Figure 2.1a are respectively:

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \quad \text{and} \quad \mathbf{L} = \begin{pmatrix} 1 & 0 & 0 & -1 & 0 \\ 0 & 2 & -1 & 0 & -1 \\ 0 & -1 & 3 & -1 & -1 \\ -1 & 0 & -1 & 3 & -1 \\ 0 & -1 & -1 & -1 & 3 \end{pmatrix}.$$

2.2 Paths and Components

Definition 2.2.1 (Walk, Trail and Path)**.** A walk on a graph $G = (V, E, w)$ is an alternating sequence of vertices $(u_0, u_1, \ldots, u_\ell)$ and edges $(e_1, e_2, \ldots, e_\ell)$ where $e_i = (u_{i-1}, u_i) \in E$ for all $0 < i \leq \ell$. The number ℓ of edges in the walk is called the *length* of the walk. A *trail* is a walk where all edges are distinct and a *path* is a walk with distinct vertices.

Definition 2.2.2 (Random Walk)**.** A random walk of length ℓ on a graph $G = (V, E, w)$ is a walk $(u_0, u_1, \ldots, u_\ell)$ obtained in a random fashion. Starting at vertex u_0, its next vertex u_1 is chosen at random from u_0's (out) neighbors, and so on.

Hereafter, we denote $P_{s,t}$ a path where $s = u_0$ and $t = u_\ell$. Additionally, a *circuit* is a trail where $u_0 = u_\ell$ and a *cycle* is a circuit with no repeated vertices. If a graph does not have cycles it is called *acyclic*. Directed acyclic graphs are abbreviated with the acronym DAG.

On weighted graphs, the weight of a walk W is defined as the sum of the weights of the edges it traverses:

$$w(W) := \sum_{i=1}^{\ell} w(e_i).$$

Clearly, if the graph is unweighted, the length and the weight of a walk coincide.[9]

If there exists a path from a vertex s to a vertex t, we say that s *reaches* t or, alternatively, that t is *reachable* from s. Reachability is important to determine connectivity and the components in a graph.

Definition 2.2.3 (Connected Graph)**.** An undirected graph is *connected* if any vertex reaches any other vertex.

[9] Recall that an unweighted graph is also a weighted graph where all edges have weight 1.

Definition 2.2.4 (Connected Component)**.** A *connected component* (or CC) of a graph G is a maximal connected subgraph C of G, i.e., no additional vertex or edge can be added to C without breaking its property of being connected.

Clearly, a connected graph has only one connected component – e.g., the graph in Figure 2.1a is connected. Moreover, we call *tree* an undirected acyclic connected graph and *forest* a disjoint union of trees.

Definition 2.2.5 (Strongly Connected Graph)**.** A directed graph is *strongly connected* if any vertex reaches any other vertex.

Definition 2.2.6 (Strongly Connected Component)**.** A *strongly connected component* (or SCC) of a directed graph G is a maximal strongly connected subgraph of G.

For example, the graph in Figure 2.1b has three SCCs: $\{\{0\}, \{3\}, \{1, 2, 4\}\}$. Analogously to the undirected case, strongly connected graphs consist of only one strongly connected component. By ignoring edge directions in directed graphs, we also have *weakly connected graphs* and *weakly connected components*.

Definition 2.2.7 (Weakly Connected Graph)**.** A directed graph is *weakly connected* all vertices are connected to each other by a path that ignores edge directions.

Definition 2.2.8 (Weakly Connected Component)**.** A *weakly connected component* (or WCC) of a directed graph G is a maximal weakly connected subgraph of G.

For example, the graph in Figure 2.1b is weakly connected because it has only one weakly connected component. Finally, we introduce biconnectivity, i.e., the property of graphs to remain connected after the removal of a vertex.

Definition 2.2.9 (Biconnected Graph)**.** A [directed] graph is *biconnected* if it remains [strongly] connected after the removal of any one vertex.

Definition 2.2.10 (Biconnected Component)**.** A *biconnected component* of a graph G is a maximal biconnected subgraph of G.

Clearly, the graph in Figure 2.1a is not biconnected – the removal of vertex 3 splits the graph into two connected components – but it has two biconnected components: $\{\{0, 3\}, \{1, 2, 3, 4\}\}$.

2.3 Distances in Graphs

In mathematics, a *metric* on a set S is a function $d : S \times S \to \mathbb{R}_{\geq 0}$ that satisfies the following axioms for all $x, y, z \in S$:

1. identity of indiscernibles: $d(x, y) = 0 \Leftrightarrow x = y$;
2. symmetry: $d(x, y) = d(y, x)$;
3. triangle inequality: $d(x, y) \leq d(x, z) + d(z, y)$.

We call such a function a *distance function*. Also, we define the distance between a point $s \in S$ and a subset $P \subseteq S$ as the distance from s to the nearest point in P:

$$d(s, P) := \min_{p \in P} d(s, p). \tag{2.1}$$

In an undirected graph $G = (V, E, w)$ with positive edge weights, the *vertex distance* is a metric $d : V \times V \to \mathbb{R}_{\geq 0}$ defined on all the pairs of vertices in G. Notwithstanding that the symmetry property does not necessarily hold in directed graphs, we follow the standard convention and use the term "distance" even in the directed case. We now introduce three vertex distance measures that are necessary for the definition of the centrality measures described in Sections 2.4 and 2.5.

2.3.1 Shortest-path Distance

The shortest-path distance, also known as the *geodesic* distance, is one of the most natural and common definitions of vertex distance in graphs. The *shortest path* between two vertices s and t is the path $P_{s,t}$ with minimum weight – note that such a path might not be unique.

Definition 2.3.1 (Shortest-path Distance)**.** The *shortest-path distance* is the function $d : V \times V \to \mathbb{R}_{\geq 0}$ such that $d(s, t)$ is the weight of the path $P_{s,t}$ with minimum weight.

Conventionally, if t is not reachable from s, the distance $d(s, t)$ is defined as infinite.

Definition 2.3.2 (Diameter)**.** The *diameter* of a graph of a graph G is the maximum shortest-path distance between two reachable vertices u and v where u reaches v:

$$\text{diam}(G) := \max_{\substack{u,v \in V, \\ d(u,v) < +\infty}} d(u, v).$$

Definition 2.3.3 (Eccentricity)**.** The *eccentricity* of a vertex $u \in V$ is the maximum shortest-path distance from u to a vertex v reachable from u:

$$\text{ecc}(u) := \max_{\substack{v \in V, \\ d(u,v) < +\infty}} d(u, v).$$

Computing the shortest path between two vertices is a classic problem in algorithmic graph theory and can be solved by the well-known Dijkstra's algorithm in $\mathcal{O}(m + n \log n)$ time – if the priority queue is implemented using a Fibonacci Heap. The same algorithm can also solve the more general *Single-Source Shortest Path* problem (or SSSP), i.e., finding the shortest distances from a vertex to *all* the other vertices in a graph, in $\Theta(m + n \log n)$ time. If the graph is unweighted, both the shortest path and the SSSP problems can be solved in $\mathcal{O}(n + m)$ time with a Breadth-First Search (or BFS). The problem of computing the shortest-path distances between all pairs of vertices is known as the *All-Pairs Shortest Path* problem (or APSP), and can be solved in $\mathcal{O}(nm + n^2 \log n)$ time by running Dijkstra's algorithm from each vertex or – for graphs with arbitrary edge weights – Johnson's algorithm. On unweighted graphs, in turn, it suffices to run a BFS from each vertex, which requires $\Theta(n^2 + nm)$ time.

In addition to the aforementioned *combinatorial* techniques, shortest-path distance problems can also be solved algebraically via *matrix multiplication*. APSP can be solved in $\mathcal{O}(n^\omega \log n)$ by repeatedly squaring the adjacency matrix of a graph, where $\omega < 2.373$ [9] is the matrix multiplication exponent, i.e., the smallest constant such that the product of two $n \times n$ matrices can be performed within $\mathcal{O}(n^{\omega+o(1)})$ algebraic operations. The logarithmic factor of the time complexity can be saved by a recursive decomposition of the problem [4, Theorem 5.7, p. 204]. In case of integral edge weights, further improvements have been proposed for undirected [267] and directed graphs [296, 297]. Despite being asymptotically faster than combinatorial algorithms on dense graphs,[10] methods based on matrix multiplication are in practice not applicable to large graphs because they require to store dense matrices.

2.3.2 Resistance Distance

In the context of determining the distance between two vertices in a graph, some applications require to take into account not only the *shortest* paths, but *all* the paths between the two vertices [57, 70, 274]. A typical example are electrical networks: an electrical network can be regarded as an undirected weighted graph $G = (V, E, r)$ where edges are *resistors* and $r : E \to \mathbb{R}_{>0}$ is the *resistance* of an edge. The *conductance* of an edge e is defined as the reciprocal its resistance: $c(e) = 1/r(e)$. Let $e = \{a, b\} \in E$ and let $p(a, b) = p(a) - p(b)$ be the electric *potential difference* between a and b; then, according to Ohm's law, an amount

$$i(a, b) = \frac{p(a, b)}{r(e)}$$

of *electrical current* flows from a to b. Notice that, although the graph is undirected, the electrical current has a *direction* which is determined by the sign of potential difference, and thus $p(a, b) = -p(b, a)$ and $i(a, b) = -i(b, a)$. The current always flows from the vertex with the highest potential to the vertex with the lowest potential. Potential differences and currents in an electrical network are governed by the two well-known Kirchhoff's potential and current laws [54, 196].

Kirchhoff's potential law. The potential differences round any cycle $(u_1, u_2, \ldots, u_\ell)$ sum to zero:

$$p(u_1, u_2) + p(u_2, u_3) + \ldots + p(u_\ell, u_1) = 0.$$

Kirchhoff's current law. The total current outflow from any vertex u is zero:

$$i(u, \infty) + \sum_{v \in N(u)} i(u, v) = 0.$$

Here $i(u, \infty)$ denotes the amount of current leaving the network at vertex u – in accordance with our notation, $i(\infty, u) = -i(u, \infty)$ is the amount of current entering the network at vertex u.

[10] The *density* of a graph is $m/\binom{n}{2}$, i.e., the fraction of edges actually present in the graph over the maximum possible number of edges. Consider a sequence of graphs of increasing number of vertices n. If the density approaches zero as n becomes large, the graphs are said to be *sparse*. Otherwise, a graph where the density remains non-zero in the limit of large n is said to be *dense*. Clearly, we cannot take the limit of real-world networks and thus we cannot determine formally whether they are sparse or dense. Informally, however, for these networks "sparse" means that most of the edges that could exist in the network are not present [227, Section 6.10.1].

In most problems, current enters the network at some vertices and leaves it at others. Such a general problem can be reduced to the fundamental problem where a unit of current enters the network at a vertex s, called the *source*, and leaves it at another vertex t, called the *sink*.

The *effective conductance* from s to t, $c_{\text{eff}}(s,t)$, is the amount of current flowing from s to t when $p(s,t) = 1$. The *effective resistance* $r_{\text{eff}}(s,t) = 1/c_{\text{eff}}(s,t)$ is the potential difference between s to t when a unit of current flows from s to t [54, Ch. IX]. This quantity is also referred to as the *resistance distance* between s and t [120].

Definition 2.3.4 (Resistance Distance)**.** Let $G = (V, E, r)$ be a graph. The resistance distance (or effective resistance) is a metric $\rho : V \times V \to \mathbb{R}_{\geq 0}$ defined as the potential difference between two vertices s and t ensuring s as the source vertex, t as the sink vertex, and a unit of current from s to t.

Let $z \in V$ and $\mathbf{e}_z$ be the canonical unit vector for z, i.e., $\mathbf{e}_z[u] = 0$ for all $u \neq z$ and $\mathbf{e}_z[z] = 1$. The resistance distance between s and t can be computed as follows:

$$\rho(s,t) = (\mathbf{e}_s - \mathbf{e}_t)^\top \mathbf{L}^\dagger (\mathbf{e}_s - \mathbf{e}_t) = \mathbf{L}^\dagger[s,s] - 2\mathbf{L}^\dagger[s,t] + \mathbf{L}^\dagger[t,t], \tag{2.2}$$

where $\mathbf{L}^\dagger$ is the Moore-Penrose pseudoinverse of the Laplacian matrix [126, p. 290]. $\mathbf{L}^\dagger$ can be expressed as [281]:

$$\mathbf{L}^\dagger = \left(\mathbf{L} + \frac{1}{n}\mathbf{J}\right)^{-1} - \frac{1}{n}\mathbf{J}, \tag{2.3}$$

where $\mathbf{J}$ is the $n \times n$-matrix with all elements equal to one. $\mathbf{L}^\dagger$ has numerous applications in physics and engineering [281] as well as applied mathematics [126] and graph (resp. matrix) algorithms [183]. A straightforward way to compute the resistance distance between two vertices is to compute $\mathbf{L}^\dagger$. However, this approach is limited to small graphs because it requires $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ memory as $\mathbf{L}^\dagger$ is generally a dense matrix. Clearly, to compute $\rho(s,t)$, only three elements of $\mathbf{L}^\dagger$ are needed, and selected elements of $\mathbf{L}^\dagger$ can be computed in practice more efficiently than computing the whole $\mathbf{L}^\dagger$ [151, 187].

The resistance distance between two vertices u and v is proportional to the *commute time* between u and v, namely:

$$t_c(u,v) := t_h(u,v) + t_h(v,u) = \text{vol}(G) \cdot r_{\text{eff}}(u,v),$$

where $t_h(u,v)$ is the *hitting time* from u to v, i.e., the expected length of a random walk (see Definition 2.2.2) that starts in u and ends in v, and $\text{vol}(G)$ is the *volume* of G, i.e., the sum of the edge weights of the graph. The commute time can be interpreted as the expected length of a random walk for going from u to v and back to u again [62].

2.3.3 Forest Distance

Forest distance is a one-parametric family of metrics introduced by Chebotarev and Shamis [69, 71] that takes into account all the paths between two vertices. Contrary to resistance distance, forest distance (i) applies to disconnected graphs out of the box[11] and (ii) allows to control the influence of the length of the paths between the two vertices over the value of their distance via a parameter $\alpha > 0$. Forest distances have

[11] Resistance distance requires some adjustments to handle disconnected graphs.

been shown to effectively capture sensitive relationship indices such as social proximity and group cohesion and thus has found application in sociology [72].

Definition 2.3.5 (Rooted Spanning Forest)**.** Let $G = (V, E, w)$ be an undirected graph with c connected components $(C_1, C_2, \ldots, C_c)$ and let T_i be a spanning tree of the component C_i. A *spanning forest* on G is the disjoint union $\cup_{i=1}^{c} T_i$. A spanning forest is *rooted* if each spanning tree T_i has a vertex marked as its root.

Let $G = (V, E, w)$ be an undirected graph and let $\alpha > 0$. The forest distance is defined in terms of the parametric *forest matrix* of G, i.e.,

$$\mathbf{\Omega}_\alpha := (\mathbf{I} + \alpha \mathbf{L})^{-1}. \tag{2.4}$$

The name *forest* stems from the fact that the element $\mathbf{\Omega}_\alpha[u, v]$ is the fraction of rooted spanning forests where u is the root of a tree and v belongs to the same tree [69, 153]. An alternative definition of the forest matrix preferred in other works such as Ref. [70] is $(\alpha\mathbf{I} + \mathbf{L})^{-1}$, which is equivalent to the one given in Eq. (2.4) up to a scaling factor of the edge weights of the graph. The forest matrix $\mathbf{\Omega}_\alpha$ is symmetric and doubly stochastic [213], i.e., for all $i, j \in \{1, \ldots, n\}$:

$$\mathbf{\Omega}_\alpha[i, j] \geq 0, \quad \mathbf{j}^\top \mathbf{\Omega}_\alpha = \mathbf{j}^\top, \quad \mathbf{\Omega}_\alpha \mathbf{j} = \mathbf{j},$$

where $\mathbf{j}$ is the all-ones vector. Further, $\mathbf{\Omega}_\alpha[i, j]$ is 0 iff i and j are in two different components, whereas $\mathbf{\Omega}_\alpha[i, i] \leq 1$ with equality iff i is an isolated vertex [214].

Definition 2.3.6 (Forest Distance [69])**.** Let $G = (V, E, w)$ be an undirected graph, $\alpha > 0$, and $s, t \in V$. The forest distance is a one-parametric metric $\zeta_\alpha : V \times V \to \mathbb{R}_{\geq 0}$ defined as:

$$\zeta_\alpha(s, t) := (\mathbf{e}_s - \mathbf{e}_t)^\top \mathbf{\Omega}_\alpha (\mathbf{e}_s - \mathbf{e}_t) = \mathbf{\Omega}_\alpha[s, s] - 2\mathbf{\Omega}_\alpha[s, t] + \mathbf{\Omega}_\alpha[t, t].$$

Higher values of α give greater importance to long paths. To see this we analyze the asymptotic behavior of $\zeta_\alpha(s, t)$ with respect to α. As $\alpha \to 0$, $\zeta_\alpha(s, t)$ tends to the discrete metric:

$$\zeta_0(s, t) = \begin{cases} 0 & \text{if } s = t \\ 1 & \text{otherwise.} \end{cases}$$

Let us denote by V_s the set of vertices in the connected component that contains vertex s. As $\alpha \to \infty$, $\zeta_\alpha(s, t)$ tends to [69]:

$$\zeta_\infty(s, t) = \begin{cases} 0 & \text{if } t \in V_s \\ \frac{1}{2}\left(\frac{1}{|V_s|} + \frac{1}{|V_t|}\right) & \text{otherwise.} \end{cases}$$

2.4 Centrality Measures

Given a graph $G = (V, E, w)$, a *centrality measure* is a function $c : V \to \mathbb{R}$ that assigns a *centrality score* to each vertex depending on its structural position in the graph. The main intuition is that, if a vertex has a central position in the graph (i.e., it is *structurally central*), then it is also important or has some degree of

influence in the network under consideration. Clearly, what makes a vertex central in a network is highly application-dependent, and thus a universal definition of centrality cannot be given. Over the past decades, numerous centrality measures were introduced and applied in a multitude of contexts and applications [3, 35, 81, 197, 241, 269]. In this section, we describe the centrality measures that are relevant to our contributions, for a broader survey see [57, 132, 227, Ch. 7.1]. As in [53], we categorize centrality measures into three main classes: *distance-based* measures, *spectral* measures and *path-based* measures. For the sake of comparison, in the heatmaps in Figure 2.2, we exemplify the centrality of the members of the popular Zachary's karate club social network [289] according to different centrality measures.

2.4.1 Distance-based Measures

Distance-based centrality measures express the importance of a vertex u as a function of a distance metric between u and other vertices (see Section 2.3). The distance to other vertices is one of the most natural ways to assess the importance of a vertex in a network, and therefore these measures are among the first ever defined.

Degree Centrality. The degree of a vertex is probably the simplest and oldest centrality measure ever used [109]. Even though it only takes into account the immediate neighbors of a vertex, degree centrality has been shown to be strongly correlated with other centrality measures such as closeness, betweenness, and eigenvector centrality in real-world networks [186, 280]. This is also clear from the example in Figure 2.2: the vertices with highest degree (see Figure 2.2a) are also central according to other measures.

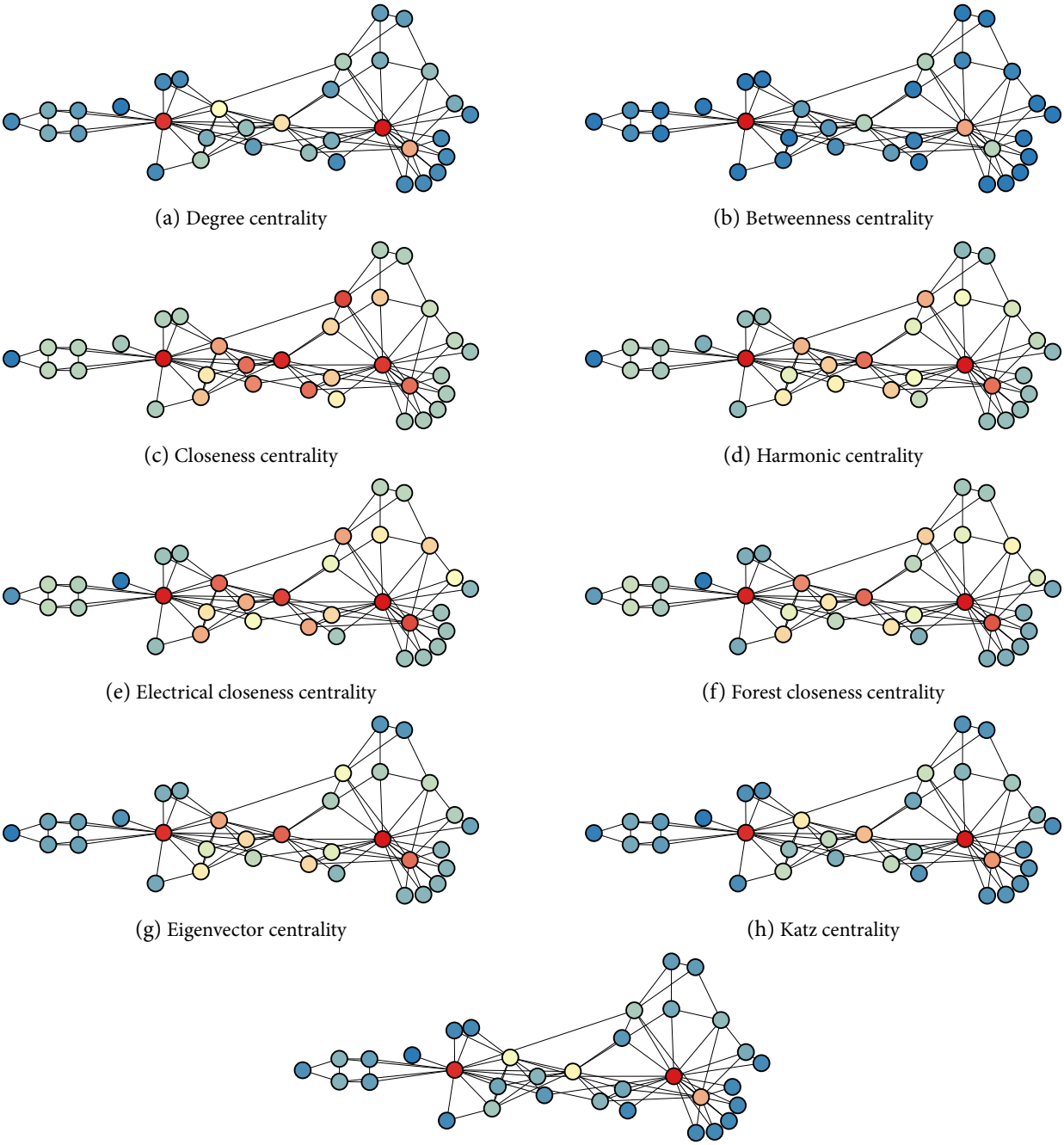
Closeness Centrality. Recall from Section 1.1 that closeness was introduced in 1950 by Bavelas in his attempt of capturing how a communication pattern may affect the performance of a group of people [34]. A vertex in a network is considered *central* if the sum of the geodesic distances to all the other vertices is low. Intuitively, the time required by a message to be spread throughout an entire network is minimized if the message originates from the most central vertex.

Such a definition of closeness depends upon the number of vertices in the network from which it is calculated, and therefore comparing the closeness of vertices in networks with different sizes would be meaningless. Beauchamp [35] solved this issue by suggesting a normalization factor for closeness of $n-1$. The closeness centrality of a vertex u is thus defined as:

$$c_c(u) := \frac{n-1}{\sum_{v \in V} d(u,v)}. \tag{2.5}$$

The denominator is also known as the *farness* of u, or $f(u)$. Closeness centrality may also be interpreted as the reciprocal of the average distance from u to the other vertices. Notice that $c_c(\cdot)$ is only defined on (strongly) connected graphs – if some vertex u cannot reach some other vertex v, then $d(u,v)$ is infinity. To overcome this limitation, Lin introduced Lin's index [188]:

$$c_{\text{Lin}}(u) := \frac{r^2(u)}{\sum_{v \in R(u)} d(u,v)},$$



(a) Degree centrality

(b) Betweenness centrality

(c) Closeness centrality

(d) Harmonic centrality

(e) Electrical closeness centrality

(f) Forest closeness centrality

(g) Eigenvector centrality

(h) Katz centrality

(i) PageRank

Figure 2.2: Heatmaps of vertex centrality scores of the vertices in the Zachary's karate club social network [289]. Red means high centrality score, blue means low centrality score. Images were generated with the Gephi software [31].

where $R(u)$ is the set of vertices reachable from u and $r(u)$ is $|R(u)|$. Olsen et al. [231] proposed a further definition of closeness for disconnected graphs:

$$c_c(u) := \frac{(r(u)-1)^2}{(n-1)\sum_{v \in R(u)} d(u,v)}.$$

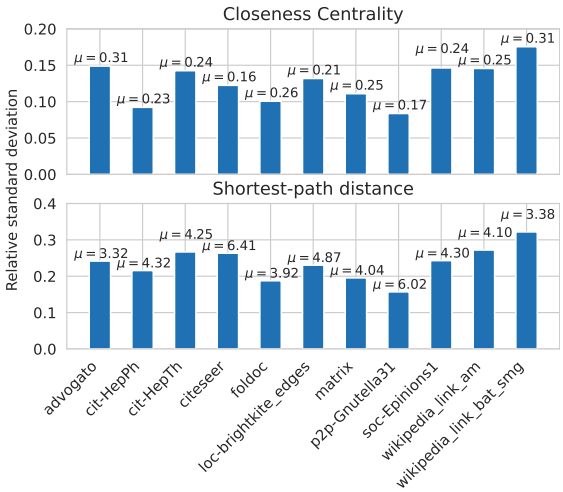


Figure 2.3: Relative standard deviation for shortest-paths distances and for closeness centrality for the networks in Table 2.1.

Network	n	m	Diameter
advogato	5,042	52,195	9
cit-HepPh	34,401	421,529	14
cit-HepTh	27,400	352,580	15
citeseer	365,154	1,742,596	34
foldoc	13,356	120,238	8
loc-brightkite_edges	56,739	212,945	18
matrix	79,116	515,397	12
p2p-Gnutella31	62,561	147,878	11
soc-Epinions1	75,877	508,836	15
wikipedia_link_am	20,883	105,714	12
wikipedia_link_bat_smg	21,814	123,756	13

Table 2.1: Graphs used in Figure 2.3 (downloaded from the public repository KONECT [173]).

Unless stated differently, hereafter we refer to the definition in eq. (2.5), which is well-established in the literature [53].

Apart from requiring some adjustments to handle disconnected graphs, another known weakness of closeness centrality is its limited capability to discriminate different vertices, especially on complex networks. A typical characteristic of complex networks is having a small diameter [6]. Hence, all distances – and thus all closeness values – lie within a narrow interval [132]. To show this in practice, we pick 100 vertices uniformly at random from the networks of Table 2.1 and we compute their closeness centrality and the shortest-path distances from those vertices to the others. Figure 2.3 reports the mean and the relative standard deviation of the shortest-path distances and of the closeness centrality for the networks of Table 2.1. The relative standard deviation is the standard deviation (σ) divided by the mean (μ) and shows the extent of variability with respect to the mean of the distribution. For example, for the "advogato" network, the standard deviation is 14.88% of the mean $\mu = 0.31$, which is quite low.[12] Such limitation can also be observed from Figure 2.2c: compared to other measures (e.g., betweenness or PageRank), closeness fails in discriminating highly-central vertices from the others – as several vertices have a high or very similar centrality score.

Harmonic Centrality. Marchiori and Latora [204] introduced the *connectivity length* of a graph in order to measure the *efficiency* of a network in terms of information propagation. The connectivity length of a graph is defined as the *harmonic mean* of all the pairwise distances between the vertices in a graph. Later, harmonic centrality was independently devised by Dekker [86] (with the name "valued centrality") and by Rochat [253]. Harmonic centrality is defined as:

$$c_h(u) := \sum_{v \in V \setminus \{u\}} \frac{1}{d(u,v)}. \tag{2.6}$$

[12] In this example, $\sigma = 0.05$ which means that, assuming a normal distribution of the closeness scores, $\approx 68\%$ of the values of $c_c(\cdot)$ are in the interval $[0.26, 0.35]$ while $\approx 95\%$ of the values are in $[0.22, 0.40]$.

Under the reasonable assumption that the reciprocal of infinity is zero, harmonic centrality is well-defined also on disconnected graphs. Further, from an axiomatic point of view, harmonic centrality enjoys several desirable properties [53].

Electrical Closeness. Electrical closeness [62] – also known as *current-flow closeness* or *information centrality* [274] – ranks vertices according to their average resistance distance to the others:

$$c_e(u) := \frac{n-1}{\sum_{v \in V \setminus \{u\}} \rho(u,v)}. \tag{2.7}$$

Analogously to combinatorial closeness, electrical closeness is not directly applicable to disconnected graphs due to the aforementioned infinite distances issue. However, generalizations to disconnected graphs such as Lin's [188] or Olsen's [231] apply as well.

Because it takes into considerations all the paths between two vertices, electrical closeness solves two issues concerning the vertex rankings computed by combinatorial closeness: (i) having a low discriminative power, especially on complex networks, and (ii) being highly susceptible to changes in the graph [132]. These claims are corroborated by the experiments in Ref. [41].

Forest Closeness. Forest distance closeness centrality (abbreviated with *forest closeness*) is a further variation of closeness centrality where the shortest-path distance is replaced by the *forest distance*:

$$c_{f,\alpha}(u) := \frac{n}{\sum_{v \in V \setminus \{u\}} \zeta_\alpha(u,v)}. \tag{2.8}$$

The denominator of $c_{f,\alpha}(u)$ is also called the *forest farness* of u. Compared to other centrality measures, forest closeness presents two main advantages: (i) by taking all paths into account it has a high discriminative power [153] and (ii) it handles disconnected graphs out of the box.

2.4.2 Spectral Centrality Measures

Spectral centrality measures compute the importance of the vertices of a graph G by using the left dominant eigenvector of a matrix derived from a matrix representation of G. The existence and the uniqueness of most of these measures is guaranteed by the theory developed by Perron and Frobenius [110, 240] about non-negative matrices [42].

Eigenvector Centrality. Eigenvector centrality implements the idea that a vertex in a network is important if adjacent to vertices that are themselves important. It can also be interpreted as an extension of degree centrality where the centrality of a vertex is *proportional to the centrality of its neighbors*.

Let λ be the dominant eigenvalue of the adjacency matrix $\mathbf{A}$ of a graph. Assuming that $V = \{1, \ldots, n\}$, i.e., all vertices in G are indexed from 1 to n, the eigenvector centrality of a vertex $u \in V$ is defined as:

$$c_{\text{eig}}(u) = \frac{1}{\lambda} \sum_{v=1}^{n} A[u,v] \cdot c_{\text{eig}}(v), \tag{2.9}$$

In matrix notation, Eq. (2.9) is equivalent to $c_{\text{eig}}(u) = \mathbf{x}[u]$ where $\mathbf{x}$ is the right leading eigenvector that solves the linear system $\mathbf{Ax} = \lambda\mathbf{x}$.

If $\mathbf{A}$ is irreducible,[13] the Perron-Frobenius theorem implies that $\lambda > 0$ and that all the entries of $\mathbf{x}$ are strictly positive. $\mathbf{A}$ is irreducible iff the corresponding graph is strongly connected [42]. In case of disconnected graphs, however, the eigenvector centrality of some vertices could be zero [227], which makes this measure suitable for (strongly) connected graphs only.

Katz Centrality. The limitation of eigenvector centrality of being restricted to strongly connected graphs is resolved by Katz centrality [159]. Conversely to distance-based based centrality measures that only consider shortest paths, this measure takes into account *all the walks* between two vertices, with shorter walks having a greater contribution to the centrality score than longer walks. The Katz centrality of a vertex u is:

$$c_{\text{K}}(u) := \sum_{i=1}^{\infty} \alpha^i \sum_{v \in V} \mathbf{A}^i[u, v], \tag{2.10}$$

which converges if the *attenuation factor* $\alpha > 0$ is less than the reciprocal of the largest singular value of $\mathbf{A}$. Eq. (2.10) can be rewritten in matrix terms:

$$\mathbf{c}_{\text{K}} = \Big((\mathbf{I} - \alpha\mathbf{A})^{-1} - \mathbf{I} \Big)\mathbf{j}, \tag{2.11}$$

where $\mathbf{c}_{\text{K}}[u]$ is the Katz centrality of $u \in V$.

PageRank. Originally designed with the purpose of developing a search engine, PageRank [235] is among the most recent and popular spectral centrality measures. Despite being tailored to the Internet graph, as of today this measure is used in applications beyond web graphs [122]. PageRank models the stationary distribution of a random walk over the vertices of a graph. At any step, the random walk jumps to another random vertex with probability $1 - \alpha$, where $\alpha \in (0, 1)$ is called the *damping factor*. The PageRank score of a vertex u is defined as:

$$c_{\text{PR}}(u) := \alpha \sum_{v \in N(u)} \frac{c_{\text{PR}}(v)}{\deg(v)} + \frac{1 - \alpha}{n}.$$

2.4.3 Path-based Centrality Measures

Path-based centrality measures take into account all the (shortest) paths hitting a vertex. Notice that, in unweighted graphs, degree centrality can also be considered a path-based measure since it evaluates the number of incoming or outgoing paths of length one.

Betweenness Centrality. Freeman [108] introduced betweenness centrality as a measure of how much a vertex *controls* the information flow in a network. A vertex is considered central if it stands *between* other

[13] A square matrix $\mathbf{M}$ is called *reducible* if there exists a permutation matrix $\mathbf{P}$ such that:

$$\mathbf{PMP}^{-1} = \begin{pmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ 0 & \mathbf{B}_{2,2} \end{pmatrix},$$

where $\mathbf{B}_{1,1}$ and $\mathbf{B}_{2,2}$ are non-empty square matrices. A matrix that cannot be reduced is called *irreducible*.

vertices, as it can control or influence their communications – under the assumption that information flows through shortest paths exclusively. More formally, the betweenness centrality of a vertex u is defined as the probability that a shortest path between any two vertices $x, y \in V \setminus \{u\}$ passes through u. Let $\sigma_{x,y}$ be the number of shortest paths from x to y, and $\sigma_{x,y}(u)$ be the number of such paths that cross u. The betweenness centrality of u is defined as:

$$c_b(u) := \sum_{\substack{x,y \in V \setminus \{u\} \\ x \neq y,\ \sigma_{x,y} \neq 0}} \frac{\sigma_{x,y}(u)}{\sigma_{x,y}}. \tag{2.12}$$

Vertices with high betweenness centrality can also be interpreted as crucial actors whose removal from the network would disrupt most communications between other vertices [52]. Indeed, in the example in Figure 2.2b, the removal of the vertex with highest betweenness (the red one) would isolate six vertices from the entire network.

ED-Walk Centrality. Inspired by Katz centrality, in Ref. [13], we define ED-Walk, a new centrality measure designed to possess two main properties: (i) to take into consideration *all walks* (not just shortest paths) of *any length* that cross a vertex and (ii) to admit a natural generalization to sets of vertices, leading to a monotone and submodular group centrality measure (see Section 2.5) and to a scalable greedy maximization algorithm.

ED-Walk stands for *exponentially decaying walk* as the contribution of a walk to the centrality score decays exponentially with its length according to a parameter $\alpha > 0$. Let $\phi_i(S)$ be the number of i-walks (i.e., walks of length i) that contain at least one vertex in $S \subseteq V$. The ED-Walk for $u \in V$ is defined as [13]:

$$c_{\text{ED}}(u) := \sum_{i=1}^{\infty} \alpha^i \phi_i(\{x\}).$$

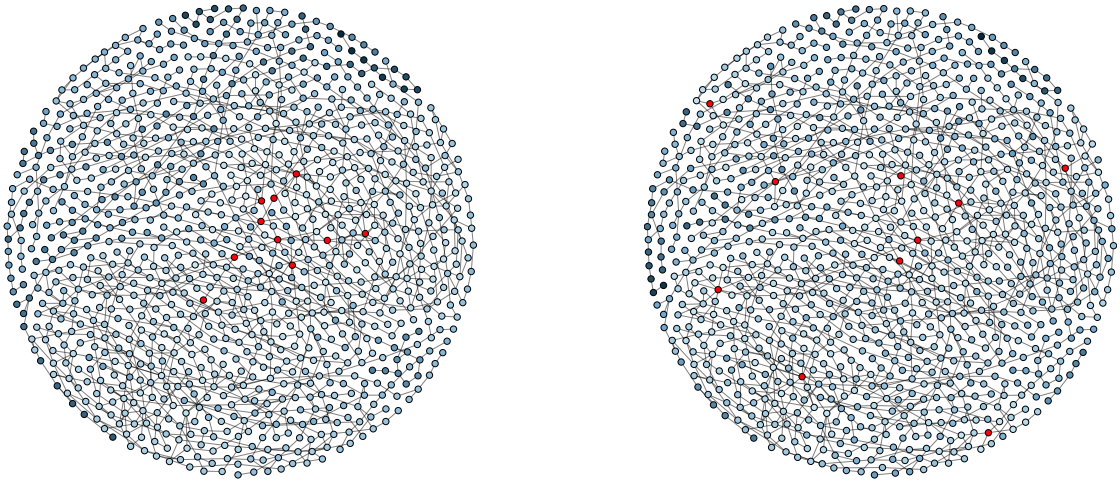
For the series to converge, α needs to be chosen small enough. As for Katz centrality, α must be less than the reciprocal of the largest singular value of the adjacency matrix of the graph [13, Sec. 2.2].

2.5 Group Centrality Measures

The centrality measures we describe in Section 2.4 are also called *single-vertex* centrality measures because they indicate the importance of a *single* vertex in a graph with respect to the others. However, determining the centrality of a *group* of vertices *as a whole*, or to find the most central group of vertices – or group centrality maximization – are two frequent problems that arise in graph mining applications such as influence maximization [162, 291], facility location [121, 185], congestion avoidance [285], and others.

Everett and Borgatti [104] proposed a general framework to generalize single-vertex centrality measures to groups of vertices.[14] Given a graph $G = (V, E, w)$ a group centrality measure is a set function $g : 2^V \to \mathbb{R}$ that assigns to each subset of V a group centrality score. A group centrality measure is a proper generalization of a single-vertex centrality measure if the two yield the same score when applied to a group consisting

[14] In their work, Everett and Borgatti extended degree, closeness, betweenness, and flow betweenness centrality to groups of vertices.



(a) Red vertices: top-k vertices with highest closeness.

(b) Red vertices: group of vertices with high group-closeness (computed with the *Grow-Shrink* algorithm, see Chapter 6).

Figure 2.4: Comparison between top-k closeness centrality and group-closeness centrality ($k = 10$) on the European road network [276] (downloaded from the public repository KONECT [173] and drawn with the Fruchteman Reingold algorithm [111]). Red vertices represent in Figure 2.4a the top-k vertices with highest closeness and, in Figure 2.4b, the k vertices in a set with high group-closeness centrality. For the other vertices, the bluest they are the greater is their distance to the nearest red vertex. Images were generated with the Gephi software [31].

of a single vertex [104]. In general, the vertices in the most central group "cover" well the entire graph together; thus, they often differ considerably from the top-k most individually central vertices. Figure 2.4 exemplifies this difference in the closeness centrality case. The top-10 vertices with highest $c_c(\cdot)$ in Figure 2.4a are clustered in the center of the graph and, consequently, far from peripheral vertices. The set S of 10 vertices in Figure 2.4b, in turn, has high *group-closeness*: compared to Figure 2.4a, vertices in S are much more scattered around the network and thus the average distance from a vertex to the nearest in S is lower.

Properties of Set Functions. Properties of set functions such as monotonicity and submodularity have played a crucial role in combinatorial optimization [106, 193, 226, 282].

Definition 2.5.1 (Monotonicity)**.** Let X be a non-empty finite set. A set function $f : 2^V \to \mathbb{R}$ is called *non-decreasing* if $f(T) \geq f(S)$ for all $T \supseteq S$. Similarly, f is called *non-increasing* if for every $S \subseteq T$, we have $f(S) \geq f(T)$. In both cases, f is called *monotone*.

Definition 2.5.2 (Submodularity and Marginal Gain)**.** A set function $f : 2^X \to \mathbb{R}$ is called *submodular* if, for every $S \subseteq T \subseteq X$ and every $x \in X \setminus T$, we have that:

$$f(S \cup \{x\}) - f(S) \geq f(T \cup \{x\}) - f(T \cup \{x\}).$$

In this context, the value $f(S \cup \{x\}) - f(S)$ is called the *marginal gain* of x w.r.t. the set S. Similarly, f is called *supermodular* if $f(S \cup \{x\}) - f(S) \leq f(T \cup \{x\}) - f(T)$ for every $S \subseteq T \subseteq X$ and every $x \in X \setminus T$.

The relevance of these properties stems from a classical result about optimization of monotone and submodular set function:

Proposition 2.5.1 ([226])**.** Let $f : 2^X \to \mathbb{R}$ be a non-decreasing submodular set function. Consider the problem of maximizing f over all possible subsets $S \subseteq X$ w.r.t. the cardinality constraint $|S| \leq k$ for some $k \in \mathbb{N}$. Let $S^\star$ be the optimal solution to this problem:

$$S^\star := \underset{S \subseteq X, |S| \leq k}{\operatorname{argmax}} \; f(S).$$

The greedy algorithm that constructs S by iteratively adding the element $x \in X$ with highest marginal gain $f(S, x) := f(S \cup \{x\}) - f(S)$ to S yields a $(1 - 1/e)$-approximation for this problem. Specifically, if $\tilde{S}$ is the result of the greedy algorithm, it holds that $f(\tilde{S}) \geq (1 - 1/e) f(S^\star)$.

Group-Degree Centrality. The *group-degree centrality* of a group $S \subset V$ is the number of vertices in $V \setminus S$ that are neighbors of at least a vertex in S [104]:

$$g_d(S) := |\{v \in V \setminus S : (u, v) \in E \text{ for some } u \in S\}|.$$

It is simple to verify that group-degree is not monotone: in a triangular graph the group-degree of a single vertex is two, whereas the group-degree of any set with two vertices is one. On the other hand, group-degree is submodular.

Lemma 2.5.1. Group-degree is submodular.

Proof. Let $\delta_u(S)$ be 1 if $\{u, v\} \in E$ for some $v \in S$, 0 otherwise. We need to show that $g_d(S \cup \{u\}) - g_d(S) \geq g_d(T \cup \{u\}) - g_d(T)$. Because $T \supseteq S$ it holds that:

$$g_d(S \cup \{u\}) - g_d(S) = |N(u) \setminus S| - \delta_u(S) \geq |N(u) \setminus T| - \delta_u(T) = g_d(T \cup \{u\}) - g_d(T).$$

□

Group-Closeness Centrality. For a group $S \subset V$, its *group-closeness centrality* is defined as [38]:

$$g_c(S) := \frac{n}{\sum_{v \in V \setminus S} d(S, v)}, \tag{2.13}$$

where $d(S, v)$ is $\min_{s \in S} d(s, v)$, see Eq. (2.1). The denominator of $g_c(S)$ is also called the *group-farness* of S. As with closeness centrality, group-closeness is only defined on strongly connected graphs. Further, it is easy to see that $g_c(\cdot)$ is monotone: for each $S \subseteq T \subset V$ and $u \in V \setminus T$ we have that $d(T, u) \leq d(S, u)$, and thus $g_c(T) \geq g_c(S)$. However, group-closeness is not submodular, and this can be demonstrated with a simple counterexample.

Lemma 2.5.2. Group-closeness is not submodular.

Proof. Consider a complete undirected graph with five vertices numbered from 0 to 4, let $S = \{0, 1\}$, $T = \{0, 1, 2\}$, and $v = 3$:

$$g_c(S \cup \{v\}) - g_c(S) = \frac{5}{2} - \frac{5}{3} = \frac{5}{6} < g_c(T \cup \{v\}) - g_c(T) = \frac{5}{1} - \frac{5}{2} = \frac{5}{2}. \qquad \square$$

Group-Harmonic Centrality. The *group-harmonic centrality* of a group $S \subset V$ is defined as [12]:

$$g_h(S) := \sum_{v \in V \setminus S} \frac{1}{d(S, v)},$$

where, as for $c_h(\cdot)$, $1/d(S, v) = 0$ if no vertex in S reaches v. Hence, group-harmonic also handles disconnected graphs. This measure is submodular but not monotone [12]. Although such a definition is a natural generalization of harmonic centrality to groups of vertices, the way it handles vertices in the group may seem questionable. Indeed, vertices in S count as 0 in the harmonic centrality score of S while they are the closest ones to S. On the other hand, assigning them an arbitrary value greater than 0 would be unsatisfactory. A workaround for this problem is to always compare the group-harmonic centrality of groups with equal cardinality, so that the value assigned to vertices in the group does not have any impact [12].

GED-Walk Centrality. As described in Section 2.4, ED-Walk was designed to naturally generalize to groups of vertices $S \subseteq V$. This leads us to GED-Walk:

$$g_{\text{ED}}(S) := \sum_{i=1}^{\infty} \alpha^i \phi_i(S).$$

As we show in Chapter 8, GED-Walk not only is both monotone and submodular, but it is also faster to maximize (for groups of small size) than existing shortest-path based group centrality measures.

2.6 Matchings

Let $G = (V, E, w)$ be an undirected weighted graph. A *matching* in G is a subset of pairwise disjoint edges $M \subseteq E$. A vertex is called *matched* in M if it is incident to an edge $e \in M$; otherwise, it is called *free* or *unmatched*. A matching M is called *maximal* if no further edge can be added to M while retaining the matching property, and *maximum* if no other matching with higher cardinality exists. Computing a maximum cardinality matching (or MCM) of a graph is a popular combinatorial optimization problem that can be solved in $\mathcal{O}(m\sqrt{n})$ by the algorithm of Micali and Vazirani [215]. By restricting the input algorithms to planar graphs, MCM can be solved in $\mathcal{O}(n^\omega)$ [221].

The weight of a matching M is the sum of the weights of the edges in M and a maximum *weighted* matching (or MWM) is a matching with maximum weight. The fastest known algorithm for finding a MWM is by Galil et al. [116] and it takes $\mathcal{O}(mn \log n)$ time. Algorithms for MWM with lower time complexity exist but they are specialized for bipartite graphs [257] or graphs with integral edge weights [114]. A broader overview over matching theory and algorithms is provided in Refs. [Ch. 5 49, 194].

2.7 Dynamic Graphs

So far we only considered *static* graphs, i.e., graphs that do not change over time. However, graphs that occur in real-world scientific and commercial applications are often *dynamic*, i.e., they evolve over time: vertices and edges are inserted, removed, or edge weights are updated. Examples include social network analysis [294], computational biology [105], advertisements on search engines [211] and many more.

We target *fully-dynamic* graphs, i.e., dynamic graphs where update operations are restricted to edge insertions and removals [101].[15] These operations are also called *edge updates*. More formally, let $G = (V, E, w)$ be a graph; an edge update is an operation that transforms G into another graph $G' = (V, E', w')$. In case of the insertion of an edge $(u, v) \notin E$ with weight α, we have that $E' = E \cup \{(u, v)\}$, $w'(e) = w(e)$ for all $e \in E$ and $w'(u, v) = \alpha$. Conversely, in case of the removal of an edge $(u, v) \in E$, we have that $E' = E \setminus \{(u, v)\}$ and $w'(e) = w(e)$ for all $e \in E'$. Further, we denote with superscript $'$ any additional property of the graph G', e.g., $d'(u, v)$ is the shortest-path distance from u to v in G'.

2.8 Performance and Quality Indicators

An algorithm's performance is commonly measured by its *running time* and the wall-clock time is a widely used indicator for running time. For solution quality, indicators are usually problem-specific; we often measure the gap between the algorithm's solution and the optimum, if known, or a highly accurate solution. In the following, we describe the indicators we use in our experiments to evaluate the performance and the solution quality of our algorithms.

Running Time. We often compare the running time of two algorithms A and B on several instances. As recommended in Ref. [14], in order to make a concise evaluation, we aggregate the wall-clock times over *ratios*, which means computing the *algorithmic speedup* of A w.r.t. B. For a fair comparison of the algorithmic aspects, the algorithmic speedup is often measured over the *sequential* executions of the algorithms – i.e., using a single thread. To summarize multiple algorithmic speedup values we use the geometric mean [50]:

$$\text{GM(speedup)} = \left(\prod_{i=1}^{\text{\# of values}} \text{speedup on instance } i \right)^{\frac{1}{\text{\# of values}}}$$

as it has the fundamental property that $\text{GM(speedup)} = \frac{\text{GM(running times of } A)}{\text{GM(running times of } B)}$.

Concerning parallel algorithms, we evaluate how efficiently they can be parallelized. To do so we use the *parallel speedup* of an algorithm A, i.e., the speedup of a parallel execution of A over its sequential execution.

Solution Quality. The quality of the results yielded by the algorithms we consider in this work is represented by either a scalar, a vector of scores, or a ranking. The first category concerns problems such as group centrality maximization: the solution is a set S of vertices and its quality is the group centrality score of S. In this case, we measure the gap between the computed solution and the optimum as a percentage and we aggregate multiple results with the geometric mean.[16]

[15] On weighted graphs, edge weight updates can be represented as an edge removal followed by an edge insertion.
[16] If the optimum is too computationally expensive to compute, we resort to high-quality approximations of it.

In the second category, we have solutions where a score is computed for each vertex (or edge) in the graph. Let us assume that all vertices in the graph are indexed from 1 to n, let $\widetilde{\mathbf{x}}$ be the computed vector (where $\widetilde{\mathbf{x}}[i]$ is the score computed for the i-th vertex) and let $\mathbf{x}$ be vector of the exact scores. Recall the definition of a vector norm.

Definition 2.8.1 (Vector Norm [244])**.** Let Z be a vector space over $\mathbb{R}$. A vector norm on Z is a function $\|\cdot\| : Z \to \mathbb{R}$ such that:

1. for each $\mathbf{z} \in Z$, $\|\mathbf{z}\| \geq 0$ and $\|\mathbf{z}\| = 0 \Leftrightarrow \mathbf{z} = 0$;
2. for each $\mathbf{z} \in Z$ and every $\alpha \in \mathbb{R}$, $\|\alpha\mathbf{x}\| = |\alpha| \cdot \|\mathbf{x}\|$;
3. for each $\mathbf{z}_1, \mathbf{z}_2 \in Z$, $\|\mathbf{z}_1 + \mathbf{z}_2\| \leq \|\mathbf{z}_1\| + \|\mathbf{z}_2\|$ (triangle inequality).

For error estimation purposes, we are interested in three vector norms on $\mathbb{R}^n$, the 1-norm, the 2-norm, and the max-norm, which are defined as follows for a vector $\mathbf{z} \in \mathbb{R}^n$ [244]:

$$\begin{aligned}
\|\mathbf{z}\|_1 &:= \sum_{i=1}^{n} |\mathbf{z}[i]|, \\
\|\mathbf{z}\|_2 &:= \sqrt{\sum_{i=1}^{n} \mathbf{z}[i]^2}, \\
\|\mathbf{z}\|_\infty &:= \max(|\mathbf{z}[1]|, |\mathbf{z}[2]|, \ldots, |\mathbf{z}[n]|).
\end{aligned}$$

Depending on the problem under consideration and the algorithm's quality guarantees, we evaluate the gap between the computed result $\widetilde{\mathbf{x}}$ and the baseline $\mathbf{x}$ on the basis of the 1-norm, 2-norm, and/or the max-norm of the absolute error vector $(|\widetilde{\mathbf{x}}[1] - \mathbf{x}[1]|, \ldots, |\widetilde{\mathbf{x}}[n] - \mathbf{x}[n]|)$ and/or of the relative error vector $\left(\frac{|\widetilde{\mathbf{x}}[1]-\mathbf{x}[1]|}{\mathbf{x}[1]}, \ldots, \frac{|\widetilde{\mathbf{x}}[n]-\mathbf{x}[n]|}{\mathbf{x}[n]}\right)$.

Finally, the third category accounts for solutions where the quality is measured by a ranking of the vertices according to their individual scores, which is often more relevant than the scores per se [227, 229]. As we did above, let $\mathbf{x}$ be the vector of exact scores (or a high-quality approximation of it) and let $\widetilde{\mathbf{x}}$ be the vector of the computed scores. A pair of scores $\langle(\mathbf{x}[i], \widetilde{\mathbf{x}}[i]), (\mathbf{x}[j], \widetilde{\mathbf{x}}[j])\rangle$ with $i < j$ is said to be *concordant* if either $(\mathbf{x}[i] > \mathbf{x}[j]) \wedge (\widetilde{\mathbf{x}}[i] > \widetilde{\mathbf{x}}[j])$ or $(\mathbf{x}[i] < \mathbf{x}[j]) \wedge (\widetilde{\mathbf{x}}[i] < \widetilde{\mathbf{x}}[j])$, ties are neglected for simplicity; otherwise, they are said to be *discordant*. We measure the quality of a ranking it terms of:

- Percentage of concordant pairs:
$$\frac{\text{\# of concordant pairs}}{\binom{n}{2}};$$
- Kendall τ coefficient:
$$\tau := \frac{(\text{\# of concordant pairs}) - (\text{\# of discordant pairs})}{\binom{n}{2}}.$$

Part II

Algorithms for Single-Vertex Centrality Measures

Introduction

Vertex centrality is arguably one of the most popular concepts in network analysis. Given a graph G, centrality measures assign to each vertex v in G a *centrality score* that represents the importance of v in G by considering the structural properties of the graph. Because importance highly depends on the application, several centrality measures have been proposed and none is universal [53, 132].

Today, graph datasets easily reach millions (sometimes billions) of edges and thus the efficiency and scalability of network analysis algorithms is a major concern. Kang et al. [156] observed that "measuring centrality in billion-scale graphs poses several challenges. Many of the 'traditional' definitions such as closeness and betweenness were not designed with scalability in mind". This often implies that complete exact computations of such measures on large graphs is prohibitively expensive and thus not practical – especially for *global* measures that take the whole graph into account. On the other hand, complete exact computation is often not necessary as several applications require either a reliable ranking of the vertices [227, 229] or to identify the top-k most central vertices [231].

Another challenge is posed by dynamic graphs, i.e., graphs that change over time – see Section 2.7. Edges can be inserted or deleted and this might change the centrality score of some vertices. Recomputing centrality scores from scratch every time the graph changes is expensive and, if the changes are frequent, it easily becomes computationally infeasible. A promising strategy implemented by several dynamic algorithms [39, 40, 129, 157] is to re-use previously computed information – e.g., from an initial static run – to identify the vertices *affected* by the edge update. Then dynamic algorithm can then ignore all the unaffected vertices and this results in better performance in practice.

Contribution and Outline. In the following, we describe successful attempts to scale up vertex centrality computations in three main settings: fully-dynamic graphs, top-k rankings, and approximation. In Chapter 3, we present new dynamic algorithms top-k closeness centrality ranking in fully-dynamic graphs. Our dynamic algorithms are developed on top of the static ones by Bergamini et al. [37]: after an edge update, they only process the affected vertices that could change their position in the top-k ranking, and ignore all the others. Our algorithms are *exact*, i.e., they provide the correct top-k ranking and closeness centrality scores of the top-k vertices; furthermore, they can handle multiple edge updates at a time. Our experimental results show that, compared to a static recomputation, our algorithms are up to four orders of magnitude faster for single edge updates and up to two orders of magnitude faster for batches of 100 edge updates.

In Chapter 4, we present new parallel sampling-based approximation algorithms for betweenness centrality. Approximation via sampling is a widely adopted strategy for computational problems that cannot be solved exactly within a reasonable time budget [127]. We focus on *adaptive* sampling (ADS), a particular subclass of sampling algorithms (also called *progressive* sampling algorithms) where the number of required

samples is not computed statically (e.g., from the input instance). Instead, the algorithm determines dynamically when to stop by checking a stopping condition that also depends on the data sampled so far. While non-adaptive sampling algorithms are often trivial to parallelize (e.g., by drawing multiple samples in parallel), this is not necessarily true for adaptive sampling. Indeed, checking the stopping condition for ADS constitutes a challenge as the algorithm requires to access all the data generated so far and thus mandates some form of synchronization. We introduce two new parallel ADS algorithms we call *local-frame* and *shared-frame*, both of which try to avoid expensive synchronization overheads when checking the stopping condition. We also propose a third variant called *indexed-frame* which, at the cost of additional synchronization, guarantees deterministic results. To demonstrate the effectiveness of the proposed algorithms, we turn to the state-of-the-art KADABRA betweenness approximation algorithm from Borassi and Natale [56] – note, however, that our techniques can easily be adjusted to other ADS algorithms. Experimental results show that, on 32 cores, our algorithms are up to $2.9\times$ faster than a straightforward OpenMP-based parallelization. Moreover, also due to implementation improvements and parameter tuning, our best best algorithm performs adaptive sampling $65.3\times$ faster than the existing implementation of KADABRA.

Finally, in Chapter 5, we introduce algorithms to approximate electrical centrality measures. Those measures interpret the graph under consideration as an electrical network [192] and determine the centrality of a vertex by taking into accounts paths of arbitrary lengths. We consider two well-known electrical centrality measures: electrical closeness centrality [62] – or current-flow closeness or information centrality [274] – as well as other generalizations of electrical closeness such as normalized random-walk betweenness and Kirchhoff-related indices, and forest closeness centrality [153]. A straightforward way to compute electrical closeness and related centralities is to compute the Moore-Penrose pseudoinverse $\mathbf{L}^\dagger$ of the Laplacian matrix $\mathbf{L}$ of the graph under consideration, which is as expensive as dense matrix multiplication and standard tools in practice even require cubic time [249]. Further, $\mathbf{L}^\dagger$ requires $\mathcal{O}(n^2)$ space which is not practical for large graphs. State-of-the-art approximation algorithms use the Johnson-Lindenstrauss transform [155] and require the solution of $\mathcal{O}(\log n/\varepsilon^2)$ Laplacian linear system to guarantee a relative error, which is still very expensive for large inputs. We observe that, to compute electrical closeness as well as related centralities, the only relevant part of $\mathbf{L}^\dagger$ is its diagonal $\mathrm{diag}(\mathbf{L}^\dagger)$. Our algorithm approximates $\mathrm{diag}(\mathbf{L}^\dagger)$ of a Laplacian matrix $\mathbf{L}$ corresponding to a weighted undirected graph by exploiting a strong connection between uniform spanning trees and resistance distance. It requires the solution of only one Laplacian linear system while the remaining part is based on the sampling of uniform (i.e., random) spanning trees (USTs). For small-world graphs, our algorithm achieves a $\pm\varepsilon$-approximation guarantee with high probability in $\mathcal{O}(m \log^4 n \cdot \varepsilon^{-2})$ time. Experiments show that, compared to the state of the art, our strategy is much faster and more memory-efficient, it yields a better approximation of $\mathrm{diag}(\mathbf{L}^\dagger)$, which results in a more accurate complete ranking the elements of $\mathrm{diag}(\mathbf{L}^\dagger)$. We then generalize our algorithm for electrical closeness to approximate forest closeness. This results in a nearly-linear time algorithm with an absolute probabilistic error guarantee that outperforms the state of the art in terms of both running time and solution quality.

3 Closeness Centrality Ranking in Fully-Dynamic Networks

3.1 Introduction

Closeness centrality (see Section 2.4.1) is among the oldest and most widely-studied centrality measures. The intuition of closeness is that information often travels through shortest paths and thus a vertex is important or influential if its distance to the others is short. Popular applications that require to identify such highly central vertices are influence maximization [162, 291], facility location [121, 185], game theory [142], biology [21] and sociology [34].

Formally, the closeness centrality of a vertex u is defined as the reciprocal of the average distance from u to the others and computing it exactly requires a complete exploration of the graph – i.e., a BFS on unweighted graphs or a complete run of Dijkstra's algorithm on weighted graphs. Therefore, computing the closeness centrality of every vertex of a graph requires to solve the APSP problem, which is impractical on large real-world instances.

Related Work. In practical applications, centrality is often used to find the top-k most central vertices and the actual computational effort for this problem can be substantially cheaper than APSP for large real-world networks [37, 231]. Despite their superior performance in practice, these strategies have the same asymptotics as APSP. In particular, assuming the Strong Exponential Time Hypothesis (SETH) [145], for any $\varepsilon > 0$, no algorithm can compute the vertex with highest closeness centrality in $\mathcal{O}(n^{2-\varepsilon})$ time for sparse graphs and in $\mathcal{O}(m^{2-\varepsilon})$ time for general graphs [2].

To overcome this limitation, several approximation techniques were introduced. Eppstein and Wang's method [102] selects $1 \leq k < n$ *pivot* vertices, runs a complete SSSP from each of them, and estimates the closeness of a vertex v as $\tilde{c_c}(u) := \frac{k(n-1)}{n} \sum_{i=1}^{k} d(u, v_i)$. It is shown that $1/\tilde{c_c}(u)$ is an unbiased estimator of $1/c_c(u)$, i.e., $\mathbf{E}[1/\tilde{c_c}(u)] = 1/c_c(u)$. By choosing $k \in \Theta\big(\log(n)/\varepsilon^2\big)$ pivot vertices, the algorithm is guaranteed to approximate $c_c(u)$ for each $u \in V$ within an absolute error of $\varepsilon \cdot \mathrm{diam}(G)$ with probability at least $1 - 1/n$. Cohen et al. [79] refine this approach and present a 3-approximation algorithm for closeness. Chechik et al. [73], in turn, propose an algorithm that approximates closeness centrality either within a fixed relative standard deviation (i.e., the ratio between the standard deviation and the mean) ε by performing $\mathcal{O}(\varepsilon^{-2})$ SSSPs, or within a maximum relative error of ε with probability at most $1-1/poly(n)$ by performing $\mathcal{O}(\log(n)/\varepsilon^2)$ SSSPs.

Even though these algorithms provide a precise approximation the closeness centrality scores, they often fail in ranking the top-k most central vertices exactly. This is not surprising considering the limited discriminative power of closeness centrality (as we saw in Section 2.4.1), especially for complex networks [227,

Ch. 7]. In order to provide a correct ranking, approximation algorithms would lose their competitiveness. For example, Bergamini et al. [37] argue that the algorithm by Chechik et al. [73] would require $\mathcal{O}(n^2m)$ time in unweighted graphs.

Motivation. Many real-world networks undergo continuous changes (see Section 2.7). Edge insertions and removals may impact the closeness centrality score of some vertices and recomputing the ranking after each edge update is not a scalable solution. A more efficient strategy that was shown to achieve promising results on related problems [39, 40, 129, 157] is to exploit previously computed information to update the ranking of the top-k most central vertices more efficiently in practice than a static recomputation.

Contribution. In this chapter, we present new dynamic algorithms for top-k closeness centrality ranking in fully-dynamic graphs. Our algorithms are developed upon the static algorithm by Bergamini et al. [37]: they use the information computed on an initial run of the static algorithm to efficiently update the top-k ranking after multiple edge updates. Further, they are *exact*, meaning that they provide the correct top-k ranking and the exact closeness centrality scores. Because the traditional definition of closeness centrality (Eq. (2.5), Section 2.4.1) does not apply to not strongly connected graphs, our algorithms compute the *harmonic centrality* (Eq. (2.6), Section 2.4.1), which does not have such a restriction. However, our techniques can be easily adapted to the traditional closeness centrality as well.

In Section 3.5, our experimental evaluation shows that, compared to a static recomputation, our dynamic algorithms are up to four orders of magnitude faster for single edge updates and up to two orders of magnitude faster for batches of 100 edge updates.

Bibliographic Notes. My contributions among those presented in this chapter involve the re-implementation of all the dynamic algorithms (the algorithms for single edge updates were rewritten with additional improvements that avoid to run BFScut or BFSbound when not needed), the extension of the algorithms to batch updates, and carrying out the experiments. The remaining contributions are joint work with Patrick Bisenius, Elisabetta Bergamini, and Henning Meyerhenke. A preliminary version [48] of this work was published in the Proceedings of the *Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX 2018)*. The aforementioned improvements to the dynamic algorithms for single edge updates, the extension of the dynamic algorithms to batch updates, and additional experimental results are presented in an extended version accepted for publication as a chapter in the "Massive Graph Analytics" book edited by David A. Bader and expected to be released in February 2022. Preliminary results were part of Patrick Bisenius's Master Thesis, entitled "Computing Top-k Closeness Centrality in Fully-dynamic Graphs".

3.2 Overview of Algorithms for Closeness Centrality

In this section, we provide an overview of existing algorithms for top-k closeness centrality ranking for both static and dynamic graphs.

3.2.1 Static Algorithms

The problem of finding the top-k closeness centrality has been targeted with heuristics, probabilistic approaches and exact algorithms. Proposed heuristics [186, 212] are based on sampling or they exploit the correlation between closeness and degree centrality. Okamoto et al. [229] introduced a probabilistic algorithm to compute the top-k closeness centrality ranking in $\mathcal{O}((k + n^{2/3} \log^{1/3} n)(n \log n + m))$ time on general graphs with probability at least $1 - 1/n$, which is faster than APSP if $k \in o(n)$. Their strategy is to approximate the centrality scores of every vertex in the graph with the algorithm by Eppstein and Wang [102] and then to compute the exact centrality scores for a set of "promising" candidate vertices. Olsen et al. [231] presented an exact algorithm that efficiently finds the top-k ranking by scheduling centrality computations in order to minimize the running time and by reusing intermediate results. These approaches were outperformed by Bergamini et al. [37] who proposed two functions to compute exactly the top-k vertices with highest closeness centrality on unweighted undirected graphs: NBCut and NBBound, optimized for complex and high-diameter networks, respectively. Since our dynamic algorithms are based on these two functions, we provide a detailed description of them in Section 3.3.

3.2.2 Dynamic Algorithms

Proposed dynamic algorithms for closeness centrality either maintain the scores of all vertices or the score of just one vertex. In any case, the main strategy is to run a static algorithm on the initial graph in order to reduce the computation after an edge update.

Kas et al. [157] extended the dynamic APSP algorithm by Ramalingam and Reps [248] to also update the closeness centrality scores of all the vertices of a graph. A similar strategy was adopted by Khopkar et al. [164] who developed a partially dynamic APSP algorithm that only handles vertex or edge insertions; the algorithm was extended to update the closeness and the betweenness centrality of all the vertices of a graph as well. However, these methods require to compute – and store – all the exact pairwise distances, resulting in unfeasible time and memory requirements on large networks.

Yen et al. [287] overcame this limitation by designing a data structure that efficiently identifies the vertices whose average distance to the other vertices changes after an edge update. The data structure takes a linear amount of memory w.r.t. the size of the graph but is restricted to undirected and unweighted graphs. Sariyüce et al. [259, 260, 261] present further optimizations tailored to complex networks to identify vertices whose closeness centrality score is unaffected by an edge update and therefore can be skipped. These optimizations were extended to harmonic centrality by Putman et al. [245]. Santos et al. [258] proposed a partially dynamic algorithm that only handles edge deletions. The main drawback of the aforementioned dynamic algorithms is that they require to compute the closeness centrality of every vertex of the initial graph, which is not practical on large-scale graphs.

Finally, the fully-dynamic algorithm by Ni et al. [228] address the problem of updating the closeness centrality of a single vertex under the assumption that all edge updates are already known.

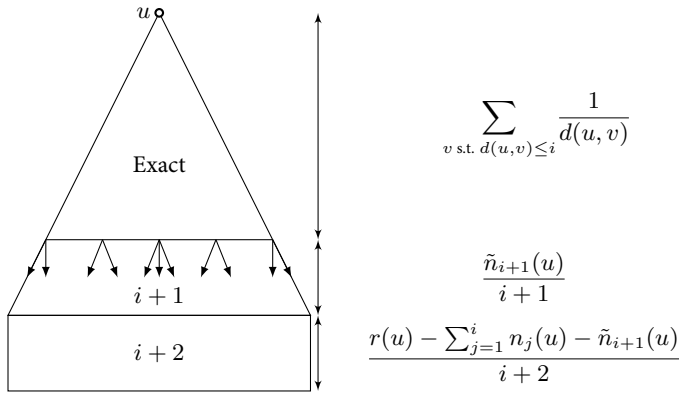


Figure 3.1: Upper bound $\widetilde{c_h}(u)$ of $c_h(u)$ computed by the NBCut algorithm. For all the vertices up to distance i from u we know their exact distance from u. Then, we assume that $\tilde{n}_{i+1}(u)$ vertices are at distance $i+1$ and that the remaining vertices reachable from u are at distance $i+2$.

3.3 Static Algorithm for Top-k Closeness Centrality

The NBCut and NBBound functions try to reduce the computational cost of running a complete BFS from each vertex in the graph by exploiting upper bounds of closeness centrality and lazy evaluation. More precisely, NBCut starts a BFS from each vertex u in the graph and interrupts the BFS as soon as it is certain that u cannot be in the top-k ranking. Conversely, NBBound does not attempt to prune BFSs, but it runs complete BFSs from a limited number of vertices.

3.3.1 The NBCut Algorithm for Complex Networks

Assume that we already computed the exact closeness centrality for at least k vertices and let h_k be the k-th highest closeness centrality. While running a BFS from a vertex u, let $\widetilde{c_h}(u)$ be an upper bound of $c_h(u)$. Clearly, if $\widetilde{c_h}(u) < h_k$, then u cannot be among the top-k vertices with highest closeness centrality, and thus we can interrupt the BFS. This possibly pruned BFS is named BFScut. In the following, we illustrate how $\widetilde{c_h}(u)$ is defined.

While running a BFScut from u, assume that we just visited all the vertices up to distance i and thus all the remaining vertices are at distance at least $i+1$. Assuming that all the unvisited vertices are at distance $i+1$ would already give us a (rather weak) bound. However, we can make further observations. Let $N_i(u)$ be the set of vertices at distance exactly i from u and let $n_i(u)$ be its cardinality; each vertex in $N_{i+1}(u)$ must have an (incoming) neighbor in $N_i(u)$, and thus we have that $N_{i+1}(u) \subseteq \bigcup_{v \in N_i(u)} N_{\text{out}}(v)$ – recall from Section 2.1 that $N_{\text{out}}(v)$ are the out-neighbors of v, i.e., $N_1(v)$. This implies that the number of vertices at distance $i+1$ from u is, in directed graphs, at most $\tilde{n}_{i+1}(u) := \sum_{v \in N_i(u)} \deg_{\text{out}}(v)$ and, in undirected graphs, at most $\tilde{n}_{i+1}(u) := \sum_{v \in N_i(u)} (\deg(v) - 1)$ – the latter holds because we can always discard v's parent in the BFS tree. The distance from u to all the remaining reachable vertices have to be at least $i+2$. These observations are summarized in the following bound on $c_h(u)$:

$$\widetilde{c_h}(u) := \sum_{v \text{ s.t. } d(u,v) \leq i} \frac{1}{d(u,v)} + \frac{\tilde{n}_{i+1}(u)}{i+1} + \frac{r(u) - \sum_{j=1}^{i} n_j(u) - \tilde{n}_{i+1}(u)}{i+2}. \tag{3.1}$$

Algorithm 1 NBCut algorithm for top-k closeness centrality in static graphs [37].

Input: A graph $G = (V, E)$, an integer $1 \leq k < n$.
Output: Top-k vertices with highest closeness centrality.

```
1:  TopK ← empty min-priority queue with keys c_h(u) and values u
2:  compute the number of reachable vertices r(u) for each vertex u ∈ V
3:  h_k ← 0
4:  for each u ∈ V sorted by an ordering O do
5:      ⟨c̃_h(u), isExact(u)⟩ ← BFScut(u, h_k)
6:      if isExact(u) and c̃_h(u) > h_k then
7:          TopK.push(c̃_h(u), u)                    ▷ Note that here c̃_h(u) = c_h(u)
8:          if TopK.size() > k then
9:              TopK.removeMin()
10:         if TopK.size() = k then
11:             h_k ← TopK.getMinKey()
12: return TopK
```

As shown in Figure 3.1, the first summand is the contribution to $c_h(u)$ due to the vertices up to distance i from u – we know their exact distance from u. In the second summand, we assume that $\tilde{n}_{i+1}(u)$ vertices are at distance $i+1$ from u[17] whereas in the third summand that all the remaining vertices reachable from u are at distance $i+2$ from u. As i increases the bound tightens: the more BFScut proceeds the fewer are the vertices whose distance from u is underestimated. If $\widetilde{c_h}(u) < h_k$ for some i, then NBCut interrupts BFScut and $d_{\text{cut}}(u) = i$ is called the *cutoff distance* for u. Otherwise, BFScut is equivalent to a BFS: it visits all the vertices reachable from u and we have that $\widetilde{c_h}(u)$ equals $c_h(u)$.

Notice that the third therm of Eq. (3.1) requires the number of vertices reachable from any vertex u. In undirected graphs, this is the size of the connected component of u and can be computed in linear time w.r.t. the graph size. In directed graphs, this is the transitive closure, which would be too expensive to compute. Thus, the authors of Ref. [37] replace $r(u)$ with an upper bound based on a topological sorting of the SCCs DAG. Furthermore, we observe that the numerator of the third summand of Eq. (3.1) cannot be greater than $n - \sum_{j=1}^{i} n_j(u) - \tilde{n}_{i+1}(u)$. Again, for simplicity, this detail is omitted in the equation but implemented in practice.

Algorithm 1 shows the NBCut algorithm. Because h_k is not known before k BFSs are completed, in Line 3 the k-th highest closeness centrality is set to 0. Then, for each vertex u in the graph, the algorithm starts a BFScut from u (Line 5), which returns the upper bound $\widetilde{c_h}(u)$ of u and `isExact`(u), i.e., whether $\widetilde{c_h}(u)$ equals $c_h(u)$. If BFScut computes the exact closeness centrality of u, then in Lines 7–11 the algorithm updates the top-k ranking. Clearly, BFScut is more likely to be interrupted if h_k is high. Hence, ideally, we want to process the vertices by decreasing closeness centrality. Two promising ordering criteria proposed by the authors of [37] are degree centrality or a measure based on the number of walks.

3.3.2 The NBBound Algorithm for High-Diameter Networks

NBBound computes an initial upper bound $\widetilde{c_h}(u)$ for each vertex $u \in V$, and stores all the vertices in a max priority queue `PrioQ` (Lines 2 and 3 of Algorithm 2). Note that $\widetilde{c_h}(u)$ is unrelated to the upper bound $\widetilde{c_h}(u)$ used by NBCut described in Section 3.3.1, for more details about $\widetilde{c_h}(u)$ see [37]. In Lines 5–7

[17] The numerator can be further tightened by taking $\min(n - n_i(u), \tilde{n}_{i+1}(u))$. However, to simplify our notation, we keep $\tilde{n}_{i+1}(u)$ in the text, but implement the better bound in practice.

Algorithm 2 NBBound algorithm for top-k closeness centrality in static graphs [37].

Input: A graph $G = (V, E)$, an integer $1 \leq k < n$.
Output: Top-k vertices with highest closeness centrality.

```
1:  TopK ← empty min-priority queue with keys c_h(u) and values u
2:  compute the initial upper bound c̃_h(u) for each u ∈ V
3:  PrioQ ← max-priority queue with keys c̃_h(u) and values all u ∈ V
4:  while PrioQ is not empty do
5:      u ← PrioQ.extractMax()
6:      if TopK.size() = k and TopK.getMinKey() > c̃_h(u) then
7:          return TopK
8:      c_h(u) ← BFSbound(u)                    ▷ Might modify c̃_h(v) for some v ∈ PrioQ
9:      Update PrioQ according to the new upper bounds
10:     TopK.push(c_h(u), u)
11:     if TopK.size() > k then
12:         TopK.removeMin()
13: return TopK
```

the algorithm extracts the vertex u in PrioQ with highest $\widetilde{c_h}(u)$ and terminates if $\widetilde{c_h}(u)$ is smaller than the k-th largest closeness centrality computed exactly so far. Otherwise, in Lines 8 and 9 the closeness centrality of u is computed exactly by the BFSbound function. BFSbound might also modify the upper bound of other vertices in PrioQ: assume we are running BFSbound from u and let x and y be any two visited vertices; from the triangle inequality if follows that $d(x, y) \geq |d(u, x) - d(u, y)|$. By assuming that $d(x, y) = |d(u, x) - d(u, y)|$ for each $x, y \in V$ we obtain an upper bound of the closeness centrality of all vertices. If for some vertex v the newly computed upper bound is smaller than their current one, then $\widetilde{c_h}(v)$ and PrioQ are updated. Eventually, the TopK ranking is updated (Lines 10–12).

3.4 Dynamic Top-k Closeness Centrality

When an edge is inserted into or removed from the graph, the top-k closeness centrality ranking might change. Our goal is to update the top-k ranking faster than re-running NBCut or NBBound. In the following, we present new dynamic algorithms to handle single edge updates and how we generalize them to batches of edge insertions and removals. We use hereafter the notation for dynamic graphs described in Section 2.7.

3.4.1 Updating the Number of Reachable Vertices

Recall that, in Eq. (3.1), the upper bound $\widetilde{c_h}(u)$ computed by BFScut requires the number of reachable vertices $r(u)$ or an upper bound of it. As mentioned in Section 3.3.1, in undirected graphs, this is the number of vertices in the connected component of u. Instead of recomputing the connected component from scratch after each edge or batch update, we update them with a simple dynamic algorithm similar to the one presented in [98]. Briefly, we compute a spanning forest of the graph. When an edge (u, v) is inserted where u and v belong to different components, we merge the components and add (u, v) to the forest. When an edge (u, v) that is part of the forest is removed, we simultaneously run two BFSs: one from u and one from v. We interrupt the two BFSs as soon as one vertex is explored by both of them. If u does not

reach v anymore, we split the components; otherwise, we add to the forest the edge connecting u's spanning tree to v's.

In directed graphs, as described in Section 3.3.1, we do not compute $r(u)$ exactly but we compute an upper bound of it based on a topological sorting of the SCCs DAG. Preliminary experiments showed that updating this after each edge update is the bottleneck of our dynamic algorithm. Hence, we replace this bound with the number of vertices in the weakly connected component of u, which still represents an upper bound of $r(u)$ and can be updated efficiently with the same strategy we use for connected components in undirected graphs.

3.4.2 Finding Affected Vertices

Assume that an edge (u, v) is inserted into or removed from a graph G. This could change the closeness centrality of some vertices. We call such vertices *affected*. More formally, the set of affected vertices is defined as:

$$A := \{x \in V : \exists y \in V \text{ s.t. } d'(x, y) \neq d(x, y)\}.$$

Clearly, if $x \in A$, then either $d'(x, u) \neq d(x, u)$ or $d'(x, v) \neq d(x, v)$. In other words, if the distance between a vertex x and both u and v remains the same after the edge update, then x cannot be affected because the BFS DAG rooted at x is unchanged. The set of affected vertices is computed by running two complete BFSs from u and v in G followed by two pruned BFSs from u and v in G' that only visit the vertices whose distance from u or to v change due to the edge update – i.e., the affected vertices. In directed graphs, the BFSs from v run on G transposed and G' since we want to determine the vertices that change their distance *to* v. A rather obvious optimization is possible when the insertion/removal of an edge connects/disconnect two (weakly) connected components: in this case we already know that the affected vertices will be the ones in the two components, and thus we only need to run two BFSs – one from u and one from v – instead of four.

3.4.3 Update After an Edge Insertion – Based on NBCut

We focus on the NBCut algorithm first. We assume that, after the initial static run of NBCut, the quantities $d_{\text{cut}}(u)$, $\widetilde{c_h}(u)$, and `isExact`(u) are know for every vertex $u \in V$. Further, we assume that the top-k vertices are stored in a min-priority queue `TopK`. A simple strategy to update the top-k vertices after an edge insertion would be to run BFScut for every affected vertex. This would already be an improvement compared to the static algorithm – unaffected vertices are skipped – but further optimizations are possible.

Algorithm 3 shows our dynamic algorithm. In Lines 3–5 we compute the number of vertices reachable after the update and the set A of affected vertices as described in Sections 3.4.1 and 3.4.2. Then, in Lines 7 and 8, all the affected top-k vertices are removed from `TopK`. In Lines 13–20 we try to avoid starting a new BFScut from each affected vertex x by computing the new bound $\widetilde{c_h}'(x)$ – more details below – and we store x in a max-priority queue `PrioQ` sorted by $\widetilde{c_h}'(x)$. Finally, in Lines 21–31, we resume the NBCut algorithm but only for the affected vertices in `PrioQ` and we interrupt it as soon as it is certain that `TopK` contains the exact top-k vertices. In the following, we illustrate how $\widetilde{c_h}'(x)$ is computed depending on $d_{\text{cut}}(x)$ and the distance from x to the newly added edge. If G is undirected, let us assume w.l.o.g. that $d(x, u) < d(x, v)$.[18]

[18] Recall that, as described in Section 3.4.2, $d(\cdot, u)$ is known from the computation of the affected vertices.

Algorithm 3 Dynamic NBCut algorithm to update the top-k vertices after an edge insertion.

Input: A graph $G = (V, E)$, an edge $(u, v) \notin E$, `TopK`.
Output: Top-k vertices with highest closeness centrality in $G' = (V, E \cup \{(u, v)\})$.

1: `PrioQ` $\leftarrow$ empty max-priority queue with keys $\widetilde{c_h}'(x)$ and values x
2: $h_k \leftarrow$ `TopK.getMinKey()`
3: compute $r'(x)$ for each $x \in V$
4: $A \leftarrow$ compute the set of affected vertices
5: compute $d(u, x), d'(u, x), d(x, v), d'(x, v)$ for each $x \in A$
6: **for** each $x \in A$ **do** ▷ Compute the new bound $\widetilde{c_h}'(x)$ for each $x \in A$
7: **if** $x \in$ `TopK` **then**
8: `TopK.remove`(x)
9: $h_k \leftarrow 0$
10: $y \leftarrow u$
11: **if** G is undirected **then**
12: $y \leftarrow \operatorname{argmin}_{z \in \{u,v\}} d(x, z)$
13: **if not** `isExact`(x) **and** $d_{\text{cut}}(x) < d(x, y)$ **then**
14: $\widetilde{c_h}'(x) \leftarrow \widetilde{c_h}(x) - \frac{r(x)}{d_{\text{cut}}(x)+2} + \frac{r'(x)}{d_{\text{cut}}(x)+2}$ ▷ x is a far-away vertex
15: **else if not** `isExact`(x) **and** $d_{\text{cut}}(x) = d(x, y)$ **then**
16: $\widetilde{c_h}'(x) \leftarrow \widetilde{c_h}(x) - \frac{r(x)-r'(x)+1}{d_{\text{cut}}(x)+2} + \frac{1}{d_{\text{cut}}(x)+1}$ ▷ x is a boundary vertex
17: **else**
18: $\widetilde{c_h}'(x) \leftarrow \widetilde{c_h}(x) + \sum_{i=1}^{i_{\max}} \frac{1}{i+d(x,y)}(n'_i(y) - n_i(y))$ ▷ Distance-based bound
19: $d_{\text{cut}}(x) \leftarrow \infty$
20: `PrioQ.push`(x)
21: **while** `PrioQ` is not empty **do**
22: $x \leftarrow$ `PrioQ.extractMax()`
23: **if** $\widetilde{c_h}'(x) \geq h_k$ **then**
24: $\langle \widetilde{c_h}'(x), \texttt{isExact}(x) \rangle \leftarrow \text{BFScut}(x, h_k)$ ▷ We have to run a new BFScut from x
25: **else break**
26: **if** `isExact`(x) **and** $\widetilde{c_h}'(x) \geq h_k$ **then**
27: `TopK.push`$(\widetilde{c_h}'(x), x)$ ▷ Note that here $\widetilde{c_h}'(x) = c_h(x)$
28: **if** `TopK.size()` $> k$ **then**
29: `TopK.removeMax()`
30: **if** `TopK.size()` $= k$ **then**
31: $h_k \leftarrow$ `TopK.getMinKey()`
32: update $\widetilde{c_h}(x), r(x)$, and $n_i(u)$ to $\widetilde{c_h}'(x), r'(x)$, and $n'_i(x)$, respectively for each $x \in V$
33: **return** `TopK`

Far-away vertices. Let x be an affected vertex such that the BFScut rooted at x has been interrupted at the cutoff distance $d_{\text{cut}}(x)$. If $d(x, u) > d_{\text{cut}}(x)$ and $r'(x) = r(x)$, then $\widetilde{c_h}(x)$ is still valid because, as shown in Figure 3.2a, u has not been by visited by the BFScut rooted at x, and therefore the new edge (u, v) does not affect $\widetilde{c_h}(x)$. If $d(x, u) > d_{\text{cut}}(x)$ but $r'(x) \neq r(x)$ – i.e., the insertion increased the number of vertices reachable from x – then in Eq. (3.1) we simply replace $r(x)$ with $r'(x)$, and we obtain:

$$\widetilde{c_h}'(x) = \widetilde{c_h}(x) - \frac{r(x)}{d_{\text{cut}}(x) + 2} + \frac{r'(x)}{d_{\text{cut}}(x) + 2}.$$

We call u and v *far-away* vertices w.r.t. x, and they are handled in Line 14 of Algorithm 3.

Boundary vertices. If $d(x, u)$ equals $d_{\text{cut}}(x)$, then the bound $\widetilde{c_h}(x)$ is affected because, as shown in Figure 3.2b, $u \in N_{d_{\text{cut}}(x)}(x)$ and the (out)-degree of u increases by one. Recall from Section 3.3.1 that in directed graphs $\tilde{n}_{i+1}(x) := \sum_{y \in N_i(x)} \deg_{\text{out}}(y)$ while in undirected graphs $\tilde{n}_{i+1}(x) := \sum_{y \in N_i(x)} (\deg(x) -$

1). Hence, the upper bound $\tilde{n}'_{d_{\text{cut}}(x)+1}(x)$ of the number of vertices at distance $d_{\text{cut}}(x)+1$ from x after the insertion equals $\tilde{n}_{d_{\text{cut}}(x)+1}(x)+1$. We can compute the new bound $\widetilde{c_h}'(x)$ from the old one as follows:

$$\begin{aligned}\widetilde{c_h}'(x) - \widetilde{c_h}(x) &= \frac{\tilde{n}_{d_{\text{cut}}(x)+1}(x)+1}{d_{\text{cut}}(x)+1} + \frac{r'(u) - \sum_{j=1}^{d_{\text{cut}}(x)} n_j(x) - \tilde{n}_{d_{\text{cut}}(x)+1}(x) - 1}{d_{\text{cut}}(x)+2} - \\ &\quad \frac{\tilde{n}_{d_{\text{cut}}(x)+1}(x)}{d_{\text{cut}}(x)+1} - \frac{r(u) - \sum_{j=1}^{d_{\text{cut}}(x)} n_j(x) - \tilde{n}_{d_{\text{cut}}(x)+1}(x)}{d_{\text{cut}}(x)+2} \\ &= \frac{1}{d_{\text{cut}}(x)+1} + \frac{r'(x) - 1 - r(x)}{d_{\text{cut}}(x)+2}.\end{aligned}$$

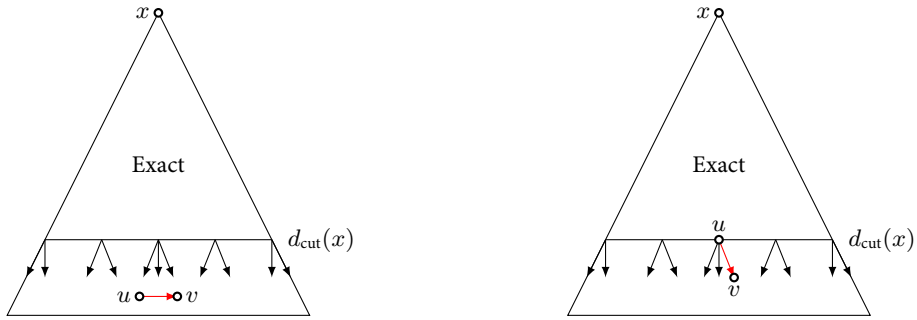
Vertices such as u are called *boundary* vertices w.r.t. x and they are handled in Line 16 of Algorithm 3.

Distance-based Bounds. Let x be an affected vertex such that $d(x,u) < d_{\text{cut}}(x)$ and let y be any vertex such that $d'(x,y) < d(x,y)$ – if x is affected then such a vertex y has to exist. Since all new shortest paths from x have to go through (u,v) (and thus through u), we can write $d'(x,y)$ as $d'(x,u) + d'(u,y) = d(x,u) + d'(u,y)$. This holds because the distance from x to u cannot change as a consequence of the insertion of (u,v) – recall our assumption at the beginning of this section that $d(x,u) < d(x,v)$ if G is undirected. As for $d(x,y)$ before the insertion there are two options: either u was part of a shortest path from x to y, and thus $d(x,y) = d(x,u)+d(u,y)$, or there was a shortest path from x to y not going through u, and therefore $d(x,y) < d(x,u)+d(u,y)$. In any case, $d(x,y) \leq d(x,u)+d(u,y)$. Putting this together we have that $\frac{1}{d'(x,y)} - \frac{1}{d(x,y)} \leq \frac{1}{d(x,u)+d'(u,y)} - \frac{1}{d(x,u)+d(u,y)}$ and thus:

$$\begin{aligned}c'_h(x) - c_h(x) &= \sum_{y \in V} \left(\frac{1}{d'(x,y)} - \frac{1}{d(x,y)} \right) \\ &\leq \sum_{y \in V} \left(\frac{1}{d(x,u) + d'(u,y)} - \frac{1}{d(x,u) + d(u,y)} \right) \\ &\leq \sum_{i=1}^{i_{\max}} \frac{1}{i + d(x,u)} (n'_i(u) - n_i(u)),\end{aligned} \tag{3.2}$$

where $i_{\max}$ is the highest value of i such that $n'_i(u) > n_i(u)$. Eq. (3.2) implies that, if $\widetilde{c_h}(x)$ is an upper bound of $c_h(x)$, then $\widetilde{c_h}'(x) := \widetilde{c_h}(x) + \sum_{i=1}^{i_{\max}} \frac{1}{i+d(x,u)}(n'_i(u) - n_i(u))$ is an upper bound of $c'_h(x)$. The values $(n'_i(u) - n_i(u))$ can easily be computed with two BFSs from u: one in G and one in G' – we integrate this in the BFSs we run to find the affected vertices, see Section 3.4.2. Notice that the computation of the new bound requires $i_{\max} \in \mathcal{O}(\text{diam}(G))$ operations. Since the diameter of complex networks is very small – often assumed to be logarithmic in n, or constant [6, 271] – this is much faster than running BFScut, which takes up to $\mathcal{O}(n+m)$ time. For each affected vertex x for which u is neither far-away nor boundary, we compute the new upper bound as in Eq. (3.2) – Line 18 of Algorithm 3. In Line 19, we set $d_{\text{cut}}(x)$ to ∞ to indicate that the new bound $\widetilde{c_h}'(u)$ is not a result of NBCut anymore and thus it should not be used in future to identify far-away or boundary vertices.

Handling multiple edge insertions – based on NBCut. A straightforward strategy to handle multiple edge insertions would be to run Algorithm 3 for each individual edge insertion, which is unnecessarily



(a) u and v are far-away vertices for x.

(b) u is a boundary vertex for x.

Figure 3.2: Example of far-away (left) and boundary (right) vertices.

expensive because it recomputes the top-k ranking after each edge insertion – we are interested in the top-k ranking after multiple edge insertions. A more efficient approach is to modify Algorithm 3 so that it computes all the affected vertices $x \in A$ and their new bounds $\widetilde{c_h}'(x)$. The top-k ranking is recomputed only once afterwards. This can be done as follows: for each edge insertion, we update the bounds of the affected vertices (Lines 3–20); then, in Line 20, we push x into `PrioQ` only if $x \notin$ `PrioQ`, otherwise, we just update its key. Finally, we recompute the top-k ranking by running Lines 21–31.

3.4.4 Update After an Edge Insertion – Based on NBBound

Algorithm 4 shows the pseudocode of our dynamic algorithm for single edge insertions based on NBBound. Let us assume that NBBound has been executed on the initial graph. Recall from Section 3.3.2 that NBBound runs complete BFSs until it finds k vertices whose exact closeness centrality is higher than the upper bounds on the remaining vertices. Thus, there is no cutoff distance that we can exploit to identify far-away or boundary vertices, but we can still make some considerations. First, for unaffected vertices $x \notin A$, $c_h(x)$ and $\widetilde{c_h}(x)$ are still valid and do not need to be changed. Further, the distance-based bounds described in Section 3.4.3 can be applied to NBBound as well (Line 13 of Algorithm 4) and, if there are vertices whose new bound is higher than the k-th highest closeness centrality value, we run a BFSbound from them (Line 19). Similarly to the static NBBound, we interrupt the algorithm when there are no affected vertices x whose upper bound $\widetilde{c_h}(x)$ is higher than the k-th highest closeness centrality (Line 18).

Handling multiple edge insertions – based on NBBound. Similarly to the NBCut case, Algorithm 4 can be easily adapted to handle multiple edge insertions: Lines 3 and 4 are executed for each edge insertion and then Lines 5–24 are executed only once.

3.4.5 Update After an Edge Removal

Edge removals are, in some sense, easier to handle than edge insertions: closeness centrality can only decrease as a consequence of a removal – distances can only increase – and thus nothing needs to be done if none of the top-k vertices are affected. For edge insertions, in turn, there could be some vertex that increases its closeness centrality and "overtakes" the previous k-th highest closeness centrality. Further, for an affected vertex x, its previous upper bound $\widetilde{c_h}(x)$ is still valid – although it might be less tight.

Algorithm 4 Dynamic NBBound algorithm to update the top-k vertices after an edge insertion.

Input: A graph $G = (V, E)$, an edge $(u, v) \notin E$, TopK.
Output: Top-k vertices with highest harmonic centrality in $G' = (V, E \cup \{(u, v)\})$.

```
1:  PrioQ ← empty max-priority queue with keys c̃_h(x) and values x
2:  h_k ← TopK.getMinKey()
3:  A ← compute the set of affected vertices
4:  compute d(u,x), d'(u,x), d(x,v), d'(x,v) for each x ∈ A
5:  for each x ∈ A do                          ▷ Compute the new bound c̃_h'(x) for each x ∈ A
6:      if x ∈ TopK then
7:          TopK.remove(x)
8:          h_k ← 0
9:      y ← u
10:     if G is undirected then
11:         y ← argmin_{z∈{u,v}} d(x,z)
12:     /* update the bounds of the affected vertices */
13:     c̃_h'(x) ← c̃_h(x) + Σ_{i=1}^{i_max} 1/(i+d(x,y)) (n'_i(y) − n_i(y))
14:     PrioQ.push(c̃_h'(x), x)
15: while PrioQ is not empty do
16:     x ← PrioQ.extractMax()
17:     if TopK.size() = k and c̃_h(x) < h_k then
18:         return TopK
19:     c_h(x) ← BFSbound(x)                   ▷ Might modify c̃_h(z) for some z ∈ PrioQ
20:     Update PrioQ according to the new upper bounds
21:     TopK.push(c_h(x), x)
22:     if TopK.size() > k then
23:         TopK.removeMin()
24: return TopK
```

Algorithm 5 shows our dynamic algorithm for single edge removals based on NBCut. If we know the exact $c_h(x)$ of an affected vertex x before the removal, this becomes an upper bound on the closeness centrality of x in G'; then, we set isExact(x) to false and, if $x \in$ TopK, we remove it from TopK (Lines 6–8). Once this is done, the algorithm processes the vertices in the max-priority queue PrioQ sorted by $\widetilde{c_h}(x)$ and terminates when TopK contains k vertices and the highest $\widetilde{c_h}(x) \in$ PrioQ is smaller than the k-th highest closeness centrality (Line 14). Note that this strategy works for both NBCut and NBBound, with the only difference that, in Line 17, we either run BFScut or BFSbound.

Handling multiple edge removals. As with multiple edge insertions, multiple edge removals are handled in two steps. First, all the affected vertices are computed by executing Lines 2–9 for each edge removal. Then, the top-k ranking is updated by running Lines Lines 10–25 only once.

3.4.6 Time Complexity and Memory Requirements

Updating the number of reachable vertices described in Section 3.4.1 and computing the set of affected vertices take $\mathcal{O}(n + m)$ time in the worst case (the time complexity of a complete BFS). Then, the algorithms described in Sections 3.4.3–3.4.5 have to run, in the worst case, one BFScut (or BFSbound, if we are considering the algorithm based on BFSbound) for each affected vertex. Since worst-case time complexity of both BFScut and BFSbound is $\mathcal{O}(n + m)$ [37], the running time of our dynamic algorithms is

Algorithm 5 Dynamic NBCut algorithm to update the top-k vertices after an edge removal.

Input: A graph $G = (V, E)$, an edge $(u, v) \in E$, TopK.
Output: Top-k vertices with highest closeness centrality in $G' = (V, E \setminus \{(u, v)\})$.

```
1:  h_k ← TopK.getMinKey()
2:  compute r'(x) for each x ∈ V
3:  A ← compute the set of affected vertices
4:  compute d(u,x), d'(u,x), d(x,v), d'(x,v) for each x ∈ A
5:  for each x ∈ A do
6:      isExact(x) ← false
7:      if x ∈ TopK then
8:          TopK.remove(x)
9:          h_k ← 0
10: PrioQ ← max-priority queue with keys c̃_h'(x) and values all x ∈ V \ TopK
11: while PrioQ is not empty do
12:     x ← PrioQ.extractMax()
13:     if c̃_h(x) < h_k then
14:         break
15:     if not isExact(x) then
16:         /* we run a new BFScut from x */
17:         ⟨c̃_h'(x), isExact(x)⟩ ← BFScut(x, h_k)
18:     if isExact(x) and c̃_h'(x) > h_k then
19:         TopK.push(c̃_h'(x), x)                    ▷ Note that here c̃_h(x) = c_h(x)
20:     if TopK.size() > k then
21:         TopK.removeMin()
22:     if TopK.size() = k then
23:         h_k ← TopK.getMinKey()
24: update c̃_h(x), r(x), and n_i(u) to c̃_h'(x), r'(x), and n'_i(x), respectively for each x ∈ V
25: return TopK
```

$\mathcal{O}(|A|(n + m))$ in the worst case, where $|A|$ is the number of affected vertices. However, as shown in Section 3.5, in real-world networks the actual number of calls to BFScut is usually a small fraction of the total number of affected vertices – which is, in turn, often only a small fraction of the total number of vertices.

Concerning memory requirements, for each vertex $u \in V$ our algorithms need to store the bound $\widetilde{c_h}(u)$, isExact(u), the number of reachable vertices $r(u)$, the cutoff distance $d_{\text{cut}}(u)$, and the TopK priority queue with the k vertices with highest closeness centrality. This takes only $\Theta(n)$ memory, which is asymptotically the same as the static top-k algorithms.

3.5 Experimental Results

In this section, we present the results of our experimental study. We study the impact of the optimizations proposed in Section 3.4.3 and the algorithmic speedup (ratio between the sequential running times, see Section 2.8) of our dynamic algorithms on the static ones.

3.5.1 Experimental Setup

We test our algorithms on numerous directed and undirected real-world complex and high-diameter networks we retrieved from the public repositories SNAP [181], KONECT [173], OpenStreetMap [83], and from

the 9th DIMACS Implementation Challenge [87]. The graphs are listed in Tables B.1–B.4 in Appendix B.1. For high-diameter networks, we only test road networks because they can be retrieved easily from public repositories, but we are confident that our dynamic algorithms can handle other types of high-diameter networks as well without significant difference in performance. To test more undirected networks, we read as undirected three of the directed road networks ("seychelles", "comores", and "liechtenstein") by ignoring the direction of the edges.

For each tested graph, we either add or remove 100 batches of edges selected uniformly at random and then run both the static and the dynamic algorithms. For edge insertions, we remove from the original graph an equivalent number of edges selected uniformly at random before running the algorithms and then we add them one-by-one, whereas for removals we just remove edges uniformly at random. Due to time constraints, we only run the static algorithms once every 10 edge updates – this does not affect the results considerably, since the running time of the static algorithms is always approximately the same.

Implementation and settings. Since NBCut outperforms NBBound on complex networks, for our experiments on complex networks we only use NBCut and our new dynamic algorithms based on it – i.e., Algorithms 3 and 5. Similarly, we only use NBBound (Algorithm 2) and the dynamic algorithms based on it (Algorithm 4) for our experiments on road networks. We recall that, for performance reasons, our dynamic algorithms based on NBCut for directed graphs use the number of vertices in the WCCs instead of the bound originally proposed in [37]. However, in order to make a fair comparison with previous work, for the static case we use the original bound. Further, we recall that all algorithms are *exact*, i.e., the find the k vertices with highest closeness centrality and their exact scores, so they only differ by their running time and not in the results they find.

The machine we use for our experiments is a shared-memory Linux server equipped with 192 GiB of RAM and an Intel Xeon Gold 6126 with 2×12 cores, of which we use only one because we execute all algorithms sequentially – i.e., using a single thread. To ensure reproducibility, all experiments are managed by the SimexPal software [14]. The code has been written in C++ on top of the open-source NetworKit framework [273].

3.5.2 Speedups on Recomputation

Dynamic Complex Networks. Table 3.1a shows the average number of affected vertices over single edge insertions and the percentage of affected vertices that are far-away, boundary, or that have been updated using the distance bound on undirected graphs (results are for $k = 10$). The average number of affected vertices is at most 4.36% of the total number of vertices and for the great great majority of them $\widetilde{c_h}'(\cdot)$ can be computed in constant time as they are either far-away or boundary. The "BFScuts" column shows the percentage of affected vertices from which a new BFScut is executed after $\widetilde{c_h}'(\cdot)$ has been computed for all the affected vertices. For single edge insertions, our dynamic algorithm always run BFScut for less than 0.01% of the number of affected vertices, while for batches of 10 and 100 edge insertions the number of BFScut is at most 0.08% and 1.0% the number of vertices in the graph, respectively. The higher number of BFScut the batch size can be explained by the fact that each edge insertion weakens the bounds $\widetilde{c_h}'(\cdot)$ and thus reduces the possibility for Algorithm 3 to terminate early at Line 25.

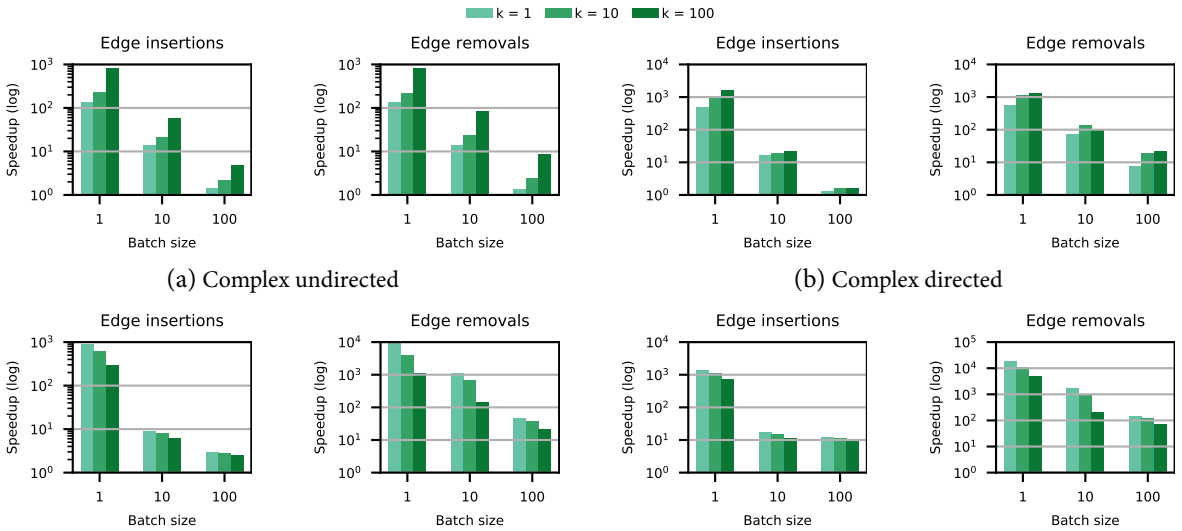
Table 3.1: Impact of optimizations in complex networks for $k = 10$, averaged over 100 batches of 1, 10, 100 random edge insertions using the geometric mean. The column "Aff." shows the average number of vertices affected by a single edge insertion, while "Aff. (%)" the percentage of affected vertices over the total number of vertices of the graph. The next three columns report the percentage of affected vertices that are far-away, boundary, or that are updated using the distance bound. The last column shows the percentage of affected vertices for which a BFScut has been run.

(a) Undirected networks

Network Batch size	Aff.	Aff. (%)	F (%)	B (%)	D (%)	BFScuts (%) 1	10	100
petster-cat-household	3,505	3.33	99.45	0.32	0.23	< 0.01	0.03	0.22
petster-catdog-household	14,518	4.36	89.09	10.68	0.23	< 0.01	0.04	0.22
petster-cat-friend	26	0.01	84.69	14.16	1.15	< 0.01	< 0.01	< 0.01
petster-friendships-cat	30	0.02	86.89	12.07	1.05	< 0.01	< 0.01	< 0.01
petster-friendships-dog	1,440	0.34	88.76	9.58	1.66	< 0.01	0.01	0.17
petster-dog-friend	446	0.10	24.59	66.61	8.80	< 0.01	0.02	0.17
petster-catdog-friend	1,620	0.26	92.43	6.37	1.20	< 0.01	< 0.01	0.15
higgs-twitter-social	10,342	2.26	84.54	13.59	1.87	< 0.01	0.08	1.05
petster-carnivore	269	0.04	66.75	27.55	5.70	< 0.01	0.02	0.18

(b) Directed networks

Network Batch size	Aff.	Aff. (%)	F (%)	B (%)	D (%)	BFScuts (%) 1	10	100
wikipedia_link_li	2,531	5.15	0.10	0.30	99.60	3.51	32.53	94.26
cit-HepPh	615	1.78	65.31	4.97	29.73	0.37	3.66	29.08
slashdot-zoo	1,888	2.39	51.99	11.59	36.43	0.53	7.53	41.64
wiki_talk_sv	1,397	0.23	94.37	5.27	0.36	< 0.01	1.16	5.31
munmun_twitter_social	139	0.03	47.80	20.74	31.46	< 0.01	< 0.01	0.04
wiki_talk_ja	1,475	0.14	15.98	18.33	65.68	0.05	0.47	2.61
wikipedia_link_gu	191	0.63	11.43	12.11	76.46	0.47	4.73	25.54
web-NotreDame	6,332	1.94	91.81	3.33	4.86	0.08	0.36	3.74
digg-friends	708	0.02	66.00	18.05	15.95	< 0.01	0.03	0.26



(a) Complex undirected (b) Complex directed

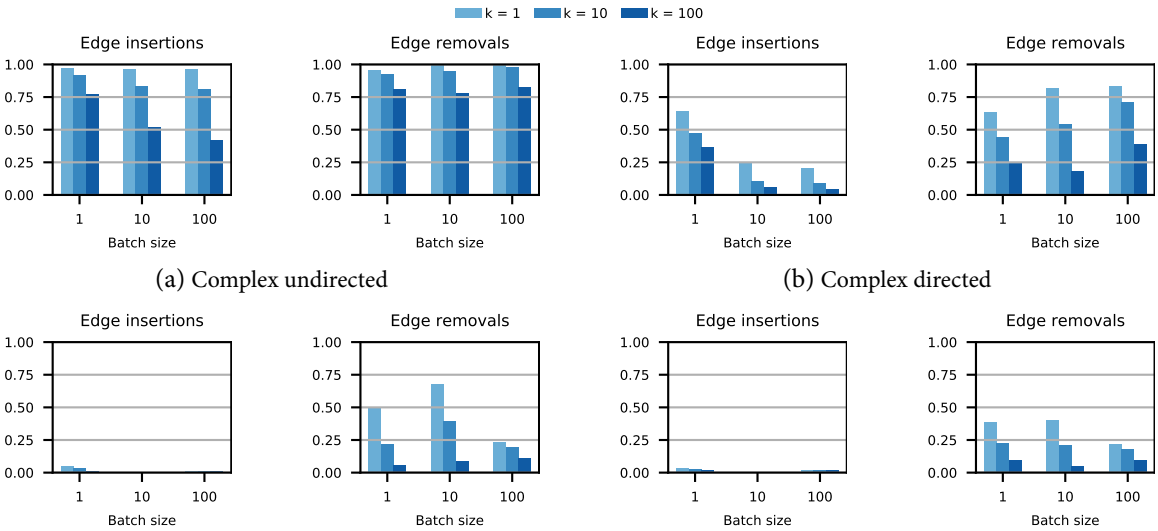
(c) High-diameter undirected (d) High-diameter directed

Figure 3.3: Geometric mean of the average speedups over all tested networks, for different values of k and batch sizes. Figures 3.3a and 3.3b show the results for complex networks, whereas Figures 3.3c and 3.3d show the results for road networks. Detailed numbers can be found in Sections B.2.1 and B.2.2.

Table 3.1b shows the results for the directed case. Compared to undirected graphs, insertions typically affect smaller portions of the graph – average values are at most 5.15%. However, a smaller percentage of affected vertices is far-away, probably because most of the vertices that are very close to the inserted edge are affected. Therefore, the new bounds $\widetilde{c_h}'(\cdot)$ of the affected vertices are less tight than in the undirected case and Algorithm 3 runs BFScut more often – up to 3.5% for single edge insertions and up to 94.3% for batches of 100 edge insertions.

Figures 3.3a and 3.3b summarize the speedup of our dynamic algorithm on the static one for complex networks while Figures 3.4a and 3.4b show the fraction of time spent by the dynamic algorithm computing $\widetilde{c_h}'(\cdot)$ of the affected vertices w.r.t. the algorithm's overall running time (i.e., computing $\widetilde{c_h}'(\cdot)$ and the new top-k ranking) for complex networks. Detailed results for insertions in complex networks are reported in Tables B.5 and B.9. For undirected networks, the geometric mean of the speedups over 100 single edge insertions are always at least in the double-digit range for every instance. Also, the speedups grow for bigger values of k, reaching an average speedup (over all tested undirected instances) of $827.8\times$ for $k = 100$. A possible explanation for this pattern is that separating the top-k vertices with highest closeness centrality from the others is harder for larger values of k and the dynamic algorithm does this faster because it exploits precomputed values of $\widetilde{c_h}'(\cdot)$. This is also clear from the left plot of Figure 3.4a, as updating the ranking becomes more expensive for larger values of k.

For larger batches of edge insertions, our dynamic algorithm yields diminishing returns. This is expected because, the more the graph changes, the less accurate the bounds $\widetilde{c_h}'(\cdot)$ become. As we can clearly see from Figure 3.4a, inaccurate bounds jeopardize the performance of the dynamic algorithm since the recomputation of the ranking becomes more expensive as we increase the batch size. Nevertheless, averaging over all



(a) Complex undirected

(b) Complex directed

(c) High diameter undirected

(d) High diameter directed

Figure 3.4: Geometric mean of the average time spent by the dynamic algorithm computing $\widetilde{c_h}'(\cdot)$ w.r.t. the total time spent in updating the top-k nodes with highest closeness centrality over all the tested networks, for different values of k and batch sizes. Figures 3.4a and 3.4b show the results for complex networks, whereas Figures 3.4c and 3.4d show the results for road networks.

the undirected networks, with $k = 1$ the dynamic algorithm handles batches of 100 edge insertions $1.5\times$ faster than the static algorithm, $2.2\times$ faster with $k = 10$, and $4.9\times$ faster with $k = 100$.

Concerning directed graphs, the average speedups for single edge insertions are better than for the undirected case: the average speedup over all the tested directed networks are $465.7\times$ for $k = 1$, $978.4\times$ for $k = 10$ and $1{,}583.8\times$ for $k = 100$. This can be explained by the smaller number of affected vertices in directed networks (see Table 3.1b). The performance of the dynamic algorithm for directed networks seems to be more susceptible to the batch size than for undirected networks; for example, for $k = 10$ and 10 edge insertions, the average speedup is $19.4\times$ on directed networks and $21.2\times$ on undirected networks, while for 100 edge insertions the average speedup is $1.6\times$ and $2.2\times$ respectively. Similarly to the undirected case, this is likely due to the inaccuracy of the updated bounds as the dynamic algorithm spends the vast majority of its running time in recomputing the ranking (see the left plot in Figure 3.4b).

Speedups for edge removals on complex networks are summarized in the right of Figures 3.3a and 3.3b – detailed results are reported in Tables B.6 and B.10. Interestingly, for directed graphs, our algorithm handles removals faster than insertions, whereas the performance does not change substantially w.r.t. insertions in the undirected case. In general, for shortest-path based problems insertions are easier to handle than removals; for example, pairwise distances can be updated in $\mathcal{O}(n^2)$ time after an edge insertion, but not after an edge removal [88]. In our case, we know that removals can only *decrease* centrality. Thus, the upper bounds of the centralities are still valid and, if none of the top-k vertices is affected, nothing needs to be done. In insertions, on the contrary, any vertex could *increase* its centrality and become one of the top-k. If the number of affected vertices is small – as in most cases for directed graphs, see Table 3.1b – it is quite unlikely that a top-k vertex is affected. This happens more often in undirected graphs, where often a larger number of vertices are affected. Consequently, as shown in Figures 3.4a and 3.4b, updating the ranking

is less expensive after an edge removal than after an edge insertion. As for insertions, speedups increase with k and decrease with the batch size: for single edge removals, the geometric mean of the speedups in undirected graphs is $135.5\times$ for $k = 1$, $223.8\times$ for $k = 10$, and $845.9\times$ for $k = 100$, whereas for directed graphs it is $548.2\times$ for $k = 1$, $1{,}112.7\times$ for $k = 10$, and $1{,}299.3\times$ for $k = 100$. For edge removals, the decrease of the speedups w.r.t. the batch size is less severe than for edge insertions; the bounds on the closeness centrality after a batch of edge removals are likely to be tighter than the ones computed after a batch of edge insertions, allowing Algorithm 5 to terminate often earlier than Algorithm 3.

Dynamic Road Networks. Figures 3.3c and 3.3d summarize the speedups for directed and undirected road networks, respectively – detailed results are shown in Tables B.7, B.8, B.11 and B.12 – whereas Figures 3.4c and 3.4d show the fraction of time spent by the dynamic algorithm updating $\widetilde{c_h}'(\cdot)$ w.r.t. the algorithm's overall running time. As for complex networks, speedups in road networks are generally higher in the directed case and they decrease with the batch size. However, differently from complex networks, speedups generally decrease with k: If k is large, it is also more likely that some affected vertices are either among the top-k or "overtake" one of the top-k, making the algorithm run BFSbound more often and thus slowing it down. The bar plots in the left of Figures 3.4c and 3.4d also show that the running time of the dynamic algorithm is dominated by the recomputation of the ranking, which is expected because closeness centrality distinguishes vertices in complex networks less efficiently than in high-diameter networks [227, Ch. 7]. Nevertheless, even for $k = 100$, for a single edge insertion the dynamic algorithm is on average $417.1\times$ faster than a static recomputation in undirected road networks and $660.1\times$ faster in directed road networks. Furthermore, regarding multiple edge insertions, the dynamic algorithm is faster than a static recomputation on all the considered instances and for all the considered batch sizes.

Results for removals are significantly better: for $k = 1$ and single edge removals the dynamic algorithm is on average $12{,}505.6\times$ faster in the undirected case and $18{,}236.9\times$ in the directed one, while for $k = 100$ the speedups are $1{,}950.4\times$ and $4{,}252.6\times$, respectively. As we described previously, the impact on the ranking due to edge removals is in general lower than edge insertions, so the dynamic algorithm spends less time on recomputing the ranking. This is also clear from the bar plots in the right of Figures 3.4c and 3.4d: after edge removals the fraction of time spent by the dynamic algorithm updating $\widetilde{c_h}'(\cdot)$ w.r.t. the total running time is greater than after edge insertions. Finally, concerning batches of 10 [100] edge removals, the dynamic algorithm is on average two to three [one to two] orders of magnitude faster than a static recomputation.

3.6 Conclusions

In this chapter, we addressed the problem of preserving an exact top-k ranking of the vertices with highest closeness (including their exact score). We implemented batch-dynamic algorithms for top-k closeness centrality tailored to both complex and high-diameter networks. Our dynamic algorithms are developed on top of the static algorithms for top-k closeness centrality by Bergamini et al. [37]; in particular, they maintain upper bounds $\widetilde{c_h}(\cdot)$ on the closeness centrality of every vertex, which accelerates the computation of the top-k vertices in practice. By re-using such bounds as well as other precomputed information, we are able to significantly reduce the number of operations required to update the most central vertices in the graph after multiple graph updates.

As a result, for single edge updates, we achieve high speedups on a static recomputation, in line with results obtained by other dynamic algorithms for related problems [39, 40, 129, 157] – confirming that efforts in developing dynamic algorithms are well spent. As we increase the batch sizes, our strategy yields diminishing returns. This is expected because, the more the graph changes, the less precise the upper bounds $\widetilde{c_h}'(\cdot)$ become and this impacts the performance of our dynamic algorithms. Nevertheless, experimental data show that, averaging results over the tested instances, our dynamic algorithms are always faster than a static recomputation. In contrast to most existing algorithms for updating shortest-path based centralities that require $\mathcal{O}(n^2)$ additional memory, the techniques we propose require an amount of memory that is only linear in n. Although storing more data (e.g., the distances computed during BFScut on the initial graph) might lead to even higher speedups, a quadratic memory footprint would not allow us to target networks with millions of vertices. An interesting question is whether the memory requirements of other dynamic algorithms for related problems can be reduced by using techniques similar to the ones we presented in this chapter.

A possible direction for future research is the extension of our strategies for batch updates to other centrality measures such as betweenness – for which a static algorithm for finding the top-k vertices with highest betweenness has already been proposed in Ref. [180]. Thus, an interesting question is whether this algorithm can be further improved and/or efficiently updated in fully-dynamic networks.

4 Parallel Approximation of Betweenness Centrality

4.1 Introduction

With large datasets being the rule and not the exception today, approximation is frequently applied [127] to problems that cannot be solved exactly within a desired time budget, including polynomial-time problems [56]. We focus on a particular subclass of approximation algorithms: *sampling algorithms*. They sample data according to some (usually algorithm-specific) probability distribution, perform some computation on the sample and induce a result for the full dataset.

More specifically, we consider *adaptive* sampling (ADS) algorithms – also called *progressive* sampling algorithms. Here, the number of samples that are required is not statically computed (e.g., from the input instance) but also depends on the data that has been sampled so far. While non-adaptive sampling algorithms can often be parallelized trivially by drawing multiple samples in parallel, adaptive sampling constitutes a challenge for parallelization: checking the stopping condition of an ADS algorithm requires access to all the samples drawn so far and thus mandates some form of synchronization.

Motivation and Contribution. Our initial motivation was a parallel implementation of the sequential state-of-the-art approximation algorithm KADABRA [56] for betweenness centrality (c_b) approximation. Betweenness is a very popular centrality measure in network analysis, see Sections 2.4.3 and 4.2.2 for more details. To the best of our knowledge, parallel adaptive sampling has not received a generic treatment yet. Hence, we propose techniques to parallelize ADS algorithms in a generic way, while scaling to large numbers of threads. While we turn to KADABRA to demonstrate the effectiveness of the proposed algorithms, our techniques can be adjusted easily to other ADS algorithms.

We introduce two new parallel ADS algorithms, which we call *local-frame* and *shared-frame*. Both algorithms try to avoid extensive synchronization when checking the stopping condition. This is done by maintaining multiple copies of the sampling state and ensuring that the stopping condition is never checked on a copy of the state that is currently being written to. *Local-frame* is designed to use the least amount of synchronization possible – at the cost of an additional memory footprint of $\Theta(n_s)$ per thread, where n_s denotes the size of the sampling state. This algorithm performs only atomic `load-acquire` and `store-release` operations for synchronization, but no expensive read-modify-write operations (like `CAS` or `fetch-add`). *Shared-frame*, in turn, aims instead at meeting a desired trade-off between memory footprint and synchronization overhead. In contrast to *local-frame*, it requires only $\Theta(1)$ additional memory per thread, but uses atomic read-modify-write operations (e.g., `fetch-add`) to accumulate samples. We also propose the deterministic *indexed-frame* algorithm; it guarantees that the results of two different executions is the same for a fixed random seed, regardless of the number of threads.

Algorithm 6 Generic Adaptive Sampling

```
1: /* Variable initialization */
2: d ← new sampling state structure
3: d.data ← (0, ..., 0)                                  ▷ Sampled data
4: d.num ← 0                                             ▷ Number of samples
5: /* Main loop */
6: while not checkForStop(d) do
7:     d.data ← d.data ∘ sample()
8:     d.num ← d.num + 1
```

Our experimental results show that *local-frame*, *shared-frame*, and *indexed-frame* achieve parallel speedups of $15.9\times$, $18.1\times$, and $10.8\times$ on 32 cores, respectively. Using the same number of cores, our OpenMP-based parallelization (functioning as a baseline) only yields a speedup of $6.3\times$; thus, our algorithms are up to $2.9\times$ faster. Moreover, also due to implementation improvements and parameter tuning, our best algorithm performs adaptive sampling $65.3\times$ faster than the existing implementation of KADABRA (when all implementations use 32 cores).

Bibliographic Notes. The contributions presented in this chapter were published in the Proceedings of the *Twenty-Fifth European Conference on Parallel Processing (Euro-Par 2019)*. My contributions involve the development of the *indexed-frame* algorithm with bounded memory complexity (Section 4.4.5) in collaboration with Alexander van der Grinten and the implementation of all presented algorithms. The rest is joint work with Alexander van der Grinten and Henning Meyerhenke. Proofs to which I did not contribute are omitted and can be found in the original paper [131].

4.2 Preliminaries and Baseline for Parallelization

4.2.1 Basic Definitions

Memory Model. Throughout this chapter, we target a multi-threaded shared-memory machine with T threads. We work in the C11 memory model [147]. The weakest operations in this model are `load-relaxed` and `store-relaxed` operations; those only guarantee the atomicity of the memory access (i.e., they guarantee that no *tearing* occurs) but no ordering at all. Hence, the order in which `store-relaxed` writes become visible to `load-relaxed` reads can differ from the order in which the stores and loads are performed by individual threads. Additionally, `load-acquire` and `store-release` do provide ordering guarantees: if thread t_0 writes a word X to a given memory location using `store-release` *and* thread t_1 reads X using `load-acquire` from the same memory location, then all store operations – whether atomic or not – done by thread t_0 *before* the store of X become visible to all load operations done by thread t_1 *after* the load of X. We note that C11 defines even stronger ordering guarantees that we do not require in this chapter. Furthermore, on a hardware level, x86_64 implements a stronger *total store order*; thus, `load-acquire` and `store-release` compile to plain load and store instructions and our *local-frame* algorithm does not perform *any* synchronization instructions on x86_64.

Adaptive sampling. For our techniques to be applicable, we expect that an ADS algorithm behaves as depicted in Algorithm 6: it iteratively samples data (in sample) and aggregates it (using some operator $\circ$) until a stopping condition (checkForStop) determines that the data sampled so far is sufficient to return an approximate solution within the required accuracy. This condition does not only consider the number of samples (d.num), but also the sampled data (d.data). Throughout this chapter, we denote the size of that data (i.e., the number of elements of d.data) by n_s. We assume that the stopping condition needs to be checked on a *consistent* state, i.e., a state of d that can occur in a sequential execution.[19] Furthermore, to make parallelization feasible at all, we need to assume that $\circ$ is associative. For concrete examples of stopping conditions, we refer to Sections 4.2.3 and 4.2.4.

4.2.2 Betweenness Centrality and its Approximation

Betweenness Centrality (c_b) is one of the most popular vertex centrality measures for network analysis (see Section 2.4.3). Th betweenness of a vertex $u \in V$ is defined as:

$$c_b(u) := \sum_{\substack{x,y \in V \setminus \{u\} \\ x \neq y,\ \sigma_{x,y} \neq 0}} \frac{\sigma_{x,y}(u)}{\sigma_{x,y}},$$

where $\sigma_{x,y}$ is the number of shortest x-y paths and $\sigma_{x,y}(u)$ is the number of shortest x-y paths that contain u. Betweenness is extensively used to identify the key vertices in large networks, e.g., cities in a transportation network [136] or lethality in protein networks [152].

Unfortunately, $c_b(\cdot)$ is rather expensive to compute: the standard exact algorithm by Brandes [61] has time complexity $\Theta(n \cdot m)$ for unweighted graphs. Moreover, unless the Strong Exponential Time Hypothesis (SETH) fails, this asymptotic running time cannot be improved [55]. Numerous approximation algorithms for betweenness centrality have been developed – we refer to Section 4.6 for an overview. The state of the art of these approximation algorithms is the KADABRA algorithm [56] of Borassi and Natale, which happens to be an ADS algorithm. With probability $(1 - \delta)$, KADABRA approximates the $c_b(\cdot)$ values of the vertices within an additive error of $\pm\varepsilon$ in nearly-linear time complexity, where δ and ε are user-specified constants.

While our techniques apply to any ADS algorithm, we recall that, as a case study, we focus on scaling the KADABRA algorithm to large number of threads.

4.2.3 The KADABRA Algorithm

At each iteration, KADABRA samples a vertex pair (s, t) of $G = (V, E, w)$ uniformly at random and then selects a shortest s-t-path uniformly at random (sample in Algorithm 6). After τ iterations, this results in a sequence of randomly selected shortest paths $(\pi_1, \pi_2, \ldots, \pi_\tau)$. From those paths, $c_b(u)$ is estimated as:

$$\tilde{c}_b(u) = \frac{1}{\tau} \sum_{i=1}^{\tau} x_i(u), \quad x_i(u) = \begin{cases} 1 & \text{if } v \in \pi_i \\ 0 & \text{otherwise.} \end{cases}$$

[19] That is, d.num and all entries of d.data must result from an integral sequence of samples – parallelization would be trivial otherwise.

$\sum_{i=1}^{\tau} x_i$ is exactly the sampled data (d.`data`) that the algorithm has to store – the accumulation $\circ$ in Algorithm 6 sums x_i over i. To compute the stopping condition (CHECKFORSTOP in Algorithm 6), KADABRA maintains the invariants:

$$\Pr(c_b(u) \leq \tilde{c}_b(u) - f) \leq \delta_L(u) \quad \text{and} \quad \Pr(c_b(u) \geq \tilde{c}_b(u) + g) \leq \delta_U(u) \tag{4.1}$$

for two functions $f = f(\tilde{c}_b(u), \delta_L(u), \tau_{\max}, \tau)$ and $g = g(\tilde{c}_b(u), \delta_U(u), \tau_{\max}, \tau)$ depending on a maximum number $\tau_{\max}$ of samples and per-vertex probability constants δ_L and δ_U (more details in the original paper [56]). The value of those constants are computed in a preprocessing phase – mostly consisting of computing an upper bound of the diameter of the graph. δ_L and δ_U satisfy $\sum_{u \in V} \delta_L(u) + \delta_U(u) \leq \delta$ for a user-specified parameter $\delta \in (0, 1)$. Thus, the algorithm terminates once $f, g < \varepsilon$; with probability $(1 - \delta)$, the result is correct with an absolute error of $\pm\varepsilon$. We note that checking the stopping condition of KADABRA on an inconsistent state leads to incorrect results. For example, this can be seen from that fact that g is increasing with $\tilde{c}_b(\cdot)$ and decreasing with τ, see Section 4.2.4.

4.2.4 The Stopping Condition in Detail

In this section, we illustrate the stopping condition more in detail and show that evaluating it in a consistent state is crucial for the correctness of the algorithm. The functions f and g we mentioned in Eq. (4.1) are defined as [56]:

$$f(\tilde{c}_b(u), \delta_L(u), \tau_{\max}, \tau) = \frac{1}{\tau}\left(\log \frac{1}{\delta_L(u)}\right)\left(\frac{1}{3} - \frac{\tau_{\max}}{\tau} + \sqrt{\left(\frac{1}{3} - \frac{\tau_{\max}}{\tau}\right)^2 + \frac{2\tilde{c}_b(u)\tau_{\max}}{\log \frac{1}{\delta_L(u)}}}\right)$$

$$g(\tilde{c}_b(u), \delta_U(u), \tau_{\max}, \tau) = \frac{1}{\tau}\left(\log \frac{1}{\delta_U(u)}\right)\left(\frac{1}{3} + \frac{\tau_{\max}}{\tau} + \sqrt{\left(\frac{1}{3} + \frac{\tau_{\max}}{\tau}\right)^2 + \frac{2\tilde{c}_b(u)\tau_{\max}}{\log \frac{1}{\delta_U(u)}}}\right),$$

where $\tilde{c}_b(u)$ is the approximation of the betweenness centrality of vertex u obtained after τ samples. When the stopping condition is evaluated, f and g are computed for every vertex of the graph and the algorithm terminates if:

$$f(\tilde{c}_b(u), \delta_L(u), \tau_{\max}, \tau) \leq \varepsilon \quad \text{and} \quad g(\tilde{c}_b(u), \delta_U(u), \tau_{\max}, \tau) \leq \varepsilon$$

hold for every vertex $u \in V$. It is straightforward to verify that both f and g grow with $\tilde{c}_b(u)$ but that g decreases with τ. Thus, evaluating the stopping condition with inconsistent data (e.g., if accesses to τ and $\tilde{c}_b(u)$ are not synchronized) could lead to an erroneous termination of the algorithm.

4.2.5 First Attempts at KADABRA Parallelization

In the original KADABRA implementation,[20] a lock is used to synchronize concurrent access to the sampling state. As a first attempt to improve the scalability, we consider an algorithm that iteratively computes a fixed number of samples in parallel (e.g., using an OpenMP `parallel for` loop) and checks the stopping condition afterwards. While sampling, atomic increments are used to update the global sampling data. This algorithm is arguably the "natural" OpenMP-based parallelization of an ADS algorithm and can be

[20] Available at https://github.com/natema/kadabra

int	epoch	$\leftarrow e$
int	num	$\leftarrow 0$
int	data$[n_s]$	$\leftarrow (0, \ldots, 0)$

(a) Structure of a state frame (SF) for epoch e. num: Number of samples, data: Sampled data

bool	stop	$\leftarrow$ false
int	epochToRead	$\leftarrow 0$
SF $*$	sfFin$[T]$	$\leftarrow$ (null, ..., null)

(b) Shared variables

Figure 4.1: Data structures used in epoch-based algorithms, including initial values

implemented in a few extra lines of code. Moreover, it already improves upon the original parallelization. As shown by the experiments in Section 4.5 however, further significant improvements in performance are possible by switching to more lightweight synchronization.

4.3 Scalable Parallelization Techniques

To improve upon the OpenMP parallelization from Section 4.2.5, we have to avoid to synchronization barrier before the stopping condition can be checked. This is the objective of our *epoch-based* algorithms that constitute the main contribution of this chapter. In Section 4.3.1, we formulate the main idea of our algorithms as a general framework. The subsequent subsections present specific algorithms based on this framework and discuss the trade-offs between them.

4.3.1 Epoch-based Framework

In our epoch-based algorithms, the execution of each thread is subdivided into a sequence of discrete *epochs*. During an epoch, each thread iteratively collects samples; the stopping condition is only checked at the end of an epoch. The crucial advantage of this approach is that the end of an epoch *does not* require global synchronization. Instead, our framework guarantees the consistency of the sampled data by maintaining multiple copies of the sampling state.

As an invariant, it is guaranteed that no thread writes to a copy of the state that is currently being read by another thread. This is achieved as follows: each copy of the sampling state is labeled by an *epoch number* e, i.e., a monotonically increasing integer that identifies the epoch in which the data was generated. When the stopping condition has to be checked, all threads advance to a new epoch $e + 1$ and start writing to a new copy of the sampling state. The stopping condition is only verified after all threads have finished this transition and it only takes the sampling state of epoch e into account.

More precisely, the main data structure that we use to store the sampling state is called a *state frame* (SF). Each SF f (depicted in Figure 4.1a) consists of (i) an epoch number (f.epoch), (ii) a number of samples (f.num), and (iii) the sampled data (f.data). The latter two symbols directly correspond to d.num and d.data in our generic formulation of an ADS algorithm (Algorithm 6). Aside from the SF structures, our framework maintains three global variables that are shared among all threads (depicted in Figure 4.1b): (i) a simple Boolean flag stop to determine if the algorithm should terminate, (ii) a variable epochToRead that stores the number of the epoch we want to check the stopping condition on, and (iii) a pointer sfFin$[t]$ for each thread t that points to a SF finished by thread t. Incrementing epochToRead is our synchronization mechanism to notify all threads that they should advance to a new epoch. Figure 4.2 visualizes such an epoch

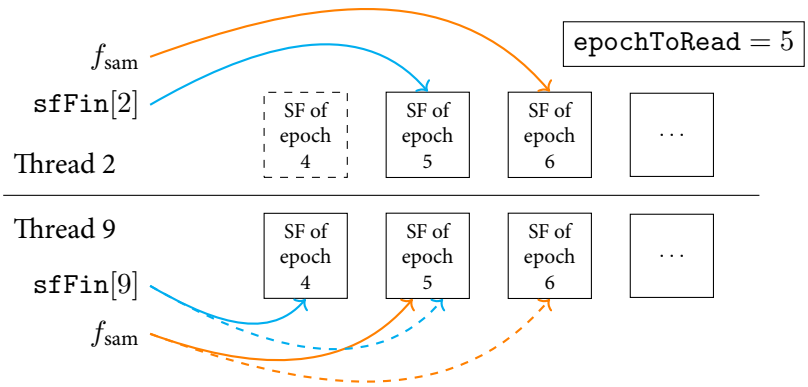


Figure 4.2: Transition after `epochToRead` is set to 5. Thread 2 already writes to the SF of epoch 6 (using the f_{sam} pointer). Thread 9 still writes to the SF of epoch 5 but advances to epoch 6 once it checks `epochToRead` (dashed orange line). Afterwards, thread 9 publishes its SF of epoch 5 to `sfFin` (dashed blue line). Finally, the stopping condition is checked using both SFs of epoch 5 (i.e., the SFs now pointed to by `sfFin`).

transition. In particular, it depicts the update of the `sfFin` pointers after an epoch transition is initiated by incrementing `epochToRead`.

Algorithm 7 states the pseudocode of our framework. By $\leftarrow_{\mathsf{relaxed}}$, $\leftarrow_{\mathsf{acquire}}$, and $\leftarrow_{\mathsf{release}}$, we denote relaxed memory access, `load-acquire` and `store-release`, respectively (see Section 4.2.1). In the algorithm, each thread maintains an epoch number e_{sam}. To be able to check the stopping condition, thread 0 maintains another epoch number e_{chk}. Indeed, thread 0 is the only thread that evaluates the stopping condition (in CHECKFRAMES) after accumulating the SFs from all threads. The CHECKFRAMES procedure determines whether there is an ongoing check for the stopping condition (*inCheck* is `true`; Line 16). If that is not the case, a check is initiated (by incrementing e_{chk} and all threads are signaled to advance to the next epoch (by updating `epochToRead`). Note that *inCheck* is needed to prevent thread 0 from repeatedly incrementing e_{chk} without processing data from the other threads. Afterwards, CHECKFRAMES only continues if all threads t have published their SFs for checking (i.e., `sfFin`$[t]$ points to a SF of epoch e_{chk}; Line 20). Once that happens, those SFs are accumulated (Line 27) and the stopping condition is checked on the accumulated data (Line 31). Eventually, the termination flag (`stop`, Line 32) signals to all threads that they should stop sampling. The main algorithm, on the other hand, performs a loop until this flag is set (Line 2). Each iteration collects one sample and writes the results to the current SF (f_{sam}). If a thread needs to advance to a new epoch (because an incremented `epochToRead` is read in Line 7), it publishes its current SF to `sfFin` and starts writing to a new SF (f_{sam}; Line 12). Note that the memory used by old SFs can be reclaimed (Line 9); note, however, that there is no SF for epoch 0). How exactly this is done is left to the algorithms described in later subsections.

Proposition 4.3.1 ([131]). Algorithm 7 always checks the stopping condition on a consistent state; in particular, the epoch-based approach is correct.

Algorithm 7 Epoch-based Approach

Per-thread variable initialization:

```
    e_sam ← 1
    f_sam ← new SF for e_sam = 1
    if t = 0 then
        e_chk ← 0
        inCheck ← false
```

Main loop for thread t:

```
 1: loop
 2:     doStop ←_relaxed stop
 3:     if doStop then
 4:         break
 5:     f_sam.data ← f_sam.data ∘ SAMPLE()
 6:     f_sam.num ← f_sam.num + 1
 7:     r ←_relaxed epochToRead
 8:     if r = e_sam then
 9:         reclaim SF of epoch e_sam − 1
10:         sfFin[t] ←_release f_sam
11:         e_sam ← e_sam + 1
12:         f_sam ← new SF for e_sam
13:     if t = 0 then
14:         CHECKFRAMES()
```

Check of stopping condition by thread 0:

```
15: procedure CHECKFRAMES()
16:     if not inCheck then
17:         e_chk ← e_chk + 1
18:         epochToRead ←_relaxed e_chk
19:         inCheck ← true
20:     for t = 0 to T − 1 do
21:         f_fin ←_acquire sfFin[t]
22:         if f_fin = null then
23:             return
24:         if f_fin.epoch ≠ e_chk then
25:             return
26:     d ← new SF for accumulation
27:     for t = 0 to T do
28:         f_fin ←_relaxed sfFin[t]
29:         d.data ← d.data ∘ f_fin.data
30:         d.num ← d.num + f_fin.num
31:     if CHECKFORSTOP(d) then
32:         stop ←_relaxed true
33:     inCheck ← false
```

4.3.2 *Local-frame* and *shared-frame* Algorithm

We present two epoch-based algorithms relying on the general framework from the previous section: namely, the *local-frame* and the *shared-frame* algorithm. Furthermore, in Sections 4.4.4 and 4.4.5, we present two variants of the deterministic *indexed-frame* algorithm – as both *local-frame* and *shared-frame* are non-deterministic. *Local-frame* and *shared-frame* are both based on the pseudocode of Algorithm 7. They differ, however, in their allocation and reuse of SFs (Line 9 of the pseudocode). The *local-frame* algorithm allocates one pair of SFs per thread and cycles through both SFs of that pair (i.e., epochs with even numbers are assigned to the first SF while odd epochs use the second SF). This yields a per-thread memory requirement of $\mathcal{O}(n_s)$; as before, n_s denotes the size of the sampling state. The *shared-frame* algorithm reduces this memory requirement to $\mathcal{O}(1)$ by only allocating F pairs of SFs in total, for a constant number F. Thus, T/F threads share a SF in each epoch and atomic `fetch-add` operations need to be used to write to the SF. The parameter F can be used to balance the memory bandwidth and synchronization costs – a smaller value of F lowers the memory bandwidth required during aggregation but leads to more cache contention due to atomic operations.

4.3.3 Synchronization Costs

In Algorithm 7, all synchronization of threads $t > 0$ is done wait-free in the sense that the threads only have to stop sampling for $\Theta(1)$ instructions to communicate with other threads (i.e., to check `epochToRead`, update per-thread state and write to `sfFin`$[t]$). At the same time, thread $t = 0$ generally needs to check all `sfFin` pointers. Taken together, this yields the following statement:

Proposition 4.3.2 ([131]). In each iteration of the main loop, threads $t > 0$ of *local-frame* and *shared-frame* algorithms spend $\Theta(1)$ time to wait for other threads. Thread $t = 0$ spends up to $\mathcal{O}(T)$ time to wait for other threads.

In particular, the synchronization cost does not depend on the problem instance – this is in contrast to the OpenMP parallelization (Section 4.2.5) in which threads can idle for $\mathcal{O}(\mathcal{S})$ time, where $\mathcal{S}$ denotes the time complexity of a sampling operation (e.g., $\mathcal{S} = \mathcal{O}(n + m)$ in the case of KADABRA).

Nevertheless, this advantage in synchronization costs comes at a price: the accumulation of the sampling data requires additional evaluations of $\circ$. The *local-frame* algorithm requires $\mathcal{O}(Ts)$ evaluations, whereas the *shared-frame* requires $\mathcal{O}(Fs)$. No accumulation is necessary in the OpenMP baseline. As can be seen in Algorithm 7, we perform the accumulation in a single thread (i.e., thread 0). Compared to a parallel implementation (e.g., using parallel reductions), this strategy requires no additional synchronization and has a favorable memory access pattern (as the SFs are read linearly). A disadvantage, however, is that there is a higher latency (depending on T) until the algorithm detects that it is able to stop. In Section 4.4.3, we discuss how a constant latency can be achieved heuristically.

4.4 Optimization and Tuning

4.4.1 Improvements to the KADABRA Implementation

In the following, we document some improvements to the sequential KADABRA implementation of Borassi and Natale [56]. First, for undirected graphs, we avoid searching for non-existing shortest paths between a pair (s, t) of randomly selected vertices by checking if s and t belong to the same connected component.[21] Then, we reduce the memory footprint of the sampling procedure: the original KADABRA implementation stores all predecessors on shortest paths in a separate graph G', which is used to backtrack the path starting from the last explored vertices. Our implementation avoids the use of G' by reconstructing shortest s-t-paths from the original graph G and a distance array. Furthermore, for each shortest s-t-path sampled, the original KADABRA implementation needs to reset a Boolean "visited" array with an overall additional cost of $\Theta(n)$ time per sample. We avoid doing this by using 7 bits per element in this array to store a *timestamp* that indicates when the vertex was last visited; therefore, the array needs to be reset only once in $2^7 = 128$ SSSPs.

4.4.2 Balancing Costs of Termination Checks

Although the pseudocode of Algorithms 6 and 7 checks the stopping condition after every sample, this amount of checking is excessive in practice. Hence, both the original KADABRA and the OpenMP ADS algorithms check the stopping condition after a fixed number N of samples. N represents a trade-off between the time required to check the stopping condition and the time required to sample a shortest path. In the original KADABRA implementation, N is set to 11; in our experiments, however, this choice turned out to be inefficient. Thus, we formed a small set of the instances for parameter tuning [14], and ran experiments with different values of N.[22] As a result, we found that $N = 1000$ empirically performs best.

[21] Connected components are computed along with the diameter during preprocessing.

[22] We chose the instances com-amazon, munmun_twitter_social, orkut-links, roadNet-PA, wikipedia_link_de, and wikipedia_link_fr.

4.4.3 Termination Latency in Epoch-based Approach

In the epoch-based approach, we also need to balance the frequency of checking the stopping condition and the time invested into sampling; however, we face a different problem: the accumulation of all SFs before the stopping condition is checked takes $\mathcal{O}(Tn_s)$ time, thus the length of an epoch depends on T (see Section 4.3.3). This is an undesirable artifact as it introduces an additional delay between the time when the algorithm could potentially stop (because enough samples have been collected) and the time when the algorithm actually stops (because the accumulation is completed). It would be preferable to check the stopping condition after a constant number of samples (summed over all thread) – as the sequential and OpenMP variants naturally do.[23]

While it seems unlikely that a constant number of samples per epoch can be achieved (without additional synchronization overhead), we aim to satisfy this property heuristically. Checking the stopping condition after $N_0 = (1/T)N$ samples *per thread* seems to be a reasonable heuristic. However, it does not account for the fact that only one thread performs the check while all additional threads continue to sample data. Thus, we check the stopping condition after

$$N_0 = \frac{1}{T^\xi} N$$

samples from thread 0. Here, ξ is another parameter that can be tuned. Using the same approach as in Section 4.4.2 (and running the algorithm on 32 cores), we empirically determined $\xi = \log_{32}(N/10) \approx 1.33$ to be a good choice.

4.4.4 *Indexed-frame* Algorithm

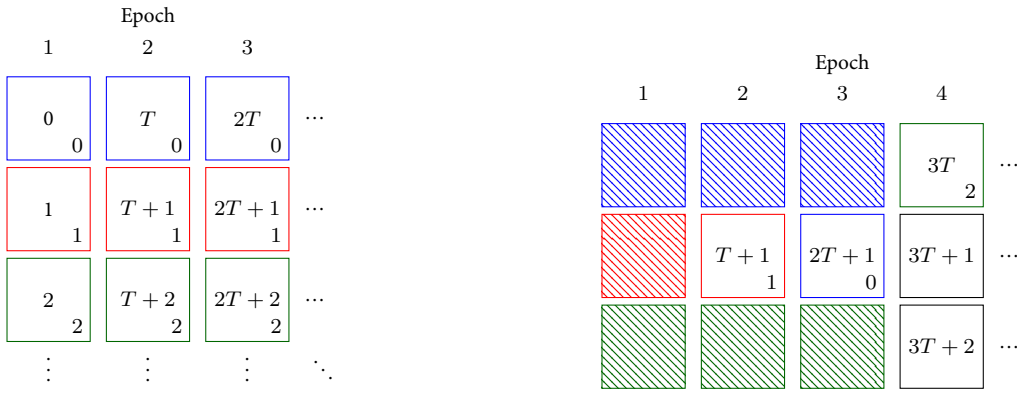
In this subsection, we introduce the *indexed-frame* algorithm that is a variant of *local-frame* but always obtains deterministic results. In particular, we highlight the modifications compared to *local-frame* that are necessary to avoid non-determinism.

There are two sources of non-determinism in the epoch-based algorithms: First, because threads generate random numbers independently from each other and the pseudo-random number generator (PRNG) of each thread is seeded differently, the sequence of generated random numbers depends on the number of threads. Secondly, and more importantly, the point in time where a thread notices that the stopping condition needs to be checked (i.e., `epochToRead` is read in Line 7 of Algorithm 7) is non-deterministic. Thus, among multiple executions of the algorithm, the SFs that are checked differ in (i) the number of samples and (ii) in the PRNG state used to generate the samples.

Indexed-frame avoids the first problem by re-seeding the random number generator of each thread whenever the thread moves to a new epoch. To avoid a dependence on the number of threads, the new seed should only vary based on a unique *index* of the generated SF (not to be confused with the epoch number). As an index for the SF of epoch e, we choose $(eT + t)$, as every thread t contributes exactly one SF to each epoch e. This scheme is depicted in Figure 4.3a.

Handling the second issue turns out to be more involved. As we need to ensure that the stopping condition is always checked on exactly the same SFs, the point in time where a thread moves to a new epoch must be

[23] As a side effect, doing so improves the comparability of those algorithms.



(a) SF indices in *indexed-frame* algorithm (not to be confused with epoch numbers).

(b) Relaxation of the *indexed-frame* condition with $T = 3$. Thread 0 (blue) and Thread 2 (green) have both finished the SFs of epochs 1, 2, and 3, while Thread 1 (red) did not finish SF $T + 1$ yet. Assuming that Thread 0 finished SF $2T$ before Thread 2, Thread 0 starts computing the SF with index $2T + 1$. Then, Thread 2 starts computing SF $3T$.

Figure 4.3: Indices of SFs in *indexed-frame* algorithm. Central numbers indicate SF indices. Numbers in bottom right corners (and colors) denote the thread that will compute the SF. Dashed SFs are already finished.

independent of the time when the stopping condition is checked. To achieve that, *indexed-frame* writes a fixed number of samples to each SF. That, however, means that by the time a check is performed, a thread can have finished multiple SFs. To deal with multiple finished SFs, we use a per-thread queue of SFs which have already been finished but which were not considered by the stopping condition yet. While the size of this queue is unbounded in theory, in our experiments we never observed a thread buffering more than 12 SFs at a time – with an average of 3 SFs allocated per thread. Thus, we do not implement in *indexed-frame* a sophisticated strategy to bound the queue length. The following subsection discusses such a strategy for ADS algorithms where this becomes a problem.

4.4.5 Bounded Memory Complexity in *Indexed-frame*

As our experiments in Section 4.5.5 demonstrate, the SF buffering overhead of the deterministic algorithm is not problematic in our betweenness centrality case study. However, at the cost of additional synchronization, it is possible to bound the theoretical memory complexity of the algorithm as well. In particular, if there are lower and upper bounds $\mathcal{C}_\ell$ and $\mathcal{C}_u$ on the time to compute a single SF,[24] we can relax the condition that every thread t samples the SF of epoch e with index $(eT + t)$: instead of computing the SFs with indices $(t, t+T, t+2T, \ldots)$, each thread t determines (by synchronizing with all other threads) the smallest index i of a SF that is not being computed yet by any other thread – see Figure 4.3b for an illustration of this process. Determinism is guaranteed because, before starting to compute a new SF, every thread re-seeds its random number generator with a seed that depends exclusively on the SF index, not on the thread sampling the SF. The bounds on the computation time of a single SF imply that all other threads can only compute a constant number $\mathcal{C}_u/\mathcal{C}_\ell$ of SFs until an epoch is finished (and all SFs of the epoch can be reclaimed).

[24] Such bounds trivially exist if the algorithmic complexity of a single sampling operation is bounded.

Table 4.1: List of instances used for the experiments.

Network name	n	m	Diameter	Category
tntp-ChicagoRegional	12,979	20,627	106	Infrastructure
dimacs9-NY	264,346	365,050	720	Infrastructure
dimacs9-COL	435,666	521,200	1,255	Infrastructure
munmun_twitter_social	465,017	833,540	8	Social
com-amazon	334,863	925,872	47	Co-purchase
loc-gowalla_edges	196,591	950,327	16	Social
web-NotreDame	325,729	1,090,108	46	Hyperlink
roadNet-PA	1,088,092	1,541,898	794	Infrastructure
roadNet-TX	1,379,917	1,921,660	1,064	Infrastructure
web-Stanford	281,903	1,992,636	753	Hyperlink
petster-dog-household	256,127	2,148,179	11	Social
flixster	2,523,386	7,918,801	8	Social
as-skitter	1,696,415	11,095,298	31	Computer
dbpedia-all	3,966,895	12,610,982	146	Relationship
actor-collaboration	382,219	15,038,083	13	Collaboration
soc-pokec-relationships	1,632,803	22,301,964	14	Social
soc-LiveJournal1	4,846,609	42,851,237	20	Social
livejournal-links	5,204,175	48,709,621	23	Social
wikipedia_link_ceb	7,891,015	63,915,385	9	Hyperlink
wikipedia_link_ru	3,370,462	71,950,918	10	Hyperlink
wikipedia_link_sh	3,924,218	76,439,386	9	Hyperlink
wikipedia_link_de	3,603,726	77,546,982	14	Hyperlink
wikipedia_link_it	2,148,791	77,875,131	9	Hyperlink
wikipedia_link_sv	6,100,692	99,864,874	10	Hyperlink
wikipedia_link_fr	3,333,397	100,461,905	10	Hyperlink
wikipedia_link_sr	3,175,009	103,310,837	10	Hyperlink
orkut-links	3,072,441	117,184,899	10	Social

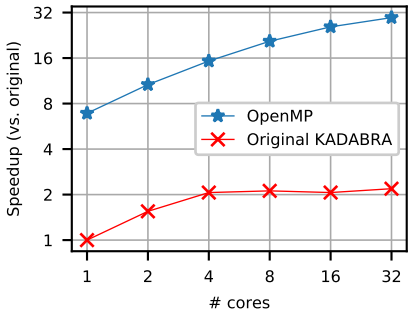
4.5 Experiments

4.5.1 Settings

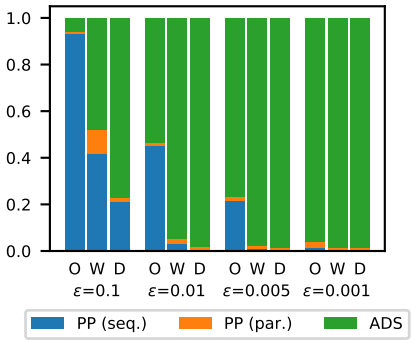
The platform we use for our experiments is a Linux server equipped with 1.5 TiB of RAM and two Intel Xeon Gold 6154 with 18 cores (for a total of 36 cores) at 3.00 GHz. Each thread of the algorithm is pinned to a unique core; hyperthreading is disabled. Our implementation is written in C++ building upon the NetworKit toolkit [273]. In the experiments, we use the 27 undirected real-world graphs reported in Table 4.1. The largest instances take tens of minutes for our OpenMP baseline and multiple hours for the original implementation of KADABRA. The error probability for KADABRA is set to $\delta = 0.1$ for all experiments. Absolute running time are reported in Appendix C.1.

4.5.2 OpenMP Baseline

In a first experiment, we compare our OpenMP baseline against the original implementation of KADABRA (see Section 4.2.5 for these two approaches). We set the absolute approximation error to $\varepsilon = 0.01$. The overall speedup (i.e., both preprocessing and ADS) are reported in Figure 4.4a. The results show that our OpenMP baseline outperforms the original implementation considerably (by a factor of $1.7\times$), even in a single-core setting. This is mainly due to implementation tricks (see Section 4.4.1) and parameter tuning (as discussed in Section 4.4.2). Furthermore, for 32 cores, our OpenMP baseline performs $13.5\times$ better than the original implementation of KADABRA – or $22.7\times$ if only the ADS phase is considered. Hence, for the remaining experiments, we discard the original implementation as a competitor and focus on the parallel speedup of our algorithms.

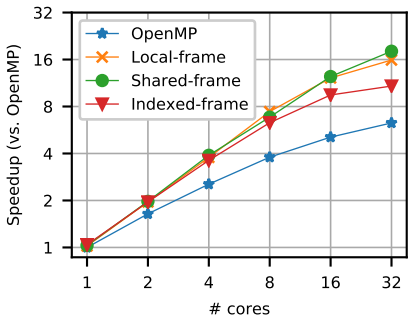


(a) Average speedup (preprocessing + ADS, geom. mean) of OpenMP baseline over the original sequential implementation of KADABRA.

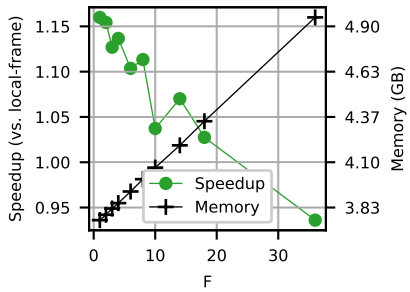


(b) Breakdown of sequential KADABRA running times into preprocessing and ADS (in percent) on instances orkut-links (O), wikipedia_link_de (W), and dimacs9-COL (D)

Figure 4.4: Performance of OpenMP baseline.



(a) Average ADS speedup (geom. mean) of epoch-based algorithms over sequential OpenMP baseline.



(b) Average ADS speedup (over 36-core *local-frame*, geom. mean) and memory consumption of *shared-frame*, depending on the number of SFs.

Figure 4.5: Performance of epoch-based algorithms.

4.5.3 Preprocessing and ADS Costs

To understand the relation between the preprocessing and ADS phases of KADABRA, we break down the running times of the OpenMP baseline in Figure 4.4b. In this plot, we present the fraction of time that is spent on ADS on three exemplary instances and for different values of ε. Especially if ε is small, the ADS running time dominates the overall performance of the algorithm. Thus, improving the scalability of the ADS phase is of critical importance. For this reason, we neglect the preprocessing phase and only consider ADS when comparing to our *local-frame* and *shared-frame* algorithms.

4.5.4 Parallel Speedup

In Figure 4.5a we report the parallel speedup of the ADS phase of our epoch-based algorithms relative to the OpenMP baseline. All algorithms are configured to check the stopping condition after a fixed number of samples (see Section 4.4.3 for details). The number F of SF pairs of *shared-frame* has been configured to 2, which we found to be a good setting for $T = 32$. On 32 cores, *local-frame* and *shared-frame* achieve parallel speedups of $15.9\times$ and $18.1\times$; they both significantly improve upon the OpenMP baseline, which

can only achieve a parallel speedup of $6.3\times$ (i.e., *local-frame* and *shared-frame* are $2.5\times$ and $2.9\times$ faster, respectively; they also outperform the original implementation by factors of $57.3\times$ and $65.3\times$, respectively). The difference between *local-frame* and *shared-frame* is insignificant for lower numbers of cores; this is explained by the fact that the reduced memory footprint of *shared-frame* only improves performance once memory bandwidth becomes a bottleneck. For the same reason, both algorithms scale very well until 16 cores; due to memory bandwidth limitations, this nearly ideal scalability does not extend to 32 cores. This bandwidth issue is known to affect graph traversal algorithms in general [26, 195].

4.5.5 Indexed-frame Algorithm

The *indexed-frame* algorithm is not as fast as *local-frame* and *shared-frame* on the instances depicted in Figure 4.5a; it achieves a parallel speedup of $10.8\times$ on 32 cores. However, it is still considerably faster than the OpenMP baseline (by a factor of $1.7\times$). There are two reasons why the determinism of *indexed-frame* is costly: *indexed-frame* has similar bandwidth requirements as *local-frame*; however, it has to allocate more memory as SFs are buffered for longer periods of time. On the other hand, even when enough samples are collected, the stopping condition has to be checked on older samples first, while *local-frame* and *shared-frame* can just check the stopping condition on the most recent sampling state.

4.5.6 Impact of Parameter F

In a final experiment, we evaluate the impact of the parameter F of *shared-frame* on its performance. Note that this experiment also demonstrates the difference in memory consumption of *shared-frame* ($F \in \{1, \ldots, T\}$) and *local-frame* (equivalent to $F = T$). Figure 4.5b depicts the results. The experiment is done with 36 cores; hence, memory pressure is even higher than in the previous experiments. The plot demonstrates that, in this situation, minimizing the memory bandwidth requirements at the expense of synchronization overhead is a good strategy. Hence, for larger number of cores, we can minimize memory footprint and maximize performance at the same time.

4.6 Related Work

ADS Algorithms. Our parallelization strategy can be applied to arbitrary ADS algorithms. ADS was first introduced by Lipton and Naughton to estimate the size of the transitive closure of a digraph [189]. It is used in a variety of fields, e.g., in statistical learning [243]. In the context of betweenness centrality, ADS has been used to approximate distances between pairs of vertices of a graph [230], to approximate the $c_b(\cdot)$ value of the vertices in a graph [27, 252], and to approximate the betweenness centrality of a single vertex [75]. An analogous strategy is exploited by Mumtaz and Wang [222] to find approximate solutions to the group-betweenness maximization problem.

Betweenness Centrality Approximation Algorithms. Regarding more general (i.e., not necessarily ADS) algorithms for $c_b(\cdot)$, a survey from Matta et al. [205] provides a detailed overview of the state of the art. The RK [251] algorithm represents the leading non-adaptive sampling algorithm for betweenness centrality approximation; KADABRA was shown to be $100\times$ faster than RK in undirected real-world graphs, and

70× faster thank RK in directed graphs [56] McLaughlin and Bader [210] introduced a work-efficient parallel algorithm for betweenness centrality approximation, implemented for single- and multi-GPU machines. Madduri et al. [198] presented a lock-free parallel algorithms optimized for specific massively parallel non-x86_64 architectures to approximate or compute $c_b(\cdot)$ exactly in massive networks. Unlike our approach, this lock-free algorithm parallelizes the collection of individual samples and is thus only applicable to betweenness centrality and not to general ADS algorithms. Additionally, according to the authors of [198] this approach hits performance bottlenecks on x86_64 even for 4 cores.

Concurrent Data Structures. The SFs used by our algorithms are concurrent data structures that enable us to minimize the synchronization latencies in multi-threaded environments. Devising concurrent (lock-free) data structures that scale over multiple cores is not trivial and much effort has been devoted to this goal [59, 216]. A well-known solution is the Read-Copy-Update mechanism (RCU); it was introduced to achieve high multi-core scalability on read-mostly data structures [209], and was leveraged by several applications [19, 77]. Concurrent hash tables are another popular example [85].

4.7 Conclusions

In this chapter, we observed that previous techniques to parallelize ADS algorithms are insufficient to scale to large numbers of threads. However, we found that significant speedups can be achieved by employing adequate concurrent data structures. Using such data structures in connection with our new epoch-based mechanism, we were able to devise parallel ADS algorithms that, in our betweenness centrality case study, consistently outperform the state of the art but also achieve different trade-offs between synchronization costs, memory footprint, and determinism of the result.

Sampling-based strategies are used to estimate other centralities such as closeness [63, 102], group betweenness [198], electrical closeness [17], or (group-) forest closeness [133]. Hence, regarding future work, a promising direction for our algorithms is to adapt them to approximate other centrality measures beyond betweenness.

5 Approximation of the Diagonal of a Laplacian's Pseudoinverse for Complex Network Analysis

5.1 Introduction

In this chapter, we address the problem of approximating electrical centrality measures for the analysis of complex networks – also known as small-world networks, i.e., graphs whose diameter is bounded by $\mathcal{O}(\log n)$, see Section 1.3. The small-world feature is typical of many real-world networks such as social networks, biological networks, information networks, etc. [227]. As described in Section 2.4, many centrality measures exist, some of which are based on shortest paths (or shortest-path distances) while others consider paths of arbitrary lengths. Electrical centrality measures interpret the graph as an electrical network [192] and thus fall in the latter category. Here we consider two electrical centrality measures: electrical closeness centrality (c_e) [62] – a.k.a. current-flow closeness or information centrality [274] – and forest closeness centrality ($c_{f,\alpha}$) [153].

Electrical Closeness. The electrical closeness of a vertex $u \in V$ (see Eq. (2.7)) is defined as the reciprocal of the average effective resistance $\rho(u, \cdot)$ from u to all other vertices.[25] The effective resistance between two vertices $u, v \in V$ can be computed by solving the linear system $\mathbf{L}\mathbf{x} = \mathbf{e}_u - \mathbf{e}_v$ for $\mathbf{x}$, where $\mathbf{e}_z$ is the canonical unit vector for vertex z. Then, $\rho(u, v) = \mathbf{x}[u] - \mathbf{x}[v]$. It is well-known that $\mathbf{L}$ does not have full rank and is thus not invertible. Its Moore-Penrose pseudoinverse [126] $\mathbf{L}^\dagger$, however, can be used to compute $\rho(u, v)$ as shown in Eq. (2.2).

A straightforward way to compute the electrical closeness of any vertex would be to compute $\mathbf{L}^\dagger$ which, without exploiting the structure of $\mathbf{L}$, takes in practice cubic time in n, cf. Ref. [249]. Further, this strategy would require $\Theta(n^2)$ memory since $\mathbf{L}^\dagger$ is in general a dense matrix – even for sparse $\mathbf{L}$. Therefore, full (pseudo)inversion is clearly limited to small inputs.

Conceptually similar to inversion would be to solve $\Theta(n)$ Laplacian linear systems. Fewer linear systems suffice in applications with lower accuracy requirements: the Johnson-Lindenstrauss Transform (JLT) combined with a fast Laplacian solver (such as Ref. [80]) achieve a relative approximation guarantee by solving $\mathcal{O}(\log n/\varepsilon^2)$ systems [272] in $\tilde{\mathcal{O}}(m \log^{1/2} n \log(1/\varepsilon))$ time each, where $\tilde{\mathcal{O}}(\cdot)$ hides a $\mathcal{O}((\log\log n)^{3+\delta})$ factor for $\delta > 0$.

As pointed out by Bozzo and Franceschet [60], the (only) relevant part of $\mathbf{L}^\dagger$ for computing the electrical closeness is its diagonal – we will see that this is true for other electrical centrality measures as well.

[25] We remark that effective resistance has numerous applications well beyond its usage in electrical centrality measures – cf. Refs. [8, 120].

Numerical methods for sampling-based approximation of the diagonal of implicitly given matrices already exist [36]. Yet, for our purpose, they solve $\mathcal{O}(\log n/\varepsilon^2)$ linear system as well to obtain an ε-approximation with high probability. While Laplacian linear systems can be solved independently in parallel, their solution can still be time-consuming in practice, in part due to high constant overheads hidden in the $\mathcal{O}$-notation.

Forest Closeness. Forest closeness (see Eq. (2.8)) is based on the forest distance, a metric introduced by Chebotarev and Shamis [69]. It is defined as closeness centrality where the shortest-path distance is replaced by the forest distance. In sociology, forest distances are shown to effectively capture sensitive relationship indices such as social proximity and group cohesion [72]. Hence, forest closeness has two main advantages over many other centrality measures [153]: (i) unlike electrical closeness, it can handle disconnected graphs out of the box and, by considering not only shortest paths, (ii) it has a high discriminative power.

Jin et al. [153] provided an approximation algorithm for forest closeness centrality with nearly-linear time complexity. Similarly to the aforementioned strategy for electrical closeness, their algorithm uses the JLT and fast linear solvers; it is, however, still time-consuming. For example, in their experimental study [153], graphs with $\approx$ 1M to $\approx$ 2-3M edges require more than 2.3 to 4.7 *hours* for a reasonably accurate ranking. Clearly, this hardly scales to larger graphs with more than a few million edges.

As already realized by Chebotarev and Shamis [69], forest distance is closely connected to effective resistance. In particular, as we describe in Section 5.5, any algorithm that computes the electrical closeness can easily be adapted to compute the forest closeness with just an additional linear-time overhead.

5.1.1 Related Work

Solving Laplacian Systems. As mentioned above, a straightforward approach to compute electrical closeness is to compute $\mathbf{L}^\dagger$ by solving a number of Laplacian systems. Brandes and Fleischer [62] compute electrical closeness from the solution of n linear systems using Conjugate Gradient (CG) in $\mathcal{O}(mn\sqrt{\kappa})$, where κ is the condition number of the appropriately preconditioned Laplacian matrix.[26] Later, Spielman and Srivastava [272] proposed an approximation algorithm to compute effective resistance distances. The main components of the algorithm are (i) a dimension reduction with JLT [155] and (ii) the use of a fast Laplacian solver for $\mathcal{O}(\log n/\varepsilon^2)$ Laplacian systems. The algorithm approximates effective resistance values for all edges with a factor of $(1 \pm \varepsilon)$ in $\mathcal{O}(I(n,m)\log n/\varepsilon^2)$ time, where $I(n,m)$ is the running time of the Laplacian solver, assuming that the solution of the Laplacian systems is exact. With an approximate Laplacian solution, the algorithm yields a $(1+\varepsilon)^2$-approximation. Significant progress in the development of Laplacian solvers with theoretical guarantees [80, 161, 169, 170, 174] has resulted in the currently best one running in $\mathcal{O}(m\log^{1/2} n \log 1/\varepsilon)$ time (up to polylogarithmic factors) [80]. Parallel algorithms for solving linear systems on the more general SDD matrices also exist in the literature [51, 239]. To date, the fastest algorithms for electrical closeness and spanning edge centrality[27] extend the idea of Spielman and Srivastava [41, 139, 207] – similar ideas are also used for centrality measures based on the Kirchhoff index, see Section 5.4. Since theoretical Laplacian solvers rely on heavily graph-theoretic machinery such as low-stretch spanning trees, multigrid solvers [41, 171, 190] are used in practice instead.

[26] Brandes and Fleischer provide a rough estimate of κ as $\Theta(n)$, leading to a total time of $\mathcal{O}(mn^{1.5})$.

[27] The spanning edge centrality of an edge e in a graph G is defined as the number of spanning trees in G that contain e and it is equivalent to the effective resistance of e [207].

Diagonal Estimation. Recall that only the diagonal of $\mathbf{L}^\dagger$ [or of $\mathbf{\Omega}$, resp.] is enough to compute the electrical closeness – or, in case of the Kirchhoff index, only the sum of $\mathrm{diag}(\mathbf{L}^\dagger)$, i.e., the trace $\mathrm{tr}(\mathbf{L}^\dagger)$ is required – [or the forest closeness]. Algorithms that approximate the diagonal (or the trace) of matrices that are only implicitly available often use iterative methods [268], sparse direct methods [10, 151], Monte Carlo [144] or deterministic probing techniques [36]. A popular approach is the standard Monte-Carlo method for the trace of a matrix $\mathbf{B}$, due to Hutchinson [144]. The idea is to estimate the trace of $\mathbf{B}$ by observing the action of $\mathbf{B}$ (in terms of matrix-vector products) on a sufficiently large sample of random vectors. In our case, this would require to solve a large number of Laplacian linear systems with random vectors as right-hand sides. Avron and Toledo [24] proved that the method requires $\mathcal{O}(\log n/\varepsilon^2)$ samples to achieve a maximum error of ε with probability at least $1-\delta$. The approach from Hutchinson [144] was extended by Bekas et al. [36] for estimating $\mathrm{diag}(\mathbf{B})$. Finally, Barthelemé et al. [30] proposed a combinatorial algorithm to approximate the trace (not the diagonal) of the inverse of a matrix closely related to the Laplacian. Their algorithm can be seen as a special case of our algorithm in a situation where a universal vertex exists (i.e., a vertex connected to all the other vertices of the graph).

5.1.2 Contribution and Outline

We introduce a new algorithm for approximating $\mathrm{diag}(\mathbf{L}^\dagger)$ of a Laplacian matrix $\mathbf{L}$ that corresponds to weighted undirected graphs (Section 5.3). Our main technique is the approximation of effective resistances between a pivot vertex $u \in V$ and all other vertices of G. It is based on sampling uniform (= random) spanning trees (USTs). The resulting algorithm is highly parallel and (almost) purely combinatorial – it relies on the connection between Laplacian linear systems, effective resistances, and USTs. Because effective resistance also plays a major role in *normalized random-walk betweenness* [224] and *Kirchhoff index centrality* [184], in this chapter we consider these measures as well.

For small-world graphs, our algorithm obtains an absolute $\pm\varepsilon$-approximation guarantee with high probability in (sequential) time $\mathcal{O}(m \log^4 n \cdot \varepsilon^{-2})$. In particular, compared to the fastest theoretical Laplacian solvers in connection with JLT, our approach is off by only a polylogarithmic factor. More importantly from a practical perspective, after some algorithm engineering (Section 5.6), our algorithm performs much better than the state of the art, already in our sequential experiments (Section 5.7): (i) it is much faster and more memory-efficient, (ii) it yields a maximum absolute error that is one order of magnitude lower, and (iii) results in a more accurate complete centrality ranking of elements of $\mathrm{diag}(\mathbf{L}^\dagger)$. Furthermore, due to good parallel speedups, we can even compute a reasonably accurate diagonal of $\mathbf{L}^\dagger$ on a small-scale cluster with 16 compute nodes in less than 8 minutes for a graph with ≈ 13.6M vertices and ≈ 334.6M edges.

Concerning forest closeness centrality, we adapt our UST sampling technique to approximate $c_{f,\alpha}(\cdot)$ for each vertex in the graph. Our experiments in Section 5.8 show that our algorithm for ranking individual vertices is always substantially faster than the state of the art [153]; specifically, for sufficiently large networks and in a sequential setting, it is one to two orders of magnitude faster while often achieving better accuracy. Our new algorithm can rank all vertices in networks with up to 334.6M edges with reasonable accuracy in less than 20 minutes if executed in an MPI-parallel setting on a small-scale cluster with 16 compute nodes.

Bibliographic Notes. The algorithms for approximating $\mathrm{diag}(\mathbf{L}^\dagger)$ were published in the Proceedings of the *Twenty-Eighth Annual European Symposium on Algorithms (ESA 2020)* while the approximation algo-

rithms for forest closeness were presented in the Proceedings of the *Twenty-First SIAM International Conference on Data Mining (SDM 2021)*. Among those presented in this chapter, my contributions involve the implementation of all presented algorithms and carrying out the experiments. The rest is joint work with Maria Predari, Alexander van der Grinten, and Henning Meyerhenke. Proofs to which I did not contribute are omitted and can be found in in the original papers [17, 133].

5.2 Preliminaries

As input we consider simple undirected graphs $G = (V, E, w)$ with non-negative edge weights – for electrical closeness, we also assume that G is connected. For the complexity analysis, we usually assume that $\operatorname{diam}(G) \in \mathcal{O}(\log n)$, but our algorithm works correctly even without this assumption.

Graphs as Electrical Networks. Recall our notation for electrical networks provided in Section 2.3.2. We interpret G as an electrical network in which every edge $e \in E$ represents a resistor with resistance $1/w(e)$. The effective resistance between two vertices $u, v \in V$, denoted by $\rho(u, v)$, is defined as the potential difference between u and v when a unit of current is injected into G at u and extracted at v – see Definition 2.3.4. To compute $\rho(u, v)$, we can either use the Moore-Penrose pseudoinverse as shown in Eq. (2.2):

$$\rho(u, v) = (\mathbf{e}_u - \mathbf{e}_v)^\top \mathbf{L}^\dagger (\mathbf{e}_u - \mathbf{e}_v) = \mathbf{L}^\dagger[u, u] - 2\mathbf{L}^\dagger[u, v] + \mathbf{L}^\dagger[v, v], \tag{5.1}$$

or, equivalently, $\rho(u, v) = \mathbf{x}[u] - \mathbf{x}[v]$, where $\mathbf{x}$ is the solution vector of the Laplacian linear system $\mathbf{L}\mathbf{x} = \mathbf{e}_u - \mathbf{e}_v$. Recall from Eq. (2.3) that $\mathbf{L}^\dagger$ can be expressed as:

$$\mathbf{L}^\dagger = \left(\mathbf{L} + \frac{1}{n}\mathbf{J}\right)^{-1} - \frac{1}{n}\mathbf{J},$$

where $\mathbf{J}$ is the $n \times n$-matrix with all entries being 1. Also, note that the effective resistance between the endpoints of an edge $e \in E$ equals the probability that e is an edge in a UST, i.e., a spanning tree selected uniformly at random among all spanning trees of G, cf. [54, Ch. II].

Electrical Closeness. The combinatorial counterpart of electrical closeness (Eq. (2.5)) is based on shortest-path distances: $c_c(u) := (n-1)/f(u)$, where the denominator is the *combinatorial farness* of u:

$$f(u) := \sum_{v \in V \setminus \{u\}} d(u, v). \tag{5.2}$$

Electrical farness (f_e) is defined analogously to combinatorial farness – shortest-path distance in Eq. (5.2) is replaced by effective resistance $\rho(u, v)$. Both combinatorial and electrical farness are not defined for disconnected graphs due to infinite distances. A solution to this issue already exists: Lin's index [188] is a generalization of combinatorial closeness to disconnected graphs and can easily be adapted to the electrical case as well. Thus, our assumption of G being connected is no limitation.

Normalized Random-Walk Betweenness. Contrary to classical betweenness, which is based on shortest paths (see Eq. (2.12)), normalized random-walk betweenness ($c_{\text{nrwb}}(\cdot)$, abbreviated with NRWB) con-

siders random walks [224]. More precisely, let $s \in V$ be a source vertex, $t \in V \setminus \{s\}$ be a destination vertex, and assume we are trying to obtain the NRWB of some other vertex $u \in V \setminus \{s,t\}$; $c_{\text{nrwb}}(u)$ counts the fraction $\mu_{s,t}(u)$ of random walks starting from s passing through u only once. To compute $c_{\text{nrwb}}(u)$, $\mu_{s,t}(u)$ is averaged over all $s \in V$ and $t \in V \setminus \{s\}$:

$$c_{\text{nrwb}}(u) = \frac{1}{n(n-1)} \sum_{\substack{s,t \in V \setminus \{u\} \\ s \neq t}} (\mu_{s,t}(u) + (n-1)).$$

Narayan and Saniee [224] obtain a closed-form expression of NRWB:

$$c_{\text{nrwb}}(u) = \frac{1}{n} + \frac{1}{n-1} \sum_{t \in V \setminus \{u\}} \frac{\mathbf{M}^{-1}[t,t] - \mathbf{M}^{-1}[t,u]}{\mathbf{M}^{-1}[t,t] + \mathbf{M}^{-1}[u,u] - 2\mathbf{M}^{-1}[t,u]}, \tag{5.3}$$

where $\mathbf{M} := \mathbf{L} + \mathbf{P}$, with $\mathbf{P}$ the projection operator onto the zero eigenvector of the Laplacian $\mathbf{L}$, i.e., $\mathbf{P}[i,j] = 1/n$. In Section 5.4 (Lemma 5.4.1), we show how to simplify this expression and how our algorithm can be adapted to NRWB.

Kirchhoff Index and Related Centrality Measures. The Kirchhoff index $\mathcal{K}(G)$ [100, 167] – a.k.a. (effective) graph resistance – is the sum of the effective resistance distances over all pairs of vertices in G and is an important measure for network robustness. The Kirchhoff index is often computed via the closed-form expression $\mathcal{K}(G) = n \operatorname{tr}(\mathbf{L}^{\dagger})$ [167].

Li and Zhang [184] adapted the Kirchhoff index to obtain two new *edge* centrality measures for $e \in E$. Let $G \backslash_{\theta} e$ be the graph obtained by deactivating edge e, i.e., decreasing its weight of e from $w(e)$ to $\theta w(e)$ for some small $\theta \in (0, 1/2]$, and let $\mathbf{L} \backslash_{\theta} e$ be the Laplacian matrix of $G \backslash_{\theta} e$. The new measures are:

- $c_{\theta}(e) := n \operatorname{tr}(\mathbf{L}^{\dagger} \backslash_{\theta} e)$, i.e., the Kirchhoff index of the graph $G \backslash_{\theta} e$;
- $c_{\theta}^{\Delta}(e) := c_{\theta}(e) - \mathcal{K}(G)$ i.e., the difference between the Kirchhoff indices of graph $G \backslash_{\theta} e$ and G.

To calculate the Kirchhoff edge centralities, Li and Zhang [184] use techniques such as partial Cholesky factorization [175], fast Laplacian solvers, and the Hutchinson estimator. For $c_{\theta}^{\Delta}(e)$, which is the more interesting measure in our context, they propose an ε-approximation algorithm that approximates $c_{\theta}^{\Delta}(e)$ for all edges in $\mathcal{O}(m\varepsilon^{-2}\theta^{-2} \log^{2.5} n \log(1/\varepsilon))$ time – up to polylogarithmic factors. The algorithm uses the Sherman-Morrison formula [266], which gives a fractional expression of $(\mathbf{L}^{\dagger} \backslash_{\theta} e - \mathbf{L}^{\dagger})$. The numerator is approximated by the Johnson-Lindenstrauss lemma and the denominator by effective resistance estimates for all edges.

Following the definition of $\mathcal{K}(G)$, it is easy to see that an algorithm that approximates $\operatorname{diag}(\mathbf{L}^{\dagger})$ also approximates the Kirchhoff index.

Forest Closeness. Similarly to its combinatorial counterpart, forest closeness is inversely proportional to *forest farness*, i.e., the sum of the forest distances from a vertex to all other vertices. The forest distance

between two vertices $u, v \in V$ (see Definition 2.3.6 in Section 2.3.3) is a one-parametric metric $\zeta_\alpha(u, v)$ defined in terms of the forest matrix $\mathbf{\Omega}_\alpha := (\alpha\mathbf{L} + \mathbf{I})^{-1}$ [69]:

$$\zeta_\alpha(u, v) := (\mathbf{e}_u - \mathbf{e}_v)^\top \mathbf{\Omega}_\alpha(\mathbf{e}_u - \mathbf{e}_v) = \mathbf{\Omega}_\alpha[u, u] - 2\mathbf{\Omega}_\alpha[u, v] + \mathbf{\Omega}_\alpha[v, v]. \tag{5.4}$$

Hence, the forest farness and the forest closeness of a vertex u are defined as:

$$f_{f,\alpha}(u) := \sum_{v \in V \setminus \{u\}} \zeta_\alpha(u) \quad \text{and} \quad c_{f,\alpha}(u) := \frac{n}{f_{f,\alpha}(u)}, \tag{5.5}$$

respectively. Non-parametric variants of forest closeness fix α to 1 [70]. To simplify our notation, in the following we omit α when clear from the context. As we explain in more detail in Section 5.5, a close connection between forest closeness and electrical closeness allows us to easily adapt our algorithm for electrical closeness approximation to forest closeness.

5.3 Approximation Algorithm for Electrical Closeness

5.3.1 Overview

In order to compute the electrical closeness for all vertices in V, the main challenge is obviously to compute their electrical farness $f_e(\cdot)$. Recall from Section 5.1 that the diagonal of $\mathbf{L}^\dagger$ is sufficient to compute $f_e(\cdot)$ for all vertices simultaneously – comp. Ref. [60, Eq. (15)]) with a slightly different definition of electrical closeness. This follows from Eq. (5.1) and from the fact that each row/column in $\mathbf{L}^\dagger$ sums to 0:

$$f_e(u) := \sum_{v \in V \setminus \{u\}} \rho(u, v) = n\mathbf{L}^\dagger[u, u] + \mathrm{tr}\left(\mathbf{L}^\dagger\right) - 2\sum_{v \in V} \mathbf{L}^\dagger[u, v] = n\mathbf{L}^\dagger[u, u] + \mathrm{tr}\left(\mathbf{L}^\dagger\right),$$

since $\mathrm{tr}(\cdot)$ is the sum over the diagonal entries. We are interested in an approximation of $\mathrm{diag}\left(\mathbf{L}^\dagger\right)$, since we do not necessarily need exact values for our particular applications. To this end, we propose an approximation algorithm for which we give a rough overview first. Our algorithm works best for small-world networks – thus, we focus on this important input class. Let G be unweighted for now; we discuss the extension to weighted graphs in Section 5.4.

1. Select[28] a pivot vertex $u \in V$ and solve the linear system $\mathbf{L}\mathbf{x} = \mathbf{e}_u - \frac{1}{n}\mathbf{j}$, (recall that $\mathbf{j} = (1, \ldots, 1)^\top$). Out of all solutions $\mathbf{x}$, we want the one such that $\mathbf{x} \perp \mathbf{j}$, since this unique normalized solution is equal to $\mathbf{L}^\dagger[:, u]$, i.e., the column of $\mathbf{L}^\dagger$ corresponding to u – see Ref. [281, pp. 6-7].

2. As a direct consequence from Eq. (5.1), the diagonal entries $\mathbf{L}^\dagger[v, v]$ for all $v \in V \setminus \{u\}$ can be computed as:
$$\mathbf{L}^\dagger[v, v] = \rho(u, v) - \mathbf{L}^\dagger[u, u] + 2\mathbf{L}^\dagger[v, u].$$

3. It remains to approximate these $n - 1$ effective resistance values $\rho(u, \cdot)$. To do so, we employ Kirchhoff's theorem, which connects electrical flows with spanning trees [54, Ch. II]. Let N be the total

[28] As we will see later on, one can improve the empirical running time when u is not chosen arbitrarily, but so as to have a low eccentricity. The correctness and the asymptotic time complexity of the algorithm are not affected by the selection, though.

Algorithm 8 Approximation algorithm for $\operatorname{diag}(\mathbf{L}^\dagger)$.

Input: Undirected graph $G = (V, E, w)$, pivot $u \in V$, error bound $\varepsilon > 0$, probability $\delta \in (0, 1)$
Output: $\operatorname{diag}\left(\widetilde{\mathbf{L}^\dagger}\right)$, i.e., an (ε, δ)-approximation of $\operatorname{diag}(\mathbf{L}^\dagger)$

1: $\tilde{\rho}(u, v) \leftarrow 0 \; \forall v \in V \setminus \{u\}$ ▷ $\mathcal{O}(n)$
2: Pick a constant $\kappa \in (0, 1)$ arbitrarily
3: $\eta \leftarrow \frac{\kappa\varepsilon}{3\sqrt{mn \log n} \operatorname{diam}(G)}$
4: Compute the BFS tree B_u of G with root u ▷ $\mathcal{O}(n + m)$
5: $\tau \leftarrow \operatorname{ecc}(u)^2 \cdot \left\lceil \frac{\log(2m/\delta)}{2(1-\kappa)^2\varepsilon^2} \right\rceil$ ▷ $\mathcal{O}(1)$
6: **for** $j \leftarrow 1$ **to** τ **do**
7: Sample UST T_i of G with root u ▷ $\mathcal{O}(m \log n)$
8: $\tilde{\rho}(u, \cdot) \leftarrow \textsc{Aggregate}(T_i, \tilde{\rho}(u, \cdot), B_u)$ ▷ $\mathcal{O}(n \log n)$, see Algorithm 9
9: Solve $\mathbf{L}\mathbf{x} = \mathbf{e}_u - \frac{1}{n}\mathbf{j}$ for $\mathbf{x}$ with accuracy η ▷ $\tilde{\mathcal{O}}(m \log^{1/2} n \log(1/\eta))$
10: $\widetilde{\mathbf{L}^\dagger}[u, u] \leftarrow \mathbf{x}[u]$
11: **for** $v \in V \setminus \{u\}$ **do** ▷ Overall: $\mathcal{O}(n)$
12: $\widetilde{\mathbf{L}^\dagger}[v, v] \leftarrow \tilde{\rho}(u, v)/\tau - \mathbf{x}[u] + 2\mathbf{x}[v]$
13: **return** $\operatorname{diag}\left(\widetilde{\mathbf{L}^\dagger}\right)$

number of spanning trees of G and let $N_{s,t}(a, b)$ be the number of spanning trees in which the unique from s to t traverses the edge $\{a, b\}$ *in the direction from a to b*. Further, recall from Section 2.3.2 that $i(a, b)$ denotes the amount of electrical current that flows from a to b.

Theorem 5.3.1 (Kirchhoff, comp. [54])**.** Let $i(a, b) := (N_{s,t}(a, b) - N_{s,t}(b, a))/N$. Distribute the current flows on the edges of G by sending a current of size $i(a, b)$ from a to b for every edge $\{a, b\}$. Then there is a total current of size 1 from s to t satisfying Kirchhoff's laws.

As a result of Theorem 5.3.1, the effective resistance between s and t is the potential difference between s and t induced by the current-flow given by i. Conversely, since the current flow is induced by potential differences (Ohm's law), one simply has to add the currents on a path from s to t to compute $\rho(s, t)$ – see Eq. (5.6) in Section 5.3.2. Actually, as a proxy for the current flows, we use the (approximate) $N(\cdot)$-values mentioned in Theorem 5.3.1.

4. For large graphs, it is impractical to compute the exact values for N (e.g., by Kirchhoff's matrix-tree theorem [123], which would require the determinant or all eigenvalues of $\mathbf{L}^\dagger$) or $N(\cdot)$. Instead, we obtain approximations of the desired values via sampling: we sample a number uniform spanning trees and determine the $N(\cdot)$-values by aggregation over the sampled trees. This approach provides a probabilistic absolute approximation guarantee.

Note that Steps 2-4 of the algorithm are entirely combinatorial. Step 1 may or may not be combinatorial, depending on the Laplacian solver used. Corresponding implementation choices are discussed in Section 5.6.

Overall Algorithm. Algorithm 8 shows the pseudocode of our algorithm. The pivot vertex u is already received as an input parameter. Lines 1–8 approximate the effective resistances $\rho(u, \cdot)$. To do so, Lines 1–5 perform initializations: first the estimate of the effective resistance $\tilde{\rho}(u, v)$ is set to 0 for all vertices $v \in V$. Then, the accuracy η of the linear solver is computed so as to ensure an absolute ε-approximation for the

whole algorithm. A BFS is performed from u in order to compute shortest paths from u to all other vertices (more details in Section 5.3.2). The sample size τ depends on the parameters ε and δ, among others.

The first `for`-loop in Line 6 performs the actual sampling and aggregation of the USTs – the latter with Algorithm 9. Afterwards, Lines 9–12 fill the u-th column and the diagonal of $\mathbf{L}^\dagger$ – to the desired accuracy.

Remark 5.3.1. Note that, due to the fact that Laplacian linear solvers provide a *relative* error guarantee (and not an absolute $\pm\varepsilon$ guarantee), the (relative) accuracy η for the initial Laplacian linear system (Lines 3 and 9) depends in a non-trivial way on our guaranteed absolute error ε. For details, see [17, Appendix A.4].

We also remark that the value of the constant κ does not affect the asymptotic running time (nor the correctness) of the algorithm. However, it does affect the empirical running time by controlling which fraction of the error budget is invested into solving the initial linear system vs. UST sampling.

In the reminder of Section 5.3, we describe more in detail the components and properties of Algorithm 8.

5.3.2 Effective Resistance Approximation by UST Sampling

Extending and generalizing work by Hayashi et al. [139] on spanning edge centrality, our main idea is to compute a sufficiently large sample of USTs and to aggregate the $N(\cdot)$-values of the edges in those USTs. Recall that, given an electrical flow with source u and sink v, the effective resistance between u and v equals the difference $\mathbf{x}[u] - \mathbf{x}[v]$, where $\mathbf{x}$ is the solution vector of the Laplacian linear system $\mathbf{Lx} = \mathbf{e}_u - \mathbf{e}_v$. Since $\mathbf{x}$ is a potential and the electrical flow i flows from its difference, $\rho(u, v)$ can be computed given *any* path $(u = v_0, v_1, \ldots, v_{\ell-1}, v_\ell = v)$ as:

$$\rho(u, v) = \sum_{j=0}^{\ell-1} i(v_j, v_{j+1}) = \frac{1}{N} \sum_{j=0}^{\ell-1} (N_{u,v}(v_j, v_{j+1}) - N_{u,v}(v_{j+1}, v_j)). \tag{5.6}$$

Recall that the sign of the current flow changes if we traverse an edge against the flow direction. This is reflected by the second summand of Eq. (5.6). Since we can choose any path from u to v, for efficiency reasons we use *one shortest* path $P_{u,v}$ per vertex $v \in V \setminus \{u\}$. We compute these paths with one breadth-first search (BFS) with root u, resulting in a tree B_u whose edges are considered implicitly directed from the root to the leaves. For each vertex $v \in V \setminus \{u\}$, we maintain an estimate $\tilde{\rho}(u, v)$ of $\rho(u, v)$, which is initially set to 0 for all v. After all USTs have been sampled and processed, we divide all $\tilde{\rho}(u, \cdot)$ by τ, the number of sampled trees – i.e., τ takes the role of N in Eq. (5.6).

Sampling USTs. In total, we sample τ USTs, where τ depends on the desired approximation guarantee and is determined later. The choice of the UST algorithm depends on the input: for general graphs, the algorithm by Shild [262] with time complexity $\mathcal{O}(m^{1+o(1)})$ is the fastest. Among others, it uses a sophisticated shortcutting techniques using fast Laplacian solvers to speed up the classical Aldous-Broder [7, 64] algorithm. For unweighted small-world graphs, however, Wilson's [284] simple algorithm using loop-erased random walks takes $\mathcal{O}(m \log n)$ time, as we outline in the following. Thus, for our class of inputs, Wilson's algorithm is preferred.

Wilson's UST Algorithm. Given a path P, its loop erasure is a simple path created by removing all cycles of P in chronological order. Wilson's algorithm grows a sequence of sub-trees of G, in our case

starting with u as root of T. Let $M = \{v_1, \ldots, v_{n-1}\}$ be an enumeration of $V \setminus \{u\}$. Following the order in M, a random walk starts from every unvisited v_i, until it reaches (some vertex in) T and its loop erasure is added to T.

Proposition 5.3.1 ([284] comp. [139])**.** For a connected and unweighted undirected graph $G = (V, E)$ and a vertex $u \in V$, Wilson's algorithm samples a UST of G with root u. The expected running time is the mean hitting time of G, $\sum_{v \in V \setminus \{u\}} \phi_G(v) t_{c,G}(v, u)$, where $\phi_G(v)$ is the probability that a random walk stays at v in its stationary distribution and where $t_{c,G}(v, u)$ is the commute time between v and u.

Lemma 5.3.1 ([133])**.** Let G be as in Proposition 5.3.1. Its mean hitting time can be rewritten as $\sum_{v \in V \setminus \{u\}} \deg(v) \cdot \rho(u, v)$, which is $\mathcal{O}(\mathrm{ecc}(u) \cdot m)$. In small-world graphs, this is $\mathcal{O}(m \log n)$.

Data Structures. When computing the contribution of a UST T to $N(\cdot)$, we need to update for each edge $e = (a, b) \in E(T)$ its contribution to $N_{u,v}(a, b)$ and $N_{u,v}(b, a)$, respectively – for exactly every vertex v for which (a, b) [or (b, a)] lies on $P_{u,v}$. Hence, the algorithm that aggregates the contribution of UST T to $\tilde{\rho}(u, \cdot)$ needs to traverse $P_{u,v}$ for each vertex $v \in V$. To this end, we represent the BFS tree B_u as an array of parent pointers for each vertex $v \in V$. On the other hand, the tree T can conveniently be represented by storing a child and a sibling for each vertex $v \in V$. Compared to other representations (e.g., adjacency lists), this data structure can be constructed and traversed with low constant overhead.

Tree Aggregation. After constructing a UST T, we process it to update the intermediate effective resistance values $\tilde{\rho}(u, \cdot)$. Note that we can discard T afterwards as we do not need to store the full sample, which has a positive effect on the memory footprint of our algorithm. The aggregation algorithm is shown in Algorithm 9. Recall that we need to determine for which vertex v and each edge $(a, b) \in P_{u,v}$, whether (a, b) or (b, a) occurs on the unique u-v path in T. To simplify this test, we root T at u; hence, it is enough to check if (a, b) [or (b, a)] appears above v in T. For general graphs, such a test still incurs quadratic overhead in running time – in particular, the number of vertex-edge pairs that need to be considered is $f(u) = \sum_{v \in V \setminus \{u\}} |P_{u,v}| \in \mathcal{O}(n^2)$. We remark that, perhaps surprisingly, a bottom-up traversal of T does not improve on this, either; it is similarly difficult to determine all $\tilde{\rho}(u, v)$ that a given $(a, b) \in E(T)$ contributes to – those v form an arbitrary subset of descendants of b in T. However, we can exploit the fact that, on small-world networks, the depth of B_u can be controlled, i.e., $f(u)$ is sub-quadratic. To accelerate the test, we first compute a DFS data structure for T, i.e., we determine discovery and finish timestamps for all vertices in V, respectively (Line 2). For an arbitrary $v \in V$ and $(a, b) \in V \times V$, this data structure allows us to answer in constant time (i) whether either (a, b) or (b, a) is in T and (ii) if $(a, b) \in E(T)$, whether v appears below (a, b) in T. Finally, the first `for`-loop iterates over all $v \in V \setminus \{u\}$ while the second `for`-loop iterates over all $e = (a, b) \in P_{u,v}$ and aggregates the contribution of T to $N_{u,v}(a, b)$. To do so, we test whether (a, b) [or (b, a)] is in T in Line 5 [or Line 8, respectively]. If that is indeed the case, in Line 6 [or Line 9] we check whether v is below (a, b) [or (b, a), respectively] and, if that is the case, we add [subtract] 1 to [from] $\tilde{\rho}(u, v)$ in Line 7 [Line 10]. If e is not in B_u, $\tilde{\rho}(u, v)$ does not change.

Algorithm 9 Aggregation of T's contribution to $\tilde{\rho}(u, \cdot)$.

Input: Spanning tree T, effective resistance estimates $\tilde{\rho}(u, \cdot)$, BFS tree B_u
Output: $\tilde{\rho}(u, \cdot)$ updated with T's contribution

```
function AGGREGATE((T, ρ̃(u,·), B_u))
    ⟨t_dis, t_fin⟩ ← DFS(T)                    ▷ t_dis(v), t_fin(v): discovery and finish timestamps of vertex v
    for v ∈ V \ {u} do
        for (a, b) ∈ P_{u,v} obtained from B_u do
            if parent(b) = a then
                if t_dis(b) < t_dis(v) and t_fin(v) < t_fin(b) then
                    ρ̃(u, v) ← ρ̃(u, v) + 1
            else if parent(a) = b then
                if t_dis(a) < t_dis(v) and t_fin(v) < t_fin(a) then
                    ρ̃(u, v) ← ρ̃(u, v) − 1
    return ρ̃(u, ·)
```

5.3.3 Algorithm Analysis

The choice of the pivot u has en effect on the time complexity of our algorithm. The intuitive reason is that the BFS tree B_u should be shallow in order to have short paths to the root u. This is achieved by a root u with small eccentricity. Regarding tree aggregation, we obtain:

Lemma 5.3.2 ([17])**.** Tree aggregation (Algorithm 9) has time complexity $\mathcal{O}(f(u))$, which can be bounded by $\mathcal{O}(n \cdot \mathrm{ecc}(u)) = \mathcal{O}(n \cdot \mathrm{diam}(G))$.

In high-diameter networks, the farness of u can become quadratic in n (consider a path graph) and thus problematic for large inputs. In small-world graphs, however, we obtain $\mathcal{O}(n \log n)$ time per aggregation. We continue the analysis with the main algorithmic results.

Theorem 5.3.2 ([17])**.** Let G be an undirected and unweighted graph with Laplacian matrix $\mathbf{L} = \mathbf{L}(G)$. Then, Algorithm 8 computes an approximation of $\mathrm{diag}\left(\mathbf{L}^{\dagger}\right)$ with absolute error $\pm\varepsilon$ with probability $1 - \delta$ in time $\mathcal{O}(m \cdot \mathrm{ecc}^3(u) \cdot \varepsilon^{-2} \cdot \log(m/\delta))$. For small-world graphs and with $\delta := 1/n$ to get high probability, this yields a time complexity of $\mathcal{O}(m \log^4 n \cdot \varepsilon^{-2})$.

Thus, for small-world networks, we have an approximation algorithm whose running time is nearly-linear in m (i.e., linear up to a polylogarithmic factor), quadratic in $1/\varepsilon$, and logarithmic in $1/\delta$. By choosing a "good" pivot u, it is often possible to improve the running time of Algorithm 8 by a constant factor (i.e., without affecting the $\mathcal{O}$-notation). In particular, there are vertices u with $\mathrm{ecc}(u)$ as low as $\frac{1}{2}\mathrm{diam}(G)$.

Remark 5.3.2. If G has constant diameter, Algorithm 8 obtains an absolute ε-approximation guarantee in $\mathcal{O}(m \log n \cdot \varepsilon^{-2})$ time. This is faster than the best JLT-based approximation – which, in turn, provides a relative approximation guarantee.

5.4 Generalizations

In this section, we show how our algorithm can be adapted to work for weighted graphs, for normalized random-walk betweenness, and for Kirchhoff-related indices.

Extension to Weighted Graphs. For an extension to weighted graphs, we need a weighted version of Kirchhoff's theorem. To this end, the weight of a spanning tree T is defined as the product of the weights (i.e., the conductances) of its edges. Then, let N^* be the sum of the weights of all spanning trees of G; also, let $N^*_{s,t}(a,b)$ be the sum of the weights of all spanning trees in which the unique path from s to t traverses the edge $\{a,b\}$ in the direction from a to b.

Theorem 5.4.1 (comp. [54], p. 46)**.** There is a distribution of currents satisfying Ohm's law and Kirchhoff's laws in which a current of size 1 enters at s and leaves at t. The value of the current on an edge $\{a,b\}$ is given by $(N^*_{s,t}(a,b) - N^*_{s,t}(b,a))/N^*$.

Consequently, our sampling approach needs to estimate N^* as well as the $N^*(\cdot)$-values. It turns out that no major changes are necessary. Wilson's algorithm also yields a UST for weighted graphs (if its random walk takes edge weights for transition probability into account [284]). Yet, the running time bound for Wilson needs now to mention the graph volume $\mathrm{vol}(G)$, specifically: $\mathcal{O}(\mathrm{ecc}(u) \cdot \mathrm{vol}(G))$. The weight of each sampled spanning tree can be accumulated during each run of Wilson. It has to be integrated into Algorithm 9 by adding [subtracting] the tree weight in Line 7 [Line 10] instead of 1. For the division at the end (Algorithm 8, Line 12), one has to replace τ by the total weight of the sampled trees. Finally, the tree B_u remains a BFS tree. The eccentricity and farness of u still refer in the analysis to their unweighted versions, respectively, as far as B_u is concerned.

To conclude, the only important change regarding bounds happens in Theorem 5.3.2. In the time complexity, m is replaced by $\mathrm{vol}(G)$.

Normalized Random-Walk Betweenness. Narayan and Saniee [224] propose NRWB as a measure for the influence of a vertex in the network, but the paper does not provide an algorithm to compute it – beyond implicit (pseudo)inversion. We propose to compute NRWB with Algorithm 8 and derive the following Lemma.

Lemma 5.4.1 ([17])**.** Normalized random-walk betweenness $c_{\mathrm{nrwb}}(v)$ (Eq. (5.3)) can be rewritten as:

$$c_{\mathrm{nrwb}}(v) = \frac{1}{n} + \frac{\mathrm{tr}\left(\mathbf{L}^\dagger\right)}{(n-1)f_e(v)}.$$

Hence, since Algorithm 8 approximates the diagonal of $\mathbf{L}^\dagger$ and both trace and electrical farness depend only on the diagonal, the following proposition holds:

Proposition 5.4.1. Let $G = (V,E)$ be a small-world graph as in Theorem 5.3.2. Then, Algorithm 8 approximates with high probability $c_{\mathrm{nrwb}}(v)$ for all $v \in V$ with absolute error $\pm\varepsilon$ in $\mathcal{O}(m \log^4 n \cdot \varepsilon^{-2})$ time.

Kirchhoff Index and Edge Centralities. It is easy to see that Algorithm 8 can approximate Kirchhoff index by exploiting the expression $\mathcal{K}(G) = n \cdot \mathrm{tr}\left(\mathbf{L}^\dagger\right)$ [167]. As a direct consequence, we have:

Proposition 5.4.2. Let G be a small-world graph as in Theorem 5.3.2. Then, Algorithm 8 approximates with high probability $\mathcal{K}(G)$ with absolute error $\pm\varepsilon$ in $\mathcal{O}(m \log^4 n \cdot \varepsilon^{-2})$ time.

We also observe that we can use a component of Algorithm 8 to approximate $c_\theta^\Delta(e)$. Recall from Section 5.2 that $c_\theta^\Delta(e) = c_\theta(e) - \mathcal{K}(G) = n(\mathrm{tr}(\mathbf{L}^\dagger \backslash_\theta e) - \mathrm{tr}(\mathbf{L}^\dagger))$. Using the Sherman-Morrison formula, as done in Ref. [184], we have that:

$$\theta^\Delta(e) = n(1-\theta)\frac{w(e)\mathrm{tr}(\mathbf{L}^\dagger \mathbf{b}_e \mathbf{b}_e^\top \mathbf{L}^\dagger)}{1-(1-\theta)w(e)\mathbf{b}_e^\top \mathbf{L}^\dagger \mathbf{b}_e}, \tag{5.7}$$

where $\mathbf{b}_e$ for $e = (u, v)$ is the vector $\mathbf{e}_u - \mathbf{e}_v$.

Li and Zhang [184] approximate $c_\theta^\Delta(e)$ with an algorithm that runs in time $\mathcal{O}(m\theta^{-2}\log^{2.5} n \log(1/\varepsilon)\mathrm{poly}(\log\log n) \cdot \varepsilon^{-2})$. The algorithm is dominated by the denominator of Eq. (5.7), which takes $\mathcal{O}(m\theta^{-2}\log^{2.5} n\mathrm{poly}(\log\log n) \cdot \varepsilon^{-2})$ time. For the numerator of Eq. (5.7), they use the following Lemma:

Lemma 5.4.2 (paraphrasing from Ref. [184])**.** Let $\mathbf{L}$ be a Laplacian matrix and ε be a scalar such that $0 < \varepsilon \leq 1/2$. There is an algorithm that achieves an ε-approximation of the numerator of Eq. (5.7) with high probability in $\mathcal{O}(m\log^{1.5} n \log(1/\varepsilon) \cdot \varepsilon^{-2})$.

The algorithm in Lemma 5.4.2 uses the Monte-Carlo estimator with $\mathcal{O}(\varepsilon^{-2}\log n)$ random vectors $\mathbf{z}_j$ to calculate the trace of the implicit matrix $\mathbf{y}_j^\top \mathbf{b}_e \mathbf{b}_e^\top \mathbf{y}_j$, where $\mathbf{y}_j$ is the approximate solution of $y_j := \mathbf{L}^\dagger \mathbf{z}_j$ – derived from solving the corresponding linear system involving $\mathbf{L}^\dagger$. For each system, the Laplacian solver runs in $\mathcal{O}(m\log^{1/2} n \log(1/\varepsilon))$ time.

We notice that a UST-based sampling approach works again for the denominator: The denominator is just $1-(1-\theta)w(e)\rho(e)$, where $e \in E$ and $\rho(e) = \mathbf{b}_e^\top \mathbf{L}^\dagger \mathbf{b}_e$. Approximating $\rho(e)$ for every $e \in E$ then requires sampling USTs and counting for each edge e the number of USTs it appears in. Moreover, we only need to sample $q = \lceil 2\varepsilon^{-2}\log(2m/\delta)\rceil$ to get an ε-approximation of the effective resistances for all edges (using [139, Theorem 8]). Since $\rho(e)$ are approximate, we need to bound their approximation when subtracted from 1. Following Ref. [184], we use the fact that $\theta \in (0, 1)$ and that for each edge $w(e)\rho(e)$ is between 0 and 1, bounding the denominator. The above algorithm can be used to approximate the denominator of Eq. (5.7) with absolute error $\pm\varepsilon$ in $\mathcal{O}(m\log^2 n \cdot \varepsilon^{-2})$ time. Combining the above algorithm and Lemma 5.4.2, it holds that:

Proposition 5.4.3. Let $G = (V, E)$ be a small-world graph as in Theorem 5.3.2. Then, there is an algorithm (using Lemma 5.4.2 and our Wilson-based sampling algorithm) that approximates with high probability $c_\theta^\Delta(e)$ for all $e \in E$ with absolute error $\pm\varepsilon$ in $\mathcal{O}(m\log^2 n\log(1/\varepsilon) \cdot \varepsilon^{-2})$ time.

5.5 Extension to Forest Closeness

In this section, we describe how our algorithm can be adapted to forest closeness. The key ingredient is the relation between forest distance and effective resistance which allows us to approximate the forest farness more efficiently than existing approximation algorithms. By adapting Algorithm 8 to forest closeness, we obtain an algorithm with a (probabilistic) additive approximation guarantee of $\pm\varepsilon$ that runs in nearly-linear (in m) expected time.

Algorithm 10 Approximation algorithm for diag($\mathbf{\Omega}$).

Input: Undirected graph $G = (V, E, w)$, control parameter α, error bound $\varepsilon \in (0, 1)$, probability $\delta \in (0, 1)$

Output: diag$\left(\widetilde{\mathbf{\Omega}}\right)$, i.e., an (ε, δ)-approximation of diag($\mathbf{\Omega}$)

1: Create augmented graph $G_\star = (V_\star, E_\star)$ as described in Section 5.5.1, compute vol(G) and c ▷ $\mathcal{O}(m + n)$
2: $u_\star \leftarrow$ universal vertex of $G_\star$
3: Pick constant $\kappa \in (0, 1)$ arbitrarily
4: $\eta \leftarrow \frac{\kappa\varepsilon}{6\sqrt{\alpha(c+2)\text{vol}(G)}}$
5: $\tau \leftarrow \left\lceil \frac{\log(2m/\delta)}{2(1-\kappa)^2\varepsilon^2} \right\rceil$
6: **for** $j \leftarrow 1$ **to** τ **do**
7: $\quad \tilde{\rho}_\star(u_\star, \cdot) \leftarrow$ SampleUST$(G_\star, u_\star)$ ▷ $\mathcal{O}(\alpha\text{vol}(G) + n)$
8: Solve $\mathbf{L}_\star\mathbf{x} = \mathbf{e}_{u_\star} - \frac{1}{n+1} \cdot \mathbf{j}$ for $\mathbf{x}$ with accuracy η ▷ $\tilde{\mathcal{O}}(m \log^{1/2} n \log(1/\eta))$
9: **for** $v \in V$ **do** ▷ Overall: $\mathcal{O}(n)$
10: $\quad \widetilde{\mathbf{\Omega}}[v, v] \leftarrow \tilde{\rho}_\star(u_\star, v)/\tau - \mathbf{x}[u_\star] + 2\mathbf{x}[v]$ ▷ Unweighted case, see text for weighted case
11: **return** diag$\left(\widetilde{\mathbf{\Omega}}\right)$

5.5.1 From Forest Farness to Electrical Farness (And Back Again)

As mentioned above, we exploit a result that relates forest distance to effective resistance. This requires the creation of an *augmented graph* $G_\star := G_{\star,\alpha} := (V_\star, E_\star)$ from the original graph $G = (V, E)$. To this end, a new *universal vertex* $u_\star$ is added to G, such that $V_\star = V \cup \{u_\star\}$ and $E_\star = E \cup \{\{u_\star, v\}\}\ \forall v \in V$. In particular, $u_\star$ is connected to all other vertices of $G_\star$ with edges of weight 1. Furthermore, the weights of all edges in $E_\star$ that belong to E are multiplied by α.

Proposition 5.5.1 (comp. Ref. [69])**.** For a weighted graph $G = (V, E, w)$ and any vertex pair $\langle u, v\rangle \in V \times V$, the forest distance $\zeta(u, v)$ in G equals the effective resistance $\rho(u, v)$ in the augmented graph $G_\star$.

The full proof of Proposition 5.5.1 can be found in Ref. [69]. Nevertheless, we provide here an explanation of why the above proposition holds. Recall that the effective resistance between any two vertices of G is computed by meas of $\mathbf{L}^\dagger$, while the forest distances of the same pair are computed by means of the forest matrix of G, i.e., $\mathbf{\Omega} = (\alpha\mathbf{L} + \mathbf{I})^{-1}$. When calculating the effective resistance in $G_\star$, we use its Laplacian matrix $\mathbf{L}_\star$, which consists of a block matrix corresponding to $(\alpha\mathbf{L} + \mathbf{I})$ and an additional row and column that correspond to the universal vertex $u_\star$. It turns out that the Moore-Penrose pseudoinverse of $\mathbf{L}_\star$ is the block matrix that consists of $\mathbf{\Omega}$ with an additional row and column corresponding to $u_\star$ [69]. Thus:

$$\mathbf{\Omega}[u_\star, u_\star] + \mathbf{\Omega}[v, v] - 2\mathbf{\Omega}[u_\star, v] = \mathbf{L}_\star^\dagger[u_\star, u_\star] + \mathbf{L}_\star^\dagger[v, v] - 2\mathbf{L}_\star^\dagger[u_\star, v],$$

which corresponds to the pairwise effective resistance $\rho(u_\star, v)$ in $G_\star$ – hereafter, we refer to this quantity as $\rho_\star(u_\star, v)$.

Corollary 5.5.1. Forest closeness in graph G equals electrical closeness in the augmented graph $G_\star$.

5.5.2 Forest Farness Approximation Algorithm

As mentioned, our algorithm for forest closeness exploits the algorithmic results we presented in Section 5.3 fir approximating diag$\left(\mathbf{L}^\dagger\right)$ and electrical closeness. To do so, we rewrite the forest farness $f_f(v)$ following Ref. [213]:

Algorithm 11 Sampling algorithm for USTs based on Wilson's algorithm [284]

Input: Graph $G = (V, E)$, universal vertex $u_\star \in V$
Output: Estimated effective resistance values $\tilde{\rho}_\star(u_\star, \cdot)$

```
function SampleUST(G, u_⋆))
    ρ̃_⋆(u_⋆, v) ← 0 ∀v ∈ V
    T ← {u_⋆}
    Let v_1, ..., v_n be a reordering of the vertices in V in ascending degree
    for j ← 1 to n do
        P ← random walk on G from v_j to T
        LE(P) ← loop erasure of P in order of appearance
        T ← T ∪ LE(P)
        if last vertex of LE(P) is u_⋆ then
            z ← last visited vertex before u_⋆
            ρ̃_⋆(u_⋆, z) ← ρ̃_⋆(u_⋆, z) + 1
    return ρ̃_⋆(u_⋆, ·)
```

$$f_f(v) = n \cdot \mathbf{\Omega}[v, v] + \mathrm{tr}(\mathbf{\Omega}) - 2 \sum_{z \in V} \mathbf{\Omega}[v, z] = n \cdot \mathbf{\Omega}[v, v] + \mathrm{tr}(\mathbf{\Omega}) - 2, \tag{5.8}$$

where the last equation holds since $\mathbf{\Omega}$ is doubly stochastic ($\mathbf{\Omega}[v, v] = 1 - \sum_{z \in V \setminus \{v\}} \mathbf{\Omega}[v, z]$) [213]. From Eq. (5.8) it is clear that we only require the diagonal elements of $\mathbf{\Omega}$ to compute $f_f(v)$ for any $v \in V$. We approximate the diagonal elements of $\mathbf{\Omega}$ with Algorithm 10; similarly to Algorithm 8, the main idea is to sample USTs to approximate $\mathrm{diag}\left(\mathbf{L}_\star^\dagger\right)$:

1. We build the augmented graph $G_\star$ (Line 1) and let the universal vertex $u_\star$ of $G_\star = (V_\star, E_\star)$ be the pivot vertex (Line 2) – due to its optimal eccentricity of 1. Later, in Line 8, we compute the column of $\mathbf{\Omega}$ that corresponds to $u_\star$, namely $\mathbf{\Omega}[:, u_\star]$, by solving the Laplacian linear system $\mathbf{L}_\star \mathbf{x} = \mathbf{e}_{u_\star} - \frac{1}{n+1}\mathbf{j}$. The solver's accuracy is controlled by η, which is set in Line 4 – as in Algorithm 8, κ is used to trade the accuracy of the solver with the accuray of the following sampling step.

2. We sample τ USTs in $G_\star$ with Wilson's algorithm [284] (Algorithm 11), where the sample size τ is yet to be determined. With this sample we approximate the effective resistance $\rho_\star(u_\star, v)$ for all $v \in V$ (Lines 6–7). More precisely, if an edge $\{u_\star, v\}$ appears in the sampled tree, we increase $\tilde{\rho}_\star(u_\star, v)$ by 1 (in the unweighted case) or by the weight of the current tree (in the weighted case) – and later "return" $\tilde{\rho}_\star(u_\star, v)/\tau$ (unweighted case) or the relative total weight of all sampled trees (weighted case) that contain edge $\{u_\star, v\}$ in Line 10.

3. We compute the remaining $\mathbf{\Omega}[v, v]$ in Lines 9–10 following Eqs. (5.1) and (5.4):

$$\mathbf{\Omega}[v, v] = \zeta(u_\star, v) - \mathbf{\Omega}[u_\star, u_\star] + 2\mathbf{\Omega}[v, u_\star] = \rho_\star(u_\star, v) - \mathbf{\Omega}[u_\star, u_\star] + 2\mathbf{\Omega}[v, u_\star],$$

   where $\rho_\star(u_\star, v)$ is approximated by $\tilde{\rho}_\star(u_\star, v)/\tau$. The weighted case is handled as described in the previous step.

By using $G_\star$ and thus a universal vertex $u_\star$ as pivot, there are several noteworthy changes compared to Algorithm 8. First, the graph $G_\star$ has constant diameter and the vertex $u_\star$ constant eccentricity 1. This will important for our redefined running time analysis. Second, the approximation of the effective resistances

can be simplified: while Algorithm 8 requires an aggregation along shortest paths, we notice that here $u_\star$ and all other vertices are connected by paths of one edge only; thus, the relative frequency of an edge $\{u_\star, v\}$ in the UST sample for $G_\star$ is sufficient here for our approximation:

Proposition 5.5.2 ([133])**.** Let $u_\star$ be the universal vertex in $G_\star$. Then, for any edge $\{u_\star, v\} \in E_\star$ holds: its relative frequency (or weight) in the UST sample is an unbiased estimator for $\rho_\star(u_\star, v)$.

As we will see in Theorem 5.5.1, Algorithm 10 is not only an unbiased estimator, but even provides a probabilistic approximation guarantee. To bound its running time, we analyze Wilson's algorithm for generating USTs first.

Proposition 5.5.3 ([133])**.** For an undirected graph G with constant diameter, each call to Wilson's algorithm on $G_\star$ (in Line 7) takes $\mathcal{O}(\alpha \cdot \mathrm{vol}(G) + n)$ expected time.

Note that, in the unweighted case with $\alpha = 1$ and $m = \Omega(n)$ (which is not uncommon in our context, see e.g., Ref. [153]), we obtain a time complexity of $\mathcal{O}(m)$ (the volume is $2m$ by the handshake lemma). Taking all the above into account, we can adapt Theorem 5.3.2 to forest centrality. Note that, when considering forest (as opposed to electrical) closeness centrality, we exploit the constant diameter of $G_\star$ and improve the time by a factor of $(\mathrm{ecc}(u))^3$, where u is the selected pivot vertex. This expression is $\mathcal{O}(\log^3 n)$ for small-diameter graphs, but can be larger for general graphs. In the following theorem, $\tilde{\mathcal{O}}$ hides polylogarithmic factors from the linear solver [80].

Theorem 5.5.1 ([133])**.** Let $n/(\alpha \cdot \mathrm{vol}(G))$ be bounded from above by a constant[29] and let $\varepsilon, \delta \in (0, 1)$. Then, with probability $1 - \delta$, Algorithm 10 computes an approximation of $\mathrm{diag}(\mathbf{\Omega})$ with absolute error $\pm\varepsilon$ in (expected) time $\tilde{\mathcal{O}}((m \log^{1/2} n \log(\sqrt{\alpha \cdot \mathrm{vol}(G)}/\varepsilon))) + \mathcal{O}(\log(n/\delta) \cdot \varepsilon^{-2} \cdot \alpha \cdot \mathrm{vol}(G))$.

Let us simplify the result for a common case:

Corollary 5.5.2. If G is unweighted, α a constant and $\delta := 1/n$ to get high probability, the (expected) running time of Algorithm 10 becomes $\tilde{\mathcal{O}}(m(\log^{1/2} n \log(n/\varepsilon) + \varepsilon^{-2} \log n))$. Assuming ε small enough so that $\log n \leq 1/\varepsilon$, we can further simplify this to $\tilde{\mathcal{O}}(m\varepsilon^{-2} \log^{3/2} n)$.

This is nearly-linear in m, which is also true for the JLT-based approximation (with high probability) of Jin et al. [153]. They state a running time of $\tilde{\mathcal{O}}(m\varepsilon^{-2} \log^{5/2} n \log(1/\varepsilon))$ for unweighted G and fixed $\alpha = 1$. While we save at least a factor of $\log n$, they achieve a *relative* approximation guarantee, which is difficult to compare to our absolute approximation guarantee.

5.6 Engineering Aspects and Parallelization

In this section, we illustrate important engineering decisions concerning the choice of the UST sampling algorithm, selection of the pivot u,[30] and the linear solver used for the initial linear system.

[29] This condition ensures that the algorithm is not affected by unduly heavy additional edges to $u_\star$. If the condition is met, the graph edges still play a reasonable role in the distances and in the UST computations.

[30] Note that optimizations to the pivot selection only apply to the electrical closeness case; for forest closeness, we always pick the universal vertex $u_\star$ as pivot (see Line 3, Algorithm 11).

5.6.1 UST Generation, Pivot Selection, and the Linear System

Wilson's algorithm [284] using loop-erased random walks is the best choice in practice for UST sampling and also the fastest asymptotically for unweighted small-world graphs. For our implementation in Algorithm 8, we use a variant of Wilson's algorithm to sample each tree, proposed by Hayashi et al. [139]: first, one computes the biconnected components of G, then applies Wilson to each biconnected component, and finally combines the component trees to a UST in G. In each component, we use a vertex with maximal degree as root for Wilson's algorithm. Using this approach, Hayashi et al. [139] experience an average performance improvement of 40% on sparse graphs compared to running Wilson directly.

As a consequence Theorem 5.3.2, the pivot vertex u should be chosen to have low eccentricity. As finding the vertex with lowest eccentricity with a naive APSP approach would be too expensive, we compute a lower bound on the eccentricity of all vertices of the graph and choose u as the vertex with lowest bound. These bounds are computed using a strategy analogous to the double sweep lower bound by Magnien et al. [199]: we run a BFS from a random vertex v, then another BFS from the farthest vertex from v, and so on. At each BFS we update the bounds of all the visited vertices; an empirical evaluation has shown that 10 iterations yield a reasonably accurate approximation of the vertex with lowest eccentricity.

As a result from preliminary experiments, we use a general-purpose CG solver for the single (sparse) Laplacian linear systems, together with a diagonal preconditioner. We choose the implementation of the C++ library Eigen [135] for this purpose and found that the accuracy parameter $\kappa = 0.3$ yields a good trade-off between the CG and UST sampling steps.

5.6.2 Parallel Implementation

In the following, we explain how we parallelize our algorithms to take advantage of multiple cores and multiple compute nodes. We always assume that the entire input graph fits into main memory, even in the distributed case.

Shared Memory. We turn the main sampling loops in Algorithms 8 and 10 into parallel `for`-loops. Our implementation uses OpenMP for shared-memory parallelism. We aggregate $\tilde{\rho}(u, \cdot)$ (for forest closeness, $\tilde{\rho}_\star(u_\star, \cdot)$) in thread-local arrays and perform a final parallel reduction over all $\tilde{\rho}(u, \cdot)$. We found that on the graphs that we can handle in shared memory, no sophisticated load balancing strategies are required to achieve reasonable parallel scalability. We do not employ parallelism in the other steps of the algorithm. In particular, the BFS to compute B_u is executed sequentially. We also do not parallelize over the loops in Algorithm 9 to avoid nested parallelism with multiple invocations of Algorithm 9. We note that, in contrast to the theoretical work-depth model, solving the initial Laplacian system and performing the BFS are not the main bottlenecks in practice. Instead, sampling USTs (and aggregating them, for electrical closeness) consume the majority of CPU time (see Figure 5.4).

Distributed Memory. We provide an implementation of our algorithm for replicated graphs in distributed memory that exploits hybrid parallelism based on MPI + OpenMP. On each compute node, we take samples and aggregate $\tilde{\rho}(u, \cdot)$ as in shared memory. Compared to the shared-memory implementation, however, our distributed-memory implementation exhibits two main peculiarities: (i) we still solve

the initial Laplacian system on a single compute node only; we interleave, however, this step, with UST sampling on other compute nodes, and (ii) we employ explicit load balancing. The choice to solve the initial system on a single compute node only is done to avoid additional communication overheads among nodes. In fact, we only expect distributed CG solvers to outperform this strategy for inputs that are considerably larger than the largest graphs we consider. Furthermore, since we interleave this step of the algorithm with UST sampling on other compute nodes, our strategy only results in a bottleneck on input graphs where solving a single Laplacian system is slower than taking *all* UST samples – but these inputs are already "easy".

For load balancing, the naive approach would consist of statically taking $\lceil \tau/p \rceil$ UST samples on each of the p compute nodes. However, in contrast to the shared-memory case, this does not yield satisfactory scalability. In particular, for large graphs, the running time of the UST sampling step has a high variance. To alleviate this issue, we use a simple *dynamic* load balancing strategy: periodically, we perform an asynchronous reduction (`MPI_Iallreduce`) to calculate the total number of UST samples taken so far (over all compute nodes). Afterwards, each compute node calculates the number of samples that it takes before the next asynchronous reduction – unless more than τ samples were taken already, in which case the algorithm stops. We compute this number as $\lceil \tau/(b \cdot p^{\xi}) \rceil$ for fixed constants b and ξ. We also overlap the asynchronous reduction with additional sampling to avoid idle times. Finally, we perform a synchronous reduction (`MPI_Reduce`) to aggregate $\tilde{\rho}(u, \cdot)$ on a single compute node before returning the resulting diagonal values. By parameter tuning [14], we found that choosing $b = 25$ and $\xi = 0.75$ yields the best parallel scalability.

5.7 Experiments – Electrical Closeness

We conduct experiments to demonstrate the performance of our algorithm for electrical closeness (Section 5.3) compared to the state-of-the-art competitors.

5.7.1 Settings

Unless stated otherwise, all algorithms are implemented in C++, using the NetworKit graph APIs [273]. Our own algorithm is labelled UST in the sections below. All experiments are conducted on Intel Xeon Gold 6126 machines with 2×12 cores and 192 GiB of RAM each. Unless stated otherwise, experiments run sequentially on a single core. To ensure reproducibility, all experiments were managed by the SimexPal [14] software. We executed our experiments on the graphs in Table 5.1. All graphs are unweighted and undirected and have been downloaded from the public repository KONECT [173].

Quality Measures and Baseline. To evaluate the diagonal approximation quality, we measure the maximum absolute error – i.e., $\max_v |\mathbf{L}^{\dagger}[v,v] - \widetilde{\mathbf{L}}^{\dagger}[v,v]|$. Since for some applications [227, 229] a correct ranking of the entries is more relevant than their scores, in our experimental analysis we compare the complete rankings of the diagonal entries. Note that the lowest entries of $\mathrm{diag}(\mathbf{L}^{\dagger})$ – corresponding to the vertices with highest electrical closeness – are distributed on a significantly narrow interval. Hence, to achieve an accurate electrical closeness ranking of the top-k vertices, one would need to solve the problem with very high accuracy. For this reason, all approximation algorithms we consider do not yield a precise top-k ranking, so that we (mostly) consider the complete ranking.,

Table 5.1: Real-world instances used in our experiments for electrical closeness.

(a) Medium-size instances with ground truth

Network	Type	ID	n	m	diam,	ecc(u)
slashdot-zoo	social	`sz`	79,116	467,731	12	6
petster-cat-household	social	`pc`	68,315	494,562	10	6
wikipedia_link_ckb	web	`wc`	60,257	801,794	13	7
wikipedia_link_fy	web	`wf`	65,512	921,533	10	5
loc-gowalla_edges	social	`lg`	196,591	950,327	16	8
petster-dog-household	social	`pd`	255,968	2,148,090	11	6
livemocha	social	`lm`	104,103	2,193,083	6	4
petster-catdog-household	social	`pa`	324,249	2,642,635	12	7

(b) Medium-size instances without ground truth

Network	Type	ID	n	m	diam.	ecc(u)
eat	words	`ea`	23,132	297,094	6	4
web-NotreDame	web	`wn`	325,729	1,090,108	46	23
citeseer	citation	`cs`	365,154	1,721,981	34	18
wikipedia_link_ml	web	`wm`	131,288	1,743,937	12	7
wikipedia_link_bn	web	`wb`	225,970	2,183,246	11	6
flickrEdges	images	`fe`	105,722	2,316,668	9	6
petster-dog-friend	social	`pr`	426,485	8,543,321	11	7

(c) Large instances

Network	Type	n	m	diam.	ecc(u)
hyves	social	1,402,673	2,777,419	10	7
com-youtube	social	1,134,890	2,987,624	24	12
flixster	social	2,523,386	7,918,801	8	4
petster-catdog-friend	social	575,277	13,990,793	13	7
flickr-links	social	1,624,991	15,473,043	24	12

(d) Instances used only on 16×24 cores

Network	Type	n	m	diam.	ecc(u)
petster-carnivore	social	601,213	15,661,775	15	8
soc-pokec-relationships	social	1,632,803	22,301,964	14	8
soc-LiveJournal1	social	4,843,953	42,845,684	20	10
livejournal-links	social	5,189,808	48,687,945	23	12
orkut-links	social	3,072,441	117,184,899	10	6
wikipedia_link_en	web	13,591,759	334,590,793	12	7

Using `pinv` in NumPy or Matlab as a baseline would be too expensive in terms of time (cubic) and space (quadratic) on large graphs – see Table 5.1. Thus, as a quality baseline we employ the LAMG solver [190] (see also next paragraph) as implemented within NetworKit [273] in our experiments (with 10^{-9} tolerance). The results in Table D.1 in Appendix D.1 indicate that the diagonal obtained in this way is sufficiently accurate.

Competitors in Practice. In practice, the fastest way to compute electrical closeness so far is to combine a dimension reduction via the JLT lemma [155] with a numerical solver. In this context, Algebraic MultiGrid (AMG) solvers exhibit better practical running time than fast Laplacian solvers with a worst-case guarantee [207]. For our experiments, we use JLT combined with LAMG [195] (named Lamg-jlt); the latter is an AMG-type solver for complex networks. We also compare against a Julia implementation of JLT together with the fast Laplacian solver proposed by Kyng et al. [183], for which a Julia implementation is already available in the package Laplacians.jl.[31] This solver generates a sparse approximate Cholesky decomposition for Laplacian matrices with provable approximation guarantees in $\mathcal{O}(m \log^3 n \log(1/\varepsilon))$ time; it is based purely on random sampling – and does not make use of graph-theoretic concepts such as low-stretch spanning trees or sparsifiers. We refer to the above implementation as Julia-jlt throughout the experiments. For both Lamg-jlt and Julia-jlt, we try different input error bounds – they correspond to the respective numbers next to the method names in Figure 5.1. This is a relative error, since these algorithms use numerical approaches with a relative error guarantee, instead of an absolute one – see Section 5.7.6 for results in terms of different quality measures.

Finally, we compare against the diagonal estimators due to Bekas et al. [36], one based on random vectors and one based on Hadamard rows. To solve the resulting Laplacian systems, we use LAMG in both cases. In our experiments, the algorithms are referred to as Bekas and Bekas-h, respectively. Excluded competitors are discussed in the following.

Excluded Competitors. PSelInv [151] is a distributed-memory tool for computing selected elements of a matrix $\mathbf{M}^{-1}$ – exactly those that correspond to the non-zero entries of the original matrix $\mathbf{M}$. However, when a smaller set of elements is required (such as $\mathrm{diag}(\mathbf{L}^\dagger)$), PSelInv is not competitive on our input graphs: preliminary experiments of ours have shown that, even on 4×24 cores, PSelInv is one order of magnitude slower than a sequential run of UST.

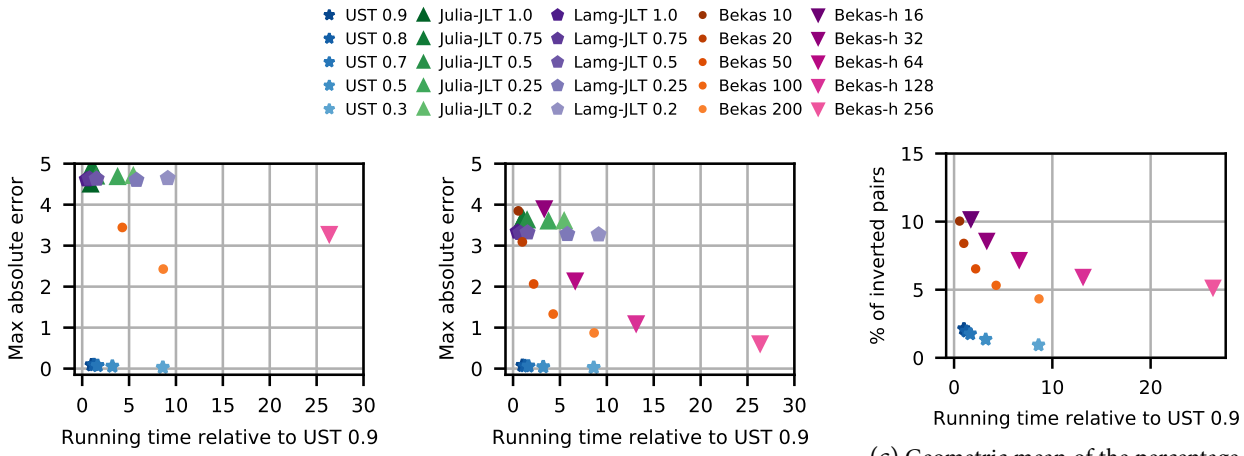
Another conceivable way to compute $\mathrm{diag}(\mathbf{L}^\dagger)$ is to extract the diagonal from a low-rank approximation of $\mathbf{L}^\dagger$ [60] using a few eigenpairs. However, our experiments have shown that this method is not competitive – neither in terms of quality nor in running time.

Hence, we do not include [60, 151] in the presentation of our experiments.

5.7.2 Running Time and Quality

Figure 5.1a shows that, in terms of maximum absolute error, every configuration of UST achieves results with higher quality than the competitors. Even when setting $\varepsilon = 0.9$, UST yields a maximum absolute error of 0.09, and it is $8.3\times$ faster than Bekas with 200 random vectors – which, in turn, achieves a maximum absolute error of 2.43. Furthermore, the running time of UST does not increase substantially for lower

[31] https://github.com/danspielman/Laplacians.jl



(a) Maximum of the maximum absolute errors.

(b) Arithmetic mean of the maximum absolute error.

(c) Geometric mean of the percentage of inverted pairs in the full ranking of $\mathrm{diag}(\mathbf{L}^{\dagger})$.

Figure 5.1: Quality measures over the instances of Table 5.1a. All runs are sequential.

values of ε and its quality does not deteriorate quickly for higher values of ε. Regarding the average of the maximum absolute error, Figure 5.1b shows that, among the competitors, Bekas-h with 256 Hadamard rows achieves the best precision. UST, however, yields an average error of 0.07 while also being $25.4\times$ faster than Bekas-h, which yields an average error of 0.62. Note also that the number next to each method in Figure 5.1 corresponds to different values of absolute (for UST) or relative (for Lamg-jlt and Julia-jlt) error bounds, and different number of samples (for Bekas and Bekas-h). For Bekas-h, the number of samples needs to be a multiple of four due to dimension of Hadamard matrices.

In Figure 5.1c, we report the percentage of inverted pairs in the full ranking of the vertices according to $\mathrm{diag}\left(\widetilde{\mathbf{L}}^{\dagger}\right)$. Note that JLT-based approaches are not depicted in this plot because they yield $> 15\%$ of rank inversions. Among the competitors, Bekas achieves the best time-accuracy trade-off. However, when using 200 random vectors, it yields 4.3% inversions while also being $8.3\times$ slower than UST with $\varepsilon = 0.9$, which yields 2.1% inversions only.

For validation purposes, we also measure how well the considered algorithms compute the *set* (not the ranking) of top-k vertices, i.e., those with highest electrical closeness centrality, with $k \in \{10, 100\}$. For each algorithm we only consider the parameter settings that yields the highest accuracy. JLT-based approaches appear to be very accurate for this purpose, as their top-k sets achieve a Jaccard index of 1.0. As expected (due to its absolute error guarantee), UST performs slightly worse: on average, it obtains 0.95 for $k = 10$ and 0.98 for $k = 100$, which still shows a high overlap with the ground truth.

5.7.3 Memory Consumption

We measure the peak memory consumption of all the algorithms while running sequentially on the instances of Tables 5.1a and 5.1b. More precisely, we subtract the peak resident set size before launching the algorithm from the peak resident size after the algorithm finished. Figure 5.2 shows that UST requires less memory than the competitors on all the considered instances. This can be explained by the fact that, unlike its competitors, our algorithm does not rely on Laplacian solvers with considerable memory overhead.

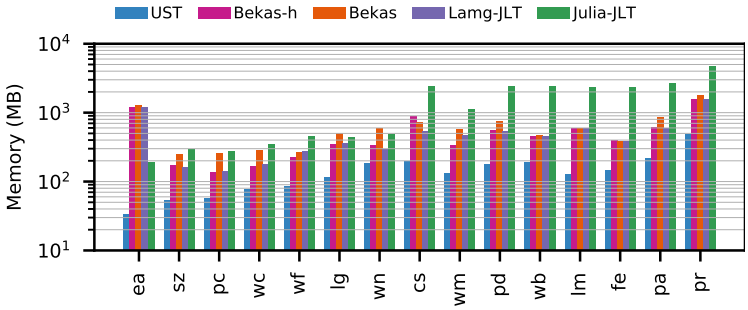
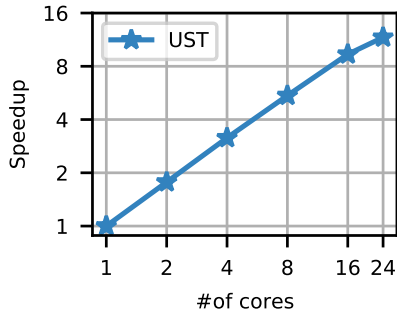
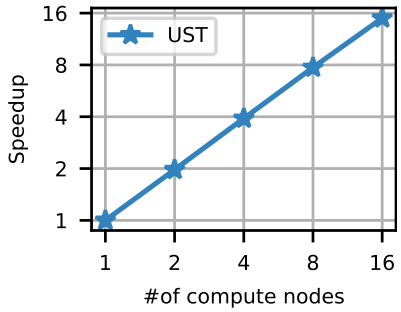


Figure 5.2: Difference between the peak resident set size before and after a sequential run of each algorithm on the instances of Tables 5.1a and 5.1b.



(a) Geometric mean of the speedup of UST on multiple cores (shared memory) w.r.t. a sequential run. Data points are aggregated over the instances of Tables 5.1a and 5.1b.



(b) Geometric mean of the speedup of UST on multiple compute nodes w.r.t. UST on a single compute node (1×24 cores). Data points are aggregated over the instances of Tables 5.1a–5.1c.

Figure 5.3: Parallel scalability of UST (with $\varepsilon = 0.3$) with shared and with distributed memory.

For the largest network in particular, the peak memory is 487.0 MB for UST, and at least 1.6 GB for the competitors.

5.7.4 Parallel Scalability

The log-log plot in Figure 5.3a shows that, on shared memory, UST achieves a moderate parallel scalability w.r.t. the number of cores; on 24 cores in particular, it is $11.9\times$ faster than on a single core. Even though the number of USTs to be sampled can be evenly divided among the available cores, we do not see a nearly-linear scalability: on multiple cores the memory performance of our NUMA system becomes a bottleneck. Therefore, the time to sample a UST increases and using more cores yields diminishing returns. Limited memory bandwidth is a known issue affecting algorithms based on graph traversals in general [26, 195].

Finally, we compare the parallel performance of UST indirectly with the parallel performance of our competitors. Specifically, assuming a perfect parallel scalability for our competitors Bekas and Bekas-h on 24 cores, UST would yield results 4.1 and 12.6 times faster, respectively, even with this strong assumption for the competition's benefit.

UST scales better in a distributed setting. In this case, the scalability is affected mainly by its non-parallel parts and by synchronization latencies. The log-log plot in Figure 5.3b shows that, on up to 4 compute nodes, the scalability is almost linear, while on 16 compute nodes UST achieves a $15.1\times$ speedup w.r.t. a single compute node.

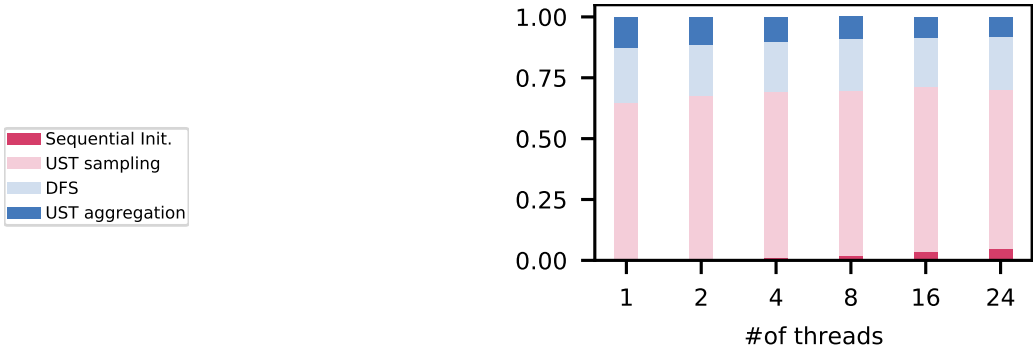
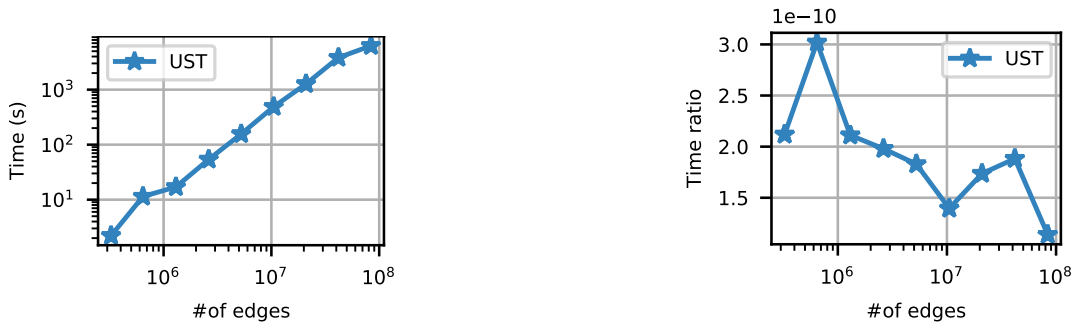


Figure 5.4: Breakdown of the running times of UST with $\varepsilon = 0.3$ w.r.t. #of cores on 1×24 cores. Data is aggregated with the geometric mean over the instances of Tables 5.1a and 5.1b.



(a) Running time of UST w.r.t. #of edges.

(b) Ratio between the running time of UST and its theoretical running time (see Theorem 5.3.2.

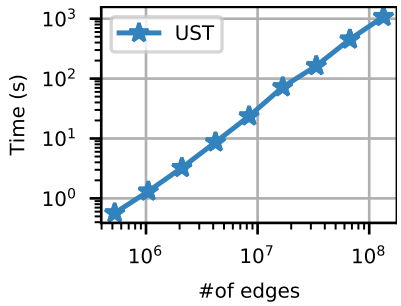
Figure 5.5: Scalability of UST on random hyperbolic graphs ($\varepsilon = 0.3$, 1×24 cores).

Figure 5.4 shows the fraction of time UST spends on different tasks depending on the number of cores. We aggregated over "Sequential Init." the time spent on memory allocation, pivot selection, solving the linear system, the computation of the biconnected components, and on computing the BFS tree B_u. In all configurations, UST spends the majority of the time in sampling, computing the DFS data structures, and aggregating USTs. The total time spent on aggregation corresponds to "UST aggregation" and "DFS" in Figure 5.4, indicating that computing the DFS data structures is the most expensive part of the aggregation. Together, sampling time and aggregation time account for 99.4% and 95.3% of the total running time on 1 core and 24 cores, respectively. On average, sampling takes 66.8% of this time, while total aggregation takes 31.2%. Since sampling a UST is on average $2.2\times$ more expensive than computing the DFS timestamps and aggregation, faster sampling techniques would significantly improve the performance of our algorithm.

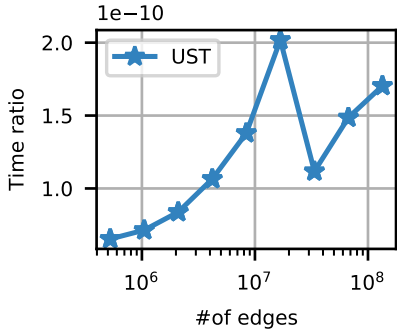
5.7.5 Scalability to Large Networks

Results on Synthetic Networks. The log-log plot in Figure 5.5a show the average running time of UST 1×24 cores on random hyperbolic networks [191].[32] For each network size, we take the arithmetic mean of the running times measured on five different randomly generated networks. Our algorithm requires 184 minutes for the largest inputs – with up to 83.9 million edges. Interestingly, Figure 5.5b shows that the algorithm scales slightly better than our theoretical bound predicts. In Figure 5.6 we present additional

[32] The random hyperbolic generator generates networks with a heavy-tailed degree distribution. We set the average degree to 20 and the exponent of the power-law distribution to 3.



(a) Running time of UST w.r.t. #of edges.



(b) Ratio between the running time of UST and its theoretical running time (see Theorem 5.3.2.

Figure 5.6: Scalability of UST on R-MAT graphs ($\varepsilon = 0.3$, 1×24 cores).

Table 5.2: Running time (s) of UST on large real-world networks (16×24 cores).

Network	n	m	Time (s) $\varepsilon = 0.3$	Time (s) $\varepsilon = 0.9$
petster-carnivore	601,213	15,661,775	16.8	4.8
soc-pokec-relationships	1,632,803	22,301,964	55.5	9.5
soc-LiveJournal1	4,843,953	42,845,684	277.0	75.5
livejournal-links	5,189,808	48,687,945	458.4	80.6
orkut-links	3,072,441	117,184,899	71.8	19.9
wikipedia_link_en	13,591,759	334,590,793	429.9	88.3

results on R-MAT graphs [65]. For this experiment, we use the Graph 500 [223] parameter setting – i.e., edge factor 16, $a = 0.57, b = 0.19, c = 0.19$, and $d = 0.05$. On these instances, the algorithm requires only 18 minutes on inputs with up to 134.2 million edges. In particular, since these graphs have a nearly-constant diameter, our algorithm is faster than on random hyperbolic graphs. Qualitatively, it exhibits a similar scalability. The comparison to the theoretical bound is, however, less conclusive.

Results on Large Real-World Networks. In Table 5.2 we report the performance of UST in a distributed setting (16×24 cores) on large *real-world* networks. With $\varepsilon = 0.3$ and $\varepsilon = 0.9$, UST always runs in less than 8 minutes and 1.5 minutes, respectively.

5.7.6 Additional Experimental Results

Relative Error Quality Measures. Because our algorithm computes an absolute $\pm\varepsilon$-approximation of $\mathrm{diag}(\mathbf{L}^{\dagger})$ with high probability, it is expected to yield better results in terms of maximum absolute error and ranking than numerical approaches with a relative error guarantee. Indeed, as we show in the following, the quality assessment changes if we consider quality measures on a relative error such as:

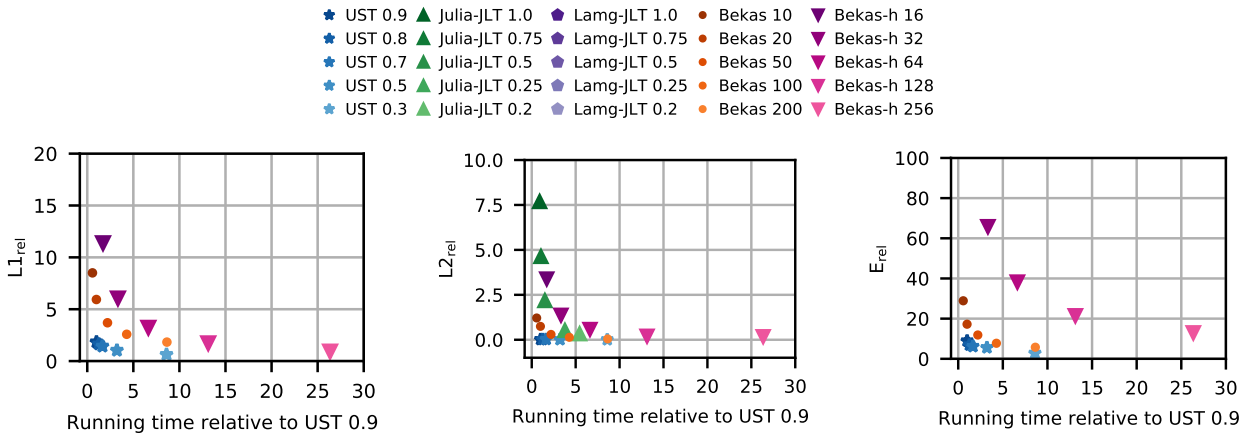


Figure 5.7: $\mathrm{L1}_{\mathrm{rel}}$, $\mathrm{L2}_{\mathrm{rel}}$, and $\mathrm{E}_{\mathrm{rel}}$ w.r.t. the running time of our algorithm with $\varepsilon = 0.9$. All data points are aggregated using the geometric mean over the instances of Table 5.1a.

$$\mathrm{L1}_{\mathrm{rel}} := \frac{\left\| \mathrm{diag}(\mathbf{L}^{\dagger}) - \mathrm{diag}\left(\widetilde{\mathbf{L}}^{\dagger}\right) \right\|_1}{\|\mathrm{diag}(\mathbf{L}^{\dagger})\|_1},$$

$$\mathrm{L2}_{\mathrm{rel}} := \frac{\left\| \mathrm{diag}(\mathbf{L}^{\dagger}) - \mathrm{diag}\left(\widetilde{\mathbf{L}}^{\dagger}\right) \right\|_2}{\|\mathrm{diag}(\mathbf{L}^{\dagger})\|_2},$$

$$\mathrm{E}_{\mathrm{rel}} := \mathrm{gmean}_v \frac{\left| \mathbf{L}^{\dagger}[v,v] - \widetilde{\mathbf{L}}^{\dagger}[v,v] \right|}{\mathbf{L}^{\dagger}[v,v]}.$$

Quality in terms of Relative Error. Figure 5.7 shows that, when assessing the error in terms of $\mathrm{L1}_{\mathrm{rel}}$, $\mathrm{L2}_{\mathrm{rel}}$, or $\mathrm{E}_{\mathrm{rel}}$, for the same running time, UST yields results that are still better in terms of quality than the competitors, but not by such a wide margin. This can be explained by the fact that the numerical solvers used by our competitors often employ measures analogous to $\mathrm{L1}_{\mathrm{rel}}$ and $\mathrm{L2}_{\mathrm{rel}}$ in their stopping conditions.

5.8 Experiments – Forest Closeness

We now study the empirical performance of our algorithm for forest closeness (Section 5.5) on real-world graphs.

Settings. Unless stated otherwise, all algorithms are implemented in C++, using the NetworKit [273] graph APIs. All experiments are conducted on Linux machines equipped with an Intel Xeon Gold 6126 with 2×12 cores and 192 GiB of RAM each. Unless stated otherwise, all experiments run on a single core. We manage our experiments with the SimexPal software [14] to ensure reproducibility. For evaluation, we use a large collection of undirected graphs of different sizes, coming from a diverse set of domains. All graphs have been downloaded from the public repositories KONECT [173], OpenStreetMap [83], and NetworkRepository [254]. We denote our proposed algorithm for forest closeness by UST and, as done in [153], we set $\alpha = 1$.

Table 5.3: Running time and KT ranking scores of UST and JLT-based algorithms. In the JLT column we report, for each instance, the competitor with highest KT score. For equal KT scores – up to the second decimal place – we choose the fastest competitor.

(a) Complex networks

Graph	n	m	Time (s)		KT	
			UST	JLT	UST	JLT
loc-brightkite_edges	58K	214K	**46.4**	186.4	**0.98**	0.95
douban	154K	327K	**80.8**	370.9	**0.71**	0.61
soc-Epinions1	75K	405K	**55.5**	339.6	**0.95**	0.90
slashdot-zoo	79K	467K	**59.9**	412.3	**0.95**	0.92
petster-cat-household	105K	494K	**61.8**	372.1	**0.98**	0.92
wikipedia_link_fy	65K	921K	**58.2**	602.9	**0.98**	0.96
loc-gowalla_edges	196K	950K	**230.9**	1,215.5	**0.99**	0.97
wikipedia_link_an	56K	1.1M	**50.7**	562.6	**0.96**	0.93
wikipedia_link_ga	55K	1.2M	**44.8**	578.6	**0.98**	0.97
petster-dog-household	260K	2.1M	**359.6**	2,472.1	**0.98**	0.96
livemocha	104K	2.2M	**107.4**	1,429.3	**0.98**	0.97

(b) Road networks

Graph	n	m	Time (s)		KT	
			UST	JLT	UST	JLT
mauritania	102K	150K	**98.1**	217.6	**0.88**	0.77
turkmenistan	125K	165K	**118.5**	273.6	**0.92**	0.85
cyprus	151K	189K	**149.4**	315.8	**0.89**	0.80
canary-islands	169K	208K	**185.5**	382.0	**0.92**	0.84
albania	196K	223K	**192.6**	430.2	**0.90**	0.82
benin	177K	234K	**188.1**	406.8	**0.92**	0.83
georgia	262K	319K	**322.1**	605.3	**0.91**	0.83
latvia	275K	323K	**355.2**	665.4	**0.91**	0.83
somalia	291K	409K	**420.1**	747.5	**0.92**	0.84
ethiopia	443K	607K	**825.9**	1,209.7	**0.91**	0.83
tunisia	568K	766K	**1,200.1**	1,629.0	**0.89**	0.79

Competitors. For the forest closeness of individual vertices, the main competitor is the JLT-based algorithm by Jin et al. [153], which uses the Laplacian solver from Ref. [175]. We compare against two implementations of this algorithm: one provided by the authors written in Julia v1.0.2 and our own implementation based on Eigen's [135] CG algorithm. We denote them by JLT-Julia and JLT-CPP, respectively. Like in Ref. [153], we compute the number of linear systems for JLT-Julia and JLT-CPP as $\left\lceil \frac{\log n}{\varepsilon^2} \right\rceil$ – which gives an $(\varepsilon \cdot c)$-approximation for a fixed constant $c > 1$.

5.8.1 Performance of UST

We measure the performance of UST compared to the state of the art. Each method is executed with multiple settings of its respective quality parameter.

Accuracy and Running Time. We report the maximum absolute error of the estimated diagonal values (i.e., $\max_v |\mathbf{\Omega}[v,v] - \widetilde{\mathbf{\Omega}}[v,v]|$) over all vertices and instances from Table 5.3.[33] As ground truth, we take $\mathbf{\Omega}[v,v]$ values that are computed using Eigen's [135] CG solver with a tolerance of 10^{-9}; exact inversion of $(\mathbf{L}+\mathbf{I})$ would be infeasible for many of the input graphs. A preliminary comparison against the values if $\mathbf{\Omega}[v,v]$ computed with the NumPy `pinv` function demonstrated the CG provides a sufficiently accurate ground truth.

Figure 5.8 shows that UST achieves the best results in terms of quality and running time for both complex and road networks. More precisely, for complex networks and $\varepsilon = 0.4$, UST yields a maximum absolute error of 0.14, which is less than the most accurate result of both competitors (0.15 achieved by JLT-Julia with $\varepsilon = 0.1$), while being $397.5\times$ faster. Also, the running time of UST does not increase substantially for lower values of ε, and its quality does not deteriorate quickly for higher values of ε. A similar pattern is observed for road networks as well. Detailed running time values are reported in Table D.2, Appendix D.2.

[33] Note that, as for electrical closeness, the top vertices in the forest closeness ranking are the ones with the *lowest* $\mathbf{\Omega}[v,v]$ (see Eq. (5.8)); hence, we also evaluate the ranking accuracy in a following experiment.

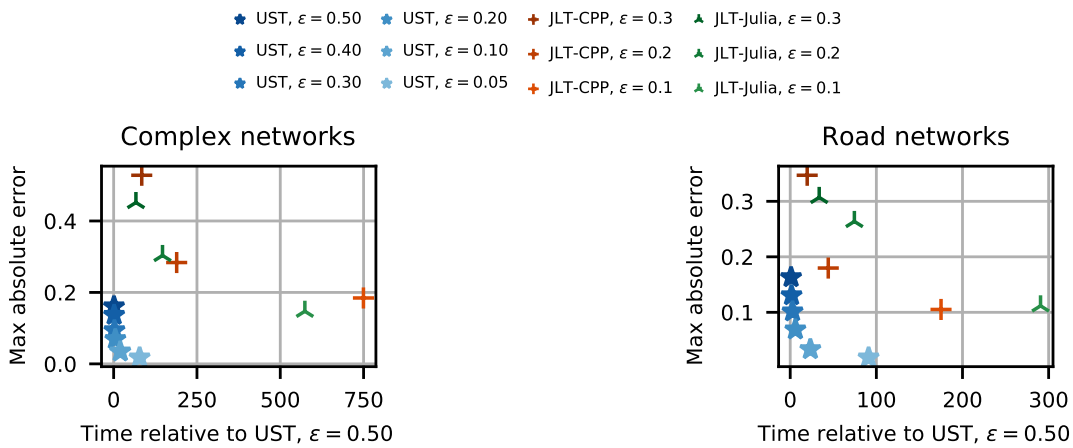


Figure 5.8: $\max_v \left| \mathbf{\Omega}[v,v] - \widetilde{\mathbf{\Omega}}[v,v] \right|$ over the instances of Table 5.3.

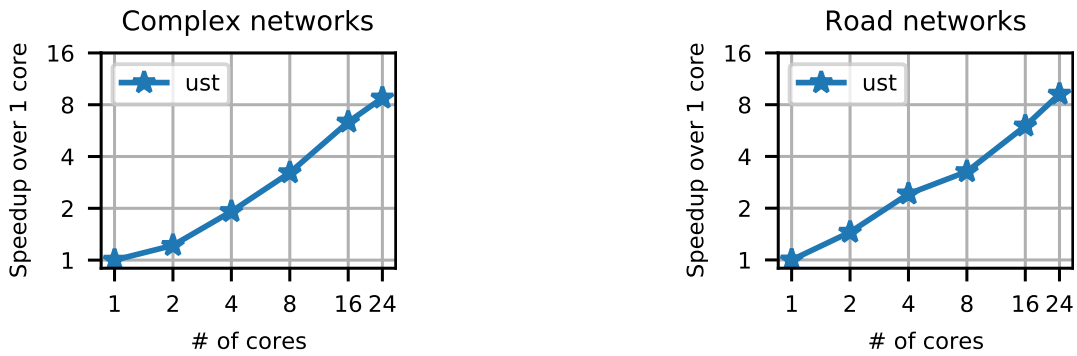


Figure 5.9: Geometric mean of the speedup of UST with $\varepsilon = 0.05$ on multiple cores over a sequential run (shared memory). Data points are aggregated over the instances of Table 5.3.

Vertex Ranking. Moreover, we measure the accuracy in terms of vertex rankings, which is often more relevant than individual scores [227, 229]. In Table 5.3, we report the Kendall's rank correlation coefficient (KT) of the vertex ranking w.r.t. the ground truth along with running times for complex networks (Table 5.4a) and for road networks (Table 5.4b). For each instance, we pick the best run, i.e., the "UST" and "JLT" columns display the run with highest respective KT value. If the values are the same up to the second decimal place, we pick the fastest one. UST has consistently the best vertex ranking scores; at the same time, it is faster than the competitors. In particular, UST is on average $7.6\times$ faster than the JLT-based approaches on complex networks and $1.9\times$ faster on road networks.

Parallel Scalability. As described in Section 5.6.2, UST is well-suited for parallel implementations since each UST can be sampled independently in parallel. Hence, we provide parallel implementations of UST based on OpenMP (for multi-core parallelism) and MPI (to scale to multiple compute nodes) for forest closeness as well.

In Figure 5.9, we report the parallel scalability of UST on multiple cores. Unsurprisingly, analogously to the results achieved in Section 5.7.4, UST exhibits a moderate scalability w.r.t. the number of cores. In particular, our OpenMP implementation on 24 cores exhibits a speedup of $8.7\times$ on complex networks and $9.2\times$ on road networks. As with electrical closeness, we hypothesize that this is mainly due to memory

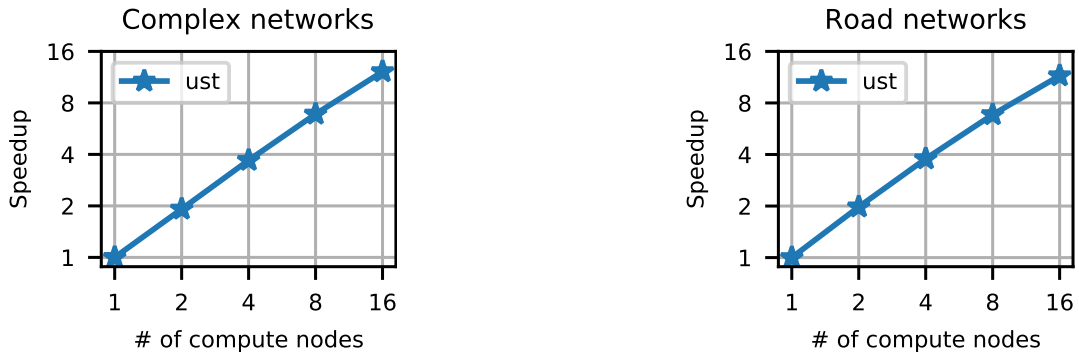


Figure 5.10: Geometric mean of the speedup of UST with $\varepsilon = 0.1$ on multiple compute nodes over a single compute node (1×24 cores). Data points are aggregated over the instances of Table 5.5.

Table 5.5: Large networks used for scalability experiments in distributed memory and running time of UST for forest closeness on 16×24 cores.

(a) Complex networks

Graph	n	m	Time (s) $\varepsilon = 0.1$	Time (s) $\varepsilon = 0.3$
soc-LiveJournal1	4,846,609	42,851,237	348.9	118.5
wikipedia_link_fr	3,333,397	100,461,905	205.4	90.7
orkut-links	3,072,441	117,184,899	293.5	92.2
dimacs10-uk-2002	18,483,186	261,787,258	1,101.3	365.8
wikipedia_link_en	13,593,032	334,591,525	919.3	295.4

(b) Complex networks

Graph	n	m	Time (s) $\varepsilon = 0.1$	Time (s) $\varepsilon = 0.3$
slovakia	543,733	638,114	28.1	9.9
netherlands	1,437,177	1,737,377	82.9	31.1
greece	1,466,727	1,873,857	74.5	29.8
spain	4,557,386	5,905,365	273.0	86.2
great-britain	7,108,301	8,358,289	419.0	136.6
dach	20,207,259	25,398,909	1,430.1	473.7
africa	23,975,266	31,044,959	1,493.4	499.3

latencies: while sampling a UST, our algorithm performs several random accesses to the graph data structure (i.e., an adjacency array [273]), which are prone to cache misses. The results for MPI are depicted in Figure 5.10. In this setting, UST obtains a speedup of $12.2\times$ on complex and $11.5\times$ on road networks on up to 16 compute nodes – for this experiment we set $\varepsilon = 0.1$ and we use the instances in Table 5.5. More sophisticated load balancing techniques are likely to increase the speedup in the MPI setting, they are left for future work. Still, the MPI-based algorithm can rank complex networks with up to 334M edges in less than 20 minutes. Road networks with 31M edges take less than 25 minutes.

5.9 Conclusions

This chapter addressed the problem of approximating electrical centrality measures, namely, electrical closeness, normalized random-walk betweenness, Kirchhoff-related indices, and forest closeness. The core contribution is a new efficient parallel algorithm for approximating $\mathrm{diag}(\mathbf{L}^\dagger)$ of Laplacian matrices $\mathbf{L}$ corresponding to small-world weighted undirected networks, which is enough to compute the aforementioned measures. Compared to the main competitors, our algorithm is about one order of magnitude faster, it yields results with superior quality in terms of absolute error and ranking of $\mathrm{diag}(\mathbf{L}^\dagger)$, and it requires less memory.

Furthermore, we extended our algorithm to forest closeness, which can now be approximated faster and more accurately than previously possible. Because the augmented graph has constant diameter, in the forest

closeness case we can target any graph – not only small-world graphs – without degrading the performance of the algorithm.

The gap between the theoretical bounds and the much better empirical error yielded by our algorithm in approximating $\mathrm{diag}(\mathbf{L}^{\dagger})$ suggests that tighter bounds on the number of samples are a promising direction for future research. Another potentially interesting research direction is to devise a new UST-based adaptive sampling strategy, as it could reduce the number of samples and thus the running time of our UST algorithm and it would benefit from our epoch-based framework for parallel ADS described in Chapter 4. Other conceivable ideas for future work are an improvement of the running time for high-diameter graphs, both in theory and in practice, and extensions of our strategies to directed graphs.

Part III

Algorithms for Group Centrality Measures

Introduction

As we saw in Part II, finding highly central vertices in a graph is a fundamental problem in network analysis (see also Ref. [227]). To this end, several centrality measures have been introduced over the past decades (see Ref. [53] and Section 2.4). The problem of identifying the top-k most central vertices in a graph has been widely studied [37, 180, 229, 231]. However, many network analysis applications do not require the k most *individually* central vertices, but rather a *group* of k vertices that is central *as a whole* [121, 162, 185, 285, 291].

Everett and Borgatti [104] were the first to extend the centrality concept to groups of vertices and defined group-degree, group-closeness, group-betweenness, and flow betweenness. Group centrality maximization problems are often $\mathcal{NP}$-hard: group-degree maximization can be reduced from vertex cover [13], for group-closeness see Ref. [76] and for group-harmonic see Ref. [12]. Consequently, exact methods for these problems such as ILP solvers [38] take too long on graphs with non-trivial size – a few thousands of vertices/edges or more. Thus, group centrality maximization is generally approached via approximation [183, 200] or heuristics [38, 76]. Yet, existing algorithms for group centrality maximization fail to scale to large real-world networks. In addition, early attempts to attribute a constant-factor approximation to a popular greedy algorithm for group-closeness maximization were flawed (see Section 7.4.1), leaving the open questions of whether this problem can be approximated and, if so, how well. Another weakness of existing group centrality measures is that, without proper adjustments, they are not designed to handle disconnected graphs.

In the following, we address the lack of scalability of group centrality maximization algorithms from two directions: (i) in the context of group-closeness centrality, we introduce new fast local search heuristics to compute highly central groups of vertices in large graphs (Chapter 6) and (ii) we introduce a novel group centrality measure called GED-Walk as well as efficient parallel algorithms compute it and to approximate the group of vertices with highest GED-Walk centrality (Chapter 8). Concerning the group-closeness approximation issue, building on top of the results presented in Ref. [12], we provide an efficient implementation of the first approximation algorithm for this problem (Chapter 7). Finally, we address the lack of electrical group centrality measures for disconnected graphs by extending forest closeness to disconnected graphs (Chapter 8); as its single-vertex counterpart, group forest closeness handles disconnected graphs out of the box.

Contribution and Outline. In Chapter 6, we introduce new fast local search heuristics for group-closeness to compute highly central groups of vertices. Our heuristics start from an initial a group of vertices S and perform exchanges between vertices within S and the rest of the graph until they reach a local optimum. Compared to the state-of-the-art greedy algorithm [38], experimental results show that, on unweighted graphs, our strategy is two orders of magnitude faster and achieves 99.4% of the solution quality;

on weighted graphs, it yields solutions of 12.4% *higher* quality while also being $127.8\times$ faster. Furthermore, our algorithms handle graphs with hundreds of million of edges in only around ten minutes, while the greedy algorithm requires more than ten *hours*.

Heuristics, however, do not provide any approximation guarantees, leaving the question of approximability for group-closeness maximization unsettled. In Chapter 7, we address this issue. By exploiting the theoretical results from Ref. [12], we build the first approximation algorithm for group-closeness maximization by adapting to group-closeness the local search algorithm for k-Median by Arya et al. [20]. In addition, we also address *group-harmonic* maximization – to the best of our knowledge, we are the first to study this optimization problem. We implement an efficient greedy algorithm as well as a local search approximation algorithm to maximize group-harmonic. In our experimental study we show that, compared to a greedy strategy, our local search approximation algorithm yields solutions with superior quality at the cost of additional running time. In particular, local search is one to three orders of magnitude slower than greedy, which is expected due to the approximation guarantees. Indeed, local search approximation algorithms often cut greedy's (empirical) gap to optimality by half or more.

Finally, in Chapter 8, we introduce two new group centrality measures: GED-Walk, a novel group centrality measure inspired by Katz centrality, and group forest closeness, that is, forest closeness (see Chapter 5) extended to sets of vertices. Similarly to Katz centrality, GED-Walk takes into account walks of any length and gives to shorter walks greater importance than to longer walks. Since it is not based on shortest paths, GED-Walk can be optimized significantly faster (for groups with moderate size) compared to shortest-path based measures such as group-closeness, group-harmonic or group-betweenness. We describe efficient parallel algorithms to compute the GED-Walk score of a given group and to efficiently approximate the group of vertices with highest GED-Walk centrality. Experimental results show that GED-Walk improves the precision of popular graph mining applications such as semi-supervised vertex classification and graph classification [278]. Also, our algorithm for maximizing GED-Walk (in approximation) is two orders of magnitude faster than state-of-the-art algorithms for group-betweenness maximization [200] and, for group sizes up to 100 vertices, one to two orders of magnitude faster than group-harmonic and group-closeness maximization [38].

Group forest closeness, in turn, is defined as the reciprocal of group forest farness, i.e., the sum of forest distances from a group S to the other vertices. Maximizing this measure turns out to be $\mathcal{NP}$-hard, so we adapt the greedy approximate algorithm from Li et al. [183] for group electrical closeness to group forest closeness. Semi-supervised vertex classification results on disconnected graphs indicate that, in comparison to existing measures, group forest closeness improves the precision to a much greater extent.

6 Local Search for Group-Closeness Maximization on Big Graphs

6.1 Introduction

Closeness centrality is one of the oldest and most widely-used vertex centrality measures (see Ref. [33] and Chapter 3). It is defined as the reciprocal of the average shortest-path distance from a vertex to all the others – see Eq. (2.5).

Group-closeness can be interpreted as a special case (on graphs) of the well-known k-Median problem for facility location. Example of applications include: (i) retailers that want to advertise their products via social media; promoters could be selected as the group of k members with highest centrality in order to maximize the influence over the other members [293]; (ii) in P2P networks, shared resources could be placed on k peers so that they are easily accessible by others [121]; (iii) in citation networks, group centrality measures can be employed as alternative indicators for the influence of journals or papers within their research field [182].

The problem of finding the group of k vertices with highest group-closeness (group closeness maximization) is shown to be $\mathcal{NP}$-hard [76]. While exact algorithms to find a group with maximal closeness are known – e.g., algorithms based on Integer Linear Programming (ILP) [38] – they do not scale to graphs with more than a few thousand edges. Hence, in practice, groups with high closeness centrality are computed through heuristics [38, 76].

Related Work. For group-betweenness maximization, sampling-based algorithms have been proposed [200, 288]. An extensive analysis of algorithms for group-betweenness estimation is provided by Chehreghani et al. [74], who also introduce a new algorithm based on an alternative definition of distance between a vertex and a set of vertices.

Concerning group-closeness maximization, in Ref. [290] the definition of group-closeness differs from the original one and only serves as an estimate of the original – we use the latter, defined in Eq. (2.13), which is widely accepted. Therefore, their $\mathcal{NP}$-hardness and submodularity proofs do not necessarily carry over to the original definition. In fact, the standard group-closeness is not submodular (see Lemma 2.5.2). Hence, submodular optimization results in [282] do not apply to this case directly. Chen et al. [76] argued that group-closeness maximization is $\mathcal{NP}$-hard by relating it to the $\mathcal{NP}$-hard k-means problem. Their proposed greedy algorithm was later improved by Bergamini et al. [38], who made algorithm more memory-efficient and exploited the supermodularity of the reciprocal of group-closeness for search pruning. Exploiting the supermodularity of the reciprocal also works when the distance function is replaced by the resistance distance (see Section 2.3.2). This leads to the so-called group *current-flow* closeness, for which Li et al. [183] proposed approximation algorithms based on greedy strategies and random projections.

Motivation. Still, even for group-closeness with the usual graph distance, the greedy algorithm can be time-consuming on large instances. Indeed, pruning is most effective when the group is already large. When performing the first addition, however, the greedy algorithm has to perform one (pruned) SSSP *for each vertex in the graph* to compute its marginal contribution, and this phase scales superlinearly in the size of the graph. As a result, real-world graphs with hundreds of millions of edges still require several hours to complete. This motivates us to develop new local search heuristics that quickly find highly central groups of vertices on large-scale real-world graphs without sacrificing too much solution quality.

Contribution. In this chapter, we develop new heuristic algorithms for group-closeness maximization. Specifically, we present two novel local search heuristics that start from an initial group of vertices $S \subset V$ and perform exchanges of vertices from S and $V \setminus S$ until a local optima is reached. The first algorithm, *Local-Swaps*, requires little time per iteration but only exchanges vertices locally. The second algorithm, *Grow-Shrink*, performs also non-local vertex exchanges, but iterations are more computationally expensive.

Experiments on unweighted graphs show that *extended Grow-Shrink*, a variation of *Grow-Shrink*, computes groups with closeness scores greater than 99.4% of the score of a greedy solution while being $127.8\times$ faster (results for $k = 10$). We see *extended Grow-Shrink* as the best trade-off between speed and solution quality. When quality is not a primary concern, our other algorithms accelerate the computation by sacrificing solution quality. For example, for groups with size 10 the non-extended variant of *Grow-Shrink* yields solutions whose quality is 91.1% compared to the state of the art while being $700.2\times$ faster. The speedup varies between $927.9\times$ and $43.0\times$ for groups with sizes 5 and 100, respectively. On weighted graphs, our algorithms improve both the quality and the running time performance of the state of the art: for example, for $k = 10$, they return solutions of 12.4% higher quality with a $793.6\times$ speedup. Different trade-offs between quality and running time are possible, we discuss them in Section 6.7.

Bibliographic Notes. The contributions presented in this chapter were published in the Proceedings of the *IEEE International Conference on Big Data, 2019*. My contributions involve the development of the initial algorithm that led to the ones presented in the paper. The development of the presented algorithms is a collaborative effort with Alexander van der Grinten. Further, I implemented all the presented algorithms (acceleration techniques with SIMD vector operations were implemented together with Alexander van der Grinten) and carried out the experiments. The rest is joint work with Alexander van der Grinten and Henning Meyerhenke. Proofs to which I did not contribute are omitted and can be found in the original paper [15].

6.2 Preliminaries

Let $S \subset V$ be a group of vertices. As reported in Section 2.5, the group-closeness centrality of S is defined as $g_c(S) := \frac{n}{\sum_{u \in V \setminus S} d(S,u)}$. In the literature, a similar objective has been addressed as well, namely, the farness of a set S, defined as $g_f(S) := \frac{1}{n} \cdot \sum_{u \in V \setminus S} d(S,u)$. Note that the farness of a group is defined as the reciprocal of its closeness.

Problem Addressed. In this chapter, we study the problem of finding groups that maximize group-closeness and subject to a cardinality constraint, i.e., for a given integer $1 \leq k < n$, our objective is to find a group with size k of large group-closeness. Formally:

Group-Closeness Maximization	
Input:	Graph $G = (V, E, w)$, integer $1 \leq k < n$.
Find:	Set $S^\star \subset V$ with $\lvert S\rvert = k$ s.t. $g_c(S^\star)$ is maximum.

We assume G to be undirected, connected, and with positive edge weights. We consider the problem of improving the group-closeness of a given set S of vertices with local search. More precisely, we consider *exchanges* of vertices from S and $V \setminus S$. Let u be a vertex in S and $v \in V \setminus S$. To simplify the presentation, we use the notation $S_{-u}^{+v} := (S \setminus \{u\}) \cup \{v\}$ to denote the set resulting by the exchange of u with v. We also use the notation $S_{-u} := S \setminus \{u\}$ and $S^{+v} := S \cup \{v\}$ to denote vertex removals and insertions, respectively.

As our algorithms can only perform vertex exchanges, before they can start they require the construction of an initial solution S. Since a superlinear initialization step would compromise our algorithms' running times, for all our local search algorithms we choose the initial set S uniformly at random. For large graphs, this initialization can be expected to cover the graph reasonably well. Exploratory experiments revealed that our algorithms do not benefit from other obvious initialization techniques – such as selecting the k vertices with highest degree.

6.3 Estimating the Quality of Vertex Exchanges

The greedy ascent algorithm starts with an empty set S and iteratively adds vertices $v \in V \setminus S$ to S that maximize $g_f(S) - g_f(S^{+v})$ [38]. Depending on the input graph and the value of k, however, the greedy algorithm might need to evaluate the difference in g_f for a substantial number of vertices, and this is rather expensive for large real-world graphs.

The algorithms we present in this section aim to improve upon the running time of the greedy algorithm. We achieve this by first considering exchanges with only *local* vertices, i.e., vertices that already "near" S. Clearly, selecting only local vertices would decrease the quality of a greedy solution – as the greedy algorithm does not have the ability to correct suboptimal choices. However, this is not necessarily true for our algorithms based on vertex exchanges. Then, we generalize this strategy by extending vertex exchanges to non-local vertices, and allowing exchanges of multiple vertices per time (Section 6.6).

To make our notion of locality more concrete, let $B_S \subseteq G$ be the DAG constructed by running a SSSP from the vertices in S. Here, all the vertices in S are considered as sources of the SSSP, i.e., they are at distance 0. Further, define

$$\Delta^-(v) := g_f(S) - g_f(S^{+v}) = \sum_{x \in V \setminus S} d(S, x) - d(S^{+v}, x).$$

To compute the greedy solution, it seems to be necessary to compute Δ^- exactly for a substantial number of vertices. Pruning techniques can avoid some of the computational costs [38], but many evaluations of Δ^- still have to be performed, which seems to be impractical for large graphs. However, a lower bound for Δ^-

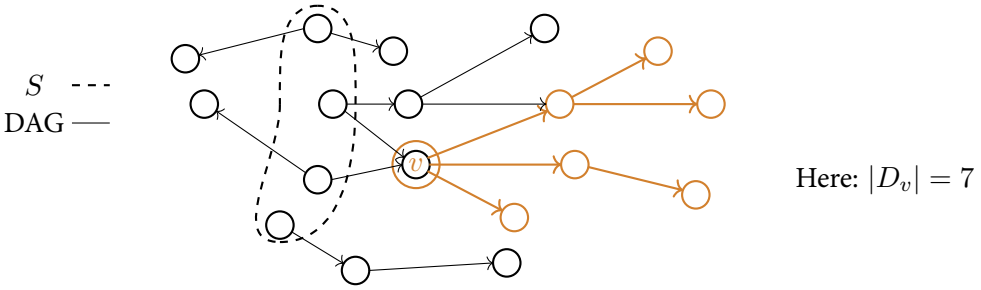


Figure 6.1: Example of shortest-path DAG B_S; orange vertices are in D_v.

can be computed from the shortest-path DAG B_S. To this end, let D_v be the set of vertices reachable from v in B_S.

Lemma 6.3.1. [15] It holds that:

$$\Delta^-(v) \geq |D_v| \cdot d(S, v)$$

In the unweighted case, equality holds if v is a neighbor of S.

An illustration of Lemma 6.3.1 is shown in Figure 6.1. This bound will be used in the two algorithms presented in Sections 6.4 and 6.5. Instead of picking vertices that maximize Δ^-, those algorithms pick vertices that maximize the right-hand side of Lemma 6.3.1, i.e., $D_v \cdot d(S, v)$. The bound is local in the sense that it is more accurate for vertices near S: in particular, let $N(S)$ be set of vertices that have a neighbor in S; the reachability sets of vertices in $N(S) \setminus S$ are larger in G than those in B_S, as B_S does not contain back edges. Unfortunately, computing the cardinality of D_v for all v seems to be prohibitively expensive: indeed, the fastest know algorithm to compute the size of the transitive closure of a DAG relies on iterated (Boolean) matrix multiplication – hence, the best known exact algorithm has a complexity of $\mathcal{O}(n^{2.373})$ [9]. However, we can use Cohen's randomized algorithm [78] to approximate the sizes of D_v for all v *at the same time*. In multiple iterations, this algorithm samples a random number for each vertex in G, accumulates in each vertex v the minimal random number of any vertex reachable from v, and estimates $|D_v|$ based on this information.

We remark that Cohen's algorithm yields an approximation, not a lower bound for the right-hand side of Lemma 6.3.1, therefore in our algorithms the inequality of the Lemma can be violated; in particular, it can happen that our algorithms pick a vertex v such that $\Delta^-(v) < 0$. In this case, instead of decreasing the closeness centrality of the current group, our algorithms terminate. Nevertheless, our experiments demonstrate that, on real-world instances, a coarse approximation of the reachability set size is enough for Lemma 6.3.1 to identify useful candidates for vertex exchanges (see Section 6.7).

6.4 The *Local-Swaps* Algorithm

Let us first focus on unweighted graphs. A straightforward idea to construct a fast local search algorithm is to allow *swaps* betweenness vertices in S and their neighbors in $V \setminus S$. This procedure can be repeated until no swap can decrease $g_f(S)$. Let u be a vertex in the group and let $v \in N(S) \setminus S$ be one of its neighbors outside the group. To determine whether swapping u and v (i.e., replacing S by S_{-u}^{+v}) is beneficial, we have to check whether $g_f(S) - g_f(S_{-u}^{+v}) > 0$, i.e., whether the farness decreases after the swap. The challenges

Algorithm 12 Overview of the *Local-Swaps* Algorithm

1: **repeat**
2: approximate D_v for all $v \in V \setminus S$ with [78]
3: $\langle u, v \rangle \leftarrow \text{argmax}_{u,\in S,\ v \in N(u) \setminus S} |D_v| - |\Lambda_u|$
4: $S \leftarrow S^{+v}_{-u}$
5: run pruned BFS from v ▷ to recompute $g_f(S)$
6: **until** previous iteration did not decrease $g_f(S)$

here are (i) to find a pair u, v that satisfies such inequality without checking all pairs u, v exhaustively and (ii) to compute the difference $g_f(S) - g_f(S^{+v}_{-u})$ quickly. Note that a crucial property that allows us to construct an efficient algorithm is that, after a swap, the distance from S to any other vertex can only change by ± 1. Hence, to compute $g_f(S) - g_f(S^{+v}_{-u})$, it suffices to count the number of vertices where the distance changes by -1 and the number of vertices where it changes by $+1$. To this end, our algorithm requires a few auxiliary data structures; in particular, we store the following:

- the distance $d(S, x)$ from S to all vertices $x \in V \setminus S$;
- a set $\lambda_x := \{s \in S : d(S, x) = d(s, x)\}$ for each vertex $x \in V \setminus S$ that contains all vertices in S that realize the shortest distance from S to x;
- the value $|\Lambda_s|$ for each $s \in S$, where $\Lambda_s := \{x \in V \setminus S : \lambda_x = \{s\}\}$ is the set of vertices for which the shortest distance is realized *exclusively* by s.

Note that the sets λ_x consume $\mathcal{O}(kn)$ memory in total. However, since $k \ll n$, we can afford this even for large real-world graphs.[34]

All of those auxiliary data structures can be maintained dynamically during the entire algorithm with little additional overhead. More precisely, after a u-v swap is done, v is added to all λ_x satisfying $d(v, x) = d(S, x)$; for $x \in V \setminus S$ such that $d(v, x) < d(S, x)$, the set λ_x is replaced by $\{v\}$. Vertex u can be removed from all λ_x by a linear-time scan through all $x \in V \setminus S$.

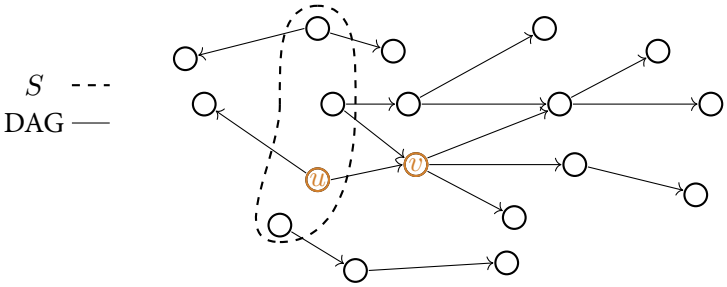
Algorithm 12 states a high-level overview of our *Local-Swaps* algorithm. In the following, we discuss how we pick a "good" swap (Line 3 of the pseudocode) and how to update the data structures after a swap (Line 5). The running time of the algorithm is dominated by the initialization of λ_x. Thus, it runs in $\mathcal{O}(kn + m)$ time per swap.

6.4.1 Choosing a Good Swap

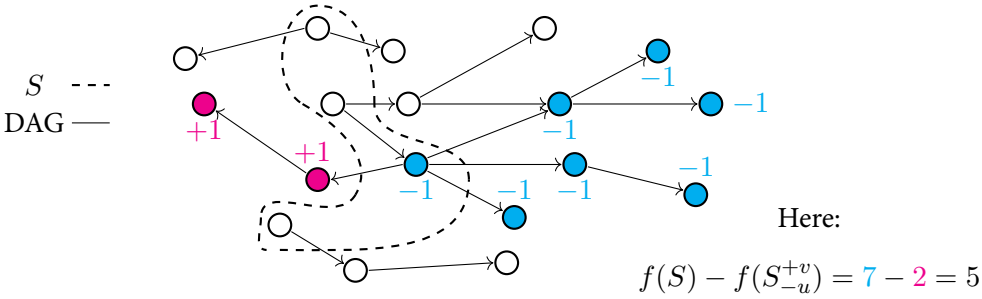
Because it would be too expensive to compute the exact difference in g_f for each possible swap, we find the pair of vertices $\langle u, v \rangle$ such that:

$$\begin{aligned} \langle u, v \rangle &= \operatorname*{argmax}_{\substack{u \in S, \\ v \in N(S) \setminus S}} |D_v| \cdot d(S, v) - |\Lambda_u| \\ &= \operatorname*{argmax}_{\substack{u \in S, \\ v \in N(S) \setminus S}} |D_v| - |\Lambda_u|. \end{aligned}$$

[34] In our implementation, we store each λ_x in only k bits.



(a) State of S and DAG before swapping u with v.



(b) State of S and DAG after the u-v swap. Vertices in magenta are the ones in $H^+_{u,v}$ (their distance from S increased by 1) whereas $H^-_{u,v}$ vertices are in blue (their distance from S decreased by 1). Overall, the farness of S decreased by 5.

Figure 6.2: Example of computation of the difference in farness after a u-v swap with the *Local-Swaps* algorithm. The states of S and of the DAG before and after the u-v swap are shown in Figures 6.2a and 6.2b, respectively.

Note that this value is a lower bound for the decrease of $g_f(S)$ after swapping u and v: In particular, Lemma 6.3.1 implies that $|D_v|$ is a lower bound for the decrease in farness when adding v to S. Additionally, $|\Lambda_u|$ is an upper bound for the increase in farness when removing u from S – and thus also for the increase in farness when removing u from S^{+v}.[35] Hence, we can expect this strategy to yield pairs of vertices that lead to a decrease of g_f. In practice, to maximize $|D_v| - |\Lambda_u|$, for each $v \in V \setminus S$ we first compute the neighbor $u \in N(v) \cap S$ that minimizes $|\Lambda_u|$, which requires $\mathcal{O}(n + m)$ time. Afterwards, we maximize $|D_v| - |\Lambda_u|$ by a linear scan over all $v \in V \setminus S$.

6.4.2 Computing the Difference in Farness

Instead of comparing actual distances, it suffices to define sets of vertices whose distance to S is increased or decreased (by 1) by the swap, namely:

$$H^+_{u,v} := \{x \in V : d(S, x) < d(S^{+v}_{-u}, x)\},$$
$$H^-_{u,v} := \{x \in V : d(S, x) > d(S^{+v}_{-u}, x)\}.$$

As $d(S, x) - d(S^{+v}_{-u}, x) \in \{-1, 0, 1\}$, it holds that:

Lemma 6.4.1 ([15]). $g_f(S) - g_f(S^{+v}_{-u}) = |H^-_{u,v}| - |H^+_{u,v}|$.

An illustration of Lemma 6.4.1 is depicted in Figure 6.2. Computing $H^-_{u,v}$ is straightforward: it suffices to run a BFS from v that counts those vertices x for which $d(v, x) < d(S, x)$. To check this condition, we

[35] Note, however, that this bound is trivial if $|D_v| - |\Lambda_u| \leq 0$.

have to store the values of $d(S, x)$ for all $v \in V$. We remark that it is not necessary to run a full BFS: indeed, we can prune the search at each vertex x such that $d(v, x) \geq d(S, x)$ – i.e., the search continues without visiting x. However, as we will see in the following, we relax the pruning condition and prune the BFS only if $d(v, x) > d(S, x)$; this allows us to update our auxiliary data structures on the fly.

$H^+_{u,v}$ ca be computed from $|\Lambda_u|$ with the help of the auxiliary data structures. We note that $H^+_{u,v} \subseteq \Lambda_u$, as only vertices x whose distance $d(S, x)$ is uniquely realized by u – out of all the vertices in the group – can have their distance from S increased by the swap. Since $H^+_{u,v} \cap H^-_{u,v} = \emptyset$, we can further restrict this inclusion to $H^+_{u,v} \subseteq \Lambda_u \setminus H^-_{u,v}$, but, in general, $\Lambda_u \setminus H^-_{u,v}$ will consist of more vertices than just $H^+_{u,v}$. More precisely, $\Lambda_u \setminus H^-_{u,v}$ can be partitioned into $\Lambda_u \setminus H^-_{u,v} = H^+_{u,v} \cup H^0_{u,v}$, where $H^0_{u,v} := \{x \in \Lambda_u : d(u, x) = d(v, x)\}$ consists of the vertices whose distance is neither increased nor decreased by the swap. By construction, $H^0_{u,v}$ and $H^+_{u,v}$ are disjoint. This proves the following Lemma:

Lemma 6.4.2 ([15]). $|H^+_{u,v}| = |\Lambda_u| - |\Lambda_u \cap H^-_{u,v}| - |H^0_{u,v}|$.

Notice that $H^0_{u,v}$ as well as $\Lambda_u \cap H^-_{u,v}$ are completely visited by our BFS. To determine $|H^0_{u,v}|$, the BFS only has to the vertices x satisfying $d(v, x) = d(S, x)$ and $\lambda_x = \{u\}$. On the other hand, to determine $|\Lambda_u \cap H^-_{u,v}|$, it has to count the vertices x that satisfy $d(v, x) < d(S, x)$ and $\lambda_x = \{u\}$.

6.5 The *Grow-Shrink* Algorithm

The main issue with the *Local-Swaps* algorithm from Section 6.4 is that it can exchange a vertex $u \in S$ exclusively with one of its neighbors $v \in N(u) \setminus S$. Due to this restriction, the algorithm might take many swaps to converge to a local optimum. It also makes it hard to escape a local optimum: indeed, the algorithm terminates if no swap with a neighbor improves the closeness.

Our second algorithm lifts those limitations. It also allows G to be a weighted graph. In particular, it allows vertex exchanges that change the distance from S to the vertices in $V \setminus S$ by arbitrary amounts. Since computing the exact differences $g_f(S) - g_f(S^{+v}_{-u})$ for all possible pairs of u and v seems to be impractical in this setting, we decompose the vertex exchange of u and v into two operations: (i) the addition of v to S and (ii) the removal of u from S^{+v}. In particular, we allow the set S to *grow* to a size of $k + 1$ before we *shrink* the size of S back to k. Thus, the cardinality constraint $|S| = k$ is temporarily violated and, eventually, restored again.

The individual differences $g_f(S) - g_f(S^{+v})$ and $g_f(S^{+v}) - g_f(S^{+v}_{-u})$ (or bounds for those differences) turn out to be efficiently computable for all possible u and v, at least in approximation. We remark, however, that while this technique finds the vertex v that maximizes $g_f(S) - g_f(S^{+v})$ and the vertex u that maximizes $g_f(S^{+v}) - g_f(S^{+v}_{-u})$, u and v are *not* necessarily the pair of vertices maximizing $g_f(S) - g_f(S^{+v}_{-u})$. Nevertheless, our experiments in Section 6.7 demonstrate that the solution quality of this algorithm is superior to the quality of the *Local-Swaps* algorithm.

To perform these computations, our algorithm maintains the following data structures:

- the distance $d_S(x)$ of each vertex $x \in V \setminus S$ to S, and a representative $r_x \in S$ that realizes this distance, i.e., it holds that $d(S, x) = d(r_x, x) = d_S(x)$;
- the distance $d'_S(x)$ from $S \setminus \{r_x\}$ to x and a representative r'_x for this distance (satisfying the analogous equality).

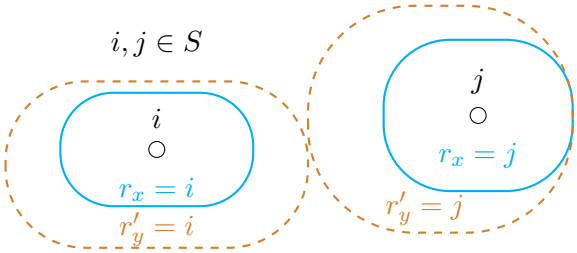


Figure 6.3: Illustration of the data structures maintained by *Grow-Shrink*. Here we show two vertices $i, j \in S$: within the blue lines are those vertices x whose representative r_x is either i (on the left) or j (on the right); the vertices y within the dashed orange lines, in turn, have either i or j as their representative r'_y.

Figure 6.3 illustrates an example of the data structures maintained by the *Grow-Shrink* algorithm. Since the graph is connected, these data structures are well-defined for all groups S of size $|S| \geq 1$. Furthermore, the sum of the differences between $d'_S(x)$ and $d_S(x)$ is exactly the difference in farness when r_x is removed from S. Later, we will use this fact to quickly determine differences in farness due to the removal of vertices from the group.

Notice that it can happen that $d_S(x) = d'_S(x)$; nevertheless, r_x and r'_x are always distinct. Indeed, there can be two different vertices $r_x, r'_x \in S$ that satisfy $d(r_x, x) = d(r'_x, x) = d(S, x)$. We also define:

$$R_u := \{x \in V \setminus S : r_x = u\},$$
$$R'_u := \{x \in V \setminus S : r'_x = u\}.$$

Algorithm 13 Overview of the *Grow-Shrink* Algorithm

1: **repeat**
2: approximate D_v for all $v \in V \setminus S$ with Cohen's algorithm [78]
3: $v \leftarrow S^{+v}$
4: run pruned BFS from v ▷ to recompute $g_f(S)$, d, and d'
5: $u \leftarrow \text{argmin}_{u \in S} \sum_{x \in R_u} d'(x) - d(x)$
6: $S \leftarrow S_{-u}$
7: run Dijkstra-like algorithm ▷ to recompute $g_f(S)$, d, and d'
8: **until** previous iteration did not decrease $g_f(S)$

Algorithm 13 gives a high-level overview of the *Grow-Shrink* algorithm. In the following, we discuss the growing phase (Lines 2–4) and the shrinking phase (Lines 5–7). The time complexity of *Grow-Shrink* is dominated by the Dijkstra-like algorithm in Line 7. Therefore, it runs in $\mathcal{O}(n + m \log n)$ time per swap – if using an appropriate priority queue. The space complexity is $\mathcal{O}(n + m)$.

6.5.1 Vertex Additions

When adding a vertex v to S, we want to select v such that $g_f(S^{+v})$ is minimized. Note that minimizing $g_f(S^{+v})$ is equivalent to maximizing the difference $g_f(S) - g_f(S^{+v}) = \Delta^-(v)$. Instead of maximizing $\Delta^-(v)$, we maximize the lower bound $|D_v| \cdot d(S, v)$. We perform a small number of iterations of Cohen's reachability set size approximation algorithm [78] (see Section 6.3) to select the vertex v with (approximatively) largest $|D_v|$.

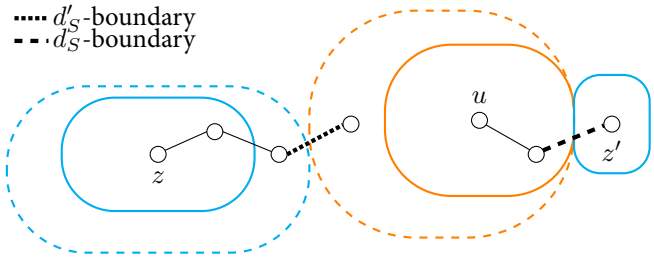


Figure 6.4: z, u and z' are vertices in S. Vertices within the solid regions belong to R_z, R_u and $R_{z'}$, respectively. Vertices within the dashed regions belong to R'_z and R'_u, respectively. After removing u from S, the vertices $x \in R'_u$ will have an invalid r'_x and $d'_S(x)$.

After v is selected, we perform a BFS from v to compute $\Delta^-(v)$ exactly. As we only need to visit the vertices whose distance to S^{+v} is smaller than to S, the BFS can be pruned at each vertex x with $d(S, x) < d(v, x)$. During the BFS, the values d_S, d'_S, r_x, and r'_x are updated to reflect the vertex addition, i.e., whether v realizes the new distance d_S or d'_S.

6.5.2 Vertex Removals

For vertex removals, we can efficiently calculate the exact increase in farness $\Delta^+(u) := g_f(S_{-u}) - g_f(S)$ for all vertices $u \in S$, even without resorting to approximation. In fact, $\Delta^+(u)$ is given as:

$$\Delta^+(u) = \sum_{x \in R_u} d'_S(x) - d_S(x).$$

We need to compute k such sums – i.e., $\Delta^+(u)$ for each $u \in S$. All of them, however, can be computed at the same time by a single linear scan through all the vertices in V.

On the other hand, it is more challenging to update d_S, d'_S, r_x, and r'_x after the removal of a vertex u from S. For vertices x with an invalid $d_S(x)$ – i.e., vertices $x \in R_u$ – we can simply update $d_S(x) \leftarrow d'_S(x)$ and $r_x \leftarrow r'_x$. This update invalidates $d'_S(x)$ and r'_x. In the following, we treat $d'_S(x)$ as infinite and r'_x as undefined for all updated vertices x; eventually, those expressions will be restored to valid values using the algorithm that we describe in the remainder of this section. Indeed, we now have to handle all vertices with an invalid $d'_S(x)$ – i.e., those in $R_u \cup R'_u$. This computation is more involved. We run a Dijkstra-like algorithm (even in the unweighted case) to "fix" $d'_S(x)$ and $r'_S(x)$. The following definition yields the starting point of our Dijkstra-like algorithm.

Definition 6.5.1 (d_S-boundary and d'_S-boundary pair)**.** Let $x \in V$ be any vertex and let $y \in N(x) \cap (R_u \cup R'_u)$ be a neighbor of x that needs to be updated.

- We call $\langle x, y \rangle$ a d_S-boundary pair for y iff $r_x \neq r_y$. In this case, we set $b(x, y) := d_S(x) + d(x, y)$.
- We call $\langle x, y \rangle$ a d'_S-boundary pair for y iff $r_x = r_y$ and $x \notin R_u \cup R'_u$. In this case, we set $b(x, y) := d'_S(x) + d(x, y)$.

In both cases, we call $b(x, y)$ the *boundary distance* of $\langle x, y \rangle$.

The definition is illustrated in Figure 6.4. Intuitively, boundary pairs define the boundary between regions of G that have a valid $d'_S(x)$ – blue regions in Figure 6.4 – and regions of the graph that have an invalid $d'_S(y)$ – orange region in Figure 6.4. The boundary distance $b(x, y)$ corresponds to the value of d'_S that a SSSP algorithm could propagate from x to y. We need to distinguish d_S-boundary pairs and d'_S-boundary pairs as the boundary distance can either be propagated on a shortest path from S over x to y (in case of a d_S-boundary pair) or on a shortest path from S_{-r_x} over x to y (in case of a d'_S-boundary pair).

Consider all $y \in V \setminus S$ such that there exists at least one (d_S- or d'_S-)boundary pair for y. For such y, let $\langle x, y\rangle$ be the boundary pair with minimal boundary distance $b(x, y)$. Our algorithm first determines all such y and updates $d'_S(y) \leftarrow b(x, y)$. If $\langle x, y\rangle$ is a d_S-boundary pair, we set $r_y \leftarrow r_x$; for d'_S-boundary pairs, we set $r_y \leftarrow r'_x$. After this initial update, we run a Dijkstra-like algorithm starting from these vertices y for which a boundary pair exists. The algorithm treats d'_S as the distance. Compared to the standard Dijkstra algorithm, ours needs the following modifications: For each vertex x, our algorithm only visits those neighbors y that satisfy $r_y \neq r'_x$; furthermore, whenever such a visit results in an update of $d'_S(y)$, we propagate $r'_y \leftarrow r'_x$. Note that these conditions imply that we never update r'_y such that $r'_y = r_y$.

Lemma 6.5.1 ([15]). After the Dijkstra-like algorithm terminates, for all the explored vertices x, $d'_S(x)$ and r'_x are correct.

6.6 Variants and Algorithmic Improvements

6.6.1 Semi-local Swaps

One weakness of the *Local-Swaps* algorithm from Section 6.4 is that it only performs local vertex exchanges. Indeed, the algorithm always swaps a vertex $u \in S$ and a vertex in $v \in N(u) \setminus S$. This condition can be generalized: in particular, it is sufficient that $u \in S$ also satisfies $u \in N(S^{+v})$. In this situation, the distances of all vertices can still only change by one and the algorithm remains correct. Note that this naturally partitions candidates u into two sets: (i) the set $N(v) \cap S$ of candidates that the original algorithm considers and (ii) the set $N(S) \cap S$. Candidates in the latter set can be determined independently of v; indeed, they can be swapped with any $v \in N(S) \setminus S$. Hence, our swap selection strategy from Section 6.4.1 continues to work with little modifications.

6.6.2 Restricted Swaps

To further improve the performance of our *Local-Swaps* algorithm at the cost of its solution quality, we consider the following variant: instead of selecting the pair of vertices $\langle u, v\rangle$ that maximizes $|D_v| - |\Lambda_u|$, we just select the vertex v that maximizes $|D_v|$ and then choose $u \in N(v) \cap S$ such that $|\Lambda_u|$ is minimized. This restricts, however, the choices for u; hence, we expect this *Restricted Local-Swaps* algorithm to yield solutions of worse quality. On the other hand, due to the restriction, we also expect it to converge faster.

6.6.3 *Local Grow-Shrink*

During exploratory experiments, it turned out that *Grow-Shrink* sometimes overestimates the lower bound $|D_v| \cdot d(S, v)$ of the decrease in farness $g_f(S) - g_f(S^{+v})$ after the addition of a vertex v. This happens

because errors in the approximation of $|D_v|$ are amplified by the multiplication with a large $d(S, v)$. Hence, we found that restricting the algorithm's choices for v to vertices near S improves the solution quality of the algorithm.

It may seem that this variant of *Grow-Shrink* makes it vulnerable to the same weaknesses of *Local-Swaps*. Namely, local choices imply that large numbers of exchanges might be required to reach a local optima, and it becomes hard to escape local optima. Fortunately, additional techniques discussed in Section 6.6.4 can be used to avoid this problem.

6.6.4 *Extended Grow-Shrink*

Even in the case of *Grow-Shrink*, the bound of Lemma 6.3.1 becomes worse for vertices at long distances from S. As detailed in Section 6.3, this happens as our reachability set size approximation strategy does not take back edges into account. This problem affects our algorithm especially on graphs with high diameter where we expect that many back edges exist. We mitigate this problem – as well as the problems mentioned in Section 6.6.3 – by allowing the group to grow by more than one vertex before shrink it again. In particular, we allow the group to grow to size $k + h$ for some $h \geq 1$, before we shrink it back to k.

In our experiments in Section 6.7, we consider two strategies to choose h. First, we consider constant values for h. However, we do not expect this to be appropriate for all graphs: specifically, we want to take the diameter of the graph into account. Hence, a more sophisticated strategy selects $h = \text{diam}(G)/k^p$ for a fixed exponent p. This strategy is inspired by mesh-like graphs (e.g., real-world road networks or other infrastructure networks): if we divide a quadratic two-dimensional mesh G into k quadratic sub-meshes (where k is a power of 2), the diameter of the sub-meshes is $\text{diam}(G)/\sqrt{k}$. Hence, if we assume that each vertex of the group covers an equal amount of vertices in the remaining graph, $h = \text{diam}(G)/\sqrt{k}$ vertex additions should be sufficient to find at least one "good" vertex that improves a size-k group. As we expect that large real-world networks deviate from ideal two-dimensional meshes to some extent, we consider not only $p = 1/2$ but also other values of p.

6.6.5 Engineering the Reachability Set Size Approximation Algorithm

Cohen's reachability set size approximation algorithm [78] has multiple parameters that need to be chosen appropriately. In particular, there is a choice of probability distribution (exponential vs. uniform), the estimator function (averaging vs. selection-based), the number of samples, and the width of each random number. For the estimator, we use the averaging estimator because it can be implemented more efficiently than a selection-based estimator – it only requires averaging numbers instead of finding the k-smallest number. We performed exploratory experiments to determine a good configuration of the remaining parameters. Our conclusions are that, while the exponential distribution empirically offers better accuracy than the uniform distribution, the algorithm can be implemented much more efficiently using the uniform distribution: specifically, for the uniform distribution, it suffices to generate and store per-vertex random numbers as (unsigned) integers, while the exponential distribution requires floating point calculations. We compensate the decrease in accuracy by simply gathering more samples.

For the uniform distribution in real-world graphs, 16 bits per integer turns out to yield sufficient accuracy. In this setting, we found that 16 samples are enough to accurately find the vertex with highest reachability set

size. In particular, while the theoretical guarantee in [78] requires the number of samples to grow with $\log n$, we found this number to have a negligible impact on the solution quality of our group-closeness heuristic (see Section 6.8.2).

6.6.6 Memory Latency in Reachability Set Size Approximation

It is well-known that the empirical performance of graph traversal algorithms (such as BFS and Dijkstra) is often limited by memory latency [26, 195]. Unfortunately, the reachability set size approximation algorithm needs to perform multiple traversals of the same graph. To mitigate this issue, we perform multiple iterations of the approximation algorithm at the same time. This technique reduces the running time of the algorithm at the cost of a higher memory footprint. More precisely, during each traversal of the graph, we store 16 random integer per vertex and we aggregate all 16 minimal values per vertex at the same time. This operation can be performed very efficiently by utilizing SIMD vector operations. Specifically, we use 256-bit AVX operations of our Intel Xeon CPUs to take the minimum of all 16 values at the same time. As mentioned above, aggregating 16 random numbers per vertex is enough for our use case; thus, using SIMD aggregation, we only need to perform a single traversal of the graph.

6.6.7 Accepting Swaps and Stopping Condition

As detailed in Sections 6.4 and 6.5, our algorithms stop once the cannot find another vertex exchange that improves the closeness score of the current group. Exchanges that worsen the score are not accepted. To prevent vertex exchanges that increase the group-closeness score only negligibly, we also set a limit on the number of vertex exchanges. In our experiments, we choose a conservative limit that does not impact the solution quality measurably (see Section 6.8.1).

6.7 Experimental Results

In this section, we evaluate the performance of our algorithms against the state-of-the-art greedy algorithm by Bergamini et al. [38] – we do not consider the naive greedy algorithm and the OSA heuristic of [76] because they are both dominated by [38]. We evaluate two variants, *LS* and *LS-restrict* (see Section 6.6.2), of our *Local-Swaps* algorithm, and three variants, *GS*, *GS-local* (see Section 6.6.3), and *GS-extended* (see Section 6.6.4) of our *Grow-Shrink* algorithm. We evaluate these algorithms for group sizes of $k \in \{5, 10, 20, 50, 100\}$ on the largest connected component of the input graph. We measure the performance in terms of running time and closeness centrality of the group computed by the algorithms. Because our algorithms construct an initial group S by selecting k vertices uniformly at random (see Section 6.2), we average the results of five runs, each one with a different random seed, using the geometric mean to aggregate speedups and relative closeness scores.[36] Unless stated otherwise, our experiments are based on the graphs listed in Table 6.1. They are all undirected and have been downloaded from the 9th DIMACS Challenge [87] and KONECT [173] public repositories. On those instances, the running time of the greedy

[36] These five runs are done to average out particularly bad (or good) selections of initial groups; as one can see from Section 6.8.2, the variance due to the randomized reachability set size algorithm is negligible.

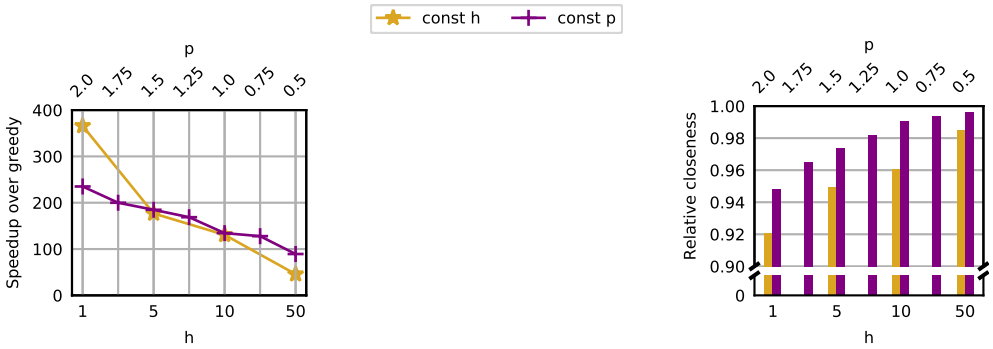
Table 6.1: Networks used in the experiments.

(a) Unweighted networks.

Network	n	m	Category
dimacs9-NY	264,346	365,050	Road
dimacs9-BAY	321,270	397,415	Road
web-Stanford	255,265	1,941,926	Hyperlink
hyves	1,402,673	2,777,419	Social
youtube-links	1,134,885	2,987,468	Social
com-youtube	1,134,890	2,987,624	Social
web-Google	855,802	4,291,352	Hyperlink
trec-wt10g	1,458,316	6,225,033	Hyperlink
dimacs10-eu-2005	862,664	16,138,468	Road
soc-pokec-relationships	1,632,803	22,301,964	Social
wikipedia_link_ca	926,588	27,133,794	Hyperlink

(b) Weighted road networks of US states.

State	n	m
DC	9,522	14,807
HI	21,774	26,007
AK	48,560	55,014
DE	48,812	59,502
RI	51,642	66,650
CT	152,036	184,393
ME	187,315	206,176
ND	203,583	249,809
SD	206,998	249,828
WY	243,545	293,825
ID	265,552	310,684
MD	264,378	312,977
WV	292,557	320,708
NE	304,335	380,004



(a) Speedup over the greedy algorithm (geometric mean).

(b) Closeness score relative to the score of the group returned by greedy (geometric mean).

Figure 6.5: Performance of the extended *Grow-Shrink* algorithm for different values of h or p; unweighted graphs, $k = 10$.

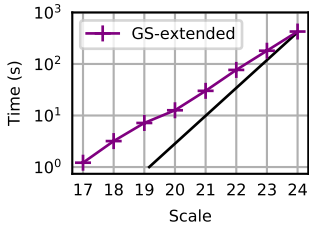
baseline always varies between 10 minutes to 2 hours – detailed running times are reported in Tables E.1 and E.2, Appendix E.1.

Our algorithms are implemented in the NetworKit [273] C++ framework and use PCG32 [234] to efficiently generate random numbers. All experiments were executed with sequential code on a Linux machine with an Intel Xeon Gold 6154 and 1.5 TiB of RAM of memory.

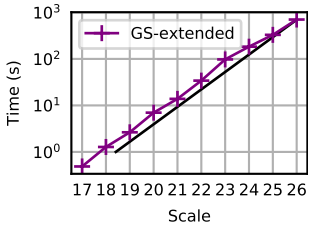
6.7.1 Results for *Extended Grow-Shrink*

In a first experiment, we evaluate the performance of our *extended Grow-Shrink* algorithm against the greedy heuristic. Because of its ability to escape local optima, we expect it to be the best algorithm in terms of quality – hence, it should be a good default choice among our algorithms. For this experiment, we set $k = 10$.

As discussed in Section 6.6.4, we implement two strategies to determine h: we either fix a constant h, or we fix a constant p. For both strategies, we evaluate multiple values for h or p; results are shown in Figure 6.5. As expected, higher values of h (or, similarly, lower values of p) increase the algorithm's running time – while $h > 1$ allows *Grow-Shrink* to perform better choices, it does not converge h times as fast. Still,



(a) R-MAT networks; 2^{17} to 2^{24} vertices (up to 268 million edges).



(b) Random hyperbolic networks; 2^{17} to 2^{26} vertices (up to 671 million edges).

Figure 6.6: Running time (in seconds) of the extended *Grow-Shrink* algorithm on synthetic graphs; $p = 0.75$, $k = 10$.

Table 6.2: Running time of the extended *Grow-Shrink* algorithm on large real-world networks; $p = 0.75$, $k = 10$.

Network	n	m	Time (s)
soc-LiveJournal1	4,843,953	42,845,684	95.3
livejournal-links	5,189,808	48,687,945	135.6
orkut-links	3,072,441	117,184,899	199.9
dbpedia-link	18,265,512	126,888,089	368.0
dimacs10-uk-2002	18,459,128	261,556,721	333.1
wikipedia_link_en	13,591,759	334,640,259	680.1

for all tested values of h or p, the extended *Grow-Shrink* algorithm is one to two orders of magnitude faster than the greedy baseline. Furthermore, values of $p < 1$ yield results of very good quality: for $p = 0.75$, for example, we achieve a quality of 99.4%. At the same time, using this setting for p, our algorithm is $127.8\times$ faster than the greedy algorithm. We remark that, for all but the smallest values of h (i.e., those corresponding to the lowest quality), choosing constant p is a better strategy than choosing constant h: for the same running time, constant p always achieves solutions of higher quality.

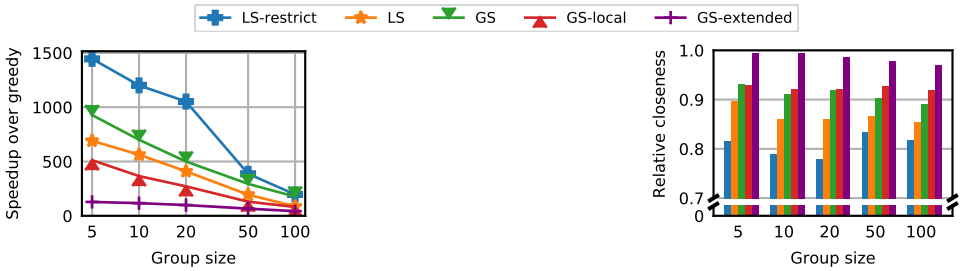
6.7.2 Scalability to Large Graphs

We also analyze the running time of our extended *Grow-Shrink* algorithm on large-scale networks. To this end, we switch to graphs larger than the ones in Table 6.1. We fix $p = 0.75$, as Section 6.7.1 demonstrated that this setting results in a favorable trade-off between solution quality and running time. The greedy algorithm is not included in this experiments as it requires multiple hours of running time, even for the smallest real-world graphs we consider in these experiments.

Results on Synthetic Data. Figure 6.6 shows the average running time of our algorithm on randomly generated R-MAT [65] as well as graphs from a generator for random hyperbolic graphs [191].[37] For the R-MAT generator, we use the same parameter setting as in the Graph 500's benchmark [223] – i.e., edge factor 16, $a = 0.57, b = 0.19, c = 0.19$, and $d = 0.05$. Concerning the random hyperbolic generator, we set the average degree to 20, and the exponent of the power-law degree distribution to 3.

In the (log-log) plot, the straight lines represent a linear regression of the running times. In both cases, the running time curves are almost as steep as the regression line, i.e., the running time grows linearly in the number of vertices for the considered network models and sizes.

[37] Like R-MAT, the random hyperbolic model yields graphs with a skewed degree distribution, similar to the one found in real-world complex networks.



(a) Speedup over the greedy algorithm (geometric mean).

(b) Closeness score relative to the score of the group returned by greedy (geometric mean).

Figure 6.7: Performance of our local search algorithms for different values of k; unweighted graphs.

Results on Large Real-World Datasets. Table 6.2 reports the algorithm's performance on large real-world graphs. In contrast to the greedy algorithm – which would requires hours, our extended *Grow-Shrink* algorithm handles real-world networks with hundreds of millions of edges in a few minutes. For the *orkut-links* network, Bergamini et al. [38] report running times for the greedy of *16 hours* on their machine – it is the largest instance in their experiments.

6.7.3 Accelerating Performance on Unweighted Graphs

While the extended *Grow-Shrink* algorithm yields results with very high quality, if quality is not a primary concern, even faster algorithms might be desirable for very large graphs. To this end, we also evaluate the performance of the non-extended *Grow-Shrink* and the *Local-Swaps* algorithms. For extended *Grow-Shrink*, we fix $p = 0.75$ again. The speedup and the quality of our algorithms over the greedy baseline, for different values of the group size k, are shown in Figures 6.7a and 6.7b, respectively. Note that the greedy algorithm scales well for large k, so that the speedup of our algorithms decreases with k.[38] However, even for large groups of $k = 100$, all of our algorithms are still at least $43.0\times$ faster.

Our non-extended local version of *Grow-Shrink* is the next best algorithm after extended *Grow-Shrink*. As explained in Section 6.6.3, this variant gives better solutions than non-local *Grow-Shrink*; further, it achieves a speedup of $3.1\times$ over extended *Grow-Shrink* with $p = 0.75$ and $k = 10$ – and a speedup of $365.8\times$ over greedy. The solution quality in this case is 92.1% of the greedy quality.

The non-restricted *Local-Swaps* algorithm is dominated by *Grow-Shrink*, both in terms of running time and solution quality. Furthermore, compared to other algorithms, the restricted *Local-Swaps* algorithm only gives a rough estimate of the group with highest closeness; it is, however, significantly faster than all other algorithms and may be employed in exploratory analysis of graph datasets.

6.7.4 Results on Weighted Road Networks

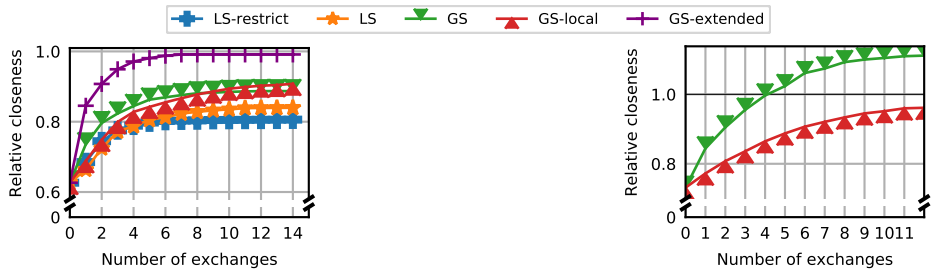
Recall that the *Local-Swaps* algorithm does not support weighted graphs; thus, in the weighted case, we report only *Grow-Shrink* data. The performance of *Grow-Shrink* and local *Grow-Shrink* on weighted graphs is shown in Figure 6.8. In contrast to the unweighted case, the quality of the non-local *Grow-Shrink* algorithm is superior to the greedy baseline for all the considered group sizes. Furthermore, contrary to unweighted

[38] Indeed, as mentioned in Section 6.1, the main bottleneck of the greedy algorithm is adding the first vertex into the group.

(a) Speedup over the greedy algorithm (geometric mean).

(b) Closeness score relative to the score of the group returned by greedy (geometric mean).

Figure 6.8: Performance of our local search algorithms for different values of k; weighted graphs.



(a) Unweighted graphs.

(b) Weighted graphs.

Figure 6.9: Behavior of the relative closeness score (compared to the group returned by greedy, geometric mean) over the execution of the algorithms (in terms of vertex exchanges); $k = 10$.

graphs, the ability to perform non-local vertex exchanges greatly benefits the non-local *Grow-Shrink* compared to local *Grow-Shrink*.[39] Thus, on the weighed graphs in our benchmark set, *Grow-Shrink* clearly dominates both the greedy and the local *Grow-Shrink* algorithms – both in terms of speedup *and* solution quality.

6.8 Additional Experiments

6.8.1 Impact of the number of vertex exchanges

Figures 6.9a and 6.9b depict the closeness score relative to the one of the group computed by the greedy algorithm depending on number of vertex exchanges performed by the algorithm. Again, for extended *Grow-Shrink*, we fix $p = 0.75$. All of the local search algorithms quickly converge to a value near their final result – additional exchanges improve the closeness score by small amounts. Thus, in order to avoid an excessive number of iterations, setting a limit on the number on the number of vertex exchanges seems a reasonable choice. In our experiments, we set a conservative limit of 100 exchanges.

(a) Speedup over the greedy algorithm (geometric mean).

(b) Closeness score relative to the score of the group returned by greedy (geom. mean).

Figure 6.10: Performance of the *Grow-Shrink* algorithm for different numbers of samples to estimate reachability set size; $k = 10$.

6.8.2 Impact of reachability set size approximation

As mentioned in Section 6.6.3, the errors in the approximation of $|D_v|$ are amplified by the multiplication with $d(S, v)$. This results in GS-local computing higher quality solutions than GS. We study how increasing the accuracy of the reachability set size approximation by increasing the number of samples impacts the performances of both GS and GS-local. Figure 6.10a shows that GS needs at least 64 samples to converge to a better local optimum than GS-local. However, in both cases increasing the number of samples degrades the speedup without leading to a substantial quality improvement (see Figure 6.10b).

6.8.3 Summary of Experimental Results

On unweighted graphs, a good trade-off between running time and solution quality is achieved by the extended *Grow-Shrink* algorithm with constant $p = 0.75$. This strategy yields solutions with at least 99.4% of the closeness score of a greedy solution – greedy, in turn, was at most 3% away from the optimum on small networks in previous work [38]. With $k = 10$, extended *Grow-Shrink* is $127.8\times$ faster than greedy. Thus, it is able to handle graphs with hundreds of millions of edges in a few minutes – while the state of the art needs multiple hours. Further, if a fast but inaccurate algorithm is needed for applications such as exploratory analysis of graph datasets, we recommend to use the non-extended *Grow-Shrink* algorithm, or, if only a very coarse estimate of the group with maximal closeness is needed, restricted *Local-Swaps*.

On weighted graphs, our recommendation is always to use our *Grow-Shrink* algorithm, as it outperforms the greedy state of the art both in terms of quality – yielding solutions that are on average 12.4% better than greedy solutions – and in terms of running time performance – with a speedup of two orders of magnitude, at the same time.

6.9 Conclusions

In this chapter, we introduced two new families of local search algorithms for group-closeness maximization in large networks. As maximizing group-closeness exactly is infeasible for graphs with more than a few

[39] For this reason, we do not include the extended *Grow-Shrink* in this experiment. In fact, we expect that it improves only slightly on GS-local (red line/bars in Figure 6.8) but cannot compete with (non-local) GS: indeed, the ability of performing non-local vertex exchanges, as done by GS (green line/bars in Figure 6.8) appears to be crucial to obtain high-quality results on weighted graphs.

thousand edges, our algorithms are heuristics – just like the state-of-the-art greedy algorithm [38]. However, for small real-world networks where the optimum can be computed reasonably fast, the results are empirically known to be close to optimal solutions [38].

Compared to previous state-of-the-art heuristics, our algorithm (extended *Grow-Shrink* in particular) allows to find groups with high closeness centrality in real-world networks with hundreds of millions of edges in seconds to minutes instead of multiple hours, while sacrificing less than 1% in quality. In weighted graphs, *Grow-Shrink* (GS) even dominates the best known heuristic: the GS solution quality is more than 10% higher and GS is two orders of magnitude faster.

7 Group-Harmonic and Group-Closeness Maximization – Approximation and Engineering

7.1 Introduction

In Chapter 6, we introduced the *group-closeness maximization* problem. This problem is $\mathcal{NP}$-hard and recent work focused on heuristics [15, 38, 76]. Heuristics, however, do not provide approximation bounds, leaving the open question of approximability of group-closeness maximization. The close relationship between group-closeness maximization and the metric k-Median problem as well as know local search algorithms with constant-factor approximation bounds for the latter [20] motivate us to investigate whether the k-Median results can be transferred not only to group-closeness maximization, but also to group-harmonic maximization.

Contribution. In this chapter, we illustrate how the new theoretical results presented in [12] can be used to develop the first approximation algorithms for group-harmonic and group-closeness maximization. For group-harmonic, we implement an efficient greedy algorithm which, as shown in [12], has approximation ratios of $\lambda(1 - 2/e)$ in directed graphs and $\lambda(1 - 1/e)/2$ in undirected graphs, where λ is the ratio of the minimal and the maximal edge weights. Additionally, since g_h is submodular [12] but not necessarily monotone, we directly apply the local search algorithm by Lee et al. [179], which evaluates $\Omega(n^2)$ swaps per iteration and achieves an approximation factor of $\frac{1+\varepsilon}{6}$.

Concerning group-closeness, we adapt the local search algorithm for k-Median by Arya et al. [20] to group-closeness maximization. The algorithm yields constant-factor approximation for undirected graphs, whereas in directed case the approximation factor is $\Theta(n^{-\varepsilon})$, with $\varepsilon < 1/2$.

In our experimental study, we show that, on small instances where an optimum can be computed in reasonable time, the quality of both the greedy and the local search algorithms are on average less than 0.5% away from the optimum. On larger instances, our local search algorithms yield results with superior quality compared to existing greedy [38] and local search solutions [15] at the cost of additional running time. In particular, local search is one to three orders of magnitude slower than greedy, but this is to be expected due to a high quality demand – indeed, unlike the local search heuristics presented in Chapter 6, local search approximation algorithms often cut greedy's (empirical) gap to optimality by half or more. We thus advocate local search approximation for scenarios where solution quality is of highest concern.

Bibliographic Notes. The contributions presented in this chapter were published in the Proceedings of the *Twenty-Third Workshop on Algorithm Engineering and Experiments (ALENEX 2021)*. My contribu-

tions involve the engineering improvements (Section 7.3.3), the implementation of all presented algorithms, and carrying out the experiments. The rest is joint work with Ruben Becker, Gianlorenzo D'Angelo, Hugo Gilbert, Alexander van der Grinten, and Henning Meyerhenke. Proofs to which I did not contribute are omitted and can be found in the original paper [12].

7.2 Preliminaries

Recall from Section 2.5 the definitions of group-harmonic and group-closeness centrality of a group $S \subset V$, namely: $g_c(S) := \frac{n}{\sum_{u \in V \setminus S} d(S,u)}$ and $g_h(S) := \sum_{u \in V \setminus S} \frac{1}{d(S,u)}$, where $\frac{1}{d(S,u)} = 0$ if there is no path from S to u.

Problems Addressed. In this chapter, we address the problems of finding groups with size $1 \leq k < n$ that maximize group-harmonic and group-closeness, more formally:

Group-Harmonic Maximization	
Input:	Graph $G = (V, E, w)$, integer $1 \leq k < n$.
Find:	Set $S^\star \subset V$ with $\lvert S \rvert = k$ s.t. $g_h(S^\star)$ is maximum.

Group-Closeness Maximization	
Input:	Graph $G = (V, E, w)$, integer $1 \leq k < n$.
Find:	Set $S^\star \subset V$ with $\lvert S \rvert = k$ s.t. $g_c(S^\star)$ is maximum.

While group-closeness maximization has already been studied in other works [15, 38, 76], to the best of our knowledge, we are the first to study the group-harmonic maximization problem.

In both problems we study, we are given a (possibly directed) weighted graph $G = (V, E, w)$ with weight function $w : E \to \mathbb{N}_{>0}$. For group-closeness maximization, we assume that G is (strongly) connected, whereas for group-harmonic we only assume that there are no isolated vertices. Further, we denote by $w_{\min} := \min_{e \in E} w(e)$ and $w_{\max} := \max_{e \in E} w(e)$ the lowest and the highest edge weights, respectively, and by $\lambda := \frac{w_{\min}}{w_{\max}}$ the ratio of the smallest and the largest edge weights.

7.3 Group-Harmonic Maximization

7.3.1 Mathematical Properties

First of all we consider the mathematical properties of g_h. We observe that the function is submodular but not monotone.

Lemma 7.3.1. The function $g_h : 2^V \to \mathbb{Q}_{\geq 0}$ is submodular [12, Lemma 3.1].

The non-monotonicity of g_h can be shown with a counterexample: consider an undirected graph with $V = \{u, v\}$ and $E = \{\{u, v\}\}$; we have that $g_h(\{u\}) = 1$, while $g_h(\{u, v\}) = 0$.

7.3.2 Approximation Algorithms

As g_h is submodular, we can directly apply the algorithm by Lee et al. [179] and obtain a $(\frac{1+\varepsilon}{6})$-approximation – the exact cardinality constraint corresponds to the case of a single matroid base constraint,

Algorithm 14 Greedy algorithm for maximizing a monotone submodular set function f under a cardinality constraint $|S| = k$.

```
1: S ← ∅
2: while |S| < k do
3:     u ← argmax_{v∈V\S}{f(S ∪ {v}) − f(S)}
4:     S ← S ∪ {v}
5: return S
```

where the matroid is the uniform one. This algorithm was notably improved by Vondrák [282], who designed a randomized local search method with an approximation factor of $\frac{1}{4} - o(1)$. Another candidate approximation algorithm for group-harmonic maximization is the greedy algorithm (see Algorithm 14). It provides an approximation factor of $1 - \frac{1}{e}$ for maximizing a monotone and submodular set function under a cardinality constraint $|S| = k$. However, as observed in Section 7.3.1, g_h is not monotone and we cannot use this result directly. Algorithm 14 still guarantees approximation bounds nonetheless, they are summarized in the following Theorem.

Theorem 7.3.1 ([12])**.** Algorithm 14 guarantees the following approximation factor for the group-harmonic maximization problem, where $\lambda := \frac{w_{\min}}{w_{\max}}$ is the ratio of the minimum and the maximum edge weights:

- $\lambda\left(1 - \frac{2}{e}\right) > 0.264\lambda$ in the directed case;
- $\frac{\lambda}{2}\left(1 - \frac{1}{e}\right) > 0.316\lambda$ in the undirected case.

While these approximation factors may be worse than the ones provided by Lee et al. [179] and Vondrák [282], they offer better guarantees for the group-harmonic maximization problem in unweighted graphs.

Lemma 7.3.2 ([12])**.** If G is unweighted, then the set returned by Algorithm 14 provides a $\frac{1}{2}\left(1 - \frac{1}{e}\right)$-approximation.

7.3.3 Engineering Improvements

In the following, we illustrate engineering techniques to accelerate the greedy and the local search algorithms for group-harmonic maximization.

Greedy Algorithm. The pseudocode of our greedy algorithm is given by Algorithm 15. The first vertex to be added to the group is the vertex with highest harmonic centrality (Line 1); this vertex can be found by a ranking algorithm such as the one from Bisenius et al. [48]. Afterwards, the algorithm iteratively adds to the group the vertex u with highest marginal gain $g_h(S \cup \{u\}) - g_h(S)$.

Since g_h is submodular, we can evaluate marginal gains lazily, i.e., the marginal gain from previous iterations $\widehat{g_h}(S, u)$ serves as an upper bound of the marginal gain $g_h(S \cup \{u\}) - g_h(S)$ in the current iteration. Since $\widehat{g_h}(S, u) \geq g_h(S \cup \{u\})$ holds after S is initialized with the vertex with highest harmonic centrality, we initialize $\widehat{g_h}(S, u)$ to $\widetilde{c_h}(u)$ for each $u \in V \setminus S$, that is the upper bound on $c_h(u)$ computed by the Bisenius et al. algorithm [48]. To determine the vertex with highest marginal gain, we use the well-known lazy strategy [217]: we evaluate the marginal gain of the vertex with highest upper bound – and adjust the

Algorithm 15 Greedy algorithm for group-harmonic maximization.

```
1: top_h ← topHarmonicCloseness() [48]
2: S ← {top_h}
3: while |S| < k do
4:     PQ ← max-priority queue with key ĝ_h(S, u) and value u
5:     for each u ∈ V \ S do
6:         PQ.push(u)
7:     x ← null                                   ▷ Vertex with highest marginal gain computed so far
8:     g_h(S ∪ {x}) ← −∞
9:     repeat                                     ▷ This loop is done in parallel
10:        u ← PQ.extractMax()
11:        if ĝ_h(S, u) ≤ g_h(S ∪ {x}) then
12:            break                              ▷ x has the highest marginal gain
13:        ⟨isExact, g_h(S ∪ {u})⟩ ← pruned SSSP(u, g_h(S ∪ {x}))
14:        if isExact and g_h(S ∪ {u}) > g_h(S ∪ {x}) then
15:            x ← u
16:    until PQ is empty
17:    S ← S ∪ {x}
18: return S
```

upper bound to the true marginal gain – until we know the true marginal gain of the top vertex w.r.t. the upper bound (Lines 4–17 of Algorithm 15, by using a priority queue).

To evaluate the marginal gain of a vertex u, we run a pruned SSSP from u (Line 13) that only visits vertices v such that $d(u,v) < d(S,v)$ and updates $\widehat{g_h}(S,u)$ after every vertex at distance i from u has been explored. The traversal is interrupted if $\widehat{g_h}(S,u) \leq g_h(S \cup \{x\})$, where x is the vertex with highest marginal gain computed so far; otherwise, the SSSP visits all the vertices that are closer to u than to S and returns the exact value of $g_h(S \cup \{u\})$. As for group-closeness, $\widehat{g_h}$ is defined differently for weighted than for unweighted graphs.

Pruning – Unweighted Graphs. In unweighted graphs, we exploit additional bounds to interrupt the SSSP earlier. Let us assume that the pruned SSSP – i.e., a BFS – has explored all vertices up to distance i. We denote be $\Phi^{\leq i}_{S,u}$ the set of vertices v such that $d(u,v) \leq i$ and $d(u,v) < d(S,v)$, with $\Phi^{i}_{S,u}$ the set of vertices v such that $d(u,v) = i$ and $d(u,v) < d(S,v)$ and with $n^{i}_{S,u}$ its cardinality. An additional upper bound on the marginal gain of u is:

$$\sum_{v \in \Phi^{\leq i}_{S,u} \setminus \{u\}} \left(\frac{1}{d(u,v)} - \frac{1}{d(S,v)} \right) + \frac{\tilde{n}^{i+1}_{S,u}}{i+1} + \frac{r(u) - |\Phi^{\leq i}_{S,u}| - \tilde{n}^{i+1}_{S,u}}{i+2} - \frac{1}{d(S,u)}. \tag{7.1}$$

The first summand is the contribution to the marginal gain due to the explored vertices up to distance i. In the second summand, we assume that $\tilde{n}^{i+1}_{S,u} \geq n^{i+1}_{S,u}$ vertices are at distance exactly $i+1$ from u, where $\tilde{n}^{i+1}_{S,u}$ is defined as $\sum_{x \in \Phi^{i}_{S,u}} \deg_{\text{out}}(x)$ for directed graphs and $\sum_{x \in \Phi^{i}_{S,u}} (\deg(x) - 1)$ for undirected graphs.[40] In the third summand we assume that all the remaining vertices reachable from u are at distance $i+2$ from u – recall that $r(u)$ is the number of vertices reachable from u.[41] Finally, we subtract the contribution of u to the centrality of S.

[40] As done with upper bound for NBCut in Eq. (3.1), a tighter numerator of the second summand of Eq. (7.1) is $\min(n - n^{i}_{S,u}, \tilde{n}^{i+1}_{S,u})$. However, to simplify our notation, we keep writing $\tilde{n}^{i+1}_{S,u}$ in the text, but implement the better bound in practice.

[41] Because in directed graphs it is too expensive to compute $r(u)$ for each vertex, we use an upper bound as described in [37].

As a further optimization for unweighted and undirected graphs, for every vertex $u \in V \setminus S$ we subtract from $r(u)$ all the vertices in u's connected component that are at distance 1 from S. In this way, we avoid to count them in the third summand of Eq. (7.1).

Pruning – Weighted Graphs. In weighted graphs, the SSSP is a pruned version of the Dijkstra algorithm. Let i be the distance from u to the last explored vertex. Upon completion of Dijkstra's relaxation step, $\widehat{g_h}(S, u)$ is updated as follows:

$$\widehat{g_h}(S, u) = \sum_{v \in \Phi_{S,u}^{\leq i} \setminus \{u\}} \left(\frac{1}{d(u, v)} - \frac{1}{d(S, v)} \right) + \frac{r(u) - |\Phi_{S,u}^{\leq i}|}{i} - \frac{1}{d(S, u)}, \tag{7.2}$$

i.e., we count the contribution to $\widehat{g_h}(S, u)$ of (i) the vertices visited by the SSSP and of (ii) the unexplored vertices assuming that they are all at distance i from u.

Local Search. The local search algorithm by Lee et al. [179] needs to evaluate $\Omega(n^2)$ swaps per iteration. This is already quite expensive, so it is desirable to perform only few iterations. To this end, we initialize the local search with a greedy solution – this does not affect its approximation guarantee but accelerates the algorithm considerably in practice.

Although we cannot use lazy evaluation for local search – we need to consider swaps, not vertex additions – we can still make use of the bound from Eq. (7.1) while evaluating a swap.

Parallelism. Both greedy and local search typically need to evaluate either the marginal gains or the objective function for several vertices before performing a single addition (or swap). Since these evaluations are independent of each other, it is desirable to utilize parallelism. We parallelize multiple evaluations of the objective function in a straightforward way: Each thread evaluates the marginal gain for one candidate vertex; this incurs of $\mathcal{O}(n)$ additional memory per thread as it needs to store the state of a single SSSP.

7.4 Group-Closeness Maximization

7.4.1 Preliminary Discussion

Different variants of the group-closeness maximization problem occur depending on whether the graph at hand is undirected or directed. From an approximation algorithm's perspective, it is tempting to observe that the group-farness g_f is a supermodular set function and conclude that g_c is submodular. In the literature, this argument was used in the paper by Chen et al. [76]. Unfortunately, this approach is flawed because the reciprocal of a supermodular set function is not necessarily submodular and the approximation question remains unanswered. Indeed, as shown in Lemma 2.5.2, g_c is not submodular.

A similar, yet not flawed, strategy was taken by Li et al. [183]. In they work they deal with the current-flow closeness centrality – also known as electrical closeness, see Section 2.4 – and they measure the approximation factor of their algorithms in a different way, allowing them to obtain constant-factor approximation results. As argued in Ref. [12, Appendix C], an analogous approach can be applied in our setting, yielding constant-factor approximation algorithms for group-closeness maximization in their sense (see Ap-

pendix F.1 for further details). We remark, however, that the notion of approximation used by Li et al. is a fundamentally different notion of approximation.

7.4.2 Approximation Algorithms

We start by introducing the metric k-Median problem and how we adapt it to our setting.

Metric k-Median	
Input:	Set of clients C, set of facilities F, cost function $c : C \times F \to \mathbb{R}_{\geq 0}$ satisfying triangle inequality, integer k.
Find:	Set $S \subseteq F$ with $\lvert S\rvert \leq k$ s.t. $c(S) := \sum_{i \in C} \min_{j \in S} c(i,j)$ is minimum.

Arya et al. [20] show that the local search algorithm that performs p swaps at each step leads to a solution with approximation ratio at most $3 + 2/p$ for Metric k-Median.

The *group-farness minimization* problem can be seen as a special case of the metric k-Median problem where C and F are both taken to be the vertex set V and the cost function being obtained using the shortest-path distances. Since g_f is monotone, the result of Arya et al. carries over to the undirected group-farness minimization problem with exact cardinality constraint, yielding an approximation factor of $\frac{p}{3p+2}$ for group-closeness maximization.

7.4.3 Engineering Improvements

Since the greedy algorithm for group-closeness has already been studied before [38], in the following, we discuss local search and engineering improvements.

Local Search. We consider the local search algorithm that, at each iteration, evaluates all possible pairs of swaps. For the k-Median case, Arya et al. [20] minimize the cost function of an initial solution S. A swap is done only if $c(S') \leq (1 - \varepsilon/Q) \cdot c(S)$, where S' is the solution after the swap, Q is the number of neighboring solutions – i.e., how many different S' are one swap away from S – and $\varepsilon > 0$. For group-closeness, the cost function is represented by g_f – minimum farness is maximum closeness – and Q is $k(n-k)$, i.e., the number of possible swaps. The algorithm has an approximation ratio of 5.

Like in the group-harmonic case, the local search algorithm is much faster in practice if we start from a good initial solution. To this end, we use the *Grow-Shrink* algorithm described in Section 6.5 which, despite being heuristic, in real-world graphs it quickly finds high-quality solutions. The lack of approximation ratio in *Grow-Shrink* is not an issue in our case, the approximation guarantee of our local search does not depend on the initial solution.

Prioritizing Swaps. The actual number of swaps that need to be evaluated before a local optimum is reached is heavily affected by the sequence of swaps that are done. Algorithm 16 summarizes how we prioritize the swaps. Similarly to *Grow-Shrink*, we prioritize swaps depending on their estimated impact on $g_f(S)$. First, we sort in ascending order the vertices in S by the increase in g_f due to their removal from S, i.e., $g_f(S \setminus \{u\}) - g_f(S)$ for all $u \in S$ (Lines 4–6 of the pseudocode). Afterwards, in Lines 13–15, we sort in descending order all the vertices $v \in V \setminus S$ by $\widetilde{g_f}((S \cup \{v\}) \setminus \{u\})$, which is an estimate of the decrease

Algorithm 16 Overview of the single-swap algorithm for group-closeness maximization.

```
1:  S ← Grow-Shrink(G, k)
2:  g_f(S) ← SSSP(S)
3:  repeat
4:      PQ_u ← min-priority queue with key (g_f(S \ {u}) − g_f(S) and value u
5:      for each x ∈ S do
6:          PQ_u.push(x)
7:      didSwap ← false
8:      repeat
9:          u ← PQ_u.extractMin()
10:         /* Compute the exact farness increase */
11:         g_f^+(u) ← g_f(S \ {u}) − g_f(S)
12:         compute g̃_f((S ∪ {v}) \ {u}) for all v ∈ V \ S
13:         PQ_v ← max-priority queue with key g̃_f((S ∪ {v}) \ {u}) and value v
14:         for each x ∈ V \ S do
15:             PQ_v.push(x)
16:         repeat
17:             v ← PQ_v.extractMax()
18:             /* Compute the exact farness decrease */
19:             g_f((S ∪ {v}) \ {u}) ← pruned SSSP from v
20:             if g_f((S ∪ {v}) \ {u}) ≤ (1 − ε/(k(n−k))) g_f(S) then
21:                 S ← (S ∪ {v}) \ {u}
22:                 g_f(S) ← SSSP(S)
23:                 didSwap ← true
24:                 break
25:         until PQ_v is empty
26:         if didSwap then
27:             break
28:     until PQ_u is empty
29: until not didSwap
30: return S
```

in farness, i.e., $g_f((S \cup \{v\}) \setminus \{u\}) - g_f(S)$. We use the same estimate based on the size of the shortest path DAGs as described in Chapter 6.

As a further optimization, we exclude swaps with vertices in $V \setminus S$ with degree 1 because, in (strongly) connected graphs, they cannot result in a decrease in g_f.

Additional Pruning. Recall from Section 6.5 that the *Grow-Shrink* algorithm performs a pruned SSSP to evaluate whether a swap is advantageous. We modify the algorithm to incorporate additional pruning conditions that interrupt the SSSP when a swap is not good enough to be considered in the local search – in contrast, *Grow-Shrink* performs any swap that improves the objective function, regardless of the difference in score. Specifically, we maintain a lower bound $\widehat{g_f}(S, u, v) \leq g_f((S \cup \{v\}) \setminus \{u\})$ so that we can interrupt the pruned SSSP as soon as $\widehat{g_f}(S, u, v) > \left(1 - \frac{\varepsilon}{k(n-k)}\right) g_f(S)$.

$\widehat{g_f}(S, u, v)$ is computed in two steps: we first compute $g_f^+(S, u) := g_f(S \setminus \{u\}) - g_f(S)$ exactly (Line 11 in Algorithm 16) that is, the increase in farness of S due to the removal of u. Then, during every pruned SSSP from v, we keep updating an upper bound of the decrease in farness of $S \setminus \{v\}$ due to the addition of v: $\widehat{g_f}^-(S, v) := g_f(S \setminus \{u\}) - \widehat{g_f}(S, u, v)$. Then, $\widehat{g_f}(S, u, v)$ is computed as $g_f(S) + g_f^+(S, u) - \widehat{g_f}^-(S, v)$.

To compute $g_f^+(S, u)$ exactly we maintain the following information for each vertex $x \in V \setminus S$: the distance $d(S, x)$, a representative vertex $r_x \in S$ such that $d(r_x, x) = d(S, x)$, and the distance $d'(S, x) =$

$d(S \setminus \{r_x\}, x)$. In this way, $g_f^+(S, u)$ can be computed in $\mathcal{O}(n)$ time as done by the original *Grow-Shrink* algorithm:

$$g_f^+(S, u) = \sum_{x \in \{V \setminus S : d(S,x) = d(u,x)\}} d(S, x) - d(S', x).$$

$\widehat{g_f}^-(S, v)$ is computed differently in unweighted and weighted graphs. In unweighted graphs, the pruned SSSP is a BFS, and we define bounds inspired by the ones used for top-k closeness centrality in Ref. [37]: For every distance $1 \leq i \leq \text{diam}(G)$ we maintain $N_{\geq i}(S)$, i.e., the set of vertices at distance $\geq i$ from S, and $\Phi_{S,v}^{\leq i}$, i.e., the set of vertices x such that $d(v, x) \leq i$ and $d(v, x) < d(S, x)$. Once every vertex in $\Phi_{S,v}^{\leq i}$ has been visited by the pruned BFS, we know that at most $\tilde{n}_{i+1}(v) := \min\Big(|N_{\geq i+2}(S)|, \sum_{x \in \Phi_{S,v}^{i}} \deg_{\text{out}}(x)\Big)$ vertices can be at distance $i+1$ from v, while the remaining unexplored vertices will be at least at distance $i+2$ – in undirected graphs, $\tilde{n}_{i+1}(v) := \min\Big(|N_{\geq i+2}(S)|, \sum_{x \in \Phi_{S,v}^{i}} (\deg(x) - 1)\Big)$. This, we update $\widehat{g_f}^-(S, v)$ as follows:

$$\begin{aligned}\widehat{g_f}^-(S, v) = &\sum_{x \in \Phi_{S,v}^{\leq i}} (d(S, x) - d(v, x)) + \\ &\sum_{x \in \Lambda} (d(S, x) - i - 1) + \sum_{x \in N_{i \geq 3}(S) \setminus \Lambda} (d(S, x) - i - 2).\end{aligned}$$

The first summand represents the decrease in farness due to the vertices that have already been visited by the BFS. In the second summand, $\Lambda \subseteq N_{\geq i+2}(S)$ contains the $\tilde{n}_{i+1}(v)$ vertices closest to S, and we assume that they are at distance $i + 1$ from v.[42] Finally, in the third summand we assume that all the remaining unvisited vertices are at distance $\geq i+3$ from S and not counted in Λ can be reachable from v in $i+2$ hops. From the third summand we exclude vertices at distance $i+2$ from S because, under our assumption, their distance from S would remain unchanged. At the cost of additional $\mathcal{O}(\text{diam}(G))$ space, $\widehat{g_f}^-(S, v)$ can be computed in $\mathcal{O}(\text{diam}(G))$ time.

For weighted graphs, we update $\widehat{g_f}^-(S, v)$ by adapting our strategy from g_h to g_f (see Eq. (7.2)).

Parallelism. We employ the same parallelization technique as for group-harmonic maximization. The fact that in the greedy and local search algorithms evaluations of the objective function can be parallelized can be seen as an advantage over the *Grow-Shrink* algorithm which, in turn, is inherently sequential.

7.5 Experiments

We conduct experiments to evaluate our algorithms in terms of solution quality and running time. For group-harmonic, we first evaluate the quality of our greedy algorithm (Greedy-H), our local search algorithm that starts from a greedy solution (Greedy-LS-H), and the best over 100 sets selected uniformly at random (Best-Random-H) against the optimal solution on small-sized networks. Then, we measure the quality and the running time performance of Greedy-H and Greedy-LS-H and we use Best-Random-H as baseline.

[42] This assumption is done in order to consider the "worst-case scenario" (in terms of decrease in farness) where the vertices ad distance $i + 1$ from v are also the closest ones to S.

Regarding group-closeness, we compare our local search algorithm against the greedy algorithm by Bergamini et al. [38], the *Grow-Shrink* algorithm (see Chapter 6),[43] and the best of 100 randomly chosen sets. Hereafter, these algorithms are referred to as Greedy-C, GS, and Best-Random-C, respectively. Our local search algorithm for group-closeness uses either Greedy-C or GS to initialize the initial solution: in the former case we label it as Greedy-LS-C, and GS-LS-C in the latter.

7.5.1 Settings

We implement all algorithms in C++ using the NetworKit [273] graph APIs and we use SCIP [118] to solve ILP instances. All experiments are conducted on a Linux machine with an Intel Xeon Gold 6126 with 2×12 cores and 192 GiB of RAM, and managed by the SimexPal [14] software for reproducibility. We aggregate approximation ratios and speedups using the geometric mean. All experiments have a timeout of one hour.

7.5.2 Instances Statistics

Datasets. Experiments are executed on real-world complex and high-diameter networks reported in Tables F.1–F.4, Appendix F.2. In the "Type" columns, the first letter indicates whether the network is undirected (U) or directed (D), while the second letter whether the network is unweighted (U) or weighted (W).

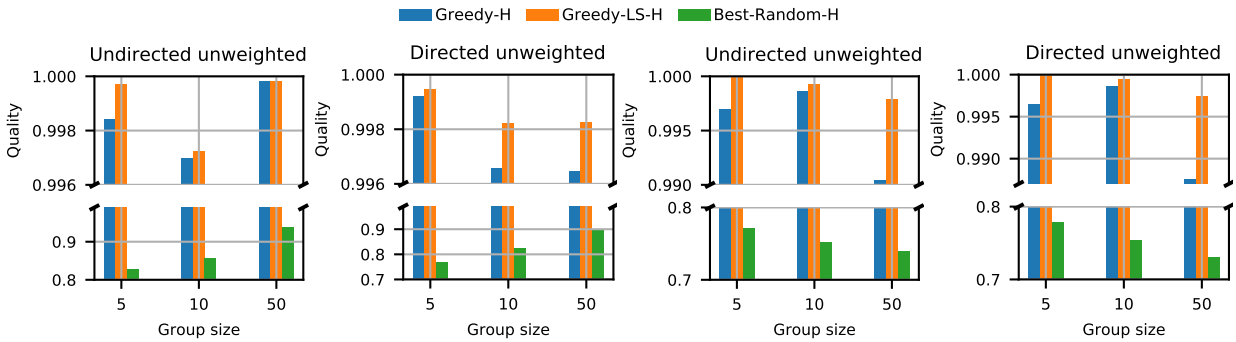
Instances have been downloaded from the public repositories KONECT [173], OpenStreetMap [83] (from which we build the car routing graph using RoutingKit [89]) and from the 9th DIMACS Implementation Challenge [87]. Small instances used for the experiments with the ILP solver are reported in Tables F.1 and F.2, while the rest of the experiments are conducted on the instances in Tables F.3 and F.4.

Because algorithms for group-closeness maximization only handle (strongly) connected graphs, we run them on the (strongly) connected components of the instances of our datasets. For high-diameter networks, we test mainly road networks because they are the most common type of networks in the aforementioned repositories. We are confident, however, that our local search algorithms are capable of handling other types of high-diameter networks as well without significant difference in performance. Because public repositories do not provide a reasonable amount of *weighted* complex networks, we omit these networks from our experiments.

7.5.3 Group-Harmonic Maximization

Comparison to Exact ILP Solutions. Figure 7.1a shows a comparison of the solution quality of our algorithms for group-harmonic maximization against to exact solutions on complex networks. We observe that random groups cover unweighted graphs reasonably well; hence, Best-Random-H already yields solutions of $> 70\%$ of the optimum. This peculiarity is amplified by the fact that the networks are rather small compared to k – they have at most 1,000 vertices. Indeed, the quality of Best-Random-H *increases* with k on complex networks, a behavior that no other algorithm shows. Still, Greedy-H yields substantially better solutions in all cases: its solutions are $> 99.5\%$ of the optimum for all group sizes. These solutions are further improved by Greedy-LS-H, which yields groups with at least 99.72% of the optimal quality.

[43] We use the extended variant of *Grow-Shrink* with $p = 0.75$. As discussed in Section 6.7.1, it achieves a reasonable time-quality trade-off.



(a) Complex networks (b) High-diameter networks

Figure 7.1: Quality relative to the optimum for group-harmonic maximization over the networks of Table F.1, Appendix F.2.

In high-diameter networks (Figure 7.1b), Best-Random-H is not a serious competitor. Its solutions are less than 80% the optimal quality. Indeed, due to the higher diameter, it is expected that a random group of vertices is less likely to be central. On the other hand, Greedy-H and Greedy-LS-C yield solution qualities from 98.76% and 99.75%, respectively. For $k = 5$ in particular, solutions returned by Greedy-LS-H have $> 99.99\%$ the quality of the optimal solution.

Concerning weighted high-diameter networks, the ILP solver runs out of time or memory on nearly all instances. Tentative results on the two remaining instances suggest that Greedy-H yields solutions that are almost optimal but, due to the small size of the dataset, we cannot conclude definitive results.

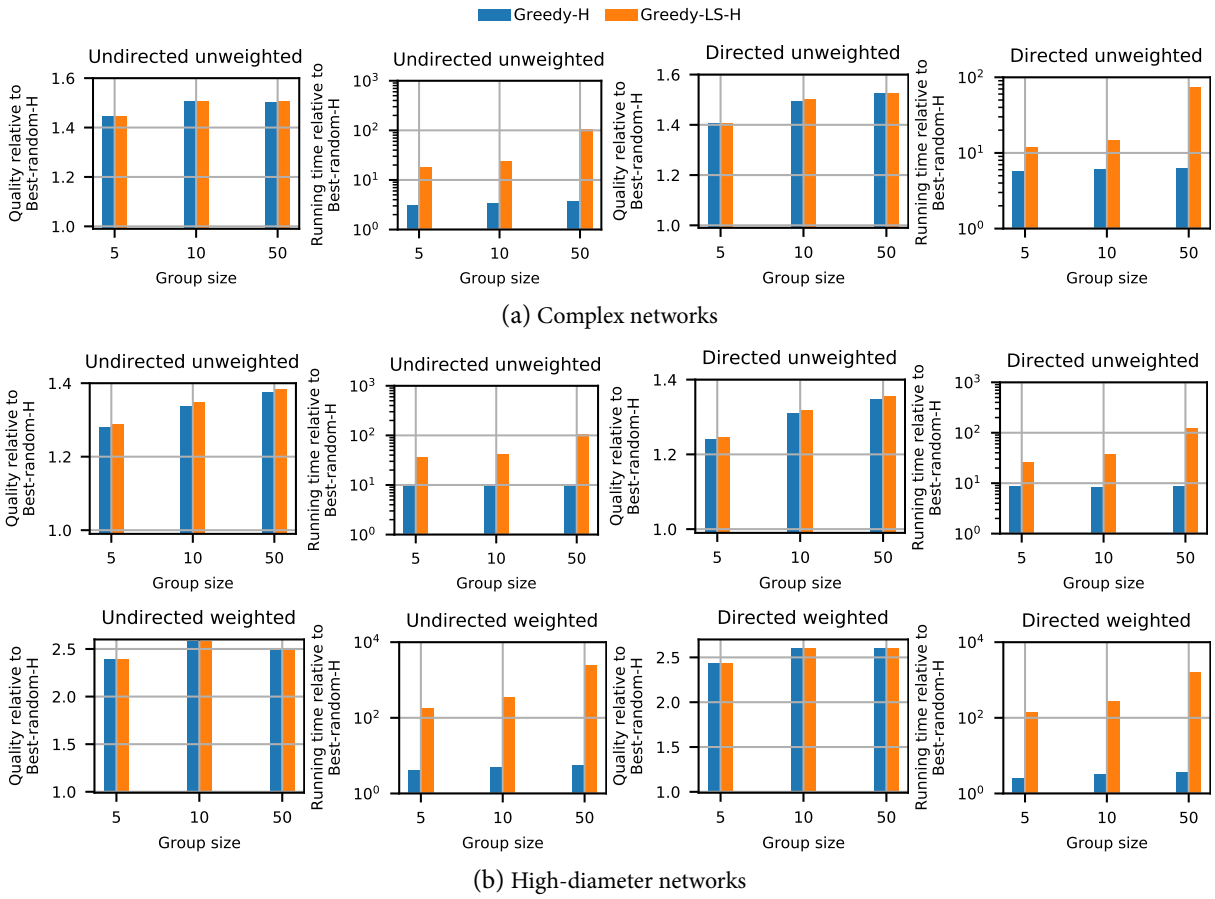
Quality and Running Time on Larger Instances. Figure 7.2 summarizes quality and running time results of Greedy-H and Greedy-LS-H (absolute running time are reported in Tables F.5 and F.6, Appendix F.3). Due to the size of these graphs, it is not feasible to compute an ILP solution. Thus, we use Best-Random-H as baseline. In unweighted complex networks (Figure 7.2a), Greedy-H finds solutions with quality (compared to Best-Random-H) from 1.407 (with $k = 5$) to 1.525 (with $k = 50$) in directed networks, and from 1.445 to 1.504 in undirected networks. Compared to Greedy-H, and Greedy-LS-H is not competitive: it improves the quality by at most 0.05% while being $5.7\times$ to $27.3\times$ slower.

Greedy-H achieves even better results in high-diameter networks: in weighted directed high-diameter networks (Figure 7.2b), Greedy-H's quality is 2.4 to 2.6 Best-Random-H's while being just $2.5\times$ to $3.6\times$ slower. Concerning Greedy-LS-H, it is even less competitive than in complex networks: it improves Greedy-H's quality by at most 0.01%, while being $54.9\times$ to $448.9\times$ slower. Results are more promising in high-diameter unweighted networks: here Greedy-LS-H improves Greedy-H's quality by 0.58% to 0.69% while being $3.2\times$ to $12.2\times$ slower.

7.5.4 Group-Closeness Maximization

Comparison to Exact ILP Solutions. Figure 7.3 summarizes the quality of our local search algorithms form group-closeness maximization and the competitors relative to the optimum.

Concerning unweighted complex networks (Figure 7.3a), in the directed case, for groups of size 5 Greedy-LS-C is the only algorithm achieving optimal solutions, while for the remaining group sizes it yields solutions with nearly the same quality as Greedy-C. In the undirected case, Greedy-LS-C and GS-LS-C achieve



(a) Complex networks

(b) High-diameter networks

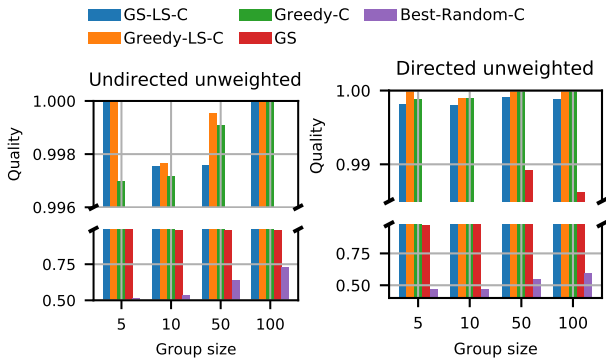
Figure 7.2: Quality and time w.r.t. Best-Random-H over the networks of Table F.3.

solutions with at least 99.77% and 99.76% the optimal quality, respectively. For $k = 5$ and $k = 100$ in particular, they achieve optimal solutions.

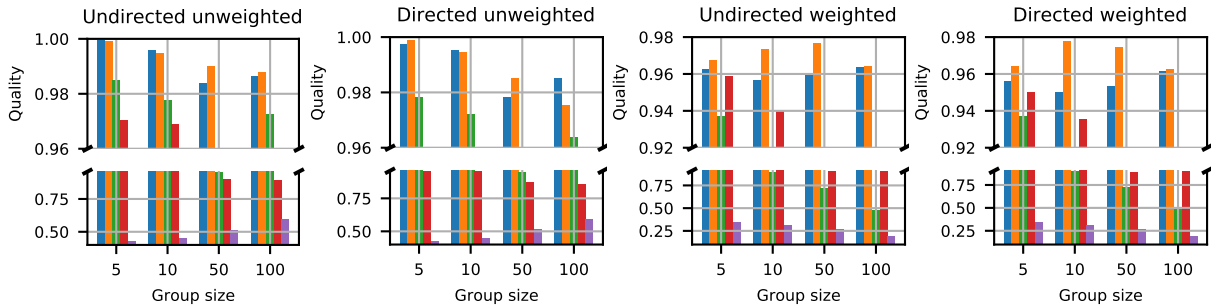
In high-diameter networks (Figure 7.3b), our local search algorithms always achieve better results than Greedy-C and GS. The best results are on unweighted graphs: here Greedy-LS-C and GS-LS-C yield solutions at least 98.66% and 98.50% away from optimality, respectively.

Interestingly, the quality of Greedy-LS-C is often higher that GS-LS-C, especially in complex networks and high-diameter weighted networks. We conjecture that local search has a narrower improvement margin on GS solutions compared to Greedy-C solutions since GS is based on local search as well.

Quality and Running Time on Larger Instances. In Figure 7.4 we report the quality and running time of GS-LS-C, Greedy-LS-C, Greedy-C, and GS compared to Best-Random-C (absolute running times are reported in Tables F.8 and F.9, Appendix F.4). In terms of quality, our local search algorithms always reach the best results in all our experiments: in directed complex networks (right of Figure 7.4a), GS-LS-C, Greedy-LS-C, and Greedy-C yield similar quality, while GS has consistently the lowest quality. On the other hand, quality can be traded for running time: GS is the fastest algorithm – even faster than Best-Random-C for small group sizes – Greedy-C is no average $16.4\times$ slower than Best-Random-C (average among all k), whereas GS-LS-C and Greedy-LS-C are respectively $28.33\times$ to $233.01\times$, and $22.99\times$ to $485.09\times$ slower than Best-Random-C. Interestingly, for small group sizes Greedy-LS-C is often faster than GS-LS-C, and



(a) Complex networks



(b) High-diameter networks

Figure 7.3: Quality relative to the optimum over the networks of Table F.2.

vice versa for larger groups. This is likely due to the difference between GS and Greedy-C solutions: Greedy-C aims to maximize the objective function regardless of the group size, while for GS the group size determines how many vertices are consecutively added and removed in a single iteration. Therefore, for larger groups, GS solutions need less swaps to reach a global optimum than Greedy-C solutions.

In high-diameter networks (Figure 7.4b), Greedy-LS-C often achieves the highest quality faster than GS-LS-C for all group sizes but 100.

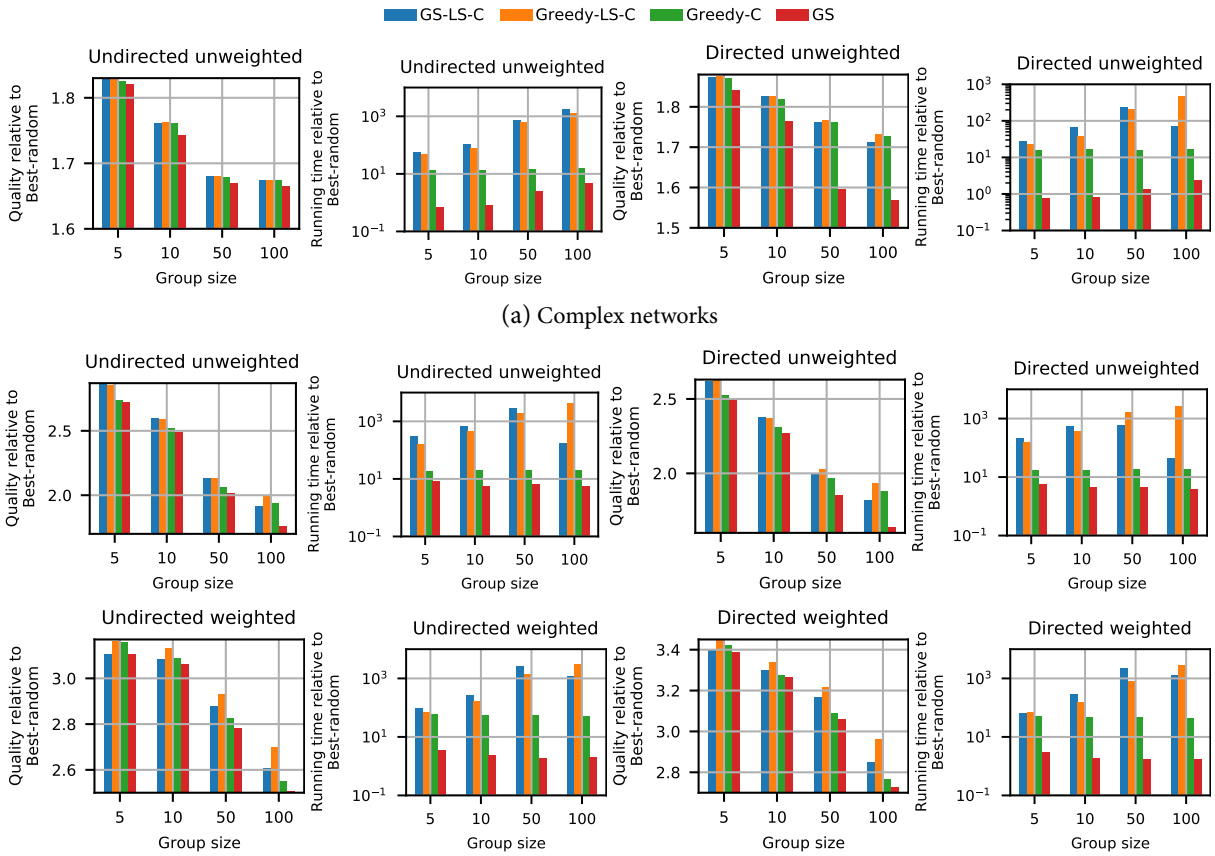
7.5.5 Parallel Scalability

Strong scaling plots for GS-LS-C, Greedy-LS-C, and Greedy-C are reported in Figure 7.5. On average, our local search algorithms scale better than Greedy-C on both complex and high-diameter networks. This is not surprising: local search needs to evaluate at least $k(n-k)$ swaps, which is a highly parallel operation, and often much more expensive than running Greedy-C.

On high-diameter networks in particular (Figure 7.5b), Greedy-C has a poor parallel scalability; we conjecture that, since closeness centrality distinguishes vertices in high-diameter networks better than in complex networks [227, Ch. 7], Greedy-C needs to evaluate only few vertices per iteration before finding the vertex with highest marginal gain. In that case, multiple cores do not speed this process up significantly.

7.6 Conclusions

This chapter investigated engineering aspects of approximating two group centrality maximization problems, namely, group-harmonic maximization and group-closeness maximization. We illustrated how to



(a) Complex networks

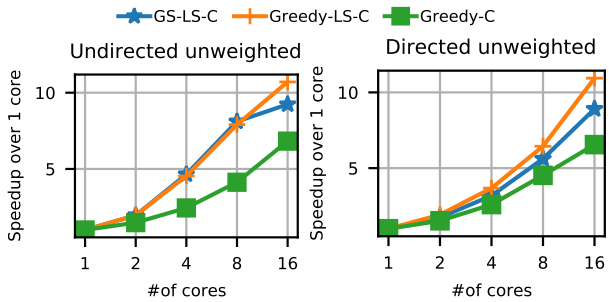
(b) High-diameter networks

Figure 7.4: Quality and time w.r.t. Best-Random-C over the networks of Table F.4.

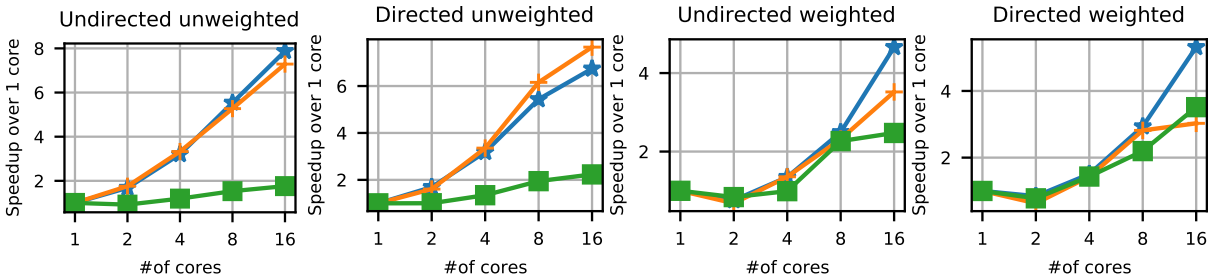
efficiently implement greedy and local search approximation algorithms and presented the results of a detailed experimental study. Our experiments suggest that the quality of both the greedy and the local search algorithms come very close to the optimum. This finding is consistent with the theoretical results illustrated in Ref. [12], which assess that, in most cases, these algorithms have good approximation guarantees. Interestingly, the two methods also perform well on directed instances for group-closeness maximization despite the hardness of approximation results which holds for this class of instances.

On the other hand, the quality guarantees come with a cost in running time. Concerning group-harmonic maximization, the results presented in Section 7.5.3 suggest that, despite the better approximation guarantees, our local search algorithm is not competitive compared to our greedy algorithm. In comparison to Greedy-H, Greedy-LS-H is $5.7\times$ to $448.9\times$ slower and yields solutions that improve Greedy-H solutions only by a thin margin. Hence, in a practical scenario, we recommend using the greedy algorithm.

For group-closeness maximization, we conclude that, unlike the group-harmonic case, our local search approximation algorithms are not dominated by existing greedy [38] and local search (see Chapter 6) heuristics. Despite being one to two orders of magnitude slower, results in Figure 7.4 convey that our GS-LS-C and Greedy-LS-C algorithms (especially the latter) improve Greedy-C and GS solutions by a solid margin. Therefore, among the available algorithms, Greedy-LS-C is the most amenable choice for applications dealing with networks of moderate size and where solution quality is of highest concern.



(a) Complex networks



(b) High-diameter networks

Figure 7.5: Parallel scalability of our algorithms for group-closeness maximization over the networks of Table F.4, Appendix F.2.

8 Algebraic Group Centrality Maximization for Large-Scale and Disconnected Graphs

8.1 Introduction

As described in Chapters 6 and 7, finding the most central group of vertices according to popular measures is an $\mathcal{NP}$-hard problem. Since real-world applications may not require exact results, approximation algorithms to maximize group centralities were introduced. Despite a nearly-linear running time, algorithms for group-betweenness [200, 288] and, to a lesser extent, group-closeness [38] and group-harmonic (see Chapter 7) fail to scale to large real-world instances due to high constant overheads.

Moreover, a limitation of popular group centrality measures is that they are based on shortest paths while some applications may be interested in all the paths between two vertices (see Section 2.3.2). This motivated the introduction of electrical single-vertex centrality measures. However, limited efforts were devoted to extend these measures to groups of vertices. Only recently, Li et al. [183] proposed current flow group closeness centrality, i.e., the extension of electrical closeness to sets of vertices. As with electrical closeness, this new group centrality measure – without proper adjustments – only handles connected graphs, leaving the problem of electrical group centrality measures for disconnected graphs open.

Contribution. In this chapter, we overcome the two aforementioned limitations by introducing two new group centrality measures: GED-Walk for *group exponentially decaying walk* and group forest closeness.

Inspired by Katz centrality, GED-Walk takes into account walks of any length and considers shorter ones as more important. We show our measure to be monotone and submodular and we develop efficient nearly-linear parallel algorithms to compute the GED-Walk score of a given group and to efficiently approximate the group of k vertices with highest GED-Walk centrality. Our greedy maximization algorithm keeps lower and upper bounds of the marginal gain of every vertex to the GED-Walk score and iteratively adds to the (initially empty) group the vertex with highest marginal gain until the desired size is reached.

Group forest closeness, in turn, is the extension to groups of vertices of forest closeness and, analogously, it handles disconnected graphs out of the box. To the best of our knowledge, we are the first to address the group case for this centrality measure. It turns out that group forest closeness centrality maximization is $\mathcal{NP}$-hard and we adapt the greedy approximation algorithm by Li et al. [183] to this problem.

Experiments show that our greedy algorithm for GED-Walk is two orders of magnitude faster than the state of the art for group-betweenness approximation [200]. Compared to the greedy group-closeness heuristic [38] and the greedy approximation algorithm for group-harmonic (see Chapter 7), we achieve a speedup of one to two orders of magnitude for groups with up to 100 vertices. Further, for GED-Walk approximation, our experiments indicate a running time that is in practice linear w.r.t. the input graph size

– despite a higher worst-case time complexity. We show potential applications for GED-Walk: (i) choosing a training set with high GED-Walk score leads to improved performance for semi-supervised vertex classification and (ii) features derived based on GED-Walk improve graph classification performance.

We repeat the semi-supervised vertex classification experiment on disconnected graphs in order to demonstrate the practical usefulness of group forest closeness as well. Results demonstrate that our new measure improves upon existing measures as well in the case of disconnected graphs.

Bibliographic Notes. The contributions presented in this chapter about GED-Walk were published in the Proceedings of the *Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX 2020)* in collaboration with Alexander van der Grinten, Aleksandar Bojchevski, Daniel Zügner, Stephan Günnemann, and Henning Meyerhenke; the ones concerning group forest closeness were published in the Proceedings of the *Twenty-First SIAM International Conference on Data Mining (SDM 2021)* in collaboration with Alexander van der Grinten, Maria Predari, and Henning Meyerhenke. About the former topic, the development of the new measure GED-Walk and the implementation of the algorithms for GED-Walk computation and maximization were a collaborative effort of Alexander van der Grinten and myself. Further, I carried out the experiments of Sections 8.5.1–8.5.5. Regarding group forest closeness, my contributions involve the implementation of the algorithm and carrying out the experiments. Contributions not mentioned above are joint work with my coauthors. Proofs to which I did not contribute are omitted and can be found in the original papers [13, 133].

Related Work. An overview of algorithms for maximizing established group centrality measures is provided in Section 6.1. The idea of developing alternative group centrality measures is not new. In fact, Ishakian et al. [146] define single-vertex centrality measures based on a generic concept of *path* and generalize them to sets of vertices. In contrast to GED-Walk, however, those measures are defined for directed acyclic graphs only. Puzis et al. [246] introduce *path betweenness centrality*, a group centrality measure that counts the fraction of shortest paths that cross *all* vertices within the group. Their proposed algorithms for finding groups with high path-betweenness are quadratic in the best case (both in time and memory); hence, they cannot scale to large networks – indeed, the largest graphs considered in their paper have only 500 vertices. Fushimi et al. [112] define a new measure called *connectedness centrality*, targeting the specific problem of identifying the appropriate locations where to place evacuation facilities on road networks. For this reason, their measure assumes the existence of edge failure probabilities and thus it is not applicable to general graphs. Additionally, due to expensive Monte Carlo simulations, computing their measure requires many hours on graphs with 100,000 edges, while GED-Walk handles graphs with hundreds of millions of edges in minutes.

Finally, concerning electrical group centrality measures, Li et al. [183] defined current flow group closeness centrality, i.e., electrical closeness extended to set of vertices, and proposed two greedy approximation algorithms to maximize this measure: (i) a deterministic algorithm with $\mathcal{O}(n^3)$ running time and quadratic memory requirements (it performs matrix inversions) and (ii) a faster randomized algorithm with $\tilde{\mathcal{O}}(km\varepsilon^{-2})$ running time ($\tilde{\mathcal{O}}$ hides a polylogarithmic factor) that exploits JLT in combination with nearly linear time solvers for Laplacian and symmetric, diagonally dominant, M-matrices.

8.2 Preliminaries

8.2.1 GED-Walk Centrality

For GED-Walk, we deal with unweighted (possibly directed) graphs. To avoid complexity-theoretic barriers typical of shortest-path based centrality measures,[44] we derive our new measure from Katz centrality. As described in Section 2.4, Katz centrality is another popular centrality measure which can be computed more efficiently compared to measures based on shortest paths. Let $a_i(u) := \sum_{v \in V} \mathbf{A}^i[u, v]$ be the number of walks of length i that start at a vertex $u \in V$, where $\mathbf{A}$ is the adjacency matrix of the graph.[45] For a small enough attenuation $\alpha > 0$, the Katz centrality of a vertex u is defined as:

$$c_{\mathrm{K}}(u) := \sum_{i=1}^{\infty} \alpha^i \sum_{v \in V} \mathbf{A}^i[u, v] = \sum_{i=1}^{\infty} \alpha^i a_i(u), \tag{8.1}$$

Note that $c_{\mathrm{K}}(\cdot)$ does not only consider *shortest paths*, but *walks* of any length – with shorter walks having higher importance due to the exponential decrease of α^i. Since it does not rely on APSP, $c_{\mathrm{K}}(\cdot)$ can be computed faster than betweenness centrality. For example, following from the algebraic formulation of Katz centrality (see Eq. (2.11), Section 2.4), iterative linear solvers – such as conjugate gradient – can be used to compute $c_{\mathrm{K}}(\cdot)$ for all vertices at the same time. Furthermore, efficient nearly-linear time algorithms for Katz centrality approximation and ranking exist [134]. In fact, the algorithms that we present in this chapter are based on those algorithms for Katz centrality.

In contrast to $c_b(\cdot)$ and $c_c(\cdot)$, however, replacing the single vertex u in Eq. (8.1) by a set leads to a rather uninteresting measure: as $a_i(u)$ and $a_i(v)$ count distinct walks for $u \neq v$, such a "group Katz" would just be equal to the sum of the individual Katz scores. Hence, a group with maximal Katz centrality would just consist of the top-k vertices with highest $c_{\mathrm{K}}(\cdot)$. Indeed, this contrasts with the intuition of group centrality: parts of a graph that are *covered* by one vertex in the group does *not* need to be covered by a second group vertex.

Fortunately, we can construct a centrality measure satisfying this intuition by replacing $c_{\mathrm{K}}(\cdot)$ by a natural variant of it: instead of considering all walks that *start* at a given vertex u, we consider all that *cross* u. This leads to our definition of ED-Walk:

Definition 8.2.1 (ED-Walk)**.** Let $\phi_i(S)$ be the number of i-walks – i.e., walks of length i – that contain at least one vertex in $S \subseteq V$. The ED-Walk (for *Exponentially Decaying* walk) centrality for $u \in V$ is:

$$c_{\mathrm{ED}}(u) := \sum_{i=i}^{\infty} \alpha^i \phi_i(\{u\}),$$

where $\alpha > 0$ is a parameter of the centrality.

[44] Recall from Chapter 3 (Section 3.1) that the existence of a sub-cubic algorithm for betweenness (on general graphs) would also imply the existence of a faster algorithm for APSP [1]. The vertex with highest closeness, in turn, cannot be computed in sub-quadratic time unless SETH [145] fails [53] (see Chapter 4, Section 4.2.2).

[45] Katz centrality is sometimes alternatively defined by taking into account walks *ending* at a given vertex; from an algorithmic perspective, however, this difference is irrelevant.

As mentioned in Section 2.4, in order for the series to converge, α needs to be chosen small enough – an upper limit for α is provided in Section 8.2.2. As claimed above, ED-Walk naturally generalizes to a group measure:

Definition 8.2.2. (GED-Walk) The GED-Walk centrality of a group $S \subseteq V$ of vertices is given by:

$$g_{\text{ED}}(S) := \sum_{i=1}^{\infty} \alpha^i \phi_i(S). \tag{8.2}$$

In the context of group centrality, we are interested in two problems:

- the problem of computing the centrality of a *given* group $S \subseteq V$;
- the problem of *finding* a group that maximizes the group centrality measure.

We deal in particular with the latter optimization problem:

GED-Walk Maximization	
Input:	Graph $G = (V, E)$, integer $1 \leq k \leq n$.
Find:	Set $S^\star \subset V$ with $\lvert S \rvert = k$ s.t. $g_{\text{ED}}(S^\star)$ is maximum.

Note again that, in contrast to Katz centrality, the GED-Walk maximization problem is different from the top-k ED-Walk centrality problem.

8.2.2 Mathematical Properties of GED-Walk

Since GED-Walk is based on similar concepts as Katz centrality, it is not surprising that the two measures have analogous mathematical properties. Indeed, the convergence properties of $c_{\text{K}}(u)$ and GED-Walk are identical:

Proposition 8.2.1 ([13])**.** The following holds:

1. $g_{\text{ED}}(V) = \sum_{u \in V} c_{\text{K}}(u)$.
2. Let σ_{max} be the largest singular value of the adjacency matrix of G. If $\alpha < 1/\sigma_{\text{max}}$, then $g_{\text{ED}}(S)$ is finite for all $S \subseteq V$.

Note that the quality of the first claim of Proposition 8.2.1 does not hold for groups $S \subset V$.

We have now established that GED-Walk is a well-defined measure. In the following, we prove further basic properties of the g_{ED} function:

Proposition 8.2.2. GED-Walk is both non-decreasing and submodular as a set function.

Proof. Monotonicity is obvious from Eq. (8.2): as the input set S becomes larger, $g_{\text{ED}}(S)$ cannot decrease. To see that GED-Walk is also submodular, we observe that the marginal gain $g_{\text{ED}}(S \cup \{u\}) - g_{\text{ED}}(S)$ is exactly the sum of the number of walks that contain u but no vertex in S, with each walk weighted by a power of α. As such, the marginal gain can only decrease if S is replaced by a superset $T \supseteq S$. □

Given an algorithm to compute g_{ED}, Proposition 8.2.2 would immediately allow us to construct a greedy algorithm to approximate GED-Walk maximization – as a consequence Proposition 2.5.1. However, as we show in Section 8.3.2, we can do better than naively applying the greedy approach.

As with other group centrality measures, maximizing GED-Walk turns out to be an $\mathcal{NP}$-hard problem [13, Theorem 2.1].

8.3 Algorithms for GED-Walk

8.3.1 Computing GED-Walk Centrality

In this section, we address the problem of computing $g_{\text{ED}}(S)$ of a given group S. As we are not aware of an obvious translation of our $g_{\text{ED}}(S)$ definition to a closed form expression, we employ an approximation algorithm that computes the infinite series of $g_{\text{ED}}(S)$ up to an arbitrarily small additive constant error $\varepsilon > 0$. To see how our algorithm works, let $\ell \in \mathbb{N}$ be a positive integer. We split the $g_{\text{ED}}(S)$ series into the first ℓ terms and a tail of infinite terms. This allows us to break the problem of approximating $g_{\text{ED}}(S)$ into two sub-problems: (i) computing (exactly) theh ℓ-th partial sum $g_{\text{ED}}^{\leq \ell}(S) := \sum_{i=1}^{\ell} \alpha^i \phi_i(S)$ and the problem of finding an upper bound on the tail $g_{\text{ED}}^{>\ell}(S) := \sum_{i=\ell+1}^{\infty} \alpha^i \phi_i(S)$. In particular, we are interested in an upper bound that converges to zero as ℓ approaches ∞. Given such an upper bound, our approximation algorithm chooses ℓ so that this bound is below the given error threshold ε and returns $g_{\text{ED}}^{\leq \ell}(S)$ once such ℓ is found.

Computing $\phi_i(S)$. To compute $g_{\text{ED}}^{\leq \ell}(S)$ is obviously enough to compute $\phi_i(S)$ for $i \in \{1, \dots, \ell\}$. The key insight to compute $\phi_i(S)$ efficiently is that, similarly to Katz centrality, it can be expressed as a series of recurrences: let $\phi_i^{\text{hit}}(u, S)$ be the number of i-walks ending in u and containing at least one vertex from S. Clearly, the number of walks that contribute to ϕ_i^{hit} partition the walks that contribute to ϕ_i in the sense that:

$$\phi_i(S) = \sum_{u \in V} \phi_i^{\text{hit}}(u, S). \tag{8.3}$$

On the other hand, ϕ_i^{hit} can be expressed in terms of $\phi_i^{\text{miss}}(u, S)$, i.e., the number of i-walks that end in u but do not contain any vertex from S:

$$\phi_i^{\text{hit}}(v, S) = \begin{cases} \sum_{(u,v) \in E} \phi_{i-1}^{\text{hit}}(u, S) + \phi_{i-1}^{\text{miss}}(u, S) & v \in S \\ \sum_{(u,v) \in E} \phi_{i-1}^{\text{hit}}(u, S) & v \notin S. \end{cases} \tag{8.4}$$

This derives from the fact that any i-walk is the concatenation of a $(i-1)$-walk and a single edge. Eq. (8.3) considers $(i-1)$-walks which do not contain a vertex in S only if the last concatenated edge ends in S. Similarly, $\phi_i^{\text{miss}}(u, S)$ can be expressed as the recurrence:

$$\phi_i^{\text{miss}}(v, S) = \begin{cases} 0 & v \in S \\ \sum_{(u,v) \in E} \phi_{i-1}^{\text{miss}}(u, S) & v \notin S. \end{cases} \tag{8.5}$$

In the $v \in S$ case, all walks ending in v have a vertex in S, so no walks contribute to ϕ_i^{miss}. Notice that the base cases of ϕ_1^{hit} and ϕ_1^{miss} can easily be computed directly from G. Collectively, this gives us an

Algorithm 17 GED-Walk Computation

Input: Graph $G = (V, E)$, parameters α, ε, and group $S \subseteq V$.
Output: $g_{\text{ED}}(S) \pm \varepsilon$.

```
1: c ← 0
2: i ← 1
3: while true do
4:     compute φ_i(S)
5:     c ← c + α^i φ_i(S)
6:     compute bound B_ℓ s.t. g_ED^{>ℓ}(S) ≤ B_ℓ
7:     if B_ℓ < ε then
8:         return c
9:     i ← i + 1
```

$\mathcal{O}(\ell(n+m))$-time algorithm for computing the GED-Walk score of a given group. Furthermore, note that the computation of ϕ_i^{hit} and ϕ_i^{miss} can be trivially done for multiple vertices in parallel.

Bounding $g_{\text{ED}}^{>\ell}(S)$. To complete our algorithm, we need to define a sequence of upper bounds on $g_{\text{ED}}^{>\ell}(S)$ that converges to zero for $\ell \to \infty$. Using Proposition 8.2.1, we can lift tail bounds of Katz centrality to GED-Walk. In particular, from the first claim of this proposition leads to:

$$g_{\text{ED}}^{>\ell}(S) \leq g_{\text{ED}}^{>\ell}(V) = \sum_{u \in V} \sum_{i=\ell+i}^{\infty} \alpha^i a_i(u). \tag{8.6}$$

Bounds for the latter term can be found in the literature on Katz centrality. For example, let $\deg_{\max}$ be the maximum degree of any vertex in the graph; for $\alpha < 1/\deg_{\max}$ it was shown that [134]:

$$\sum_{i=\ell+1} a_i(u) \leq \frac{\deg_{\max}}{1 - \alpha \deg_{\max}} \alpha^{\ell+1} a_\ell(u). \tag{8.7}$$

Combining Eq. (8.6) and Eq. (8.7) leads to the following bound for GED-Walk:

$$g_{\text{ED}}^{>\ell}(V) \leq \alpha^{\ell+1} \frac{\deg_{\max}}{1 - \alpha \deg_{\max}} \sum_{u \in V} a_\ell(u). \tag{8.8}$$

We call this bound the *combinatorial bound* – due to the nature of the proof in [134]. We generalize this statement to arbitrary α (i.e., $\alpha < 1/\sigma_{\max}$), at the cost of a factor $\sqrt{n}$.

Lemma 8.3.1. It holds that:

$$g_{\text{ED}}^{>\ell}(V) \leq \sqrt{n} \alpha^{\ell+1} \frac{\sigma_{\max}}{1 - \alpha \sigma_{\max}} \sum_{u \in V} a_\ell(u). \tag{8.9}$$

We call the bound of Eq. (8.9) the *spectral bound*.

Complexity Analysis. Algorithm 17 shows the pseudocode of the algorithm to compute $g_{\text{ED}}(S)$ of a given $S \subseteq V$. Line 4 computes $\phi_i(S)$ using Eq. (8.3) and the recurrences for ϕ_i^{hit} and ϕ_i^{miss} from Eqs. (8.4) and (8.5). This can be done in $\mathcal{O}(n+m)$ time by storing ϕ_{i-1}^{hit} and ϕ_{i-1}^{miss}, which requires an additional $\mathcal{O}(n)$

memory. Line 6 computes an upper bound on $g_{\text{ED}}^{>\ell}(S)$, which can be either the combinatorial or the spectral bound.

Lemma 8.3.2 ([13])**.** The GED-Walk score of a given group can be approximated up to an additive error of $\varepsilon > 0$ in $\mathcal{O}\left(\frac{\log(n/\varepsilon)}{\log 1/(\alpha\sigma_{\max})}\right)$ iterations.

Corollary 8.3.1. The GED-Walk score of a given group can be approximated up to an additive error of $\varepsilon > 0$ in $\mathcal{O}\left(\frac{\log(n/\varepsilon)}{\log 1/(\alpha\sigma_{\max})}(n+m)\right)$ time.

8.3.2 Maximizing GED-Walk Centrality

Let $g_{\text{ED}}(S, u)$ be the marginal gain of a vertex u w.r.t. a group S, i.e., $g_{\text{ED}}(S, u) := g_{\text{ED}}(S \cup \{u\}) - g_{\text{ED}}(S)$. As observed in Section 8.2.2, the properties given by Proposition 8.2.2 imply that a greedy algorithm that successively picks a vertex $u \in V$ with highest marginal gain $g_{\text{ED}}(S, u)$ yields a $(1 - 1/e)$-approximation for the GED-Walk maximization problem. Note, however, that we do not have an algorithm to compute $g_{\text{ED}}(S, u)$ exactly. While we could use the approximation algorithm of Section 8.3.1 to feed approximate values of $g_{\text{ED}}(S, u)$ into a greedy algorithm, this procedure would involve more work than necessary: indeed, the greedy algorithm does not need the value of $g_{\text{ED}}(S, u)$, but to ascertain that there is no $u' \in V$ with $g_{\text{ED}}(S, u') > g_{\text{ED}}(S, u)$.

This consideration is similar to a top-1 centrality problem. Hence, we adapt the ideas of the top-k Katz centrality algorithm that was introduced in [134] to the case of $g_{\text{ED}}(S, u)$.[46] Applied to the marginal gain of GED-Walk, the main ingredients of this algorithm are families $L_\ell(S, u)$ and $U_\ell(S, u)$ of lower and upper bounds on $g_{\text{ED}}(S, u)$ satisfying the following definition:

Definition 8.3.1. We say that families of functions $L_\ell(S, u)$ and $U_\ell(S, u)$ are suitable bounds on the marginal gain $g_{\text{ED}}(S, u)$ if the following conditions are all satisfied:

1. $L_\ell(S, u) \leq g_{\text{ED}}(S, u) \leq U_\ell(S, u)$;
2. $\lim_{\ell\to\infty} L_\ell(S, u) = \lim_{\ell\to\infty} U_\ell(S, u) = g_{\text{ED}}(S, u)$;
3. $L_\ell(S, u)$ and $U_\ell(S, u)$ are non-increasing in S.

In addition to L_ℓ and U_ℓ, we need the following definition:

Definition 8.3.2 (ε-separation)**.** Let $\varepsilon > 0$ and fix some $\ell \in \mathbb{N}$. Let $u, u' \in V$ be two vertices. If $L_\ell(S, u) \geq U_\ell(S, u') - \varepsilon$, we say that u is ε-separated from u'.

It is easy to see that, if there is a vertex $u \in V$ that is ε-separated from all other $u' \in V \setminus \{u\}$, then either u is the vertex with highest marginal gain $g_{\text{ED}}(S, u)$ or the highest marginal gain is at most $g_{\text{ED}}(S, u) + \varepsilon$ – this follows directly from the first property of Definition 8.3.1. Note that the introduction of ε is required here to guarantee that the separation can be achieved for finite ℓ even if u and u' have truly identical marginal gains. Furthermore, while the first condition of Definition 8.3.1 is required for ε-separation to work, the second and third conditions are required for the correctness of Algorithm 18 that we construct in the following.

[46] Note that the vertex that maximizes $g_{\text{ED}}(S, u) = g_{\text{ED}}(S \cup \{u\}) - g_{\text{ED}}(S)$ is exactly the vertex that maximizes $g_{\text{ED}}(S \cup \{u\})$. Algorithmically, it is simpler to deal with $g_{\text{ED}}(S \cup \{u\})$ because it allows us to construct a lazy greedy algorithm.

Construction of $L_\ell(S,u)$ and $U_\ell(S,u)$. In order to implement the strategy we described in this section, we need suitable families $L_\ell(S,u)$ and $U_\ell(S,u)$ of bounds. Luckily, we can re-use some of the bounds we developed in Section 8.3.1. More precisely, it holds that:

$$\begin{aligned} g_{\text{ED}}(S,u) &= g_{\text{ED}}^{\leq\ell}(S\cup\{u\}) - g_{\text{ED}}^{\leq\ell}(S) + g_{\text{ED}}^{>\ell}(S\cup\{u\}) - g_{\text{ED}}^{>\ell}(S) \\ &\geq g_{\text{ED}}^{\leq\ell}(S\cup\{u\}) - g_{\text{ED}}^{\leq\ell}(S). \end{aligned}$$

In the above calculation, $g_{\text{ED}}^{>\ell}(S\cup\{u\}) - g_{\text{ED}}^{>\ell}(S) \geq 0$ because ϕ_i is non-decreasing as a set function. Hence, $L_\ell(S,x) := g_{\text{ED}}^{\leq\ell}(S\cup\{u\}) - g_{\text{ED}}^{\leq\ell}(S)$ yields a family of lower bounds on the marginal gain of g_{ED}. On the other hand:

$$\begin{aligned} g_{\text{ED}}(S,u) &= g_{\text{ED}}^{\leq\ell}(S\cup\{u\}) - g_{\text{ED}}^{\leq\ell}(S) + g_{\text{ED}}^{>\ell}(S\cup\{u\}) - g_{\text{ED}}^{>\ell}(S) \\ &\leq g_{\text{ED}}(S\cup\{u\}) - g_{\text{ED}}^{\leq\ell}(S) + g_{\text{ED}}^{>\ell}(S\cup\{u\}) \end{aligned}$$

Thus, a family of upper bounds on the marginal gain $g_{\text{ED}}(S,u)$ is given by $U_\ell(S,u) := g_{\text{ED}}^{\leq\ell}(S\cup\{u\}) - g_{\text{ED}}^{\leq\ell}(S) + B_\ell(V)$, where B_ℓ denotes either the combinatorial or the spectral bound developed in Section 8.3.1.

Lemma 8.3.3 ([13]). Let $g_{\text{ED}}^{\leq\ell}(S,u) := g_{\text{ED}}^{\leq\ell}(S\cup\{u\}) - g_{\text{ED}}^{\leq\ell}(S)$. The two families

$$\begin{aligned} L_\ell(S,u) &:= g_{\text{ED}}^{\leq\ell}(S,u) \\ U_\ell(S,u) &:= g_{\text{ED}}^{\leq\ell}(S,u) + B_\ell(V) \end{aligned}$$

form suitable families of bounds on the marginal gain of g_{ED}.

Lazy-Greedy Algorithm. Taking advantage of the bounds defined in Lemma 8.3.3, we can construct a greedy approximation algorithm for GED-Walk maximization. To reduce the number of evaluations of the objective function – without impacting the solution quality or the (worst-case) time complexity – we use a lazy strategy inspired by the well-known lazy greedy algorithm for submodular approximation [217]. However, in contrast to the standard lazy greedy algorithm, we do not evaluate our submodular objective function g_{ED} directly; instead, we apply Definition 8.3.2 to find the vertex with highest marginal gain. To this end, we need to find the vertex $x \in V \setminus S$ that maximizes $L_\ell(S,x)$ and the vertex $y \in V \setminus (S\cup\{x\})$ that maximizes $U_\ell(S,y)$. Hence, we rank all vertices according to priorities $L(u) \geq L_\ell(S,u)$ and $U(u) \geq U_\ell(S,u)$ and lazily compute the true values $L_\ell(S,u)$ and $U_\ell(S,u)$ until x and y are identified.

Algorithm 18 depicts the pseudocode of this algorithm. Procedure lazyUpdate (Line 5) lazily updates the priorities $L(u)$ and $U(u)$ until the exact values of $L_\ell(S,u)$ and $U_\ell(S,u)$ are known for the vertex on top of the given priority queue. In Line 23, the resulting vertices x and y are checked for ε/k-separation. It is necessary to achieve an (ε/k)-separation (and not only an ε-separation) in order to guarantee that the total absolute error is below ε even after k iterations. If separation fails, ℓ is increased (Line 28) and we reset $L(u)$ and $U(u)$ for all $u \in V$ (Line 15); otherwise, x (i.e., the vertex with highest marginal gain) is added to S (Line 27).

Algorithm 18 Lazy greedy algorithm for GED-Walk maximization.

Input: Graph $G = (V, E)$, parameter α, and group size k.
Output: Group $S \subseteq V$ with $g_{\text{ED}}(S) \geq (1 - 1/e)$ times the optimum score.

```
S ← ∅, ℓ ← 1
L(u) ← ∞, U(u) ← ∞ for all u ∈ V
PQ_L ← empty max-priority queue with keys L(u) and values u ∈ V
PQ_U ← empty max-priority queue with keys U(u) and values u ∈ V
procedure LazyUpdate(PQ)
    repeat
        u ← PQ.max()
        L(u) ← L_ℓ(S, u)
        U(u) ← U_ℓ(S, u)
        PQ_L.update(u), PQ_U.update(u)
    until u = PQ.max()
    return u
loop
    for each u ∈ V do
        L(u) ← ∞, U(u) ← ∞
        PQ_L.push(u), PQ_U.push(u)
    loop
        if |S| = k then
            return S
        x ← LazyUpdate(PQ_L)
        PQ_L.remove(x), PQ_U.remove(x)
        y ← LazyUpdate(PQ_U)
        if L(x) ≤ U(y) − ε/k then                    ▷ Check if x is (ε/k)-separated from y
            /* x and y are not (ε/k)-separated, increase ℓ */
            PQ_L.push(x), PQ_U.push(x)
            break
        S ← S ∪ {x}
    ℓ ← 2ℓ                                           ▷ Increase ℓ using geometric progression
```

Lemma 8.3.4 ([13])**.** If L_ℓ and U_ℓ are suitable families of bounds according to Definition 8.3.1, then Algorithm 18 computes a group S such that $g_{\text{ED}}(S) \geq (1-1/e)g_{\text{ED}}(S^\star) - \varepsilon$, where $S^\star$ is the group with highest GED-Walk score.

Initialization of $L(u)$ and $U(u)$. To accelerate the algorithm in practice, it is crucial to choose the bounds $L(u)$ and $U(u)$ appropriately at each reset: if we reset $L(u) = U(u) = \infty$ as in Algorithm 18, the algorithm has to evaluate $L_\ell(S, u)$ and $U_\ell(S, u)$ for all $u \in V \setminus S$ whenever ℓ increases. To avoid such an inefficiency, we use $\phi_i^{\text{miss}}(x, S)$ to provide a better initialization for our bounds. Let $\psi_i^{\text{miss}}(u, S)$ be the value of $\phi_i^{\text{miss}}(u, S)$ in the reverse graph of G, i.e., $\psi_i^{\text{miss}}(u, S)$ is the number of i-walks that *start* at u but do not contain any vertex of S. Our initialization strategy is summarized in the following lemma:

Lemma 8.3.5 ([13])**.** Let

$$P_i(S, u) := \sum_{j=0}^{i} \phi_{i-j}^{\text{miss}}(u, S)\psi_j^{\text{miss}}(S, u).$$

The following holds:

1. $P_i(S, u) \geq \phi_i(S \cup \{u\}) - \phi_i(S)$;

2. Let B_ℓ denote the bound from the construction of U_ℓ. P_i yields the following bounds on L_ℓ and U_ℓ:

$$L_\ell(S,u) \leq \sum_{i=1}^{\ell} \alpha^i P_u(S,u)$$
$$U_\ell(S,u) \leq \sum_{i=i}^{\ell} \alpha^i P_i(S,u) + B_\ell(V).$$

Thus, instead of initializing $L(u) = U(u) = \infty$, we compute $P_i(S,u)$ and use the right-hand sides of the second statement of Lemma 8.3.5 as initial values for $L(u)$ and $U(u)$. Note that using the recurrence for ϕ_i^{miss} from Section 8.3.1 (and an analogous recurrence for ψ_i^{miss}), $P_\ell(S,u)$ can be computed in $\mathcal{O}(\ell(n+m))$ time simultaneously for all $u \in V$.

Complexity of the Lazy Algorithm. Assume that Algorithm 18 uses the spectral bound from Lemma 8.3.1. The worst-case value to achieve (ε/k)-separation is given by Lemma 8.3.2, namely $\mathcal{O}\left(\frac{\log(kn/\varepsilon)}{\log 1/(\alpha\sigma_{\max})}\right)$. Note that the algorithm does some iterations that do not achieve (ε/k)-separation. We calculate the cost of each iteration in terms of the number of evaluations of ϕ_i. In this measure, the cost of each iteration is ℓ. As ℓ is doubled on unsuccessful (ε/k)-separations, the cost of all unsuccessful attempts to achieve (ε/k)-separation is always smaller than the cost of a successful attempt to achieve (ε/k)-separation. Hence, the worst-case running time of Algorithm 18 is $\mathcal{O}\left(kn\frac{\log(kn/\varepsilon)}{\log 1/(\alpha\sigma_{\max})}(n+m)\right)$: in each of the k successful iterations, it can be necessary to extract $\mathcal{O}(n)$ elements from the priority queues and evaluate $L_\ell(u)$ and $U_\ell(u)$ for each of them.

We expect, however, that much fewer evaluations of $L_\ell(u)$ and $U_\ell(u)$ will be required in practice than in the worst case: in fact, we expect almost all vertices to have low marginal gains (i.e., only a few walks cross those vertices) and g_{ED} will never be evaluated on those vertices. We study the practical performance of Algorithm 18 in Section 8.5.

8.3.3 Dealing with Large k

For large groups, our algorithm for GED-Walk maximization needs to increase the walk length ℓ (see Algorithm 18, Line 28). Thus, evaluating $L_\ell(S,u)$ and $U_\ell(S,u)$ becomes more expensive as k grows. To overcome this issue, we follow the approach of [218]. This *stochastic algorithm* is similar to the lazy algorithm. However, instead of considering all vertices in $V \setminus S$ when maximizing the marginal gain – or when trying to (ε/k)-separate the vertex with maximal marginal gain from the others – stochastic greedy only considers a subset of all vertices. Specifically, when adding a new vertex to the group, the algorithm samples $\frac{n}{k}\log\frac{1}{\eta}$ vertices at random, where η is a parameter of the algorithm. Marginal gain maximization or separation only consider the sampled vertices and ignore all others – i.e., only the sampled vertices are inserted in the priority queues. Note that if (ε/k)-separation fails, the sampled vertices are not discarded; the algorithm does exactly one round of sampling for each vertex added to the group. This is necessary as the probability that we find a vertex with high marginal gain is not independent from the probability that the vertex can be separated against the other vertices of the sample.

Complexity of the Stochastic Algorithm. In contrast to the lazy greedy algorithm, stochastic greedy evaluates g_{ED} at most $\frac{n}{k}\log\frac{1}{\eta}$ times per ε/k-separation, instead of n times. Hence, its worst-case time complexity is $\mathcal{O}\Big(n\log\frac{1}{\eta}\frac{\log kn/\varepsilon}{\log 1/(\alpha\sigma_{\max})}(n+m)\Big)$. Further, the approximation ratio of stochastic greedy is $(1-1/e-\eta)$ instead of lazy greedy's $(1-1/e)$.

8.4 Group Forest Closeness Centrality

For group forest closeness, we consider undirected weighted graphs $G = (V, E, w)$ with non-negative edge weights. Recall the definitions of augmented graph $G_\star$ from Section 5.5.1 and of forest farness from Eq. (5.8). In order to extend the concept of forest closeness to groups of vertices, it is enough to define the forest farness $g_{ff}(S)$ of a vertex set $S \subset V$; the forest closeness of S is then given by $g_{fc,\alpha}(S) := \frac{1}{g_{ff,\alpha}(S)}$.

Recall from Proposition 5.5.1 that the forest farness of a single vertex v of G equals the electrical farness of v in the augmented graph $G_\star$. We use this fact to generalize the forest farness of a set S of vertices in G. In particular, we define $g_{ff}(S) := \operatorname{tr}\big(((\mathbf{L}_\star)_{-S})^{-1}\big)$, where $\mathbf{L}_\star$ is the Laplacian matrix of the augmented graph G, and by $(\mathbf{L}_\star)_{-S}$ we denote the matrix that is obtained by removing all rows and columns with indices in S. This definition is based on a corresponding definition of electrical farness by Li et al. [183]. For $|S| = 1$, it coincides with the definition of electrical closeness from Section 5.2 [150]; thus, our definition of group forest closeness is compatible with the definition of the forest closeness of individual vertices (i.e., Eq. (5.5)).

Given our definition, it is natural to ask for a set S of k vertices that maximizes $g_{fc}(S)$ over all possible size-k sets S; indeed, this optimization problem has also been considered for many other group centrality measures [132]. The complexity of this problem is settled in the following theorem:

Theorem 8.4.1 ([133]). Maximizing group forest closeness subject to a cardinality constraint is $\mathcal{NP}$-hard.

Since an efficient algorithm for maximizing group forest closeness is unlikely to exist (due to Theorem 8.4.1), it is desirable to construct an inexact algorithm for this problem. The next two results enable the construction of such an algorithm; they immediately follow from respective results on group electrical closeness on $G_\star$ (see Ref. [183, Theorems 5.4 and 6.1]).

Lemma 8.4.1 ([133]). $g_{ff}(\cdot)$ is a non-increasing and supermodular set function.

For the following corollary, we consider a greedy algorithm that constructs a set S of size k. This set is initially empty; while $|S|$ is smaller than k, the algorithm adds the vertex v to S that maximizes the marginal gain: $v = \operatorname{argmax}_{x\in V\setminus S} g_{ff}(S) - g_{ff}(S\cup\{v\})$.

Corollary 8.4.1 ([133]). The greedy algorithm computes a set S such that:

$$g_{ff}(\{v_0\}) - g_{ff}(S) \geq \left(1 - \frac{k}{e(k-1)}\right)\big(g_{ff}(v_0) - g_{ff}(S^*)\big),$$

where v_0 is the vertex with highest (individual) forest closeness and S^* is the set of size k that maximizes the group forest closeness.

Note that a naive implementation of the greedy algorithm would invert $(\mathbf{L}_\star)_{-(S\cup\{v\})}$ for each v, i.e., it would require $k\cdot n$ matrix inversions in total. By using the ideas of Li et al. for group electrical closeness [183]

Algorithm 19 Greedy algorithm for group forest closeness maximization – adapted from Li et al. [183]

Input: Undirected graph $G = (V, E)$, group size k
Output: Group $S \subseteq V$ of k vertices

1: $\mathbf{L}_\star^\dagger \leftarrow$ PSEUDOINVERSE$(\mathbf{L}_\star)$
2: $v \leftarrow \text{argmin}_{v \in V}\, n(\mathbf{L}_\star^\dagger[v,v]) + \text{tr}\left(\mathbf{L}_\star^\dagger\right)$
3: $\mathbf{M} \leftarrow$ INVERSE$((\mathbf{L}_\star)_{-v})$ ▷ Invariant: $\mathbf{M} \leftarrow (\mathbf{L}_\star)_{-S}^{-1}$ throughout the algorithm
4: $S \leftarrow \{v\}$
5: **while** $|S| < k$ **do**
6: $\quad v \leftarrow \text{argmax}_{v \in V \setminus S} \frac{(\mathbf{M}\mathbf{e}_v)^\top (\mathbf{M}\mathbf{e}_v)}{\mathbf{e}_v^\top \mathbf{M}\mathbf{e}_v}$
7: $\quad \mathbf{M} \leftarrow \left(\mathbf{M} - \frac{\mathbf{M}\mathbf{e}_v \mathbf{e}_v^\top \mathbf{M}}{\mathbf{e}_v^\top \mathbf{M}\mathbf{e}_v}\right)_{-\{v\}}$
8: $\quad S \leftarrow S \cup \{u\}$
9: **return** S

(depicted in Algorithm 19 for the case of forest closeness), these inversions can be avoided, such that only a single matrix inversion is required in total. This makes use of the fact that whenever a vertex u is added to the set S, we can decompose $(\mathbf{L}_\star)_{-S}$ into a block that consists of $(\mathbf{L}_\star)_{-(S \cup \{u\})}$ and a single row/column that corresponds to u. It is now possible to apply block-wise matrix inversion to this decomposition to avoid the need to recompute $((\mathbf{L}_\star)_{-(S \cup \{u\})})^{-1}$ from scratch (in Line 7 of the pseudocode). We remark that the greedy algorithm can be further accelerated by utilizing the JLT lemma [183]; however, since this necessarily results in lower accuracy, we do not consider this extension in our experiments.

Furthermore, we note that, by applying standard reduction by Gremban [130], it would also be possible to apply our UST-based algorithm (i.e., Algorithm 10) to the case of group forest closeness. However, if the aforementioned block-wise matrix inversion is not applied, this would require us to sample USTs for each of the $k \cdot n$ vertex evaluation. On the other hand, in order to apply block-wise inversion, the entire inverse of $(\mathbf{L}_\star)_{-S}$ must be available (and not only the diagonal). Computing this inverse via UST sampling is prohibitively expensive so far. Hence, in our experiments, we prefer the algorithmic approach by Li et al.– adapted for group forest closeness.

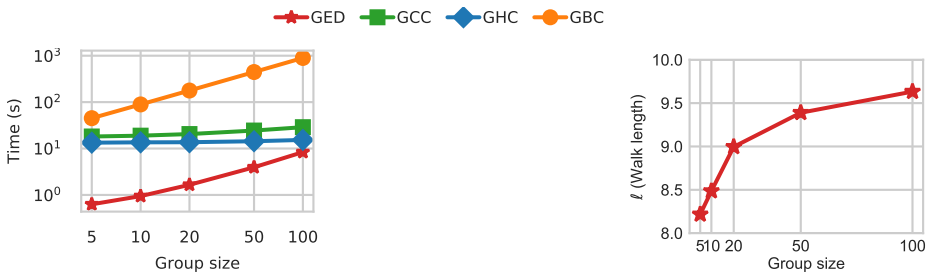
8.5 Experiments – GED-Walk

Settings. Our algorithms for GED-Walk computation and maximization are implemented in C++ on top of the open-source framework NetworKit [273], which also includes implementations of the aforementioned algorithms for group-betweenness, group-harmonic, and group-closeness maximization. All experiments are executed on a Linux server with an Intel Xeon Gold 6154 (36 cores in total) and 1.5 TiB of RAM of memory. Unless stated differently, every experiments uses 36 threads (one per core), and the default algorithm for GED-Walk maximization uses the combinatorial bound (see Eq. (8.8)) and the lazy-greedy strategy described in Section 8.3.2 with $\varepsilon = 0.5$.

Datasets. All networks are undirected; real-world graphs have been downloaded from the Koblenz Network Collection [173] and from the 9th DIMACS challenge [87], detailed statistics are reported in Table 8.1. Synthetic networks have been generated using the Erdős Rényi, R-MAT [65], and Barabàsi-Albert [28] models as well as the random hyperbolic generator from von Looz et al. [191], all of which are available in NetworKit. More precisely, for the Erdős Rényi generator, we set as probability the parameter $p = 20/n$. For

Table 8.1: Largest connected component of the real-world instances we used in our experiments.

Network	n	m	Category
dimacs9-COL	435,666	521,200	Road
munmun_twitter_social	465,017	833,540	Social
com-dblp	317,080	1,049,866	Co-author
wikipedia_link_mr	92,875	1,396,893	Hyperlink
roadNet-PA	1,087,562	1,541,514	Road
citeseer	365,154	1,721,981	Citation
roadNet-TX	1,351,137	1,879,201	Road
web-Stanford	255,265	1,941,926	Hyperlink
petster-dog-household	255,968	2,148,090	Social
wikipedia_link_bn	225,970	2,183,246	Hyperlink
petster-catdog-household	324,249	2,642,635	Social
wikipedia_link_uz	439,263	2,920,885	Hyperlink



(a) Running time (s) of GED-Walk, GCC, GHC, and GBC maximization (note that k is in log-scale).

(b) Length of the walks considered by our algorithm for GED-Walk maximization.

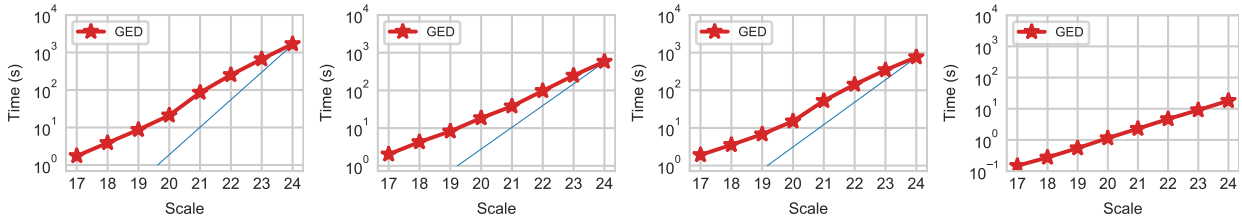
Figure 8.1: Scalability w.r.t. group size of GED-Walk, GCC, GHC, and GBC maximization (Figure 8.1a), and highest walk length considered by our GED-Walk maximization algorithm (Figure 8.1b).

R-MAT, the parameter setting is the same as in the Graph 500's benchmark [223] – i.e., edge factor 16, $a = 0.57, b = 0.19, c = 0.19$, and $d = 0.05$. For the Barabàsi-Albert generator, we set the average degree to 20 and, for the random hyperbolic generator, we set the average degree to 20 and the exponent of the power-law distribution to 3.

8.5.1 Scalability w.r.t. Group Size

Figure 8.1 shows the average running time in seconds of GED-Walk (GED), group-closeness (GCC), group-harmonic (GHC), and group-betweenness (GBC) maximization for group sizes from 5 to 100 – detailed running times are reported in Table G.1, Appendix G.1. For small group sizes, GED-Walk can be maximized much faster than the other considered measures: for $k = 5$, our algorithm for GED-Walk maximization is on average $67.7\times$, $21.26\times$, and $52.6\times$ faster than GCC, GHC and GBC maximization, respectively, whereas for $k = 100$ it is respectively $8.9\times$, $1.83\times$, and $90.1\times$ faster. On the other hand, GCC and GHC maximization scale better than both GBC and GED-Walk maximization w.r.t. the group size. This behavior is expected since the evaluation of the marginal gain becomes computationally cheaper for GCC and GHC for larger groups. This property, however, does not apply to our algorithm for maximizing GED-Walk which, in turn, needs to increase the length ℓ of the while the group grows – see Algorithm 18, Line 28.

Yet, one can also observe that the group-closeness score increases only very slowly – or more slowly than GED-Walk's – when increasing the group size (see Figure 8.4b). This means that, from a certain group size



(a) Erdős Rényi (b) R-MAT (c) Barabàsi-Albert (d) Random hyperbolic

Figure 8.2: Running time (s) on 36 cores of our lazy greedy algorithm for GED-Walk maximization on synthetic networks with 2^{17} to 2^{24} vertices, $k = 10$. Data points are aggregated over three different randomly generated networks using the geometric mean.

Table 8.2: Running time (s) of GED-Walk maximization on 36 cores on large real-world networks, $k = 10$.

Network	Category	n	m	Time (s)
petster-friendships-cat	Social	148,826	5,447,464	4.7
dimacs9-W	Road	6,262,104	7,559,642	45.3
dimacs9-CTR	Road	14,081,816	16,933,413	86.9
flickr-growth	Social	2,173,370	22,729,227	47.2
soc-LiveJournal1	Social	4,843,953	42,845,684	35.0
livejournal-links	Social	5,189,808	48,687,945	47.1
orkut-links	Social	3,072,441	117,184,899	76.5
dbpedia-link	Hyperlink	18,265,512	126,888,089	348.7
dimacs10-uk-2002	Hyperlink	18,459,128	261,556,721	47.5
wikipedia_link_en	Hyperlink	13,591,759	334,590,793	276.4

on, the choice of a new group member hardly makes a difference for group-closeness – in most cases, only the distance to very close vertices can be reduced. In that sense, GED-Walk seems to distinguish better between more and less promising candidates.

Figure 8.1 shows the length ℓ of the walks considered by our algorithm w.r.t. the group size. In accordance to our expectations about number of evaluations of $L_\ell(u)$ and $U_\ell(u)$ we stated in Section 8.3.2, ℓ grows sub-linearly w.r.t. k.

8.5.2 Scalability to Large (Synthetic) Graphs

Figure 8.2 shows the running time in seconds of GED-Walk maximization with $k = 10$ on randomly generated networks using the Erdős Rényi, R-MAT, Barabási-Albert, and random hyperbolic models models. The thin blue lines represent the linear regression on the running times. With respect to the running time curves, the regression lines either have a steeper slope (Figures 8.2a–8.2c) or, in the case of the random hyperbolic generator in Figure 8.2d, they match it almost perfectly. Thus, for the network types and sizes under consideration, GED-Walk maximization scales (empirically) linearly w.r.t. n.

Table 8.2 shows the running times in seconds of GED-Walk maximization on large real-world instances with up to hundreds of millions of edges with $k = 10$. For the largest networks, our algorithm needs up to six minutes to finish, but in most cases it requires only around one minute.

(a) Multi-core speedups of GED-Walk maximization over single-core GED-Walk maximization.

(b) Running time of GED-Walk, GCC, GHC, and GBC maximization w.r.t. the number of cores.

Figure 8.3: Parallel scalability of GED-Walk, GCC, GHC, and GBC maximization, $k = 10$. Data points are aggregated over the instances of Table 8.1 using the geometric mean.

8.5.3 Parallel Scalability

As stated in Section 8.3.1, the algorithm to compute GED-Walk can be parallelized easily by computing $\phi_i^{\text{hit}}(u)$ and $\phi_i^{\text{miss}}(u)$ in parallel for different $u \in V$. Figure 8.3a shows the parallel speedup of our algorithm for maximizing GED-Walk over itself running on a single core for $k = 10$. The scalability is moderate up to 16 cores, while on 32 cores it does not gain much additional speedup. A similar behavior was observed before for group-closeness [38]. Figure 8.3b shows the running time in seconds of GED-Walk, GBC, and GCC maximization with increasing number of cores, for $k = 10$. On a single core, GED-Walk maximization is on average $30.5\times$, $14.76\times$, and $124.1\times$ faster than GCC, GHC, and GBC maximization, respectively. As we increase the number of cores, the decrease of the running time of the three algorithms is comparable – except for GCC and GHC on 32 cores: In this case, GCC and GHC are slower than on 16 cores on the considered instances.

The limited scalability affecting all three algorithms is probably due to memory latency becoming a bottleneck in the execution on multiple cores [26, 195]. Further, Figure 8.3b shows that, on 16 cores, our algorithm for GED-Walk maximization finishes on average within a few seconds. Here the running time of the algorithm is dominated by its sequential parts, and it is not surprising that adding 16 more cores does not speed the algorithm up substantially.

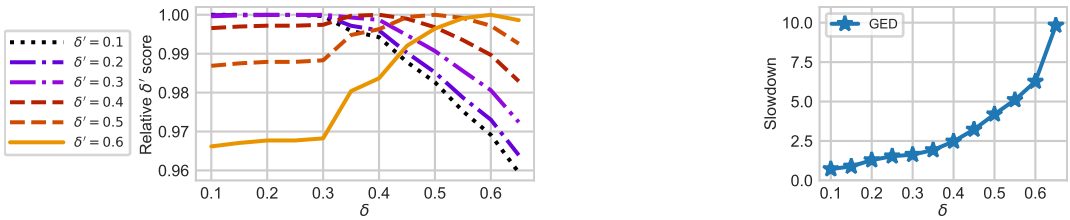
8.5.4 Scalability with Large Groups

We evaluate the performance of the stochastic algorithm with $\varepsilon = 0.5$ and $\eta = 0.1$ on large groups. Figure 8.4a shows the running time of GED-Walk (both lazy and stochastic greedy strategies) GCC, and GHC maximization for large values of k. For $k \geq 1{,}000$, the stochastic algorithm is significantly faster than the lazy algorithm. However, as explained in Section 8.5.1, GCC maximization scales better than both GED strategies w.r.t. k, and for $k \geq 1{,}000$ it is even faster than GED-S. On the other hand, Figure 8.4b shows the relative GED-Walk, group-harmonic, and group-closeness scores of the groups computed using GED, GED-S, GCC, and GHC maximization algorithms. Results demonstrate that the stochastic greedy approach computes groups with nearly the same quality as GED in less time, which makes it a reasonable alternative to lazy greedy for large values of k.

(a) Running time (s) of GED-Walk, GCC, and GHC maximization.

(b) GED-Walk, group-harmonic, and group-closeness scores of groups computed by GED-Walk, GCC, and GHC maximization. For each k, scores are divided by the scores for $k = 1$.

Figure 8.4: Running time (s) and scores of GED-Walk with lazy greedy (GED), and stochastic greedy (GED-S) strategies with large k (log-scale). Data points are aggregated over the instances of Table 8.1 using the geometric mean.



(a) Relative δ' score (i.e., $g^{\delta'}_{\text{ED}}(S^{\delta})/g^{\delta}_{\text{ED}}(S^{\delta})$) for $\delta, \delta' \in [0.1, \ldots, 0.6]$, $\varepsilon = 0.1$ and $k = 10$.

(b) Slowdown of our algorithm for maximizing GED-Walk using the spectral bound with $\alpha = \delta/\sigma_{\max}$ over the lazy algorithm using the combinatorial bound.

Figure 8.5: Quality and running time performance of our GED-Walk maximization algorithm using the spectral bound. Data points are aggregated over the instances of Table 8.1 using the geometric mean.

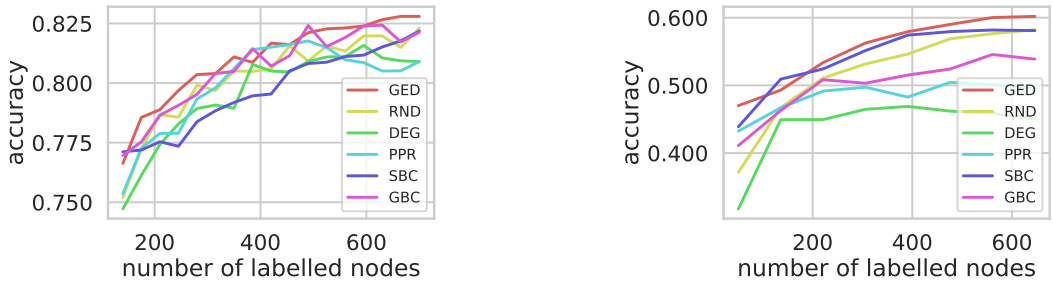
8.5.5 Impact of Parameter α

We now analyze how different settings of the parameter α impact the groups computed by our algorithm for GED-Walk maximization. For this experiment we use the spectral bound (see Eq. (8.9)). Proposition 8.2.1 implies that GED-Walk converges iff $\alpha < 1/\sigma_{\max}$. Let $\delta \in (0, 1]$; obviously, GED-Walk also converges if $\alpha < \delta/\sigma_{\max}$. In this experiment, we compute groups S^{δ} for a certain value $\alpha = \delta/\sigma_{\max}$ and we measure of the resulting group using $\alpha' = \delta'/\sigma_{\max}$. We denote this score by $g^{\delta'}_{\text{ED}}(S^{\delta})$.

Figure 8.5a shows the ratio $g^{\delta'}_{\text{ED}}(S^{\delta})/g^{\delta}_{\text{ED}}(S^{\delta})$ with $\delta, \delta' \in [0.1, \ldots, 0.6]$, $\varepsilon = 0.1$ and $k = 10$. Figure 8.5b shows the slowdown, i.e., running time of computing S^{δ} divided by the running time of the lazy algorithm using the combinatorial bound. Computing $g^{\delta'}_{\text{ED}}(S^{\delta})$ with $\delta \in [0.1, \ldots, 0.3]$ yields similar scores independently of δ, meaning that the GED-Walk score of a group does not change significantly within such an interval for δ. Increasing δ above 0.3, however, leads to a noticeable reduction of the relative scores computed using $\delta' \leq 0.3$ and to a steeper growth of the ones computed using $\delta' \geq 0.5$.

8.6 Applications of GED-Walk

We demonstrate the relevance of GED-Walk for graph mining applications by showing that it improves the performance of two popular graph mining tasks: semi-supervised vertex classification and graph classifi-



(a) Cora

(b) Wiki

Figure 8.6: Semi-supervised vertex classification accuracy for different strategies for choosing the training set.

cation. As a preprocessing step, for both tasks before applying GED-Walk we first construct a weighted graph using the symmetrically normalized adjacency matrix $\mathbf{D}^{\frac{1}{2}}\mathbf{A}\mathbf{D}^{\frac{1}{2}}$, which is often used in the literature [166, 292].[47] Here, instead of taking the contribution of an i-walk $(e_1, \ldots, e_i)$ to be α, we define it to be $\alpha \prod_{j=1}^{i} w(e_j)$, where $w(e_j) \leq 1$ is the edge weight of edge e_j. Except for the introduction of coefficients in the recurrences ϕ^{hit} and ϕ^{miss}, no modifications of our algorithms are required. Compared to our unweighted definition of GED-Walk, this weighted variant converges even faster, as the contribution of each walk is smaller.

8.6.1 Vertex Classification

Vertex classification is a fundamental graph mining problem where the goal is to predict the class labels of all vertices in a graph given a small set of labelled vertices and the graph structure [278]. The choice of which vertices we label (i.e., the vertices we include in the training set) before building a classification model can have a significant impact on the test accuracy, especially when the number of labelled vertices is small compared to the size of the graph [23, 265]. Since many models rely on diffusion to propagate information on the graph [278], we expect that selecting a training set with high GED-Walk centrality will improve diffusion, and thus also the model's accuracy. To test this hypothesis, we evaluate the performance of Label Propagation [66, 292] given different strategies for choosing the training set. The choice of the vertices for the training set can influence the accuracy of the classifier, especially when the number of labelled vertices is small compared to n [23, 265].

A key aspect in semi-supervised learning problems is the so-called *cluster assumption*, i.e., vertices that are close or that belong to the same cluster typically have the same label [67, 292]. Several models label vertices by propagating information through the graph via diffusion [278]. We expect GED-Walk to cover the graph more thoroughly than shortest-path based group centrality measures. Therefore, we conjecture that choosing vertices with high GED-Walk improves the diffusion – and thus the accuracy of propagation-based models. We test this hypothesis by comparing the classification accuracy of the label propagation model [278, 292] where the training set is chosen using different strategies.[48] The main idea of label propagation is to start from a small number of labelled vertices and each vertex iteratively propagates its label to its neighbors until convergence.

[47] Recall that $\mathbf{D}$ is the degree matrix, see Section 2.1.

[48] While this model is less powerful than state-of-the-art predictors, our strategy to select the training set could also be applied to more sophisticated models like graph neural networks.

Table 8.3: Graph classification datasets.

Dataset	# of graphs	# of classes
Mutagenicity [160, 250]	4,337	2
PROTEINS [58, 90]	1,113	2
ENZYMES [58, 263]	600	6
IMDB-BINARY [286]	1,000	2
REDDIT-BINARY [286]	2,000	2

In our experiments, we use the Normalized Laplacian variant of the Label Propagation model as our baseline[49] and we set the value for the return probability hyper-parameter to 0.85.

We evaluate the classification accuracy on two common benchmark graphs: Cora ($n = 2{,}810$, $m = 7{,}981$) and Wiki ($n = 2{,}357$, $m = 11{,}592$) [264]. We let the vertices with highest GED-Walk centrality be in the training set and the rest of the vertices be in the test set. We compare GED-Walk with the following baselines for selecting the training set: RND (select vertices at random, with results averaged over 10 trials), DEG (select the vertices with highest degree), SBC (select vertices with highest individual betweenness centrality), GBC (select vertices with highest group-betweenness centrality), and PPR (select vertices with highest Personalized PageRank).

Figure 8.6 shows that, for both the considered datasets and across different number of labelled vertices, selecting the training set using GED-Walk leads to highest (or comparable) test accuracy. Furthermore, while the second-best baseline strategy is different on different datasets – on Cora it is the GBC strategy while on Wiki it is the SBC strategy), GED-Walk is consistently better. Overall, these results confirm our hypothesis.

8.6.2 Graph Classification

Graph classification is another fundamental graph mining problem. The goal is to classify entire graphs based on features derived from their topology/structure. In contrast to the vertex classification task where each dataset is a single graph, here each dataset consists of many graphs with varying size and their associated ground-truth class labels. In this setting, our hypothesis is that groups of vertices with high GED-Walk centrality capture rich information about the graph structure and thus can be used to derive features that are useful for graph classification.

To extract features based on GED-Walk, we first compute the group of k vertices with highest (approximate) group centrality score. The group centrality score is the first feature that we extract. In addition, we summarize the marginal gains of all the remaining vertices in the graph in a histogram with b bins. We concatenate these features to get a feature vector $\mathbf{x}_i \in \mathbb{R}^{b+1}$ for each graph i in the dataset. These features are useful since graphs with similar structure will have similar group scores and marginal gains. We denote this base setting by Ged.

In addition, we obtain the topic-sensitive PageRank vector of each graph, where we specify the teleport set to be equal to the vertices in the group with highest (approximate) group centrality. Then, we summarize

[49] Our strategy to select the training set also applies to more sophisticated vertex classification models such as graph neural networks [166].

Table 8.4: Graph classification accuracy (in %) on the datasets of Table 8.3. Best performance per dataset marked in bold.

Dataset	ENZ.	IMD.	Mut.	PRO.	RED.
Eig-T	23.02	56.59	56.90	73.35	75.31
Eig-H	23.47	70.28	68.88	72.42	72.02
Ged	19.18	60.64	64.51	71.95	70.59
Ged+PPR*-H	20.85	65.18	65.46	72.18	71.59
Ged+PPR**-H	20.39	66.27	65.86	72.44	75.95
Eig-T+Ged-T	26.46	63.56	64.14	**74.06**	80.18
Eig-H+Ged-H	23.14	69.74	69.08	73.13	75.16
Eig-T+ Ged+PPR*-T	27.45	69.25	62.78	73.54	76.70
Eig-H+ Ged+PPR*-H	24.12	**71.62**	**69.18**	73.10	74.17
Eig-T+ Ged+PPR**-T	**27.88**	68.53	62.43	73.72	80.48
Eig-H+ Ged+PPR**-H	24.78	70.54	68.81	72.97	**81.43**

*: PageRank teleport probability 0.85;
**: PageRank teleport probability 0.15

this vector (i) by using a histogram of b bins and (ii) by extracting the top p values, denoted by PPR-H and PPR-T, respectively. Intuitively, these features capture the amount of diffusion in the graph.

As a strong baseline, we compute the eigenvalues of the adjacency matrix and summarize them (i) in a histogram of b bins and (ii) by extracting the top p eigenvalues, denoted by Eig-H and Eig-T, respectively. This latter strategy is inspired by recent work on graph classification [117] showing that spectral features can outperform deep-learning based approaches. Further, the ability of efficiently compute the eigenvalue histograms motivates using them as a feature for graph classification [92]. Similarly, a strong advantage of these features based on GED-Walk is that we can efficiently compute them. Last, we also combine the spectral- and GED-based features by concatenation, i.e., Eig-T+GED denotes the combination of Eig-T and Ged features. In the following experiments, we fix the value of the hyper-parameters $k = 10$, $b = 20$, and $p = 10$; in practice, however, these parameters can also be tuned using, for example, cross-validation. We split the data into 80% training and 20% test set and average the results for 10 independent random splits.

Table 8.4 summarizes graph classification results over the instances of Table 8.3. Notice that enriching the baseline features with our GED-Walk-based features improves the classification accuracy on all datasets, with variants using all available features (i.e., Eig+Ged+PPR) performing best. Moreover, as shown in Table 8.5, using the most central group according to GED-Walk as teleport set yields performance improvements over standard PageRank – i.e., with teleport to all vertices.

In summary, these results show that GED-Walk captures meaningful information about the graph structure that is complementary to baseline spectral features. We argue that GED-Walk can be used as a relatively inexpensive to compute additional source of information to enhance existing graph classification models.

Table 8.5: Graph classification accuracy (in %) on the datasets of Table 8.3. PR denotes PageRank with all vertices as the teleport set.

Dataset	ENZYMES	IMDB-BINARY	Mutagenicity	PROTEINS	REDDIT-BINARY
EIG-H+GED+PPR*-H	24.12	**71.62**	**69.18**	73.10	74.17
EIG-T+GED+PPR*-T	27.45	69.25	62.78	73.54	76.70
EIG-H+GED+PPR**-H	24.78	70.54	68.81	72.97	**81.43**
EIG-T+GED+PPR**-T	**27.88**	68.53	62.43	73.72	80.48
EIG-H+GED	23.14	69.74	69.08	73.13	75.16
EIG-T+GED	26.46	63.56	64.14	**74.06**	80.18
EIG-H+PPR*-H	24.20	**71.62**	**69.18**	73.08	74.17
EIG-T+PPR*-T	27.49	69.25	62.78	73.56	76.71
EIG-H+PPR**-H	24.78	70.54	68.80	72.97	**81.43**
EIG-T+PPR**-T	**27.88**	68.53	62.43	73.72	80.48
EIG-H+PR*-H	23.54	70.00	68.85	72.33	75.11
EIG-T+PR*-T	23.18	57.06	57.38	73.25	76.00
EIG-H+PR**-H	23.54	70.42	69.17	72.49	76.33
EIG-T+PR**-T	23.18	58.42	57.79	73.31	76.59
EIG-H	23.47	70.28	68.88	72.42	72.02
EIG-T	23.02	56.59	56.90	73.35	75.31
GED+PPR*-H	20.85	65.18	65.46	72.18	71.59
GED+PPR**-H	20.39	66.27	65.86	72.44	75.95
GED	19.18	60.64	64.51	71.95	70.59
PPR*-H	21.01	60.93	60.44	71.79	72.74
PPR**-H	19.46	66.39	62.27	71.86	73.79
PR*-H	20.11	57.04	61.31	71.78	73.19
PR*-T	16.98	51.15	55.85	71.84	69.66
PR**-H	20.11	60.84	61.75	71.77	73.43
PR**-T	16.98	56.64	57.19	72.12	73.31

*: PageRank teleport probability 0.85; **: PageRank teleport probability 0.15

8.7 Experiments – Group Forest Closeness

As we did for GED-Walk, to demonstrate the relevance of group forest closeness in graph mining applications, we apply it to semi-supervised vertex classification [278] (see Section 8.6.1).

Settings. We implement our algorithm for forest closeness maximization in C++ upon the NetworKit [273] toolkit and we manage our experiments with SimexPal [14] to ensure reproducibility.

In our experiments, we used the Normalized Laplacian variant of label propagation [292]. We set the return probability hyper-parameter to 0.85 and we evaluate its accuracy on two well-known disconnected graph datasets: Cora ($n = 2{,}708$, $m = 5{,}278$) and Citeseer ($n = 3{,}264$, $m = 4{,}536$) [264]. Since this variant of label propagation cannot handle graphs with isolated vertices (i.e., zero-degree vertices), we remove all isolated vertices from these datasets. For a fixed size k of the training set, we select its vertices as the group of vertices computed by our greedy algorithm for group forest maximization and as the top-k vertices with highest estimated forest closeness. We also include several well-known (individual) vertex selection strategies for comparison: average over 10 random trials, the top-k vertices with highest degree, the top-k vertices with highest betweenness centrality, and the top-k vertices with highest Personalized PageRank.

Results. Figure 8.7 shows that, for disconnected graphs and for a moderate number of labelled vertices, selecting the training set by group forest closeness maximization yields consistently superior accuracy than

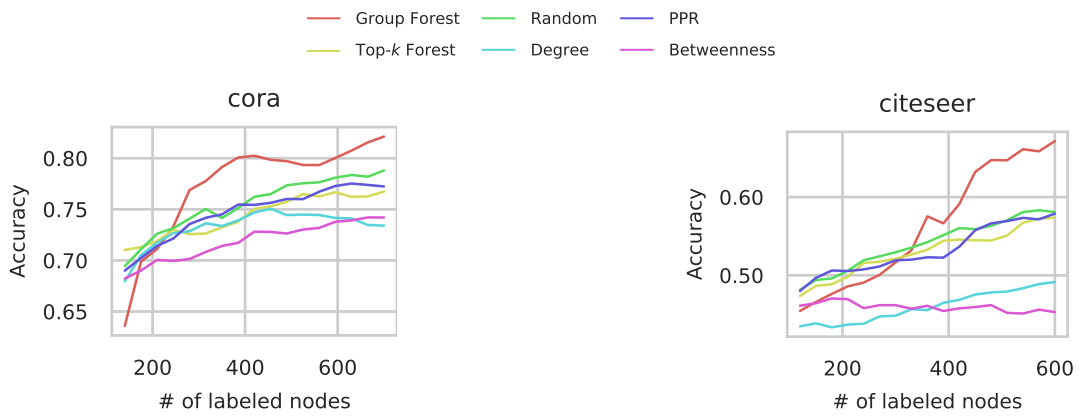


Figure 8.7: Accuracy in semi-supervised vertex classification in disconnected graphs when using different strategies to create the training set.

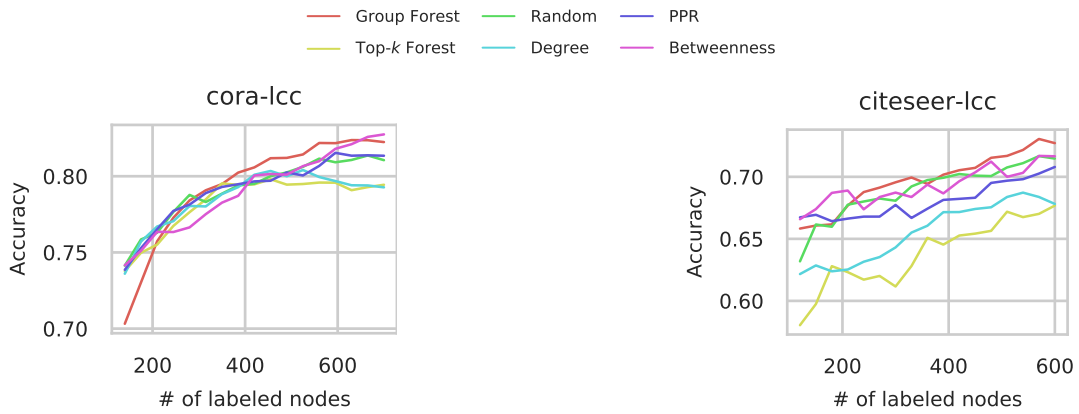


Figure 8.8: Accuracy in semi-supervised vertex classification on the largest connected component of the datasets (Cora-lcc: $n = 2{,}485$, $m = 5{,}069$; Citeseer-lcc: $n = 2{,}110$, $m = 3{,}668$) when using different strategies to create the training set.

strategies based on existing centrality measures – including top-k forest farness. As expected, the accuracy of existing measures improves if one considers connected graphs. Figure 8.8 shows the accuracy for connected graphs when using different strategies to create the training set. Compared to disconnected graphs, the competitors perform better in this setting. However, in our datasets, choosing the training set by group forest maximization yields nearly the same accuracy as the best competitors. The running time of our greedy algorithm for group forest maximization is reported in Table D.3, Appendix D.3.

8.8 Conclusions

In this chapter, we addressed the issues of scaling group centrality to large networks and the lack of electrical group centrality measures capable of handling disconnected graphs. We introduced two new centrality measures, GED-Walk and group forest closeness. For the former, we implemented efficient approximation algorithms for its maximization and for computing the GED-Walk score of a given group. For the latter, we adapted the cubic approximation algorithm from Li et al. [183].

GED-Walk's descriptive power is demonstrated by experiments on two fundamental graph mining tasks: both semi-supervised vertex classification and graph classification benefit from the new measure. As GED-Walk can be optimized faster than earlier group centrality measures, it is often a viable replacement for more expensive measures in performance-sensitive applications. Group forest closeness achieved analogous results for the semi-supervised vertex classification task on disconnected graphs. On the other hand, cubic time cannot scale to large graphs; we leave the development of faster approximation algorithms to maximize this measure to future work.

In terms of running time, our algorithm for GED-Walk maximization significantly outperforms the state-of-the-art algorithms for maximizing group-closeness, group-harmonic, and group-betweenness centrality when group sizes are at most 100. The fact that GED-Walk scales worse than group-closeness and group-harmonic w.r.t. to k may seem as a limitation; however, we expect that many applications are interested in group sizes considerably smaller than 100.

Experiments on synthetic networks indicate that our algorithm for GED-Walk maximization scales linearly with the number of vertices. For graphs with 2^{24} vertices and more than 100M edges, it needs up to half an hour – often less. In fact, our algorithm can maximize GED-Walk for small groups on real-world graphs with hundreds of millions of edges within a few minutes. A promising direction for future research is to apply GED-Walk to other practical applications. Examples include network and traffic monitoring [91, 247], the development of immunization strategies to lower the vulnerability of a network to epidemic outbreaks [236], and improving landmark-based shortest path queries [124].

Part IV

Maximum Weighted Matching in Fully-Dynamic Graphs

Introduction

Stable Marriage is the popular problem introduced by Gale and Shapley [115] of matching an equal number of women and men, each of whom has ranked the participants of the opposite sex in order of preference so that no couple prefer each other to the partners they are matched with. In other scenarios, however, there may not be a separation of the participants into two classes (e.g., same-gender couples or pairing players in a chess tournament [172]), hence the Stable Roommates Problem (SRP). SRP, originally described in the paper of Gale and Shapley [115], is essentially the stable marriage problem with just one set and appears in a variety of practical applications [149]. Every person in the set (of even cardinality n) ranks the others in order of preference. The objective is to partition the set into $n/2$ pairs of "roommates" such that no two persons that are not roommates both prefer each other to their current roommates – i.e., finding a stable *matching*.

Some commercial and scientific applications, however, demand to find matchings that are not only stable, but that also have the maximum weight, which leads to the Maximum Weighted Matching problem (MWM) [49, 58, 82, 138, 176, 219, 237, 283]. To this end, several algorithms to solve MWM have been proposed; because optimal solutions are expensive to compute,[50] approximation algorithms with nearly-linear time complexity are numerous in the literature [47, 94, 95, 202, 203, 206, 242]. As we argued in Section 1.2, networks often change over time, leading to the problem of updating an (approximate) MWM after those change(s). The naive solution is to re-run a static algorithm on the new graph; this, however, is a rather inefficient strategy: as we saw in Chapter 3, reusing the information computed on an initial snapshot of the graph can reduce enormously the amount of work required to update the results on the new graph. This would be beneficial especially for applications dealing with rapidly-evolving networks where multiple updates happen every second – running a linear-time algorithm from scratch after each update would not be practical.

Numerous dynamic algorithms for approximate MWM were proposed [11, 84, 137, 275] but just a few of them have been implemented [16] and none supports edge updates in batches. In Chapter 9, we address the problem of updating a $(1/2)$-approximate MWM after multiple edge updates by presenting a new batch-dynamic algorithm inspired by the Suitor algorithm by Manne and Halappanavar [203]. Despite having a worst-case time complexity that is linear in the size of the input graph, our extensive experimental analysis suggests that, compared to the state of the art [16], our dynamic Suitor algorithm traverses less edges to handle single edge updates and yields solutions with 0.1% quality or higher. Further, for batches with up to 10^4 edge updates, it requires milliseconds – or less, for the batches of smaller size – that is $10^2\times$ to $10^6\times$ faster than re-running Suitor from scratch.

[50] The fastest know algorithm for MWM on general graphs takes $\mathcal{O}(mn \log n)$ time [116], other faster strategies exploit assumptions on the input graph [97].

9 Approximate Maximum Weighted Matching in Dynamic Networks

9.1 Introduction

Context. Given a graph $G = (V, E, w)$, a matching $M \subseteq E$ is a set of pairwise non-adjacent edges, i.e., all the vertices in G are incident to at most one edges in M. A matching is called *maximal* if no edge can be added to it without violating the matching property. It is called *maximum*, in turn, if there is no other matching with higher cardinality. Computing the maximum cardinality matching (MCM) of a graph can be done in $\mathcal{O}(m\sqrt{n})$ time in general graphs by Micali and Vazirani [215] and in $\mathcal{O}(n^\omega)$ time in planar graph with the algorithm by Mucha and Sankowski [221].[51]

On weighted graphs, the maximum weighted matching (MWM) is the matching with maximum edge weight – the edge weight of a matching M is the sum of the weights of the edges in M. The fastest known algorithms for this problem are by Gabow [113], which takes $\mathcal{O}(nm+n^2 \log n)$ time, and by Galil [116], which takes $\mathcal{O}(mn \log n)$. Assuming integral edge weights, Gabow and Tarjan [114] require $\mathcal{O}(m\sqrt{n}\log(nW))$ time, where W is the highest edge weight, while Sankowski [257] (by restricting the input to bipartite graphs) requires $\tilde{\mathcal{O}}(Wn^\omega)$ time, where $\tilde{\mathcal{O}}$ hides a polylogarithmic factor. For a broader overview of matching algorithms, we refer the reader to Refs. [49, 178, 194].

Motivation. Computing a maximum (weighted) matching requires superlinear running time and is thus impractical on today's large real-world graph datasets. To mitigate long running times, it is common to resort to approximation. Preis's greedy algorithm [242] computes a $(1/2)$-approximation of the matching with highest weight in $\mathcal{O}(m)$ time; the same result is also achieved with the path-growing algorithm (PGA) by Drake and Hougardy [95]. The main disadvantage of those algorithms is that they are inherently sequential and thus cannot exploit parallelism, a common acceleration strategy for massive data. Birn et al. [47], in turn, provide a parallel implementation of the local max algorithm [141], which computes a maximal matching of an unweighted graph and a $(1/2)$-approximation of the MWM of a weighted graph in $\mathcal{O}(\log^2 n)$ expected time. Manne and Halappanavar [203] introduced Suitor, a parallel $(1/2)$-approximation algorithm based on local domination that is faster previous strategies and is amenable to parallelism.

Real-world networks are not only large but often change over time: edges are inserted, deleted, or change their weight – see Section 2.7. Even with a linear-time algorithm, it would be excessively expensive to compute (or approximate) a maximum (weighted) matching from scratch every time the graph changes. In recent years, several fully-dynamic algorithms for both exact and approximate MCM [18, 29, 32, 44, 45, 46, 68, 128, 137, 148, 158, 225, 232, 256, 270] and MWM [11, 137, 275] have been proposed. These algorithms perform a

[51] Recall that $\omega < 2.373$ is the matrix multiplication exponent.

static computation of the matching on an initial snapshot of the graph and exploit this information to update the matching more efficiently than a static rerun when the graph changes. Main limitations of existing fully-dynamic algorithms are either a weaker quality guarantee (e.g., Ref. [11]), or an expensive time complexity (e.g., Ref. [137]). To the best of our knowledge, none of the existing algorithms for fully-dynamic MWM support batch updates and the only existing implementation is provided in Ref. [16].

Contribution. In this chapter, we introduce a new algorithm that takes inspiration from Suitor [203] and maintains a $(1/2)$-approximation of the MWM in fully-dynamic graphs. The main idea is simple: we use the static Suitor algorithm to compute an approximate MWM on an initial snapshot of the graph. Then, after an edge update, our dynamic algorithm identifies the affected vertices (i.e., those whose matching partner needs to be updated) and updates the matching accordingly.

Our implementation also supports multiple edge insertions or removals in batches. For single edge updates, our dynamic algorithm has a worst-case time complexity of $\mathcal{O}(n + m)$, whereas for batches with b edge updates it is $\mathcal{O}(b \cdot (n + m))$. Although this does not improve the static time complexity, our dynamic algorithm performs remarkably well in practice: in our experiments, we evaluate its running time on real-world (complex and road) and synthetic networks with up to 2.5 billion edges. Our results show that, compared to the best algorithm presented in Ref. [16], our dynamic Suitor algorithm handles single graph updates faster and yields matchings with at most 99.9% lower weight. Concerning batch updates, in comparison to a static recomputation (for lack of another meaningful MWM baseline in this setting), our algorithm can handle batches of 10^4 of such updates $10^2\times$ to $10^3\times$ faster. Furthermore, the time required by our dynamic algorithm for every considered batch size is always below a millisecond. Thus, our algorithm's implementation provides real-time capabilities even without parallelism.

Bibliographic Notes. My contributions among those in this chapter involve the new dynamic algorithms and proofs, part of the implementation work, carrying out the experiments, and the presentation of the results. Preliminary work and experiments on dynamic algorithms for MWM were conducted by Michał Boroń under the supervision of Henning Meyerhenke. The contributions of this chapter are also joint work with Michał Boroń and Henning Meyerhenke and are currently in revision for international journal publication.

9.2 Preliminaries

9.2.1 Problem Definition and Notation

Let $G = (V, E, w)$ be a simple undirected graph with positive edge weights. A matching in G is a subset of pairwise non-adjacent edges $M \subseteq E$ – alternatively, one can see a matching as a subgraph of G (restricted to the edges) with degree at most 1. A vertex is *matched* if it is incident to an edge in M, otherwise it is called *unmatched* or *free*.

In the maximum-weight matching problem (MWM), the objective is to compute a matching $M^\star$ that maximizes the sum of the edge weights.

Maximum-weight Matching	
Input:	Undirected weighted graph $G = (V, E, w)$.
Find:	Matching $M^\star$ s.t. $\sum_{e \in M^\star} w(e)$ is maximal.

In the context of fully-dynamic graphs, if an edge update[52] happens to G, we denote by $G' = (V, E', w')$ the graph after the edge update. Similarly, we denote by $N'(u)$ the set of neighbors of a vertex u in G'. Given a matching M computed on G and a sequence of graph updates, our objective is to update M in G' faster (in terms of empirical running time) than recomputing a new matching in G' from scratch, while retaining the theoretical bound on the solution quality.

9.2.2 Related Work

In the following, we summarize relevant works concerning MCM and MWM in both static and dynamic settings.

Static Algorithms. Edmond's blossom algorithm [99] (time complexity: $\mathcal{O}(mn^2)$) and the later improved algorithm by Micali and Vazirani [215] ($\mathcal{O}(m\sqrt{n})$ time) are two popular strategies to compute an MCM based on augmenting paths. Goldberg and Karzanov [125] propose a blocking skew-symmetric flow algorithm that achieves the same running time as Micali and Vazirani. More recent works use data reduction rules [168] or shrink-tree data structures [96] to achieve better running times in practice on sparse real-world networks. If we restrict the input to planar graphs, the randomized algorithm by Mucha and Sankowski [221] computes an MCM in $\mathcal{O}(n^\omega)$ time via Gaussian elimination.

Concerning the MWM problem, Galil [116] provides the fastest known algorithm, which takes $\mathcal{O}(mn \log n)$ time. Assuming integral edge weights, the algorithm by Duan et al. [97] takes $\mathcal{O}(m\sqrt{n}\log(nW))$ time, improving over Gabow and Tarjan's algorithm [114]. The latter takes $\tilde{\mathcal{O}}(\log(nW))$ time, where $\tilde{\mathcal{O}}$ hides a polylogarithmic factor and W is the highest edge weight. On bipartite graphs, Sankowski [257] requires $\tilde{O}(Wn^\omega)$ time.

Running a superlinear algorithm is often too expensive on large graphs. Therefore, several approximation algorithms with (nearly-)linear running time have been introduced. A naive greedy algorithm that iteratively adds to the matching the (heaviest) edge that does not violate the matching condition takes $\mathcal{O}(m \log n)$ time and achieves a $(1/2)$-approximation for both the MCM and MWM problems [22]. Preis [242] reduced the running time to $\mathcal{O}(m)$ while retaining the same quality guarantee. Further strategies to obtain the $1/2$ bound are the one by Manne and Bisseling [202] based on dominant edges, the local max algorithm investigated by Birn et al. [47], the path growing algorithms by Drake and Hougardy [94, 95], the global paths algorithm by Maue and Sanders [206], and Suitor by Manne and Halappanavar [203].

Dynamic Algorithms. A trivial strategy maintains a $(1/2)$-approximation of a maximal matching on dynamic graphs in $\mathcal{O}(n)$ time per update by resolving all augmenting paths of length one. This result has been improved in several works. The update time was reduced for the first time to $\mathcal{O}((n+m)^{\sqrt{2}/2})$ by Ivković and Lloyd [148]. The randomized algorithm by Onak and Rubinfeld [232] maintains a $\mathcal{O}(1)$-approximation of an MCM in $\mathcal{O}(\log^2 n)$ expected amortized update time; this result was further improved

[52] Recall that by "edge update" we mean either an edge insertion, an edge removal, or an edge weight change.

by Baswana et al. [32], who reduced the update time to $\mathcal{O}(\log n)$ and the approximation ratio to $1/2$. Solomon [270] further reduced the amortized update time of Baswana et al. from logarithmic to constant. Deterministic algorithms for approximate MCM have been presented first by Bhattacharya et al. [46], who maintain a $(3+\varepsilon)$-approximate MCM in $\tilde{\mathcal{O}}(\min(\sqrt{n}, m^{1/3}/\varepsilon))$ amortized update time; the update time was later reduced to constant but at the cost of a weaker $\mathcal{O}(1)$-approximation guarantee [45]. In terms of worst-case bounds for MCM, the best known algorithms are the ones from Gupta and Peng [137] (which maintains a $(1+\varepsilon)$-approximation with $\mathcal{O}(\sqrt{m}/\varepsilon)$ update time), Neiman and Solomon [225] (which maintains a $3/2$-approximation with $\mathcal{O}(\sqrt{m})$ update time), and Bernstein and Stein [44] (which maintains a $(3/2+\varepsilon)$-approximation with $\mathcal{O}(m^{1/4} \cdot \varepsilon^{-2.5})$ update time). The first $(2+\varepsilon)$-approximation algorithms in $\mathcal{O}(\text{poly}\log n)$ update time were introduced independently by Charikar and Solomon [68] and by Arar et al. [18], whereas Grandoni et al. [128] gave a $(1+\varepsilon)$-approximation algorithm in the incremental setting requiring a constant amortized update time. For graphs with constant neighborhood independence, Barenboim and Maimon [29] present an algorithm for MCM with deterministic $\tilde{\mathcal{O}}(n)$ update time. For lax and eager algorithms, i.e., two subclasses of fully-dynamic algorithms for maintaining an MCM, Kashyop and Narayanaswamy [158] prove a conditional lower bound for the update time that is sublinear in the number of edges.

Despite the vast variety of algorithms for dynamic MCM, very little effort has been invested in implementing them and evaluating their practical performance on real-world instances. Only recently, Henzinger et al. [140] evaluated dynamic algorithms for MCM in practice. These are the algorithms by Baswana et al. [32] (2-approximate MCM in $\mathcal{O}(\sqrt{n})$ update time), by Neiman and Solomon [225] ($3/2$-approximate MCM in $\mathcal{O}(\sqrt{m})$ update time), and two novel algorithms: one based on random walks and one that uses a depth-bounded blossom algorithm to find augmenting paths. Their experimental evaluation shows that (i) the optimal matching can be maintained more than $10\times$ faster than a naive static recomputation, (ii) the considered approximation algorithms are multiple orders of magnitude faster than a naive static recomputation, and (iii) the extended random-walk based algorithms achieve the best practical performance.

Concerning the dynamic MWM problem, Anand et al. [11] propose a fully-dynamic algorithm for maintaining an 8-MWM with an expected amortized time of $\mathcal{O}(\log n \log \mathcal{C})$ per edge update, where $\mathcal{C}$ is the ratio between the maximum and the minimum edge weights of the graph. They also show that the approximation ratio can be reduced to $4.910{,}8$ without sacrificing performance by using geometric rounding. Gupta and Peng [137] maintain a $(1+\varepsilon)$-approximation in $\mathcal{O}(m^{1/2} \cdot \varepsilon^{-2-\mathcal{O}(1/\varepsilon)} \log W)$ update time in graphs with edge weights between 1 and W; their strategy runs the static algorithm from time to time, trims the graph into smaller equivalent graphs whenever possible and partitions the edges into geometrically shrinking intervals depending on their weights. Stubbs and Williams [275] present metatheorems to show that, if there exists an α-approximation algorithm for MCM with update time T, then there also exists a $2\alpha(1+\varepsilon)$-approximation algorithm for MWM with $\mathcal{O}(T \cdot \varepsilon^{-2} \log^2 W)$ update time, where W is the maximum edge weight. The main idea relies on improving and extending the algorithm by Crouch and Stubbs [84], who addressed the dynamic MWM problem in the semi-streaming model. None of the aforementioned algorithms for MWM has been implemented. For the semi-streaming model, Ghaffari and Wajc [119] describe a single-pass MWM algorithm with a $(2+\varepsilon)$-approximation ratio requiring $\mathcal{O}(n \log n)$ bits, improving previous results [84, 103, 107, 208, 238]. Again, as in the MCM case, only a few of these algorithms have recently been implemented [16]. Ref. [16] provides an experimental evaluation of dynamic algorithms for MWM inspired

Algorithm 20 Recursive findSuitor function

```
1: function FINDSUITOR(u)
2:     mate(u) ← argmax_{v∈N(u)} {w(u, v) : w(u, v) > w(v, suitor(v))}
3:     if p(u) ≠ null then
4:         y ← suitor(mate(u))
5:         suitor(mate(u)) ← u
6:         if y ≠ null then
7:             FINDSUITOR(y)
```

Algorithm 21 Static recursive Suitor algorithm [203]

Input: Undirected graph $G = (V, E, w)$
Output: $(1/2)$-approximation of the MWM of G

```
1: for each u ∈ V do
2:     mate(u) ← null
3:     suitor(u) ← null
4: for each u ∈ V do
5:     FINDSUITOR(u)                                   ▷ Algorithm 20
```

by the ones by Maue and Sanders [206] and by Stubbs and Williams [275]. Experimental data indicate that the approach by Maue and Sanders [206] – which combines random walks with dynamic programming to find augmenting paths – yields the best results in terms of running time and quality. Hereafter, as done in Ref. [16], we call this algorithm DynMWMRandom.

9.2.3 The Static Suitor Algorithm

Since our dynamic algorithm is built on top of the static one called Suitor [203], for self-containment purposes we provide in the following a brief overview of this latter algorithm.

Recall that w is the weight function of the edge. By convention, if $\{u, v\} \notin E$, or if a vertex reference v is `null`, then $w(u, v)$ is 0. To guarantee a correct execution, we assume that all vertices in the graph are indexed from 1 to n; in addition, the following total ordering of the edges incident to the same vertex u is enforced: if $\{u, x\}$ and $\{u, y\}$ have the same edge weight and $x < y$, than $w(u, x) < w(u, y)$. During the course of the algorithm, Suitor keeps two references for each vertex $u \in V$: *mate*(u) and *suitor*(u). When the execution is finished, *mate*(u) refers to the matching partner of u or is `null` if u remains unmatched. The crucial condition for selecting a matching partner is given by:

$$\begin{aligned} \textit{mate}(u) = \operatorname*{argmax}_{v \in N(u)} \{ & w(u, v) : \nexists x \in N(v) \text{ s.t.} \\ & \textit{mate}(x) = v \wedge w(x, v) > w(u, v) \}. \end{aligned} \tag{9.1}$$

In other words, $v = \textit{mate}(u)$ is the neighbor of u for which there is no other neighbor x of v "pointing" to v with $\{u, x\}$ dominating $\{u, v\}$. If no such vertex exists, *mate*(u) is set to `null` (i.e., u is unmatched).

Algorithm 22 Iterative findSuitor function

```
function FINDSUITOR(u)
    cur ← u
    done ← false
    repeat
        partner ← suitor(cur)
        heaviest ← ws(cur)
        for each v ∈ N(cur) do
            if w(cur, v) > heaviest and w(cur, v) > ws(v) then
                partner ← v
                heaviest ← w(cur, v)
        done ← true
        if heaviest > 0 then
            y ← suitor(partner)
            suitor(partner) ← cur
            ws(partner) ← heaviest
            if y ≠ null then
                cur ← y
                done ← false
    until done is true
```

Algorithm 23 Static iterative Suitor algorithm [203]

Input: Undirected graph $G = (V, E, w)$
Output: $(1/2)$-approximation of the MWM of G

```
for each u ∈ V do
    suitor(u) ← null
    ws(u) ← 0
for each u ∈ V do
    FINDSUITOR(u)                                   ▷ Algorithm 22
```

Algorithm 21 shows the pseudocode of the Suitor algorithm: *mate* and *suitor* are initially set to `null` for every vertex in the graph; then, the recursive function findSuitor (Algorithm 20) sets *mate* according to Eq. (9.1) for every vertex. The progress of the algorithm can be described in the following Lemma:

Lemma 9.2.1 ([203], Lemma 3.2)**.** Following each call to findSuitor in Algorithm 21 from the loop over the vertices of G, $mate(u)$ is set according to Eq. (9.1) for each vertex u processed so far.

Clearly, after the execution of Algorithm 21, the condition in Eq. (9.1) is true for all vertices and this leads us to the next property of the resulting matching:

Lemma 9.2.2 ([203], Lemma 3.1)**.** If $mate(u)$ is set according to Eq. (9.1) for each vertex $u \in V$, then $mate(\cdot)$ defines the same matching as the greedy algorithm.

Proof. See Ref. [163, Section 3.2]. The proof is for b-matching, but contains the MWM problem by setting b to 1. □

Thus, regardless of the order in which the `for`-loop (Line 4, Algorithm 21) processes the vertices, Suitor is a deterministic algorithm that computes the same matching as the well-known greedy algorithm that adds permissible edges in the order of decreasing weight. Due to our assumption of a total edge ordering, this matching is unique.

Algorithm 23 shows the iterative version of Suitor, which does not use *mate* any more but only *suitor*. An additional array ws stores the value $w(u, suitor(u))$ for each vertex $u \in V$. We rewrite the condition in Eq. (9.1) in terms of *suitor* as follows:

$$\begin{aligned} suitor(u) = \underset{v \in N(u)}{\text{argmax}}\{w(u,v) : \nexists y \in N(v) \text{ s.t.} \\ suitor(v) = y \wedge w(y,v) > w(u,v)\}. \end{aligned} \tag{9.2}$$

If no such vertex exists, *suitor*(u) is `null`. Similarly to its recursive counterpart, for every vertex $u \in V$, Algorithm 23 initializes *suitor* and ws to `null` and 0, respectively, and calls the iterative function findSuitor(u) (Algorithm 22). findSuitor uses a variable *cur* to store the vertex that is seeking a new partner in the current iteration – i.e., a vertex u in the recursive findSuitor. In Lines 7–10, it determines whether there exists a neighbor v of *cur* and, if so, it stores v and $w(cur, v)$ into the *partner* and *heaviest* variables, respectively. In Line 12, if *heaviest* is 0, no such neighbor exists and the function terminates because *done* is left to `true`– hence, *cur* remains unmatched. Otherwise, *partner* was set to the matching partner of *cur* and *heaviest* to $w(cur, partner)$; in Lines 13–15, y stores the previous matching partner of *partner* (if any) before making *cur* the new matching partner of *partner* by setting *suitor*(*partner*) to *cur* and ws(*partner*) to *heaviest*. Then, in Lines 16–18, if y is not `null`, then *partner* had a previous potential matching partner y; therefore y needs to seek a new potential matching partner and this is done by setting *cur* to y and *done* to `false` – which is equivalent to a recursive call of Algorithm 20.

9.3 Dynamic Suitor Algorithm for Single Edge Updates

In this section, we first describe how we extend the static Suitor algorithm to also handle single edge updates. Building upon that, we generalize our approach to multiple edge updates in batches in Section 9.4.

We index variables of the Suitor algorithm with the superscript $^{(i)}$ (for intermediate) [or $^{(f)}$ for final, resp.] if they refer to the state directly after the edge change [or after the dynamic Suitor algorithm has been run], e.g., $suitor^{(i)}(u)$ [or $suitor^{(f)}(u)$]. The matching $M^{(i)}$ as well as other Suitor variables are derived from M (and the other counterparts) by taking the edge update on G into account. For example, if an edge e is deleted from G that is part of M, then $M^{(i)} = M \setminus \{e\}$. Our goal is to update/improve the intermediate matching $M^{(i)}$ efficiently, i.e., to avoid redundant computations when computing the final matching $M^{(f)}$. We will show that $M^{(f)}$ equals the matching M' computed by the static Suitor algorithm on G'. To this end, we define the notion of *affected* vertex.

Definition 9.3.1. A vertex u is called *affected* iff $M^{(i)}$ violates Eq. (9.2) for $suitor^{(i)}(u)$, i.e., iff:

$$suitor^{(i)}(u) \neq \underset{v \in N'(u)}{\text{argmax}}\{w'(u,v) : \nexists y \in N'(v) \text{ s.t. } suitor^{(i)}(v) = y \wedge w'(y,v) > w'(u,v)\}.$$

Our dynamic algorithm computes $M^{(f)}$ after an edge update by finding all the vertices affected by the edge update (findAffected function in Algorithm 24) and by then updating their matching partner so that Eq. (9.2) is satisfied (updateAffected function in Algorithm 25). If Eq. (9.2) is satisfied for every vertex in G', it follows from Lemma 9.2.2 that the resulting matching is (the unique) M'.

Algorithm 24 Extended version of the findSuitor function (Algorithm 22) that finds the affected vertices

Input: Affected vertex z
Output: Stack of affected vertices

```
1:  function FINDAFFECTED(z)
2:      S_A ← empty stack
3:      cur ← z
4:      done ← false
5:      repeat
6:          partner ← suitor(cur)
7:          heaviest ← ws(cur)
8:          for each x ∈ N(cur) do
9:              if not affected(x) and w(cur, x) > heaviest and w(cur, x) > ws(x) then
10:                 partner ← x
11:                 heaviest ← w(cur, x)
12:         done ← true
13:         if heaviest > ws(partner) then
14:             y ← suitor(partner)
15:             suitor(partner) ← cur
16:             ws(partner) ← heaviest
17:             S_A.push(partner)
18:             affected(partner) ← true
19:             if y ≠ null then
20:                 suitor(y) ← null
21:                 ws(y) ← 0
22:                 affected(y) ← true
23:                 cur ← y
24:                 done ← false
25:         else
26:             affected(cur) ← false
27:     until done is true
28:     return S_A
```

The findAffected function (Algorithm 24) is an extended version of the findSuitor function (Algorithm 22) used by the iterative Suitor algorithm; it uses a boolean array *affected* to keep track of the affected vertices and pushes the affected vertices whose *suitor* and ws variables need to be updated onto a stack S_A. As in findSuitor, *cur* is the vertex we are trying to find a new matching partner for, *partner* is the preferred matching partner for *cur* – i.e., the vertex that satisfies Eq. (9.2) for *cur*, if any – and $heaviest = w'(cur, partner)$. Then, in Lines 15–16, if a new matching partner for *cur* is found, the $suitor^{(f)}(partner)$ and $ws^{(f)}(partner)$ variables are updated to *cur* and *heaviest*, respectively; additionally, *partner* is pushed onto S_A (Line 17). If *partner* is matched in M with another vertex y, then the edge $\{partner, y\}$ would violate the matching condition and this needs to be removed from the matching; this is done in Lines 20–21 by "invalidating" the vertex y, i.e., setting the values of $suitor^{(f)}(y)$ and $ws^{(f)}(y)$ to `null` and 0, respectively. As in findSuitor, in the next iteration we seek for a new partner for the invalidated vertex y by updating *cur* to y. This procedure is repeated until either a new partner for *cur* cannot be found (i.e., *cur* is free in M'), or *partner* is free in M. By keeping track of the affected vertices, we guarantee that a previously invalidated vertex is not selected as new matching partner in future iterations of the loop (see Line 9).

The stack S_A is later used by updateAffected (Algorithm 25) to match the affected vertices that were not updated by findAffected (i.e., the ones that were stored in the *cur* variable) to their new partner (Lines 4–5). Once a pair of matched vertices has eventually been processed, they are not affected anymore and thus

Algorithm 25 Updates *suitor* and ws of the matching partners of the vertices into the stack S_A

Input: Stack of affected vertices S_A

```
while S_A is not empty do
    x ← S_A.pop()
    y ← suitor(x)
    suitor(y) ← x
    ws(y) ← ws(x)
    affected(x) ← false
    affected(y) ← false
```

Algorithm 26 Dynamic Suitor algorithm for single edge insertions

Input: Graph $G' = (V, E \cup \{u, v\}, w')$, an edge $\{u, v\} \notin E$

```
1: if w'(u,v) > max{w'(u, suitor^(i)(u)), w'(v, suitor^(i)(u))} then
2:     affected(x) ← false ∀x ∈ V
3:     for z ∈ {v,u} do
4:         affected(z) ← true
5:         S_A ← findAffected(z)
6:         updateAffected(S_A \ {suitor^(i)(z)})
```

updateAffected marks them as unaffected (Lines 6–7). In the following, we show how these two functions are used in case of an edge insertion or an edge removal and that our dynamic algorithm yields a matching $M^{(f)}$ that equals the matching M' computed by the static Suitor algorithm on G'. Concerning edge weight updates, they can be handled as an edge removal followed by an edge insertion.

9.3.1 Edge Insertions

Let us first address the case in which an edge is inserted into G, i.e., $G' = (V, E, \cup\{u, v\})$ with $e = \{u, v\} \notin E$. Intuitively, this new edge will only be part of the new matching M' iff it is "a better deal" for both u and v. In other words: $e \in M'$ iff $w'(u, v)$ is heavier than both $w'(u, \mathit{suitor}^{(i)}(u))$ and $w(v, \mathit{suitor}^{(i)}(v))$ and is thus the dominant edge for both. We show this in the following lemma:

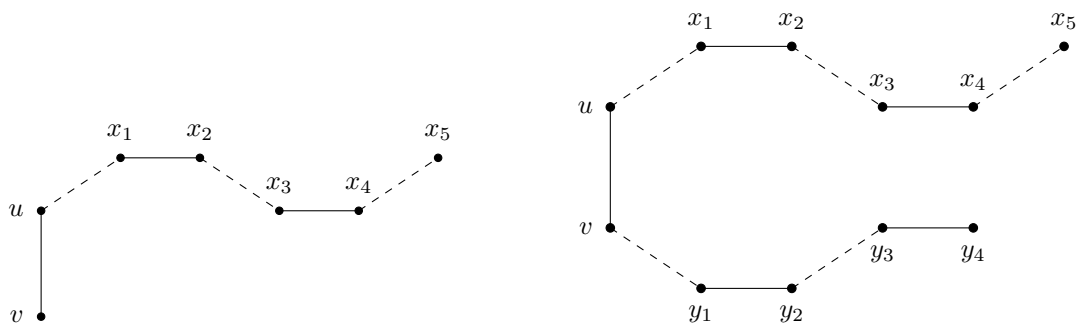
Lemma 9.3.1. Let $G' = (V, E \cup e)$ with $e = \{u, v\} \notin E$.
Then: $e \in M' \Leftrightarrow w'(u, v) > \max\{w(u, \mathit{suitor}^{(i)}(u)), w(v, \mathit{suitor}^{(i)}(v))\}$.

Proof. We develop the proof w.l.o.g. for u by assuming that $w(u, \mathit{suitor}^{(i)}(u)) > w(v, \mathit{suitor}^{(i)}(v))$; the proof for v is symmetric. Let $Y \subseteq N(u)$ be the set of vertices y that do not have any neighbor x such that $w(x, y) > \max\{w(x, \mathit{suitor}^{(i)}(x)), w(u, y)\}$, i.e., the set of vertices among which Eq. (9.2) selects $\mathit{suitor}(u)$ as the vertex y with maximum $w(y, u)$. We also define $Y^{(v)} := Y \cup \{v\}$.

"$\Rightarrow$": From $e \in M'$ we have that $\mathit{suitor}'(v) = u$, and from Eq. (9.2) it follows that $v = \operatorname{argmax}_{y \in Y^{(v)}} w'(u, y)$. Therefore, since $\mathit{suitor}^{(i)}(u) \in Y^{(v)}$, $w'(u, v) > w'(u, \mathit{suitor}^{(i)}(u))$.

"$\Leftarrow$": By definition, we have that $\mathit{suitor}^{(i)}(u) = \operatorname{argmax}_{y \in Y} w'(u, y)$ and that $\mathit{suitor}'(u) = \operatorname{argmax}_{y \in Y^{(v)}} w'(u, y)$. By hypothesis, $w'(u, v) > w(u, \mathit{suitor}^{(i)}(u))$, and thus $\mathit{suitor}'(u) = \operatorname{argmax}_{y \in Y^{(v)}} w'(u, y) = v$. The same holds if we exchange u and v in the argument. Therefore, $e \in M'$. □

Algorithm 26 shows our dynamic algorithm for single edge insertions. In Line 1, given a newly added edge $e = \{u, v\}$, the algorithm excludes e if it does not satisfy Lemma 9.3.1 because e is then also not part of M'. As we will show later, all the vertices affected by an edge insertion lie on two alternating paths that



(a) The affected vertices are covered by one alternating path P_v.

(b) The affected vertices are covered by two alternating paths P_u and P_v.

Figure 9.1: Examples of alternating paths that cover the vertices affected by the insertion of an edge $\{u, v\}$. Solid lines and dashed lines represent edges in $M' \setminus M$ and edges in $M \setminus M'$, respectively. In Figure 9.1a there is only one alternating path because u is matched in M and v is not, whereas in Figure 9.1b there are two because both u and v are matched in M.

start from u and v and that alternate edges in $M' \setminus M$ with edges in $M \setminus M'$, as shown in Figure 9.1. In the `for`-loop, our algorithm finds the affected vertices that lie on these two alternating paths and updates their matching partner according to Eq. (9.2). This is done as follows: the first vertex in the path is marked as affected, findAffected updates the *suitor* and ws variables of the affected *partner* vertices and pushes them onto a stack S_A. Finally, updateAffected matches the vertices in S_A to their new partner. Note that, in Line 6, $\mathit{suitor}^{(i)}(z)$ is removed from S_A to avoid overwriting $\mathit{suitor}(z)$ in updateAffected; this information is needed in the next iteration of the for loop to find the affected vertices in the alternating path that starts from u.

We now analyze in more detail which vertices are affected by the insertion of an edge $e = \{u, v\} \notin E$ that satisfies Lemma 9.3.1 (and is thus in M') and how $M^{(f)}$ is computed starting from $M^{(i)}$. In the following, we split our analysis into three possible scenarios: both u and v are unmatched (Case 1), only one of u and v is matched (Case 2), and both u and v are matched (Case 3).

9.3.1.1 Case 1: u and v both unmatched

Let us first cover the trivial case where both u and v are unmatched in M, i.e., there is no vertex in G that satisfies Eq. (9.2) for both u and v. In the following Lemma, we show that $M' = M^{(i)} \cup \{e\}$ and that u and v are the only affected vertices.

Lemma 9.3.2. Let $e = \{u, v\} \notin E$. If both u and v are unmatched in M, then $M' = M \cup \{e\}$ and u and v are the only affected vertices.

Proof. The proof is symmetric for u and v, we develop it for u. If u is unmatched in M, then $\mathit{suitor}^{(i)}(u) =$ `null` and there is no neighbor of u that satisfies Eq. (9.2) in G. After the insertion of e we have that $0 = w'(u, \mathit{suitor}^{(i)}(u)) < w'(u, v)$ and thus v satisfies Eq. (9.2) for u. Further, all the neighbors of u in G are already matched, and matching u with v cannot invalidate Eq. (9.2) for any of them. Therefore, $M' = M^{(i)} \cup \{e\}$ and no other vertex apart from u and v is affected. □

We now show that $M^{(f)}$ equals M', i.e., Eq. (9.2) is fulfilled by $M^{(f)}$ for every vertex in G'. The condition in Line 1 in Algorithm 26 is clearly true because both u and v are free. In findAffected(v), we have that

cur $= v$ and *partner* $= u$. Thus, *suitor*$^{(f)}(u)$ is set to v and u is pushed onto S_A. findAffected(v) performs only one iteration since $y =$ *suitor*$^{(i)}(u) =$ `null` and updateAffected has no effect since its input stack is empty. The next iteration of the for loop performs the same operations but with u and v swapped. Thus, the resulting matching is $M^{(f)} = M^{(i)} \cup \{e\} = M'$.

9.3.1.2 Case 2: u matched and v unmatched

To analyze the case where just one of u and v is matched in M, our analysis assumes w.l.o.g. that u is matched in M and v is not; the other case is symmetric. We first describe how Algorithm 26 identifies all the affected vertices and then how it updates their matching partner.

In this scenario, u and v are not the only vertices affected by the insertion of e because Eq. (9.2) is violated also for *suitor*$^{(i)}(u)$. In particular, we show in Lemma 9.3.3 that the vertices affected by the insertion of e are covered by a simple (i.e., without loops) path that, starting from v, alternates edges in $M' \setminus M$ with edges in $M \setminus M'$ as shown in Figure 9.1a. We show this in the following lemma.

Lemma 9.3.3. Let $e = \{u, v\} \notin E$ be a newly inserted edge such that $e \in M'$. If u is matched in M and v is not, then all the vertices affected by the insertion of e are connected by a simple alternating path P_v that starts from v and that alternates edges in $M' \setminus M$ with edges in $M \setminus M'$. Further, the weights of the edges along P_v are decreasing, i.e., for each $e_1, e_2 \in P_v$ where e_1 precedes e_2 in P_v, we have that $w'(e_1) > w'(e_2)$.

Proof. Clearly, $e \in M' \setminus M$ is the first edge in P_v. As shown in Figure 9.1a, let $x_1 =$ *suitor*$^{(i)}(u)$: $e \in M'$ implies that Eq. (9.2) is violated for x_1 in G'. If no vertex $x_2 \in N'(x_1)$ satisfies Eq. (9.2) for x_1, then x_1 remains unmatched in M' and no further vertex is affected. Otherwise, there exists another vertex $x_2 \in N'(x_1)$ that satisfies Eq. (9.2) for x_1. In the former case, the alternating path has only two edges: e and $e_1 = \{u, x_1\} \in M \setminus M'$. In the latter case, *suitor*$^{(f)}(x_1) = x_2$ and *suitor*$^{(f)}(x_2) = x_1$. Hence, in addition to e_1, the alternating path has at least another edge $e_2 = \{x_1, x_2\} \in M' \setminus M$. By repeating with e_1 the same logic we applied to e, it follows that the vertices affected by the insertion of e lie on a path P_u that, starting from u, follows edges in M' and in M alternately.

What's left to be shown is that P_v is simple, which can be done similarly as in the final part of the proof of [203, Lemma 3.2]. Note that, if x_1 is affected, then $w'(u, x_1) < w'(u, \textit{suitor}^{(f)}(u) = v)$ and $w'(x_1, \textit{suitor}^{(f)}(x_1) = x_2) < w'(u, x_1)$, and thus $w'(x_1, \textit{suitor}^{(f)}(x_1)) < w'(u, \textit{suitor}^{(f)}(u))$. In words, the weights of the edges along P_v are decreasing because every time an affected vertex x_1 loses its matching partner *suitor*$^{(i)}(x_1)$, it holds: if it finds a new partner *suitor*$^{(f)}(x_1) = x_2$, then $w'(x_1, x_2)$ must be smaller than $w'(x_1, \textit{suitor}^{(i)}(x_1))$. Therefore, it is not possible for x_1 to be matched in M' with a vertex that is already covered by P_v, which implies that P_v is simple. □

Note that, as shown in Figure 9.1a, the alternating path $P_v = (u, v, x_1, \ldots, x_\ell)$ alternates vertices x_i that are matched in M' with a "worse partner" (in terms of edge weight) than the one they had in M (i.e., the ones where i is odd), and vertices x_i that are matched in M' with a "better partner" than the one they had in M (i.e., u, v, and the ones where i is even). Hereafter, we will call the former ones "downgraded" and the latter ones "upgraded". More formally:

Lemma 9.3.4. For each vertex x_i in an alternating path $P_v = (v, u, x_1, \ldots, x_\ell)$, for each $1 \leq i \leq \ell$ it holds: if i is odd, then $w'(x_i, \mathit{suitor}^{(f)}(x_i)) < w'(x_i, \mathit{suitor}^{(i)}(x_i))$ (i.e., x_i is downgraded); otherwise, i is even and $w'(x_i, \mathit{suitor}^{(f)}(x_i)) > w'(x_i, \mathit{suitor}^{(i)}(x_i))$ (i.e., x_i is upgraded).

Proof. Every downgraded vertex x_i loses its initial matching partner $\mathit{suitor}^{(i)}(x_i)$ because $\mathit{suitor}^{(i)}(x_i)$ is matched in M' with another (upgraded) vertex – e.g., x_1 loses its initial matching partner u because $\{u, v\} \in M'$. Note that, in P_v, the upgraded vertex matched with $\mathit{suitor}^{(i)}(x_i)$ in M' comes always earlier than x_i; from Lemma 9.3.3 we know that the weights of the edges along P_v are decreasing, and thus $w'(x_i, \mathit{suitor}^{(i)}(x_i)) < w'(x_i, \mathit{suitor}^{(f)}(x_i))$. Further, by construction of the alternating path, P_v alternates downgraded and upgraded vertices from x_1 on. Thus, knowing that x_1 is downgraded, all the remaining x_i with odd i are also downgraded.

If x_i is upgraded, it is either one of u and v, or it is $\mathit{suitor}^{(f)}(x_{i-1})$ of the previous downgraded vertex x_{i-1} in P_v. Since by our hypothesis x_i is in P_v, we also have that $\mathit{suitor}^{(f)}(x_i) = x_{i-1}$, hence $w'(x_i, \mathit{suitor}^{(f)}(x_i) = x_{i-1}) > w'(x_i, \mathit{suitor}^{(i)}(x_i))$. □

Remark 9.3.1. Due to the total ordering of the edge weights, the alternating path P_v is unique and can be computed deterministically as the sequence of the vertices stored in the cur and $partner$ variables of the findSuitor(v) function (Algorithm 22).

From Lemma 9.3.3 it follows that, to compute M', we need to find the affected vertices in the alternating path P_v and update their $\mathit{suitor}^{(f)}$ and $ws^{(f)}$ values according to Eq. (9.2). In the following we show that Algorithm 26 finds all the vertices in P_v and updates their matching partner according to Eq. (9.2), so that $M^{(f)}$ equals M'.

Proposition 9.3.1. The findAffected(v) function (Algorithm 24) computes $\mathit{suitor}^{(f)}$ and $ws^{(f)}$ according to Eq. (9.2) for the upgraded vertices in P_v and pushes them onto a stack S_A.

Proof. The function maintains the following loop invariant: *cur* is either v or an invalidated downgraded vertex $x_i \in P_v$; in the first case, *partner* is u; in the second case, if the condition in Line 13 is true, then *partner* is an improving vertex $x_{i+1} \in P_v$ – otherwise both the loop and P_v stop. Maintaining the invariant guarantees that findAffected(v) covers all affected vertices in P_v and that all the improving vertices x_i are updated according to Eq. (9.2) and pushed onto S_A.

The invariant obviously holds in the first two iterations. In the first one, we have that $\mathit{cur} = v$, $\mathit{partner} = u$, and $y = \mathit{suitor}^{(i)}(u) = x_1$ (see Figure 9.1a). In the next one, $\mathit{cur} = x_1$ is invalidated from the previous iteration, and thus it is a downgraded vertex in P_v that is seeking a new partner to replace the previous partner u; if a new partner x_2 is found, $\mathit{partner} = x_2$ is an upgraded vertex in P_v, thus $\mathit{suitor}^{(f)}(x_2)$ and $ws^{(f)}(x_2)$ are updated to x_1 and $w'(x_1, x_2)$, respectively, and it is pushed onto S_A (Lines 15–17). By applying the same logic to the remaining vertices in P_v, the invariant also holds for the remaining iterations of findAffected(v). We remark that, by marking the vertices in P_v as affected, the function cannot iterate on the same vertices multiple times. The function terminates when either *partner* is a free vertex (hence $y = \texttt{null}$), or there is no neighbor of *cur* that satisfies Eq. (9.2), and thus *cur* remains unmatched in $M^{(f)}$. □

Note that, if P_v ends with a downgraded (hence free) vertex x_ℓ, findAffected(v) invalidates it before terminating, i.e., *suitor*$^{(f)}(x_\ell)$ and $ws^{(f)}(x_\ell)$ are updated according to Eq. (9.2). Therefore, x_ℓ is not affected anymore in G' and thus marked as unaffected (Line 26).

Proposition 9.3.2. After updateAffected finishes, all vertices in P_v satisfy Eq. (9.2) in G'.

Proof. From Proposition 9.3.1 we know that, when updateAffected is called, all the upgraded vertices satisfy Eq. (9.2) in G' and that they are stored in the stack S_A. updateAffected "completes" the matching $M^{(f)}$ by updating, for all the downgraded vertices $x_i \in P_v$ matched in M', their *suitor*$^{(f)}(x_i)$ and $ws^{(f)}(x_i)$ values to x_{i+1} and $w'(x_i, x_{i+1})$, respectively, and thus all vertices in P_v satisfy Eq. (9.2) in G'. □

Proposition 9.3.3. If u is matched in M and v is not (or vice versa), Algorithm 26 has a worst-case running time that is linear in the number of the affected vertices and in the sum of their degrees.

Proof. Algorithm 26 invokes findAffected and updateAffected twice. The number of iterations of findAffected is linear in the length of a simple alternating path that covers the vertices affected by the edge insertion (Lemma 9.3.3). In each iteration, findAffected iterates over all the neighbors of the current vertex (Line 8) and all the other operations have constant time complexity.

updateAffected performs constant-time operations on each vertex in S_A. findAffected pushes at most one vertex onto S_A in each iteration, and thus the worst-case time complexity of updateAffected is linear in the number of affected vertices. □

Remark 9.3.2. In the worst case, all vertices in the graph are affected by an edge insertion and then findAffected performs at most n iterations and visits all edges twice. Therefore, the worst-case time complexity of Algorithm 26 is $\mathcal{O}(n + m)$.

9.3.1.3 Case 3: u and v both matched

To settle the final case, we show (i) that the affected vertices are covered by two alternating paths P_v and P_u and then (ii) that the matching $M^{(f)}$ computed by Algorithm 26 equals M'.

Lemma 9.3.5. Let $\{u, v\} \notin E$ be a newly inserted edge such that $e \in M'$. If both u and v are matched in M, then all the vertices affected by the insertion of e are connected by two simple alternating paths P_v and P_u with decreasing edge weights that start from v and u, respectively, and that alternate edges in $M' \setminus M$ and edges in $M \setminus M'$.

Proof. We proceed similarly as in Lemma 9.3.3. As shown in Figure 9.1b, $e \in M' \setminus M$ is the first edge for both P_v and P_u. Thus, Eq. (9.2) is violated for both $x_1 = \textit{suitor}^{(i)}(u)$ and $y_1 = \textit{suitor}^{(i)}(v)$ (due to the matching condition, $x_1 \neq y_1$). Consequently, the edges $e_{1,u} = \{u, x_1\}$ and $e_{1,v} = \{v, y_1\}$ are removed from $M^{(i)}$, which implies that $e_{1,u}, e_{1,v} \in M \setminus M'$. Now let x_2 and y_2 be the two vertices (if any) that satisfy Eq. (9.2) for x_1 and y_1, respectively, in G'. Clearly, we have that $\{x_1, x_2\}, \{y_1, y_2\} \in M' \setminus M$. If x_2 and/or y_2 are matched in M, then we can apply recursively to their matching partners the same logic as we did with x_1 and y_1. Hence, P_v and P_u are constructed exactly as described in Lemma 9.3.3 and thus they are both simple and have decreasing edge weights. □

In the following, we show that the vertices along P_v and P_u are computed in the two iterations of Algorithm 26. A crucial observation is that our dynamic algorithm computes one alternating path at a time – P_u is computed after P_v. Therefore, when a new *partner* vertex for *cur* is found while computing P_v (i.e., in findAffected(v)), this might not be the actual vertex that satisfies Eq. (9.2) for *cur* in G' because it is computed without considering the affected vertices in P_u – they are yet to be computed. If this is the case, then it results in a wrong computation of P_v and thus, immediately after the first iteration of Algorithm 26, some of the vertices covered by findAffected(v) are in a "wrong state", i.e., Eq. (9.2) is locally satisfied for them but not for all the vertices in G'. However, we show later that, in this case, the second iteration of Algorithm 26 not only updates the affected vertices in P_u, but also corrects the vertices that are in a wrong state, so that eventually all vertices in G' fulfill Eq. (9.2) and thus $M^{(f)} = M'$. For this purpose, we expand our notation by denoting the values of the variables of Suitor and P_v immediately after the first iteration of Algorithm 26 with superscript $^{(ii)}$.

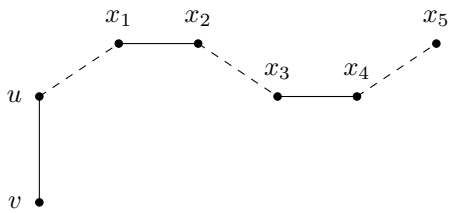
Clearly, if in $P_v^{(ii)}$ there are no vertices in a wrong state, then $P_v^{(ii)} = P_v$ and P_u is computed exactly in the same way as P_v (as in Case 2) and no further analysis is required. Otherwise, the two paths intersect: let y_i be the last vertex in P_u before P_u intersects $P_v^{(ii)}$ and let x_j be the vertex in $P_v^{(ii)}$ adjacent to y_i (e.g., vertices x_5 and y_5 in Figure 9.2b). We observe that an intersection always happens when findAffected(u) selects a vertex in $P_v^{(ii)}$ as new matching partner for the current vertex y_i in P_u. Therefore, y_i is always a downgraded vertex and x_j satisfies Eq. (9.2) for y_i in G'. Depending on the position of x_j in $P_v^{(ii)}$, we identify three possible subcases and treat them separately: (i) x_j is downgraded and it is the last vertex in $P_v^{(ii)}$ (as in Figure 9.2b), (ii) x_j is downgraded and internal in $P_v^{(ii)}$ (as in Figure 9.2c), and (iii) x_j is upgraded (as in Figure 9.2d).

Subcase (i) – x_j is downgraded and it is the last vertex in $P_v^{(ii)}$. In this case (see Figure 9.2b), x_j is free in $M^{(ii)}$ because $suitor^{(ii)}(y_i) = y_{i-1}$ and no other vertex in $N'(x_j)$ satisfies the condition in Line 9 of findAffected(v) for x_j. Hence, x_j is in a wrong state because it is not matched with y_i, its actual matching partner in M'. The wrong state of x_j is corrected by findAffected(u): when P_u reaches y_i, $suitor^{(f)}(x_j)$ and $ws^{(f)}(x_j)$ are set to y_i and $w'(x_j, y_i)$, respectively. Also, x_j is pushed onto S_A, so that y_i will eventually be matched with x_j in updateAffected.

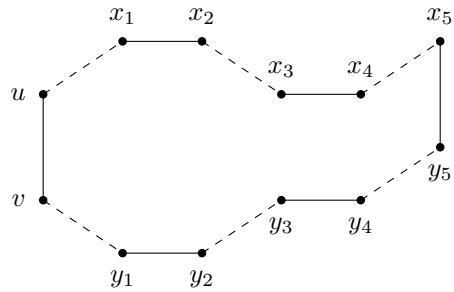
Subcase (ii) – x_j is downgraded and it is internal in P_u. Here, x_j is in a wrong state for the same reason as in subcase (i). The main difference (see Figure 9.2c) is that $P_v^{(ii)}$ does not finish in x_j because there exists a vertex *partner* $= x_{j+1} \in N'(x_j)$ that satisfies the conditions in Lines 9 and 13 in findAffected(v). Thus, all the vertices in $P_v^{(ii)}$ after x_j are in a wrong state as well, because they are not affected, but their matching partner is updated by Algorithm 26. However, in the following lemma we show that the second iteration of the algorithm matches x_j to its matching partner in M' and restores the original matching partners of the vertices in $P_v^{(ii)}$ after x_j.

Lemma 9.3.6. The second iteration of Algorithm 26 matches x_j with its matching partner in M' and restores the original matching partners of the vertices in $P_v^{(ii)}$ after x_j.

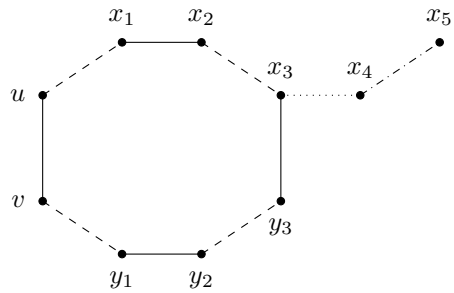
Proof. As in case (i), x_j fulfills Eq. (9.2) for y_i and once findAffected(u) reaches y_i it matches x_j to y_i. Recall that the vertices in $P_v^{(ii)}$ after x_j are in a wrong state in $M^{(ii)}$, because their matching has been



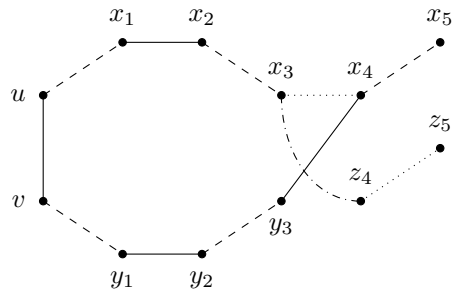
(a) Affected vertices immediately after the computation of $P_v^{(ii)}$ and before the computation of P_u. This represents $M^{(ii)}$, not $M^{(f)}$ since P_u is yet to be computed.



(b) Case 1: matching $M^{(f)}$ after P_u is computed. The last vertex x_5 in $P_v^{(ii)}$ is free in $M^{(ii)}$, P_u intersects $P_v^{(ii)}$ in x_5, and x_5 is matched with y_5 in $M^{(f)}$.



(c) Case 2: matching $M^{(f)}$ after P_u is computed. P_u intersects $P_v^{(ii)}$ in x_3, which is internal in $P_v^{(ii)}$. $P_v^{(ii)}$ from x_4 on is undone.



(d) Case 3: matching $M^{(f)}$ after P_u is computed. P_u intersects $P_v^{(ii)}$ in x_4 which is upgrade and continues with x_3.

Figure 9.2: Examples of intersecting alternating paths computed by Algorithm 26 to update the matching after the insertion of an edge $\{u, v\}$. Figure 9.2a shows the status of the affected vertices after the computation of $P_v^{(ii)}$ and before the computation of P_u. Figures 9.2b–9.2d show the three possible cases of intersection between $P_v^{(ii)}$ and P_u. Dashed and solid edges have the same meaning as in Figure 9.1, dotted edges are in $M^{(ii)} \setminus M^{(f)}$, dash-dotted edges are in $M^{(f)} \setminus M^{(ii)}$.

updated, although they are not affected by the edge insertion. Hence, we need to show that for these vertices, findAffected(u) restores the matching partner they have in M – i.e., the same as in M' since they are not affected.

Let x_{j+1} be *suitor*$^{(i)}(x_j)$; in the iteration where *partner* is x_j, findAffected(u) keeps iterating with *cur* $= x_{j+1}$. Hereafter, findAffected(u) maintains the following invariant: *cur* is an upgraded vertex in $M^{(i)}$ and, if *heaviest* $> ws^{(ii)}$(*partner*), then *partner* is the matching partner of *cur* in M; otherwise *cur* is free in M'. Due to the properties of the alternating path we have that, in $M^{(i)}$, x_{j+1} is an upgraded vertex, because it is matched with x_j and $w'(x_j, x_{j+1}) > w'(x_{j+1}, x_{j+2})$. In M', in turn, we have that x_{j+1} is either free or matched with another vertex x_{j+2}. In the first case, findAffected(u) stops at x_{j+1} and leaves it unmatched, because there is no other vertex in $N'(x_{i+1})$ that satisfies Eq. (9.2). Otherwise, we need to show that findAffected(u) selects x_{j+2} as matching partner for x_{j+1}. From Lemma 9.3.5, the vertices on $P_v^{(ii)}$ and P_u before x_{j+1} satisfy Eq. (9.2) and thus cannot be selected as partners for x_{j+1}. Furthermore, x_{j+2} satisfies Eq. (9.2) for x_{j+1} in G', and it is downgraded in $M^{(ii)}$ (hence $w'(x_{j+2}, \textit{suitor}^{(ii)}(x_{j+2})) < w'(x_{i+1}, x_{j+2})$), which means that the conditions in Lines 9 and 13 hold. Also, the edge weights are decreasing along $P_v^{(ii)}$ (Lemma 9.3.5), implying that no other neighbor of x_{i+1} in $P_v^{(ii)}$ satisfies Eq. (9.2) for x_{j+1} in G'; hence, $\{x_{i+1}, x_{i+2}\} \in M^{(f)}$. The same applies to the remaining vertices in $P_v^{(ii)}$. □

Algorithm 27 Dynamic Suitor algorithm for single edge removals

Input: Graph $G' = (V, E \setminus \{u, v\}, w')$, an edge $\{u, v\} \in E$

```
if suitor(u) = v then
    affected(x) ← false ∀x ∈ V
    suitor(u) ← null
    suitor(v) ← null
    ws(u) ← 0
    ws(v) ← 0
    for z ∈ {u, v} do
        affected(z) ← true
        S_A ← FINDAFFECTED(z)
        UPDATEAFFECTED(S_A)
```

SUBCASE (III) – x_j IS UPGRADED.. In this case (see Figure 9.2d), we have that $\{x_{j-1}, x_j\} \in M^{(ii)}$ and that $w'(y_i, x_j) > w'(x_{j-1}, x_j) > w'(x_j, x_{j+1})$. Thus, x_{j-1} is in a wrong state because $\{y_i, x_j\} \in M^{(f)}$, whereas the remaining part of $P_v^{(ii)}$ from x_j on is not necessarily wrong, because x_j is upgraded also in P_u. findAffected(u) corrects x_{j-1} by continuing the alternating path from it: once *cur* $= y_i$ is matched with x_j, in the next iteration of findAffected(u) we have that *cur* is x_{j-1}. Let z_j be the vertex that satisfies Eq. (9.2) for x_{j-1} in G' (e.g., z_4 in Figure 9.2d). If no such vertex exists, then x_{j-1} is free in $M^{(f)}$ and findAffected(u) stops; otherwise, P_u continues with z_j.

We covered now all possible cases that can occur after an edge insertion. In the following lemma we generalize our results.

Proposition 9.3.4. After an edge insertion, Algorithm 26 computes M'; its worst-case time complexity is $\mathcal{O}(n + m)$.

Proof. The correctness of the resulting matching is shown for every possible case of edge insertion described in Case 1 to 3. Concerning the worst-case time complexity, from Proposition 9.3.3 and Remark 9.3.2 it follows that the first iteration of Algorithm 26 has $\mathcal{O}(n + m)$ worst-case running time. The same also holds for the second iteration: due to Lemma 9.3.5 P_u is simple, so that its length is bounded by $\mathcal{O}(n)$, and therefore the worst-case time complexity of Algorithm 26 is $\mathcal{O}(n + m)$. □

Unfortunately, a bound using the number of affected vertices and their degrees as in Proposition 9.3.3 is difficult to determine here due to the vertices in a "wrong" state.

9.3.2 EDGE REMOVALS

We now address the case in which an edge $\{u, v\} \in E$ is removed from G, i.e., $G' = (V, E \setminus \{u, v\})$.

Lemma 9.3.7. Let $G' = (V, E \setminus e)$ with $e = \{u, v\} \in E$. Then: u and v are affected $\Leftrightarrow e \in M$.

Proof. We develop the proof w.l.o.g. for u, for v it is symmetric.

"$\Leftarrow$": If $e \in M$, then u is trivially affected since *suitor*$^{(i)}(u) = v \notin N'(u)$; u thus violates Eq. (9.2) in G'.

"$\Rightarrow$": Let $Y \subseteq N(u)$ be defined as in Lemma 9.3.1, i.e., as the set of neighbors y of u among which Eq. (9.2) selects *suitor*(u) as the vertex y with maximum $w(u, y)$. Further, let $Y^{(v)} := Y \setminus \{v\}$. Let us assume for sake of contradiction that $e \notin M$. Hence, *suitor*$^{(i)}(u) = \operatorname{argmax}_{y \in Y} w(u, y) \neq v$ and, in

G', this suitor does not change by having removed v from the neighborhood of u. Hence, we have that $\mathit{suitor}'(u) = \operatorname{argmax}_{y \in Y^{(v)}} w'(u, y) = \mathit{suitor}^{(i)}(u)$, meaning that u is not affected, which contradicts our hypothesis. □

Algorithm 27 shows our dynamic algorithm for edge removals. According to Lemma 9.3.7, Line 1 excludes all the removals where the removed edge $e = \{u, v\}$ is not in M. Similarly to edge insertions, we show later that the vertices affected by an edge removal lie on two alternating paths that start from u and v and that alternate edges in $M' \setminus M$ with edges in $M \setminus M'$. If $e \in M$, both u and v are affected downgraded vertices for which we have to find a new partner. First, Lines 3–6 invalidate them. Then, as for edge insertions in Algorithm 26, the `for`-loop uses findAffected to compute the alternating paths of the affected vertices and to update the matching partner of the upgraded vertices in the path. Then, it uses updateAffected to update the matching for the downgraded vertices in the path.

Lemma 9.3.8. Let $e = \{u, v\} \in M$. If e is removed from G, then all the vertices affected by the removal of e are connected by two simple alternating paths P_u and P_v with decreasing edge weights that start from u and v, respectively, and that alternate edges in $M' \setminus M$ and edges in $M \setminus M'$.

Proof. As we did in Lemma 9.3.5, we need to show that the construction of P_u and P_v is equivalent to the construction of an alternating path that connects the vertices affected by an edge insertion as described in Lemma 9.3.3. The only difference is that, after an edge removal, the alternating paths start from a downgraded vertex rather than from an upgraded vertex. We develop our proof w.l.o.g. for u, for v it is symmetric.

Due to Lemma 9.3.8, u is affected and downgraded. If there exists a vertex x_1 that satisfies Eq. (9.2) for u in G', then x_1 is upgraded and $e_1 = \{u, x_1\} \in M' \setminus M$; otherwise, u remains free in G' and it is the only vertex in P_u. Similarly, if x_1 is matched in M with a vertex x_2, then x_2 is affected and downgraded, $e_2 = \{x_1, x_2\} \in M \setminus M'$, and we can apply to x_2 the same logic we applied to u; otherwise, x_1 is free in G and P_u has only one edge e_1. Thus, as in Lemma 9.3.3, the resulting path P_u alternates edges in $M' \setminus M$ and edges in $M \setminus M'$ as well as downgraded and upgraded vertices. □

Thus, the vertices affected by an edge removal are covered by alternating paths as the ones shown in Figure 9.1, with the difference that the solid lines represent edges in $M \setminus M'$ and the dashed lines represent edges in $M' \setminus M$.

Proposition 9.3.5. After Algorithm 27 finishes, the resulting matching $M^{(f)}$ equals M' in G'.

Proof. Algorithm 27 works analogously to Algorithm 26, namely, it uses findAffected and updateAffected to find the affected vertices along an alternating path and to compute their $\mathit{suitor}^{(f)}(\cdot)$ and $ws^{(f)}(\cdot)$. From Lemma 9.3.8 we know that the vertices affected by an edge removal are covered by two alternating paths as described in Lemma 9.3.3. Thus, the correctness of Algorithm 27 follows from the correctness of Algorithm 26. □

Proposition 9.3.6. The worst-case running time of Algorithm 27 is $\mathcal{O}(n + m)$.

As argued in Proposition 9.3.5, Algorithm 27 works analogously to Algorithm 26. Thus, they have the same worst-case time complexity $\mathcal{O}(n + m)$ – see also Proposition 9.3.4.

Algorithm 28 Dynamic Suitor algorithm for a batch of edge insertions

Input: Graph $G' = (V, E \cup B, w')$, batch of edge insertions $B = \{\{u, v\}$ s.t. $\{u, v\} \notin E\}$

1: *affected* $\leftarrow$ `false` $\forall u \in V$
2: $i \leftarrow 0$
3: $\mathit{suitor}^{[i]}(u) \leftarrow \mathit{suitor}^{(i)}(u)\ \forall u \in V$
4: $ws^{[i]}(u) \leftarrow ws^{(i)}(u)\ \forall u \in V$
5: **for each** $\{u, v\} \in B$ **do**
6: **if** $w'(u, v) > \max\{w'(u, \mathit{suitor}^{[i]}(u)), w'(v, \mathit{suitor}^{[i]}(v))\}$ **then**
7: **for each** $z \in \{u, v\}$ **do**
8: *affected*$(z) \leftarrow$ `true`
9: $S_A \leftarrow$ FINDAFFECTED$_B(z)$ ▷ Edge weights in P_z are decreasing, see Section 9.4.1
10: UPDATEAFFECTED$(S_A \setminus \{\mathit{suitor}^{[i]}(z)\})$
11: $i \leftarrow i + 1$

9.4 Extension to Batch Updates

Our dynamic algorithms for single edge updates can be generalized to batches of edge updates. The main idea is to run the algorithms for single edge updates multiple times on the updated graph G'. When doing this, we might modify the *suitor* and ws variables multiple times. Thus, in this section we use superscript $^{[i]}$ to denote the values of these variables after we ran the algorithm for single edge updates i times. Hence, if the total number of updates in the batch is b, then $^{[b]}$ is equivalent to $^{(f)}$.

Note that the crucial difference to an algorithm that updates the matching after every single edge insertion is that our algorithm runs directly on the graph G' that already includes a batch B of edge updates. Thus, the intermediate matching computed by our algorithm after $i < |B| = b$ iterations is not necessarily the matching that Suitor computes on the initial graph with the first i edge updates in the batch. As we explain in Sections 9.4.1 and 9.4.2, this allows our algorithm to update vertices affected by different edge updates in the same iteration; albeit this does now lower the time complexity, it results in better practical performances (see Section 9.6.4).

9.4.1 Multiple Edge Insertions

Algorithm 28 shows our dynamic algorithm to handle a batch $B = \{\{u, v\}$ s.t. $\{u, v\} \notin E\}$ of edge insertions – which essentially applies Algorithm 26 to every individual edge in B. For every edge $e = \{u, v\} \in B$, Algorithm 28 checks if $e \in M^{[i+1]}$ (Line 6) and, if so, it computes the values of $\mathit{suitor}^{[i]}$ and $ws^{[i]}$ of the vertices in G' affected by the insertion of e as done by Algorithm 26. As in Section 9.3.1, these vertices lie along two alternating paths P_u and P_v that alternate edges in $M^{[i+1]} \setminus M^{[i]}$ and edges in $M^{[i]} \setminus M^{[i+1]}$. In addition to their first edge, we allow P_u and P_v to include further edges in B whose weight is lower compared to any other preceding edge in the alternating path. This condition is necessary to ensure the correctness of the resulting matching: if an alternating path does not have decreasing edge weights, this could result in violations of Eq. (9.2) for some vertices after Algorithm 28 finishes. Furthermore, allowing the alternating paths to include more than one edge in B makes our batch-dynamic algorithm more competitive in practice than an algorithm that only handles single edge insertions (see Section 9.6.4).

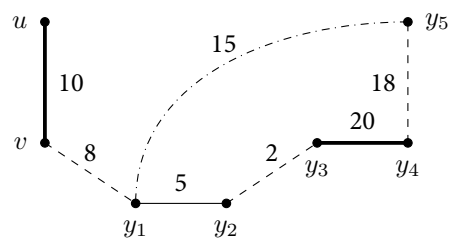


Figure 9.3: Example of alternating path (including edge weights) with non-decreasing edge weights where Eq. (9.2) is violated for y_5. Thick solid edges are in B, solid edges are in $M^{[i+1]} \setminus M^{[i]}$, and dashed edges are in $M^{[i]} \setminus M^{[i+1]}$. The dash-dotted edge shows the violation: assuming that y_1 satisfies Eq. (9.2) for y_5, findAffected ignores it because, when *cur* is y_5, y_1 is marked as affected, and thus not considered as a potential partner for y_5 (see Line 9 in Algorithm 24).

Remark 9.4.1. Let P be an alternating path computed by findAffected in the i-th iteration of Algorithm 28. If the edge weights of P are not decreasing, then Eq. (9.2) could be violated for some vertices in $M^{(f)}$.

Figure 9.3 shows a simple example when such a violation occurs. Let us assume that $B = \{\{u, v\}, \{y_3, y_4\}\}$, that Algorithm 28 is computing P_u with findAffected(u), and that $\{y_3, y_4\}$ is yet to be processed by Algorithm 28 in the `for`-loop in Line 5. Once P_u reaches y_5 (by adding $\{y_4, y_5\} \in B$), y_5 cannot choose y_1 as matching partner, because y_1 is already in P_u, and thus findAffected marked it as affected and does not consider it as a potential matching partner for y_5 (see Line 9 in Algorithm 24). Thus, Eq. (9.2) is violated for y_5 and y_1; further, updateAffected matches together y_3 and y_4, thus the next iteration of Algorithm 28 does not have any effect and the violation of Eq. (9.2) remains in $M^{(f)}$.

Consequently, in case of a batch of edge insertions, we enforce findAffected to discard heavier edges than the ones that are already part of the alternating path and, to distinguish it from the function in Algorithm 24, we denote it as findAffected$_B$ (Algorithm 29). As we prove in Lemma 9.4.1, this guarantees that, for every vertex x in an alternating path computed in the i-th iteration of Algorithm 28, if *suitor*$^{[i+1]}(x)$ does not satisfy Eq. (9.2) in $M^{[i+1]}$, then the vertex y that satisfies Eq. (9.2) for x in $M^{[i+1]}$ is such that $\{x, y\} \in B$ and $\{x, y\}$ is yet to be processed by Algorithm 28. Hence, once Algorithm 28 finishes, all vertices in G' satisfy Eq. (9.2), and thus the resulting matching $M^{(f)}$ (i.e., $M^{[b]}$) equals M'.

Lemma 9.4.1. Let P be an alternating path computed in the i-th iteration of Algorithm 28. For every vertex $x \in P$ such that *suitor*$^{[i+1]}(x)$ does not satisfy Eq. (9.2) in $M^{[i+1]}$ (if any), let y be the vertex that satisfies Eq. (9.2) in $M^{[i+1]}$ for x. Then $\{x, y\}$ is in B and it is yet to be processed by Algorithm 28 in the for loop in Line 5.

Proof. We first show that $\{x, y\} \in B$: in case of single edge insertions, as shown in Lemma 9.3.5, the resulting alternating paths have always decreasing edge weights. Therefore, an alternating path computed by findAffected (Algorithm 24) can have non-decreasing edge weights only if multiple edges are added to G at once – as shown in the example in Figure 9.3; such edges are excluded by findAffected$_B$ in Algorithm 28 (Line 9). Thus, *suitor*$^{[i+1]}(x)$ does not satisfy Eq. (9.2) in $M^{[i+1]}$ only if $\{x, y\}$ is in B and it is discarded by findAffected$_B$.

Further, if y satisfies Eq. (9.2) for x in $M^{[i+1]}$, then we have that $w'(x, y) > \max\{w'(x, \textit{suitor}^{[i]}(x)), w'(y, \textit{suitor}^{[i]}(y))\}$. Thus, if $\{x, y\}$ was processed by Algorithm 28 in an

Algorithm 29 Generalization of the findAffected function (Algorithm 24) for batches of edge updates

Input: affected vertex z
Output: Stack of affected vertices

```
function FINDAFFECTED_B(z)
    S_A ← empty stack
    nextCandidate[z] ← 0
    done ← false
    cur ← z
    repeat
        partner ← suitor(cur)
        heaviest ← ws(cur)
        found ← false
        for i ← nextCandidate[cur] to deg(cur) do
            x ← adjList[cur][i]                    ▷ i-th neighbor in the adjacency list of vertex cur
            nextCandidate[cur] ← i + 1
            if not affected(x) and w(cur, x) > heaviest and w(cur, x) > ws(x) then
                partner ← x
                heaviest ← w(cur, x)
                found ← true
                break
        done ← true
        if found then
            y ← suitor(partner)
            suitor(partner) ← cur
            ws(partner) ← heaviest
            S_A.push(partner)
            affected(partner) ← true
            if y ≠ null then
                suitor(y) ← null
                ws(y) ← 0
                affected(y) ← true
                cur ← y
                done ← false
        else
            affected(cur) ← false
    until done is true
    return S_A
```

earlier iteration than i, then findAffected$_B$ would have matched together x and y. Since $\{x, y\} \notin M^{[i]}$, $\{x, y\}$ is yet to be processed by Algorithm 28. □

9.4.2 Multiple Edge Removals

As shown in Algorithm 30, a batch of edge removals $B \subseteq E$ is handled similarly to batch insertions, namely we apply Algorithm 27 to every edge in B. For every vertex z adjacent to the current edge $e = \{u, v\} \in B$, we check in Line 7 if $suitor^{[i]}(z)$ violates Eq. (9.2) in $M^{[i]}$ due to the removal of e. If so, we update it in Lines 8–8 as done in Algorithm 27. Otherwise, z has already been updated in a previous iteration of the algorithm and no further action needs to be done.

Proposition 9.4.1. Let P be an alternating path computed in the i-th iteration of the outermost for-loop in Algorithm 30. Every vertex $x \in P$ satisfies Eq. (9.2) in G'.

Proof. Conversely to batches of edge insertions, in case of a batch of edge removals, all the alternating paths computed in Lines 11–12 of Algorithm 30 have decreasing edge weights because no edge is added to the

Algorithm 30 Dynamic Suitor algorithm for a batch of edge removals

Input: Graph $G' = (V, E \cup B, w')$, batch of edge removals $B \subseteq E$

1: $\mathit{affected}(u) \leftarrow$ `false` $\forall u \in V$
2: $i \leftarrow 0$
3: $\mathit{suitor}^{[i]}(u) \leftarrow \mathit{suitor}^{(i)}(u) \; \forall \, u \in V$
4: $ws^{[i]}(u) \leftarrow ws^{(i)}(u) \; \forall \, u \in V$
5: **for** $\{u, v\} \in B$ **do**
6: **for** $z \in \{u, v\}$ **do**
7: **if** $\mathit{suitor}^{[i]}(z) = \{u, v\} \setminus \{z\}$ **then**
8: $\mathit{suitor}^{[i+1]}(z) \leftarrow$ `null`
9: $ws^{[i+1]}(z) \leftarrow 0$
10: $\mathit{affected}(z) \leftarrow$ `true`
11: $S_A \leftarrow$ FINDAFFECTED(z)
12: UPDATEAFFECTED(S_A)
13: $i \leftarrow i + 1$

graph. The new matching partner of every vertex $x \in P$ is chosen by findAffected according to Eq. (9.2) and thus once Algorithm 30 finishes, all vertices in G' satisfy Eq. (9.2). □

From Proposition 9.4.1 it follows that, after Algorithm 30 finishes, $M^{(f)}$ equals M'. Note that an alternating path computed by Algorithm 30 can update vertices adjacent to other removed edges in B; similarly to Algorithm 28, this does not improve the worst-case time complexity of the algorithm but, as reported in Section 9.6.4, makes it faster in practice.

We can combine Algorithm 30 to handle batches with both edge insertions and removals, with the only difference that we have to use findAffected$_B$ instead of findAffected in Algorithm 30. In this way we guarantee that every alternating path P computed by our algorithm has decreasing edge weights and thus, as shown in Lemma 9.4.1, for each vertex $x \in P$ either $\mathit{suitor}^{[i]}(x)$ satisfies Eq. (9.2) in G' or x is adjacent to an edge update in B that is yet to be processed by our algorithm.

Corollary 9.4.1. Let B be a batch with $|B| = b$ edge updates. Our dynamic algorithms compute M' in $\mathcal{O}(b \cdot (n + m))$ worst-case time complexity.

As shown in Propositions 9.3.3 and 9.3.4, the worst-case time complexity of Algorithms 26 and 27 is $\mathcal{O}(n + m)$. Thus, after a batch with b edge updates, $M^{(f)} = M'$ is computed in $\mathcal{O}(b \cdot (n + m))$ time.

9.5 Implementation

We implement SortSuitor, i.e., the variant of Suitor where the adjacency list of every vertex is sorted by decreasing edge weight so that every vertex considers a neighbor as matching partner at most once [203]. If the additional preprocessing cost is not taken into account, Manne and Halappanavar show empirically that SortSuitor is faster than Suitor. We implement this by keeping, for each vertex u in the graph, an additional index in the adjacency list of u that indicates the next vertex in the adjacency list of u to be considered as potential matching partner for u. Such indices are stored in the array `nextCandidate`; they are incremented in each iteration of the `for`-loop in Line 10 of Algorithm 29, which is interrupted as soon as a new matching partner is found (Line 17).

On dynamic graphs we need to update the adjacency lists and `nextCandidate` before running both SortSuitor and our dynamic algorithm. In case of b edge insertions, we insert the new edges into the sorted edge lists. Under the reasonable assumption that the newly added edges are inserted at the back of every adjacency list, we sort the new edges and we merge the first (already sorted) part of the adjacency list with the second one. With this strategy the adjacency list of each vertex $x \in V$ can be sorted in $\mathcal{O}(\deg(x)+b_x \log b_x)$, where b_x is the number of edges in B adjacent to x. When rerunning SortSuitor, for each vertex in G', `nextCandidate` is updated to the first (i.e., heaviest) edge in the adjacency list. In our dynamic algorithm, in turn, for each vertex u adjacent to an edge insertion, `nextCandidate`$[u]$ is updated so that in G' it indicates the same edge as in G. This prevents findAffected$_B$ from computing paths with non-decreasing edge weight (as required by Algorithm 28), because every vertex x in an alternating path, when seeking a new partner, can only consider edges that are lighter than $ws^{[i]}(x)$. Newly inserted edges adjacent to x and heavier than $ws^{[i]}(x)$ are taken into account by updating the neighbor index of the current vertex x to the first (i.e., heaviest) edge in the adjacency list of x.

In case of b edge removals, the adjacency lists are updated with the same time complexity as edge insertions – for each vertex $x \in V$, the index of an edge in B adjacent to x can be found in $\mathcal{O}(\log \deg(x))$ time and all removed edges adjacent to x can be deleted from the adjacency list in $\mathcal{O}(\deg(x))$ time. Concerning the neighbor indices, in SortSuitor they are updated to the first edge in the adjacency list, whereas in our dynamic algorithm this is done only for the vertices adjacent to a removed edge.

9.6 Experimental Results

We conduct experiments to compare the performance of our dynamic Suitor algorithm against the state-of-the-art DynMWMRandom [16] for single edge updates and, since DynMWMRandom does not support batch updates, against a static recomputation for batches of edge updates.

9.6.1 Settings

We implement both the static and dynamic Suitor algorithms in C++ and we use the NetworKit [273] graph APIs. DynMWMRandom is implemented in C++ as well but the graph data structure, contrary to ours, also uses hash tables[53] for faster edge lookup and update operations. All experiments are conducted on a Linux machine equipped with 192 GiB of RAM and an Intel Xeon Gold 6126 CPU with two sockets, 12 cores each (24 cores in total) at 2.6 GHz. Our algorithms are sequential and thus in all our experiments we only use one core. All the experiments are managed by the SimexPal [14] software to ensure reproducibility; they are executed on both real-world graphs and randomly generated instances – see Tables 9.1 and 9.2. All the complex networks in Table 9.1 are downloaded from the KONECT [173] repository; the road networks, in turn, are downloaded from OpenStreetMap [83]. From the road networks we build the pedestrian routing graph using RoutingKit [89] and choose the geographic distance as weight function. Synthetic networks are generated using the R-MAT [65] and the random hyperbolic[54] models. For the R-MAT model we use the Graph500 [223] parameter setting (i.e., edge factor 16, $a = 0.57, b = 0.19, c = 0.19$, and $d = 0.05$), and the generator from Khorasani et al. [165]. For the random hyperbolic model we use the generator from

[53] See `https://github.com/sparsehash/sparsehash`.
[54] The random hyperbolic model generates networks with a power-law degree distribution.

Table 9.1: Real-world instances used in the experiments. We refer to every instance by its "ID". For complex networks, edge weights are randomly generated using either a normal distribution or an exponential distribution.

(a) Road networks

Graph	ID	n	m	Avg. Deg.
belgium	be	1,216,902	1,563,642	2.6
czech-republic	cz	1,713,252	2,181,152	2.5
finland	fi	2,177,796	2,639,775	2.4
austria	au	2,621,866	3,082,590	2.4
canada	ca	3,795,591	4,780,472	2.5
poland	po	5,567,642	7,200,814	2.6
italy	it	6,339,229	7,818,183	2.5
great-britain	gb	7,108,301	8,358,289	2.4
france	fr	11,063,911	13,785,539	2.5
russia	ru	10,984,765	14,079,238	2.6
germany	ge	15,918,055	20,266,409	2.5
dach	da	20,207,259	25,398,909	2.5
africa	af	23,975,266	31,044,959	2.6
us	us	41,256,068	51,271,328	2.5
asia	as	57,736,107	72,020,649	2.5

(b) Complex networks

Graph	ID	n	m	Avg. Deg.
hyves	hy	1,402,673	2,777,419	4.0
com-youtube	cy	1,134,890	2,987,624	5.3
flixster	fx	2,523,386	7,918,801	6.3
youtube-u-growth	yg	3,223,589	9,375,374	5.8
flickr-growth	fg	2,302,925	22,838,276	19.8
livejournal-links	ll	5,204,176	48,709,621	18.7
soc-LiveJournal1	lj	4,846,609	68,475,391	14.1
orkut-links	ol	3,072,441	117,184,899	76.3
dimacs10-uk-2002	di	18,483,186	261,787,258	28.3
wikipedia_link_en	we	13,593,032	437,167,958	32.2
twitter	tw	41,652,230	1,468,364,884	35.3
twitter_mpi	tm	52,579,682	1,963,263,507	37.3
friendster	fs	68,349,466	2,586,147,869	37.8

Table 9.2: R-MAT and random hyperbolic networks used in the experiments. For each size, we generate five networks using a different random seed. For a fixed number of vertices, the random hyperbolic generator [191] generates networks with different number of edges; thus, we report the minimum, the average, and the maximum number of edges in the m_{min}, m_{avg}, and m_{max} columns, respectively. Edge weights are randomly generated using either a normal distribution or an exponential distribution.

(a) R-MAT networks

Graph	n	m	Avg. Deg.
rmat-22	2^{22}	67,108,864	32.0
rmat-23	2^{23}	134,217,728	32.0
rmat-24	2^{24}	268,435,456	32.0

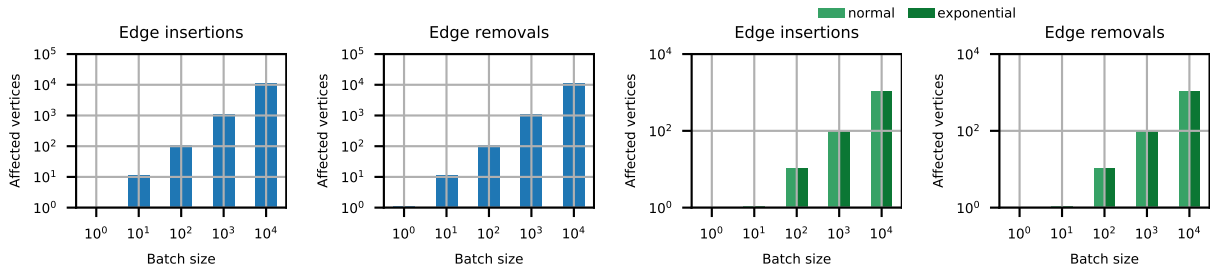
(b) Random hyperbolic networks

Graph	n	m_{min}	m_{avg}	m_{max}	Avg. Deg.
hyp-22	2^{22}	41,876,800	41,951,095.8	42,013,293	20.0
hyp-23	2^{23}	83,705,169	83,830,659.2	83,928,747	20.0
hyp-24	2^{24}	167,562,625	167,697,480.2	167,902,689	20.0

von Looz et al. [191] within NetworKit; we set the average degree to 20, and the exponent of the power-law distribution to 3. Detailed statistics about synthetic networks are reported in Table 9.2. Experiments on synthetic networks are repeated five times, in each one we generate the network using a different random seed – this results in a different graph for every experiment.

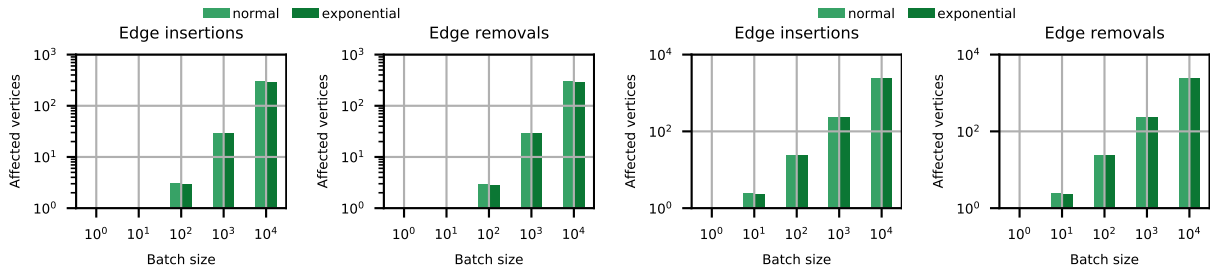
Because the real-world complex networks and the synthetic networks are initially unweighted, unless stated differently, we generate edge weights using a normal distribution with mean 1 and standard deviation 0.5 and an exponential distribution with parameter 1. Experiments with random edge weights are repeated five times; in each one we generate random weights using a different random seed.

For each tested graph, we either add or remove a batch of edges selected uniformly at random and run the dynamic algorithm(s) after each batch update (for DynMWMRandom comparison we only perform single edge updates, so the batch size is always 1). We repeat this process 100 times. For batch, insertions we first remove a random batch of edges from the original graph and re-add them back, whereas for removals we first add a batch of edges and then remove them. Therefore, after every batch of graph updates the resulting graph G' is always the same and we need to run the static Suitor algorithm only once on G' regardless of the batch size.



(a) Road networks (b) Complex networks

Figure 9.4: Average number of vertices affected by a batch of edge updates in the real-world networks of Table 9.1.



(a) R-MAT networks (b) Random hyperbolic networks

Figure 9.5: Average number of vertices affected by a batch of edge updates in the synthetic networks of Table 9.2.

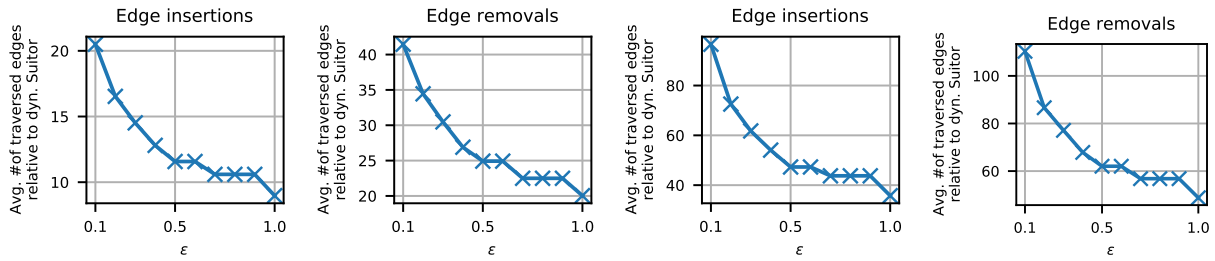
9.6.2 Affected Vertices

We first analyze how many vertices are affected by a batch of edge updates according to Definition 9.3.1. The average number of affected vertices for real-world and synthetic instances is summarized in Figures 9.4 and 9.5, respectively. In road networks, the number of affected vertices is on average moderately higher than the batch size for both edge insertions and removals: Intuitively, a random edge update is more likely to update the matching of its adjacent vertices if their degree is low, and road networks are the ones with lowest average degree – see Table 9.1. Also, as explained in Section 9.3, updating the matching of two vertices might also impact the matching of other vertices, which explains why in road networks we have a higher number of affected vertices w.r.t. the batch size.

Regarding complex networks (both real-world and synthetic), their average degree is higher than road networks and, as expected, the number of affected vertices is lower – it is on average one order of magnitude smaller than the batch size. Results do not change notably between the two distributions of edge weights.

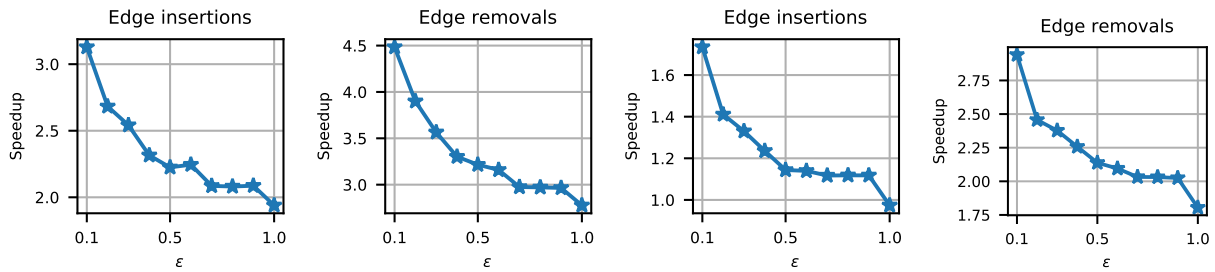
9.6.3 Comparison Against DynMWMRandom

In the following, we evaluate the performance and the quality of dynamic Suitor against the state-of-the-art DynMWMRandom algorithm on real-world networks. We remark that, as explained in Section 9.2.3, our dynamic algorithm yields the same matching as Suitor. Because DynMWMRandom maintains a $(1+\varepsilon)$-approximate MWM, we evaluate this algorithm with $\varepsilon \in \{0.1, 0.2, \ldots, 0.9, 1\}$. Further, as suggested in Ref. [16], we use the "stop early" heuristic with $\beta = 5$. Note that, with this setting, we are reducing the number of random walks sampled by the algorithm and thus the approximation guarantee does *not* hold anymore – solutions yielded by the "stop early" heuristic, however, have been shown to be much closer to the optimum than $(1+\varepsilon)$ for nearly all the instances considered in Ref. [16].



(a) Road networks (b) Complex networks

Figure 9.6: Average number of edges traversed by DynMWMRandom relative to the ones traversed by dynamic Suitor for a single edge update and for different values of ε. Results are averaged over 100 edge updates and over the networks of Table 9.1.



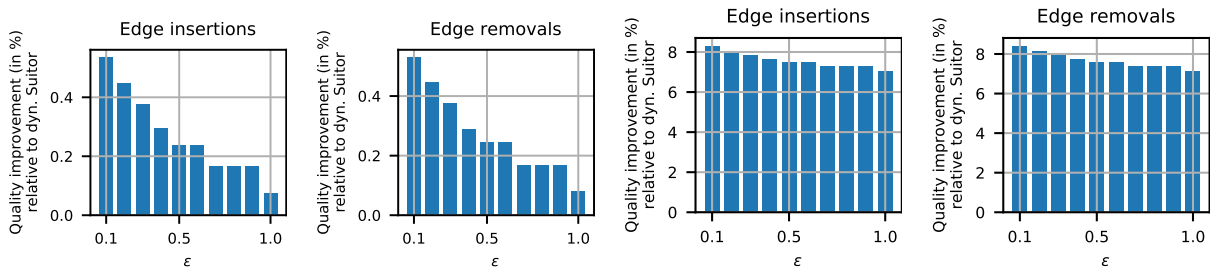
(a) Road networks (b) Complex networks

Figure 9.7: Geometric mean of the speedups of dynamic Suitor over DynMWMRandom for single edge updates and for different values of ε. Results are averaged over 100 edge updates and over the networks of Table 9.1.

Since the implementation of DynMWMRandom only supports integral edge weights, in this section, we generate random weights for complex networks using a normal distribution with mean 100 and standard deviation 10 and we round to the nearest integer value. In addition, because the two algorithms use different graph data structures, we employ an implementation-agnostic performance measurement, i.e., the number of edges traversed by the two algorithms to handle a single edge update. Detailed results are reported in Appendices H.2–H.4.

Results on High-Diameter Networks. Figures 9.6a and 9.7a summarize the average number of traversed edges and the algorithmic speedups (i.e., ratio between the running times), respectively, on high-diameter networks. On average, for edge insertions, DynMWMRandom traverses $8.9\times$ (with $\varepsilon = 1$) to $20.3\times$ (with $\varepsilon = 0.1$) more edges than our dynamic algorithm – results for removals are $19.8\times$ and $41.0\times$, respectively. This is expected because, as we explained in Section 9.3, dynamic Suitor only needs to "fix" the affected vertices which, in the case of a single update, are usually very few – as we saw in Section 9.6.2. DynMWMRandom, in turn, performs random walks after each update – apart from the trivial case when adding a new edge between two unmatched vertices. In terms of speedup, our dynamic Suitor algorithm is $1.9\times$ and $2.8\times$ (geometric mean over all instances) faster than DynMWMRandom with $\varepsilon = 1$ for insertions and removals, respectively.

Figure 9.8a shows that the two algorithms yield solutions with nearly the same quality. More precisely, compared to DynMWMRandom, dynamic Suitor solutions are 99.5% to 99.9% for both of insertions and removals. Hence, we conclude that, for high-diameter graphs, dynamic Suitor is a competitive algorithm.



(a) Road networks

(b) Complex networks

Figure 9.8: Difference (in %) of the weight of the matching computed after 100 edge updates by DynMWMRandom for different values of ε. Results are relative to dynamic Suitor solutions. Results are averaged over the networks of Table 9.1.

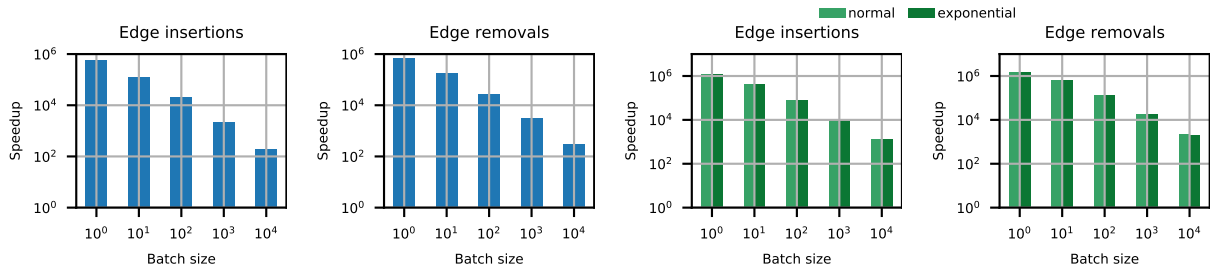
Results on Complex Networks. Results are different in complex networks (see Figures 9.6b and 9.7b). In terms of traversed edges, our dynamic algorithm achieves even better results than in high-diameter networks: w.r.t. dynamic Suitor, DynMWMRandom traverses at least $34.3\times$ more edges (insertions, $\varepsilon = 1$). As argued in Section 9.6.2, such a discrepancy is expected because, in complex networks, vertices have on average a higher degree than in high-diameter networks and thus are less likely to be affected by an update. This implies low overhead for dynamic Suitor.

Our speedup results, in turn, seem to contradict the observations we made so far: compared to DynMWMRandom with $\varepsilon = 1$, dynamic Suitor is slightly slower for edge insertions, and $1.8\times$ faster for edge removals – which is worse than what we achieved for road networks. A possible explanation is that the two algorithms use different graph data structures: hash maps (used by DynMWMRandom) enable fast edge lookup and update operations; we conjecture that this has a substantial performance impact, especially for networks with high-degree vertices – such as complex networks.

In terms of solution quality, the gap between the two dynamic algorithms is more pronounced than in high-diameter networks. Relatively to DynMWMRandom, dynamic Suitor yields solutions that are 0.1% to 93.0%. A possible explanation for this result is that, typically, vertices in complex networks tend to form densely-connected clusters (or communities) [227, Ch. 10]; we hypothesize that a random walk finds augmenting paths in such dense structures more successfully than in road networks – which, instead, are more sparse. Hence, in complex networks, while our dynamic algorithm is more likely to "skip" edge updates (because, as explained above, they do not affect any vertex), DynMWMRandom is more likely to find augmenting paths, which leads to better solution quality.

9.6.4 Speedups on the Static Algorithm

We now evaluate the speedup of our dynamic Suitor algorithm against a static recomputation, both on real-world and on synthetic networks. Because both the dynamic and the static algorithm need to sort the adjacency lists of the vertices after a batch of edge updates, we discard this step in the speedup computation (i.e., we only compare the running time of both algorithms after the adjacency lists have been sorted). As shown in Figure H.1 in Appendix H.7, in terms of running time this preprocessing step is almost negligible as it always takes less than 6% – but mostly less than 2% – of the overall running time of the static Suitor algorithm.



(a) Road networks (b) Complex networks

Figure 9.9: Geometric mean of the speedups of the dynamic algorithm over a static recomputation over the real-world networks of Table 9.1.

Detailed speedup results are reported in Tables H.8–H.11 in Appendix H.5. Running times in seconds are reported in Tables H.12–H.15 in Appendix H.6.

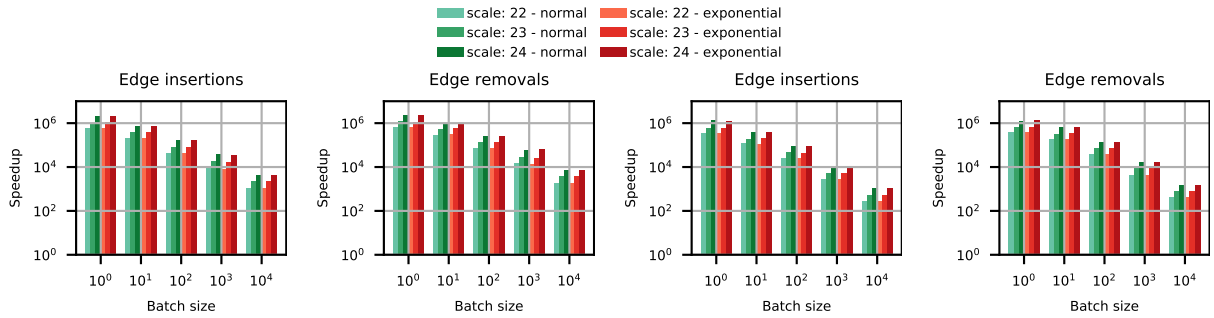
Speedups on Real-world Networks. Figure 9.9a summarizes the speedup on road networks. For single edge insertions and removals, the dynamic algorithm is 5 orders of magnitude faster than a static recomputation. As we consider larger batches the number of affected vertices increases, and thus the dynamic algorithm becomes slower. Nevertheless, the speedup is still higher than 10^3 for batches with up to 10^3 edge updates. For batches of 10^4 edge insertions and removals, the speedup is still $196.1\times$ and $310.4\times$, respectively.

Concerning complex networks, our dynamic algorithm performs even better: Figure 9.9b shows that, with both distributions of random edge weights, the speedup is always greater than 10^6 for single edge updates, and greater than 10^4 for batches of up to 10^3 edge edge updates. For batches of 10^4 edge updates with edge weights generated using a normal distribution, the dynamic algorithm is $1{,}362.1\times$ and $2{,}135.7\times$ faster than a static recomputation, respectively; using an exponential distribution to generate edge weights yields similar speedups: $1{,}352.5\times$ for edge insertions and $2{,}114.1\times$ for edge removals.

As discussed in Section 9.6.2, better speedups on complex networks can be explained by the fact that the number of affected vertices on complex networks are on average lower compared to road networks, and therefore the dynamic algorithm needs to perform less work. Further, these results show that the worst-case time complexity of our algorithms is very pessimistic compared to their practical performance, and thus that the length of the alternating paths described in Section 9.6.2 – which determine the running time of our algorithms – is, in practice, usually only a small fraction of the number of vertices in the graph.

Our speedup results are comparable to the ones achieved by Henzinger et al. for MCM [140]: their dynamic algorithms are roughly $10^5\times$ faster than a static recomputation with an *optimal* MCM algorithm. Note that they compare against an exact algorithm, which is speed-wise a weaker baseline than an approximate algorithm. This advantage, on the other hand, may be compensated by the fact that their comparisons are run on rather small networks (25K vertices), where higher speedups are more difficult to obtain.

Speedups on Synthetic Networks. Results on R-MAT and random hyperbolic networks are shown in Figures 9.10a and 9.10b, respectively. Compared to the static Suitor algorithm, for both models and for both distributions of the edge weights, our dynamic algorithm is 5 to 6 orders of magnitude faster on single edge updates, and 3 to 5 orders of magnitude faster on batches with up to 10^3 edge updates. Concerning



(a) R-MAT networks (b) Random hyperbolic networks

Figure 9.10: Geometric mean of the speedups of the dynamic algorithm over a static recomputation over the synthetic networks networks of Table 9.2. The considered graphs have 2^s vertices, where s is the scale shown in the legend.

batches of 10^4 edge updates, the speedup for edge insertions and removals on R-MAT networks is always at least $2{,}228.7\times$ and $3{,}719.2\times$, respectively, and always at least $550.5\times$ and $811.6\times$, respectively, on random hyperbolic networks.

From Figures 9.10a and 9.10b we can also see that, for every batch size, the speedups increase with the size of the networks. A possible interpretation of this result is that, as for real-world networks, even though our algorithms have a worst-case time complexity of $\mathcal{O}(n+m)$ for a single edge update (see Sections 9.3.1 and 9.3.2), in a real-world scenario, this is too pessimistic and the algorithm is instead much faster. As we have shown in Section 9.6.2, in complex networks edge updates either do not change the matching of any vertex in the graph (and thus they are handled in constant time), or they affect a very small number of vertices, leading to short processing times.

Speedup of Batch Updates on Single Updates. Finally, we measure the speedup of our batch-dynamic algorithm against the more naive approach of handling the edge updates in the batch one by one. We perform these experiments for batches of size $b = 100$ edge updates. As described in Section 9.4, although the worst-case time complexity of the two algorithms is the same, in a real-world scenario we observe that the batch-dynamic algorithm is faster than the naive one. On road networks (Table 9.1), the batch-dynamic algorithm is on average $4.3\times$ and $5.3\times$ faster than the naive one on batches of edge insertions and removals, respectively. Regarding complex networks (Table 9.1), when edge weights are generated with a normal distribution, the speedups on batches of edge insertions and removals are $7.7\times$ and $10.2\times$, respectively; the results for edge weights drawn from an exponential distribution are similar: $7.5\times$ and $10.7\times$ for batches of edge insertions and removals, respectively.

9.7 Conclusions

We have developed and implemented a batch-dynamic $(1/2)$-approximation algorithm for MWM based on the Suitor algorithm by Manne and Halappanavar [203]. Our dynamic algorithm updates the matching results from an initial static computation quickly after a batch of edge updates, leading to results that are equivalent to the static algorithm's. Our experimental data show that it can handle in less than a millisecond batch sizes of up to 10^4, thus providing real-time capabilities. Compared to the state-of-the-art

DynMWMRandom [16] algorithm, dynamic Suitor requires less work (in terms of traversed edges) and, on high-diameter networks, yields solutions with nearly the same quality. The solution quality in complex networks is, in turn, 8.4% worse w.r.t. DynMWMRandom's or higher.

In comparison with static recomputation, our dynamic algorithm is 2 to 6 orders of magnitude faster, depending on the input network and on the batch size; further, our speedup results are comparable to the ones achieved by Henzinger et al. for the related dynamic MCM problem [140]. The main reason of such high speedups is that the running time of our dynamic algorithms is determined by the length of the alternating paths. In the worst case, these paths can contain all vertices in the graph. However, as shown in Section 9.6.4, in practice these paths are much shorter. Conversely, even in a best-case scenario, the complexity of the static Suitor algorithm is linear in the size of the input network.

Note that the strategy we exploit here is very similar to the one we implemented in Chapter 3 for top-k closeness centrality: after the graph changes, we first identify the vertices *affected* by the edge update(s) and then we update the result. The better performance of our dynamic algorithm compared to DynMWM-Random and the static one is justified by the very small number (compared to n) of affected vertices (see Section 9.6.2). Hence, an interesting direction for future research is to investigate whether this strategy is successful in other scenarios beyond closeness and betweenness (cf. Ref. [40]) centrality or matching.

Further directions for future work are conceivable. One possibility is to improve the quality of the solution by adapting the *two-round approach* [203, Sec. IV] to dynamic graphs. Another one is to investigate whether the performance of dynamic Suitor would benefit from more sophisticated graph data structures – e.g., hash maps as done in Ref. [16].

Part V

Conclusion

Conclusion

In this thesis, we presented efficient and scalable algorithms that address three families of network analysis problems, namely: vertex centrality (Part II), group centrality maximization (Part III), and matching (Part IV), on either static or dynamic graphs.

Vertex Centrality. In Part II, we focused on two main aspects of vertex centrality: updating the top-k most central vertices in fully-dynamic graphs (Chapter 3) and approximating the centrality scores of *all* the vertices of large-scale networks (Chapters 4 and 5).

The algorithms presented in Chapter 3 update the exact top-k ranking of the vertices with highest closeness centrality in fully-dynamic graphs with tens of millions of edges in a matter of seconds or at most 3 minutes after batches of up to 100 edge updates. Compared to a static recomputation, they are up to $10^4\times$ faster. Similarly to other dynamic algorithms [39], these results are achieved by avoiding unnecessary work and by exploiting precomputed information on the initial graph. In particular, vertices that are not *affected* by the edge update(s) (i.e., their distance to the others is unchanged) are ignored. In addition, maintaining upper bounds of the closeness centrality of each vertex allows to prune graph traversals and to skip the computation for several affected vertices (e.g., far-away vertices in Table 3.1). This strategy has been shown to be very effective with moderately-sized batches of edge updates but, as we increase the batch size, it yields diminishing returns. This is easily explained by the fact that, if the graph undergoes many edge updates, the information we know about the initial graph is less accurate, many vertices are affected and, consequently, the possibilities of reducing the workload are narrowed down.

In Chapters 4 and 5, we introduce algorithms that target vertex centrality approximation in static graphs. Our approaches improve upon the state of the art by avoiding unnecessary work and by efficiently exploiting parallelism. In Chapter 4, we introduce a general epoch-based framework for parallel adaptive sampling, which we apply to betweenness centrality approximation as a case study. Our motivation stems from the static state-of-the-art KADABRA [56] ADS algorithm for betweenness approximation. Due to the synchronization overheads required to correctly check the stopping condition, a trivial OpenMP-based parallelization of KADABRA fails to scale to large numbers of threads. To overcome this limitation, we introduce three parallel ADS algorithms, all of which implement a novel epoch-based mechanism that avoids extensive synchronizations when checking the stopping condition by keeping multiple copies of the sampling state. On 32 cores, *shared-frame* is the fastest among our algorithms and requires less than two minutes to yield a ± 0.01 approximation for the betweenness centrality of all the vertices of networks with hundreds of millions of edges. This is an interesting result as it shows the trade-off between memory footprint and synchronization costs. *Shared-frame* uses only $\Theta(1)$ additional memory per thread (SFs are shared among threads) and thus it needs to synchronize writing operations to SFs; *local-frame*, in turn, has a greater memory footprint (uses $\mathcal{O}(n)$ additional memory per thread as SFs are thread-local) and therefore writing operations to SFs are not

synchronized. In our case study, it seems that, on 32 cores, memory latencies overcome synchronization costs. This is also clear from Figure 4.5b: on 36 cores, the fastest *shared-frame* configuration is the one with the lowest memory footprint. We also propose further variations of our algorithms which, at the cost of additional synchronizations, guarantee bounded memory footprint and determinism of the results.

Finally, in Chapter 5, we introduce a parallel UST-based algorithm to approximate $\mathrm{diag}(\mathbf{L}^{\dagger})$ (or, in the case of forest closeness, $\mathrm{diag}(\mathbf{\Omega})$), and thus several electrical centrality measures such as electrical closeness, forest closeness, and others. Compared to the state of the art, our algorithm is faster, it yields more accurate results in terms of absolute error, and requires less memory. In addition, our algorithm is trivial to parallelize – as most of the work consists in sampling USTs – and thus the workload can be easily distributed among multiple cores and/or compute nodes. For example, on our small cluster with 16×24 cores, it requires less than 20 minutes to achieve a ± 0.3 approximation of $\mathrm{diag}(\mathbf{L}^{\dagger})$ on a network with ≈ 334.6M edges. Thus, our method can be used to approximate with reasonable accuracy and in a matter of a few minutes electrical centrality measures on networks with hundreds of millions of edges with commodity hardware.

It is worth observing that the results from Chapters 3 and 5 give an empirical demonstration that worst-case analysis does not always reflect the actual performance of an algorithm on real-world instances. In fact, compared to competitors, our algorithms have either the same (Chapter 3) or slightly worse (Chapter 5) time complexity but they are consistently much faster in practice. This confirms the crucial importance of experimental evaluation in the algorithm engineering process.

Another aspect that emerges from the results in Chapters 3 and 4 is that using an additional amount of memory to store intermediate results can bring considerable benefits in terms of speed. Specifically, in Chapter 3, storing the upper bounds of the closeness centrality allows to skip unnecessary work. In Chapter 4, in turn, multiple copies of the sampling state reduce the need for synchronization and thus the overhead. On the other hand, the choice of which data to store should be made carefully because, as emerged from our experiments in Chapter 4, a large memory footprint limits the scalability of an algorithm to large instances.

Group Centrality Maximization. In Part III, we addressed the lack of algorithms for group centrality maximization capable of: (i) finding highly central sets of vertices in large-scale networks quickly and (ii) yielding solutions with approximation guarantees.

The first issue is targeted in Chapters 6 and 8. Chapter 6 introduces a family of fast local search heuristics that, for the first time, make it possible to compute sets of vertices with high group-closeness centrality on networks with hundreds of millions of edges in minutes without sacrificing too much solution quality – previous state-of-the-art methods require several hours and yield less than 1% better solution quality. We propose several variations of our local search algorithms offering to the user trade-offs between running time and solution quality. In Chapter 8, we propose an alternative solution to the lack of scalable algorithms for group centrality maximization: we observe that shortest-path based measures pose complexity-theoretic barriers that are hard to overcome; we circumvent this problem by developing GED-Walk, a new group-centrality measure inspired by Katz centrality that considers walks of any length instead of shortest paths only. Experiments in Section 8.5 show two main achievements: (i) GED-Walk captures meaningful information of a graph – as graph mining tasks such as semi-supervised vertex classification and graph classification benefit from the new measure – and (ii) GED-Walk can be maximized (in approximation)

considerably faster than other group centrality measures. Therefore, GED-Walk represents a feasible alternative to other group centrality measures in applications where performance is of major concern.

Another limitation that we target in Chapter 8 is that the only existing electrical group centrality measure (i.e., group electrical closeness [183]) does not handle disconnected graphs out of the box. We thus extend forest closeness [69] to groups of vertices. The result is group forest closeness, an electrical group centrality measure that handles disconnected graphs out of the box. We also adapt the greedy approximation algorithm by Li et al. [183] to group forest closeness. The ability of group forest closeness in finding highly-central groups in disconnected graphs is demonstrated by semi-supervised vertex classification results: as GED-Walk in connected graphs, our new measure achieves higher the precision than existing measures.

In Chapter 7, we address the second issue by implementing the first approximation algorithms for group-closeness and group-harmonic maximization. On small instances where the optimum can be computed in reasonable time, our algorithms yield nearly-optimal solutions. On larger instances, our greedy algorithm for group-harmonic maximization yields solutions with nearly the same quality as its local search counterpart while being one to two orders of magnitude faster; hence, for this problem, the greedy approach seems to dominate local search. For group-closeness maximization, in turn, the situation is different: in line with the theoretical findings presented in Ref. [12], our local search algorithm yields solutions with consistently higher quality than the best known greedy heuristics [38]. Unfortunately, this comes with a cost in running time as local search needs to evaluate $\Omega(n^2)$ swaps in each iteration. Despite our engineering efforts (see Section 7.3.3), this algorithm still cannot handle networks with millions of vertices in reasonable time, leaving the problem of group-closeness approximation on such big graphs open to future work.

Overall, our experiments in Chapters 6 and 7 indicate that local search is an effective strategy to (i) identify highly-central groups in large-scale networks and (ii) refining existing solutions. However, due to its quadratic complexity, it is not practical to use local-search to approximate group-closeness (and group-harmonic) maximization in large-scale networks. Hence, for applications dealing with large networks and where quality is of highest concern, a greedy strategy appears to offer the best trade-off between running time and quality.

Maximum Weighted Matching on Dynamic Graphs. The last problem we address in this thesis is to maintain a $(1/2)$-approximate MWM in fully-dynamic graphs (see Chapter 9). We extend the state-of-the-art Suitor algorithm by Manne and Halappanavar [203] (which yields a $(1/2)$-approximation of the MWM) to also handle multiple edge updates. As demonstrated in Chapter 9, our dynamic algorithm yields the same matching computed by Suitor. Moreover, our experiments in Section 9.6 show that our dynamic algorithm incurs less overhead (in terms of visited edges) for single edge updates than the state-of-the-art DynMWMRandom algorithm [16]. Also, it handles batches with up to 10^4 edge updates in just a fraction of a second, i.e., several orders of magnitude faster than a static recomputation.

As we observed in Chapter 3, using an additional amount of memory to store intermediate results reduces significantly the workload of the dynamic algorithm compared to a static recomputation. The amount of work that can be saved with this technique, however, heavily depends on the structure of the graph and, in a worst-case scenario, it can even be none (when all vertices in the graph are affected). On the other hand, our experiments in Section 9.6.2 suggest that, in practice, the number of vertices affected by an update is significantly smaller than n – on average, it grows at most linearly w.r.t. the batch size, see Figures 9.4

and 9.5 in Section 9.6.2. Therefore, an extensive experimental study turns out again to be an essential tool to evaluate an algorithm's practical performance, as worst-case analysis may be too pessimistic.

In conclusion, in this thesis we presented and implemented – and made publicly available in the open-source NetworKit toolkit [273] – several algorithms for solving network analysis problems that either did not have an algorithm to solve them or that could not be solved in reasonable time by existing algorithms. We devised several directions for future work following directly from our contributions finalized to either improving the proposed algorithms or extending them to other problems.

Appendices

A Publications

The research presented in this thesis appeared previously in the following publications:

1. E. Angriman, R. Becker, G. D'Angelo, H. Gilbert, A. van der Grinten, and H. Meyerhenke. "Group-Harmonic and Group-Closeness Maximization - Approximation and Engineering". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*. SIAM, 2021, pp. 154–168. DOI: 10.1137/1.9781611976472.12.
2. E. Angriman, A. van der Grinten, A. Bojchevski, D. Zügner, S. Günnemann, and H. Meyerhenke. "Group Centrality Maximization for Large-scale Graphs". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*. SIAM, 2020, pp. 56–69. DOI: 10.1137/1.9781611976007.5.
3. E. Angriman, A. van der Grinten, M. von Looz, H. Meyerhenke, M. Nöllenburg, M. Predari, and C. Tzovas. "Guidelines for Experimental Algorithmics: A Case Study in Network Analysis". *Algorithms* 12:7, 2019, p. 127. DOI: 10.3390/a12070127.
4. E. Angriman, A. van der Grinten, and H. Meyerhenke. "Local Search for Group Closeness Maximization on Big Graphs". In: *2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019*. IEEE, 2019, pp. 711–720. DOI: 10.1109/BigData47090.2019.9006206.
5. E. Angriman, M. Predari, A. van der Grinten, and H. Meyerhenke. "Approximation of the Diagonal of a Laplacian's Pseudoinverse for Complex Network Analysis". In: *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 6:1–6:24. DOI: 10.4230/LIPIcs.ESA.2020.6.
6. P. Bisenius, E. Bergamini, E. Angriman, and H. Meyerhenke. "Computing Top-k Closeness Centrality in Fully-dynamic Graphs". In: *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018, New Orleans, LA, USA, January 7-8, 2018*. SIAM, 2018, pp. 21–35. DOI: 10.1137/1.9781611975055.3.
7. A. van der Grinten, E. Angriman, and H. Meyerhenke. "Parallel Adaptive Sampling with Almost No Synchronization". In: *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*. Springer, 2019, pp. 434–447. DOI: 10.1007/978-3-030-29400-7_31.
8. A. van der Grinten, E. Angriman, and H. Meyerhenke. "Scaling up network centrality computations - A brief overview". *it Inf. Technol.* 62:3-4, 2020, pp. 189–204. DOI: 10.1515/itit-2019-0032.

9. A. van der Grinten, E. Angriman, M. Predari, and H. Meyerhenke. "New Approximation Algorithms for Forest Closeness Centrality - for Individual Vertices and Vertex Groups". In: *Proceedings of the 2021 SIAM International Conference on Data Mining, SDM 2021, Virtual Event, April 29 - May 1, 2021*. SIAM, 2021, pp. 136–144. DOI: 10.1137/1.9781611976700.16.

B Appendix of Chapter 3

B.1 Overview of Networks

Table B.1: Undirected complex networks

Network	n	m
petster-cat-household	105,138	494,858
petster-catdog-household	333,111	2,643,012
petster-cat-friend	204,473	5,448,197
petster-friendships-cat	149,700	5,448,197
petster-friendships-dog	426,820	8,543,549
petster-dog-friend	451,710	8,543,549
petster-catdog-friend	623,754	13,991,746
higgs-twitter-social	456,626	14,855,819
petster-carnivore	623,766	15,695,166

Table B.2: Directed complex networks

Network	n	m
wikipedia_link_li	49,129	375,375
cit-HepPh	34,546	421,534
slashdot-zoo	79,120	515,397
wiki_talk_sv	614,206	597,611
munmun_twitter_social	465,017	834,797
wiki_talk_ja	1,064,391	1,030,161
wikipedia_link_gu	30,430	1,327,386
web-NotreDame	325,729	1,469,679
digg-friends	3,376,023	1,731,653

Table B.3: Undirected road networks

Network	n	m
DC	9,559	14,841
DE	49,109	59,760
RI	53,658	68,706
HI	64,892	76,139
AK	69,082	77,498
seychelles	42,835	85,730
VT	97,975	106,242
comores	62,143	126,962
liechtenstein	71,878	147,187

Table B.4: Directed road networks

Network	n	m
kiribati	22,725	46,376
seychelles	42,835	85,730
comores	62,143	126,962
liechtenstein	71,878	147,187
isle-of-man	73,634	151,554
andorra	99,686	197,741
djibouti	96,682	203,142
faroe-islands	107,225	218,910

B.2 Speedup Results

B.2.1 Speedups for Complex Networks

Table B.5: Geometric mean of the speedups of the dynamic algorithm over the static one over 100 batches of edge insertions of size 1, 10, 100 in complex networks, for $k \in \{1, 10, 100\}$.

(a) Undirected networks

Network	$k = 1$			$k = 10$			$k = 100$		
Batch size	1	10	100	1	10	100	1	10	100
petster-cat-household	10.3	1.2	0.1	23.9	2.5	0.3	423.8	15.8	1.3
petster-catdog-household	33.4	3.2	0.3	48.3	3.6	0.5	116.4	4.9	0.8
petster-cat-friend	82.8	8.4	0.8	82.7	8.2	0.8	2,303.5	218.3	18.8
petster-friendships-cat	114.8	11.4	1.2	113.0	10.8	1.2	2,626.5	253.4	21.7
petster-friendships-dog	106.1	11.3	1.1	250.8	25.6	2.4	377.5	30.1	2.7
petster-dog-friend	60.2	6.0	0.6	212.1	19.5	1.9	315.4	24.6	2.3
petster-catdog-friend	899.6	91.6	9.2	974.2	94.7	9.0	1,144.4	105.0	7.0
higgs-twitter-social	177.6	17.0	2.2	505.8	43.2	3.7	709.6	43.2	2.7
petster-carnivore	6,062.6	559.0	53.8	6,115.6	535.5	50.0	6,327.9	449.8	31.9
geometric mean	139.7	14.0	1.5	229.0	21.2	2.2	827.8	57.1	4.9

(b) Directed networks

Network	$k = 1$			$k = 10$			$k = 100$		
Batch size	1	10	100	1	10	100	1	10	100
wikipedia_link_li	7,049.2	21.2	1.1	7,090.4	25.1	1.1	7,290.0	22.5	1.1
cit-HepPh	1,284.9	29.5	1.6	1,408.2	28.5	1.4	1,565.1	24.9	1.3
slashdot-zoo	623.3	6.6	0.7	791.7	7.2	0.7	1,732.7	10.1	0.9
wiki_talk_sv	14.0	1.4	0.1	20.9	1.8	0.3	43.3	1.7	0.7
munmun_twitter_social	147.0	9.6	0.9	246.7	10.7	1.4	355.2	7.8	1.9
wiki_talk_ja	12.9	1.3	0.1	1,334.5	5.7	1.0	2,066.2	6.8	1.0
wikipedia_link_gu	1,217.7	98.8	10.3	3,010.2	189.4	8.3	14,496.5	566.6	2.8
web-NotreDame	3,452.6	269.6	26.0	2,968.9	90.7	9.6	3,893.3	98.5	9.7
digg-friends	1,627.1	38.8	1.7	1,686.9	39.5	1.6	1,766.9	37.8	1.4
geometric mean	465.7	16.1	1.3	978.4	19.4	1.6	1,583.8	21.7	1.6

Table B.6: Geometric mean of the speedups of the dynamic algorithm over the static one over 100 batches of edge removals of size 1, 10, 100 in complex networks, for $k \in \{1, 10, 100\}$.

(a) Undirected networks

Network	$k = 1$			$k = 10$			$k = 100$		
Batch size	1	10	100	1	10	100	1	10	100
petster-cat-household	9.3	1.0	0.1	23.6	2.5	0.3	549.6	58.5	6.2
petster-catdog-household	33.9	3.3	0.3	55.1	5.5	0.6	147.0	11.6	1.5
petster-cat-friend	77.6	7.8	0.8	72.6	7.7	0.8	2,187.0	226.4	22.9
petster-friendships-cat	111.3	11.5	1.1	105.6	11.1	1.1	2,555.1	276.2	27.5
petster-friendships-dog	101.9	10.3	1.0	246.2	26.9	2.8	357.2	34.7	3.6
petster-dog-friend	55.5	5.7	0.6	194.0	22.2	2.2	285.4	28.9	3.1
petster-catdog-friend	882.3	89.8	9.2	965.5	104.4	9.9	1,078.2	118.5	11.2
higgs-twitter-social	196.4	18.1	1.9	523.1	55.9	5.9	729.3	79.3	7.9
petster-carnivore	5,726.3	591.3	55.9	5,864.1	585.3	56.3	6,122.9	535.7	50.7
geometric mean	135.5	13.7	1.4	223.8	23.8	2.5	845.9	84.3	8.7

(b) Directed networks

Network	$k = 1$			$k = 10$			$k = 100$		
Batch size	1	10	100	1	10	100	1	10	100
wikipedia_link_li	21,964.8	1,488.3	81.2	22,265.3	1,824.0	86.2	19,597.1	1,072.8	70.7
cit-HepPh	2,195.9	250.4	24.6	2,647.2	255.3	27.7	2,993.5	220.0	26.4
slashdot-zoo	1,357.1	154.0	15.8	1,621.5	160.0	20.2	3,208.1	195.0	35.2
wiki_talk_sv	14.6	2.3	0.3	15.3	2.6	0.6	21.3	2.0	1.3
munmun_twitter_social	77.9	13.2	1.8	58.4	13.3	3.2	33.0	5.7	3.8
wiki_talk_ja	11.3	2.3	0.3	2,038.9	333.3	48.9	1,495.7	131.6	52.0
wikipedia_link_gu	1,038.4	101.3	12.2	2,776.3	290.3	28.9	14,380.5	1,225.7	79.1
web-NotreDame	2,090.3	405.2	48.4	2,487.7	299.2	59.6	2,014.0	89.6	32.7
digg-friends	2,440.4	291.0	32.2	2,178.8	222.5	30.9	1,846.0	91.8	20.9
geometric mean	548.2	71.4	7.6	1,112.7	136.8	18.4	1,299.3	96.1	21.3

B.2.2 Speedups for Road Networks

Table B.7: Geometric mean of the speedups of the dynamic algorithm over the static one over 100 batches of edge insertions of size 1, 10, 100 in road networks, for $k \in \{1, 10, 100\}$.

(a) Undirected networks

Network	$k = 1$			$k = 10$			$k = 100$		
Batch size	1	10	100	1	10	100	1	10	100
DC	86.5	2.0	4.8	68.9	1.9	4.4	45.0	1.7	3.3
DE	617.4	4.7	1.7	411.9	4.5	1.6	159.9	3.6	1.6
RI	971.6	6.8	1.1	641.8	6.4	1.1	291.4	5.2	1.1
HI	4,962.1	34.9	5.2	4,151.2	32.1	4.9	3,017.2	23.6	4.3
AK	6,132.8	105.5	7.3	3,133.0	84.5	7.1	1,006.1	42.9	5.7
seychelles	3,021.0	85.3	77.8	1,952.6	70.1	67.8	1,102.9	40.8	44.8
VT	364.0	1.9	1.8	220.4	1.8	1.8	86.0	1.8	1.7
comores	4,263.6	27.6	55.5	3,593.0	23.4	48.9	2,416.3	15.9	34.2
liechtenstein	553.8	5.6	2.3	432.8	5.4	2.2	261.9	4.7	2.1
geometric mean	1,169.7	12.2	5.7	814.9	11.0	5.4	417.1	8.2	4.5

(b) Directed networks

Network	$k = 1$			$k = 10$			$k = 100$		
Batch size	1	10	100	1	10	100	1	10	100
kiribati	4,769.1	125.9	56.9	4,779.1	92.1	48.1	4,196.7	43.3	31.4
seychelles	3,021.0	85.3	77.8	1,952.6	70.1	67.8	1,102.9	40.8	44.8
comores	4,263.6	27.6	55.5	3,593.0	23.4	48.9	2,416.3	15.9	34.2
liechtenstein	553.8	5.6	2.3	432.8	5.4	2.2	261.9	4.7	2.1
isle-of-man	814.9	5.1	3.8	648.9	4.9	3.7	407.9	4.6	3.5
andorra	156.3	3.7	19.8	129.8	3.5	18.4	94.0	3.3	14.1
djibouti	1,176.9	10.3	2.1	879.4	9.6	2.1	559.5	7.8	2.0
faroe-islands	2,943.6	22.7	6.0	2,293.5	20.9	5.9	1,392.8	16.0	5.3
geometric mean	1,403.0	17.1	12.4	1,119.3	15.2	11.5	737.5	11.2	9.3

Table B.8: Geometric mean of the speedups of the dynamic algorithm over the static one over 100 batches of edge removals of size 1, 10, 100 in road networks, for $k \in \{1, 10, 100\}$.

(a) Undirected networks

Network	$k = 1$			$k = 10$			$k = 100$		
Batch size	1	10	100	1	10	100	1	10	100
DC	1,747.3	184.9	9.7	928.0	112.0	8.0	277.0	29.4	4.9
DE	9,620.9	1,400.1	42.6	3,436.6	878.8	35.7	708.3	180.8	21.3
RI	10,710.6	1,288.5	52.7	4,471.9	804.1	45.7	1,135.9	178.2	28.7
HI	14,180.5	1,623.4	74.8	10,152.9	925.2	60.6	5,853.5	211.6	32.6
AK	15,681.7	1,594.3	110.9	6,089.3	790.4	90.5	1,515.4	139.9	42.0
seychelles	14,249.3	883.4	199.2	8,486.8	525.9	160.4	4,141.5	175.1	75.3
VT	18,176.7	2,588.2	48.1	5,456.5	1,561.2	43.4	885.8	374.4	27.9
comores	43,088.2	2,521.7	338.6	30,364.7	1,016.6	263.1	17,394.5	190.9	149.5
liechtenstein	16,741.1	1,709.7	81.9	8,590.6	1,048.1	71.8	3,237.7	199.5	41.5
geometric mean	12,505.6	1,268.6	71.4	6,036.9	717.9	59.5	1,950.4	160.3	33.5

(b) Directed networks

Network	$k = 1$			$k = 10$			$k = 100$		
Batch size	1	10	100	1	10	100	1	10	100
kiribati	6,758.4	892.8	148.8	5,959.3	425.5	125.2	5,091.4	121.7	58.1
seychelles	14,249.3	883.4	199.2	8,486.8	525.9	160.4	4,141.5	175.1	75.3
comores	43,088.2	2,521.7	338.6	30,364.7	1,016.6	263.1	17,394.5	190.9	149.5
liechtenstein	16,741.1	1,709.7	81.9	8,590.6	1,048.1	71.8	3,237.7	199.5	41.5
isle-of-man	16,315.8	1,905.5	86.8	8,840.8	1,213.0	76.5	3,386.8	272.5	49.2
andorra	26,976.6	2,657.5	203.4	13,187.1	1,427.1	175.2	4,499.6	355.2	96.1
djibouti	24,321.3	2,149.1	135.6	10,358.8	1,082.8	109.8	3,229.7	208.8	63.8
faroe-islands	27,882.7	2,078.0	153.7	14,458.0	1,150.3	129.4	5,550.3	281.5	71.3
geometric mean	19,479.9	1,720.8	153.1	11,099.4	920.2	128.0	4,885.2	215.3	70.1

B.3 Running Time Results

B.3.1 Running Times for Complex Networks

Table B.9: Geometric mean of the update times over 100 batches of edge insertions of size 1, 10, 100 with $k = 10$ in complex networks. The columns "Static" and "Dyn." report the average time for the static and dynamic algorithm, respectively.

(a) Undirected networks

Network	Batch size 1		Batch size 10		Batch size 100	
	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)
petster-cat-household	0.535	0.022	0.539	0.219	0.569	1.980
petster-catdog-household	6.535	0.135	6.565	1.808	6.644	13.737
petster-cat-friend	8.243	0.100	8.239	1.002	7.853	9.250
petster-friendships-cat	9.466	0.084	9.163	0.846	9.457	7.740
petster-friendships-dog	55.566	0.222	55.485	2.170	56.892	23.316
petster-dog-friend	51.116	0.241	51.059	2.619	50.758	26.171
petster-catdog-friend	341.859	0.351	340.979	3.600	340.938	37.869
higgs-twitter-social	164.699	0.326	163.489	3.788	171.672	47.017
petster-carnivore	1,846.805	0.302	1,845.401	3.446	1,862.959	37.227

(b) Directed networks

Network	Batch size 1		Batch size 10		Batch size 100	
	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)
wikipedia_link_li	102.610	0.014	101.872	4.066	99.228	91.472
cit-HepPh	10.658	0.008	10.694	0.376	10.528	7.276
slashdot-zoo	17.731	0.022	17.153	2.382	17.875	25.584
wiki_talk_sv	0.173	0.008	0.173	0.097	0.174	0.596
munmun_twitter_social	3.576	0.014	3.554	0.332	3.642	2.675
wiki_talk_ja	64.789	0.049	63.567	11.103	63.453	65.358
wikipedia_link_gu	22.743	0.008	22.583	0.119	22.795	2.744
web-NotreDame	46.116	0.016	46.564	0.513	48.906	5.106
digg-friends	70.876	0.042	69.126	1.752	71.783	45.935

Table B.10: Geometric mean of the update times over 100 batches of edge removals of size 1, 10, 100 with $k = 10$ in complex networks. The columns "Static" and "Dyn." report the average time for the static and dynamic algorithm, respectively.

(a) Undirected networks

Network	Batch size 1		Batch size 10		Batch size 100	
	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)
petster-cat-household	0.549	0.023	0.541	0.214	0.578	1.930
petster-catdog-household	6.438	0.117	6.526	1.181	6.571	11.095
petster-cat-friend	7.913	0.109	7.851	1.017	8.193	9.914
petster-friendships-cat	9.188	0.087	9.136	0.821	9.466	8.240
petster-friendships-dog	55.773	0.227	57.535	2.137	57.275	20.475
petster-dog-friend	51.022	0.263	52.292	2.359	51.796	23.181
petster-catdog-friend	341.331	0.354	343.186	3.288	341.760	34.454
higgs-twitter-social	173.632	0.332	163.485	2.924	168.195	28.685
petster-carnivore	1,839.701	0.314	1,844.400	3.151	1,859.260	33.053

(b) Directed networks

Network	Batch size 1		Batch size 10		Batch size 100	
	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)
wikipedia_link_li	102.506	0.005	102.981	0.056	99.042	1.149
cit-HepPh	10.663	0.004	10.125	0.040	10.512	0.380
slashdot-zoo	16.826	0.010	17.142	0.107	18.252	0.901
wiki_talk_sv	0.172	0.011	0.173	0.066	0.176	0.294
munmun_twitter_social	3.553	0.061	3.530	0.265	3.800	1.179
wiki_talk_ja	63.633	0.031	64.686	0.194	64.092	1.310
wikipedia_link_gu	22.522	0.008	22.746	0.078	22.103	0.765
web-NotreDame	50.548	0.020	49.550	0.166	51.099	0.857
digg-friends	69.008	0.032	71.794	0.323	71.405	2.308

B.3.2 Running Times for Road Networks

Table B.11: Geometric mean of the update times over 100 batches of edge insertions of size 1, 10, 100 with $k = 10$ in road networks. The columns "Static" and "Dyn." report the average time for the static and dynamic algorithm, respectively.

(a) Undirected networks

Network	Batch size 1		Batch size 10		Batch size 100	
	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)
DC	1.987	0.029	2.054	1.090	0.546	0.125
DE	67.055	0.163	68.003	15.137	37.486	22.722
RI	71.916	0.112	71.798	11.204	53.473	47.266
HI	38.681	0.009	39.574	1.231	18.234	3.748
AK	112.411	0.036	36.267	0.429	14.521	2.035
seychelles	57.041	0.029	23.400	0.334	7.590	0.112
VT	259.058	1.176	257.643	139.302	141.648	79.793
comores	159.105	0.044	80.248	3.429	18.749	0.384
liechtenstein	195.378	0.451	151.988	27.980	55.933	25.127

(b) Directed networks

Network	Batch size 1		Batch size 10		Batch size 100	
	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)
kiribati	3.594	0.001	2.828	0.031	1.401	0.029
seychelles	57.041	0.029	23.400	0.334	7.590	0.112
comores	159.105	0.044	80.248	3.429	18.749	0.384
liechtenstein	195.378	0.451	151.988	27.980	55.933	25.127
isle-of-man	160.773	0.248	161.238	32.737	55.169	14.888
andorra	456.117	3.515	400.914	113.983	82.327	4.471
djibouti	582.899	0.663	279.441	29.029	92.643	44.369
faroe-islands	363.272	0.158	276.744	13.225	83.478	14.211

Table B.12: Geometric mean of the update times over 100 batches of edge removals of size 1, 10, 100 with $k = 10$ in road networks. The columns "Static" and "Dyn." report the average time for the static and dynamic algorithm, respectively.

(a) Directed networks

Network	Batch size 1		Batch size 10		Batch size 100	
	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)
DC	1.977	0.002	1.986	0.018	0.449	0.056
DE	67.036	0.020	66.847	0.076	33.948	0.950
RI	71.837	0.016	70.214	0.087	48.772	1.068
HI	39.809	0.004	31.872	0.034	13.693	0.226
AK	112.421	0.018	40.739	0.052	12.983	0.143
seychelles	58.981	0.007	22.552	0.043	5.903	0.037
VT	259.975	0.048	258.140	0.165	129.984	2.993
comores	173.579	0.006	95.454	0.094	15.694	0.060
liechtenstein	194.238	0.023	157.700	0.150	47.935	0.668

(b) Undirected networks

Network	Batch size 1		Batch size 10		Batch size 100	
	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)	Static (s)	Dyn. (s)
kiribati	3.646	0.001	2.560	0.006	1.266	0.010
seychelles	58.981	0.007	22.552	0.043	5.903	0.037
comores	173.579	0.006	95.454	0.094	15.694	0.060
liechtenstein	194.238	0.023	157.700	0.150	47.935	0.668
isle-of-man	161.947	0.018	158.235	0.130	48.768	0.637
andorra	458.637	0.035	399.007	0.280	60.552	0.346
djibouti	508.516	0.049	263.191	0.243	70.315	0.640
faroe-islands	351.846	0.024	258.117	0.224	62.096	0.480

C Appendix of Chapter 4

C.1 Running Time Results

In this appendix, we report the detailed running time of our algorithms on every instance. For better readability, we partitioned the instances into two categories: moderate instances achieved a total running time of less than 100 seconds (Table C.1); the others are shown in Table C.2. The deviation in running time among different runs of the same algorithm turned out to be small – e.g., around 3% for our *local-frame* algorithm using 36 cores, in geometric mean running time over all instances. As it is specifically small compared to our speedups, we report data on a single run per instance.

Table C.1: Absolute running times (s) on moderate instances. Total: ADS with preprocessing on a single core.

(a) OMP: OpenMP baseline, L: *local-frame*

Network Name		1 core		2 cores		4 cores		8 cores		16 cores		32 cores	
	Total	OMP	L	OMP	L	OMP	L	OMP	L	OMP	L	OMP	L
tntp-ChicagoRegional	6.70	6.62	5.66	3.25	2.83	1.56	1.37	0.85	0.66	0.45	0.33	0.27	0.16
munmun_twitter_social	7.99	1.72	1.49	1.41	0.83	1.09	0.45	0.89	0.24	0.84	0.23	0.78	0.17
com-amazon	10.49	9.47	9.18	4.47	4.38	3.02	2.35	2.27	1.34	1.94	0.86	1.41	0.54
loc-gowalla_edges	2.82	2.50	2.09	1.49	0.99	1.11	0.49	0.87	0.20	0.70	0.11	0.67	0.10
web-NotreDame	7.66	7.33	6.55	4.34	3.30	3.17	1.72	2.50	0.68	2.14	0.43	1.93	0.33
web-Stanford	34.62	33.87	29.95	15.76	15.54	11.62	7.95	7.96	2.79	5.49	1.75	4.48	1.33
petster-dog-household	5.31	4.83	3.89	2.67	2.12	1.82	1.09	1.43	0.67	1.30	0.56	1.32	0.42
flixster	13.99	10.94	10.03	7.87	5.77	6.61	3.20	5.49	1.91	4.90	1.32	4.78	1.32
as-skitter	17.14	13.76	13.16	9.85	7.57	7.33	3.99	5.80	2.55	5.14	1.77	5.11	2.21
actor-collaboration	8.69	5.87	6.21	3.88	3.18	2.60	1.96	1.82	1.16	1.41	0.68	1.09	0.54
soc-pokec-relationships	25.38	16.57	18.21	10.37	9.00	8.00	5.23	6.07	3.02	5.28	2.56	5.40	2.08
soc-LiveJournal1	54.91	36.52	39.08	31.53	22.37	22.29	11.68	17.79	6.12	15.57	4.69	14.82	4.03
livejournal-links	62.27	46.19	44.52	31.16	24.99	23.49	13.43	18.11	7.57	15.46	4.90	15.51	4.33
wikipedia_link_sh	41.54	21.68	17.98	17.43	9.49	14.74	4.68	13.13	2.44	12.36	2.05	12.08	2.11
wikipedia_link_sr	56.30	45.55	42.66	32.21	21.83	20.28	10.69	15.63	6.08	12.92	3.72	12.73	2.69

(b) S: *shared-frame*, I: *indexed-frame*

Network Name		1 core		2 cores		4 cores		8 cores		16 cores		32 cores	
	Total	S	I	S	I	S	I	S	I	S	I	S	I
tntp-ChicagoRegional	6.70	6.71	5.48	3.30	2.75	1.48	1.38	0.64	0.70	0.31	0.43	0.15	0.29
munmun_twitter_social	7.99	1.51	1.60	0.80	0.90	0.45	0.49	0.28	0.29	0.20	0.17	0.19	0.23
com-amazon	10.49	8.44	8.88	4.04	4.21	2.34	2.52	1.43	1.48	0.78	1.22	0.42	0.89
loc-gowalla_edges	2.82	2.23	2.34	0.79	1.11	0.46	0.53	0.28	0.31	0.14	0.17	0.08	0.11
web-NotreDame	7.66	6.34	6.81	3.18	3.52	1.40	1.58	0.76	0.99	0.52	0.66	0.22	0.60
web-Stanford	34.62	28.81	29.51	14.27	14.01	5.73	8.68	4.41	5.50	1.80	2.13	1.18	1.50
petster-dog-household	5.31	4.24	3.68	2.21	1.95	1.05	1.14	0.66	0.75	0.62	0.78	0.46	0.73
flixster	13.99	10.30	11.02	6.03	6.00	3.01	3.31	2.25	1.97	1.36	1.45	1.33	1.90
as-skitter	17.14	13.26	14.23	7.59	8.13	4.11	4.42	2.42	2.44	1.50	2.08	1.13	1.74
actor-collaboration	8.69	6.28	5.48	3.28	3.27	1.85	1.71	1.06	1.03	0.68	0.69	0.48	1.01
soc-pokec-relationships	25.38	16.22	17.18	8.77	9.38	5.57	5.79	3.40	2.98	2.15	2.00	1.40	2.49
soc-LiveJournal1	54.91	35.60	40.49	22.70	18.93	13.13	14.15	7.04	8.28	4.69	6.32	3.29	5.72
livejournal-links	62.27	44.49	44.85	25.44	24.65	12.45	15.29	7.69	8.53	5.07	6.89	3.77	6.95
wikipedia_link_sh	41.54	17.94	21.49	9.81	11.86	4.89	7.04	2.78	3.88	1.96	1.68	1.55	2.26
wikipedia_link_sr	56.30	45.93	41.54	24.91	23.70	11.36	13.18	7.09	6.54	4.44	5.11	2.95	5.23

Table C.2: Absolute running times (s) on expensive instances. Total: ADS with preprocessing on a single core.

(a) OMP: OpenMP baseline, L: *local-frame*

Network Name		1 core		2 cores		4 cores		8 cores		16 cores		32 cores	
	Total	OMP	L	OMP	L	OMP	L	OMP	L	OMP	L	OMP	L
dimacs9-NY	249	246	212	97	106	54	55	30	28	19	14	10	6
dimacs9-COL	405	397	358	177	177	101	94	55	47	30	23	17	11
roadNet-PA	1,961	1,937	1,851	1,027	942	521	458	284	235	148	121	92	59
roadNet-TX	1,965	1,937	2,001	1,042	1,035	544	496	279	250	165	130	89	64
dbpedia-all	412	402	395	227	215	126	80	76	42	54	22	41	19
wikipedia_link_ceb	1,337	1,272	1,435	701	707	415	307	238	160	156	98	121	74
wikipedia_link_ru	142	126	132	89	73	52	44	36	23	26	12	24	12
wikipedia_link_de	155	145	182	106	100	54	57	37	21	25	12	20	13
wikipedia_link_it	152	112	145	82	70	58	41	27	24	20	14	16	9
wikipedia_link_sv	444	423	472	258	255	160	105	105	61	73	32	60	31
wikipedia_link_fr	194	168	177	131	106	75	58	41	30	32	14	24	12
orkut-links	206	107	110	66	64	44	35	27	19	19	10	15	9

(b) S: *shared-frame*, I: *indexed-frame*

Network Name		1 core		2 cores		4 cores		8 cores		16 cores		32 cores	
	Total	S	I	S	I	S	I	S	I	S	I	S	I
dimacs9-NY	249	198	203	95	101	49	53	26	31	13	14	6	9
dimacs9-COL	405	345	339	177	171	94	93	50	49	22	25	11	17
roadNet-PA	1,961	1,975	1,781	998	950	471	463	242	252	113	151	60	82
roadNet-TX	1,965	1,994	1,943	1,022	1,038	504	514	232	277	119	165	63	89
dbpedia-all	412	411	391	214	205	96	83	49	66	18	25	14	36
wikipedia_link_ceb	1,337	1,421	1,268	691	665	370	399	158	162	86	135	68	117
wikipedia_link_ru	142	134	125	67	69	42	35	25	26	14	19	11	16
wikipedia_link_de	155	155	170	80	98	39	56	24	32	12	19	8	15
wikipedia_link_it	152	120	129	74	68	38	45	21	26	11	20	10	16
wikipedia_link_sv	444	438	429	272	246	111	108	55	58	30	46	24	54
wikipedia_link_fr	194	181	168	96	92	55	60	30	35	15	29	14	20
orkut-links	206	119	107	60	66	34	37	19	25	10	20	7	15

D Appendix of Chapter 5

D.1 Baseline

Table D.1: Precision of the diagonal entries computed by the LAMG solver (tolerance: 10^{-9}) compared with the ones computed by the Matlab `pinv` function.

Network	Type	n	m	diam.	max abs. err. diag.	E_{rel}	$L1_{rel}$	$L2_{rel}$	Ranking
moreno-lesmis	characters	77	254	5	0.000,0	0.00%	0.00%	0.00%	0.48%
petster-hamster-household	social	874	4,003	8	0.000,6	0.23%	0.13%	0.07%	0.02%
subelj-euroroad	infrastr.	1,039	1,305	62	0.003,1	0.12%	0.09%	0.05%	0.00%
arenas-email	emails	1,133	5,451	8	0.000,2	0.13%	0.07%	0.03%	0.00%
dimacs10-polblogs	web	1,222	16,714	8	0.000,2	0.18%	0.07%	0.02%	0.01%
maayan-faa	infrastr.	1,226	2,408	17	0.000,5	0.08%	0.06%	0.03%	0.00%
petster-hamster-friend	social	1,788	12,476	14	0.000,3	0.15%	0.07%	0.02%	0.01%
petster-hamster	social	2,000	16,098	10	0.000,1	0.09%	0.04%	0.02%	0.01%
wikipedia-link-lo	web	3,733	82,977	9	0.000,1	0.05%	0.02%	0.01%	0.03%
advogato	social	5,042	39,227	9	0.000,1	0.03%	0.02%	0.01%	0.01%
p2p-Gnutella06	computer	8,717	31,525	10	0.000,0	0.01%	0.01%	0.00%	0.00%
p2p-Gnutella05	computer	8,842	31,837	9	0.000,1	0.02%	0.01%	0.01%	0.00%

D.2 Running Time – Forest Closeness

Table D.2: Running time (s) of UST on the networks of Table 5.3.

(a) Complex networks

Graph	Time (s)					
ε	0.05	0.1	0.2	0.3	0.4	0.5
loc-brightkite_edges	46.4	11.6	3.0	1.4	0.8	0.5
douban	80.8	20.5	5.2	2.4	1.5	0.9
soc-Epinions1	55.5	14.0	3.5	1.6	1.0	0.7
slashdot-zoo	59.9	15.6	3.8	1.8	1.1	0.7
petster-cat-household	61.8	15.7	4.0	1.8	1.1	0.8
wikipedia_link_fy	58.2	15.0	3.9	1.9	1.1	0.8
loc-gowalla_edges	230.9	63.0	15.7	7.1	4.4	2.8
wikipedia_link_an	50.7	12.1	3.1	1.5	0.9	0.7
wikipedia_link_ga	44.8	11.3	3.1	1.6	1.1	0.8
petster-dog-household	359.6	87.7	22.5	10.3	6.0	4.1
livemocha	107.4	28.6	7.3	3.5	2.1	1.5

(b) Road networks

Graph	Time (s)					
ε	0.05	0.1	0.2	0.3	0.4	0.5
loc-brightkite_edges	46.4	11.6	3.0	1.4	0.8	0.5
douban	80.8	20.5	5.2	2.4	1.5	0.9
soc-Epinions1	55.5	14.0	3.5	1.6	1.0	0.7
slashdot-zoo	59.9	15.6	3.8	1.8	1.1	0.7
petster-cat-household	61.8	15.7	4.0	1.8	1.1	0.8
wikipedia_link_fy	58.2	15.0	3.9	1.9	1.1	0.8
loc-gowalla_edges	230.9	63.0	15.7	7.1	4.4	2.8
wikipedia_link_an	50.7	12.1	3.1	1.5	0.9	0.7
wikipedia_link_ga	44.8	11.3	3.1	1.6	1.1	0.8
petster-dog-household	359.6	87.7	22.5	10.3	6.0	4.1
livemocha	107.4	28.6	7.3	3.5	2.1	1.5

D.3 Running Time – Group Forest Closeness

Table D.3: Running time (s) of our greedy algorithm for group forest maximization.

Graph	Group size	Time (s)
cora	200	1,559.3
	400	2,210.6
	600	2,663.4
citeseer	200	2,518.6
	400	3,666.5
	600	4,642.4

E Appendix of Chapter 6

E.1 Running Time Results

Table E.1: Running times (s) of group-closeness maximization algorithms on the unweighted graphs of Table 6.1a; $k = 10$. For our local search algorithms, we average data over five runs using the arithmetic mean.

Network	Greedy	LS-restrict	LS	GS	GS-local	GS-extended
dimacs9-NY	1,985.0	2.0	1.0	1.6	0.2	46.8
dimacs9-BAY	3,792.9	3.8	2.5	3.4	0.4	69.1
web-Stanford	699.1	0.9	0.4	1.5	0.9	7.4
hyves	1,458.7	3.9	1.5	5.8	4.5	9.9
youtube-links	829.5	2.7	1.5	4.6	4.0	6.7
com-youtube	803.9	3.3	1.3	4.2	4.3	6.0
web-Google	760.3	3.1	0.9	5.0	2.6	8.2
trec-wt10g	5,545.3	3.6	1.6	6.3	5.0	19.3
dimacs10-eu-2005	4,987.9	2.0	1.8	6.2	3.6	11.5
soc-pokec-relationships	4,845.1	13.2	5.4	25.5	13.3	37.8
wikipedia_link_ca	1,314.7	12.0	6.8	15.1	12.8	21.6

Table E.2: Running times (s) of group-closeness maximization algorithms on the weighted graphs of Table 6.1b. For our local search algorithms, we average data over five runs using the arithmetic mean.

Network	Greedy			GS			GS-local		
	$k = 5$	$k = 10$	$k = 100$	$k = 5$	$k = 10$	$k = 100$	$k = 5$	$k = 10$	$k = 100$
DC	2.81	2.96	3.30	0.11	0.07	0.11	0.02	0.03	0.11
HI	13.12	13.29	14.53	0.08	0.08	0.29	0.04	0.07	0.31
AK	74.27	77.36	82.13	0.12	0.25	1.04	0.06	0.12	0.74
DE	76.72	80.56	85.12	0.08	0.33	0.59	0.13	0.16	0.90
RI	62.96	67.01	72.94	0.10	0.37	1.10	0.13	0.18	0.95
CT	606.73	643.37	705.91	3.08	2.42	3.79	0.59	0.87	4.08
ME	867.62	908.08	1,003.82	2.73	2.69	5.67	0.76	0.80	5.56
ND	1,679.77	1,720.69	1,874.39	2.64	2.69	5.97	0.77	1.02	5.41
SD	1,358.51	1,425.83	1,562.83	2.19	1.18	4.89	1.37	1.08	6.09
WY	1,347.51	1,462.56	1,663.29	1.14	3.37	8.75	1.07	1.32	7.71
ID	2,347.94	2,475.44	2,674.39	3.96	5.20	9.53	1.35	1.65	9.23
MD	3,037.70	3,201.99	3,408.18	3.56	4.09	8.43	1.17	2.35	8.54
WV	2,121.26	2,341.11	2,541.09	6.78	9.35	13.22	1.35	1.94	9.91
NE	3,485.90	3,707.90	3,991.72	7.23	6.90	12.80	1.94	2.01	10.22

F Appendix of Chapter 7

F.1 Approximation for Group-Closeness in the Sense of Li et al.

As explained in Ref. [12, Appendix C], the approach of Li et al. [183] works for minimizing a general supermodular monotone non-increasing function $f(\cdot)$ with respect to a cardinality constraint. They let $x_1^* \in \text{argmax}_x\{f(\emptyset) - f(\{x\})\}$ and use the greedy algorithm on the set function

$$g(S) := f(\{x_1^*\}) - f(\{x_1^*\} \cup S),$$

which is a monotone non-decreasing submodular set function with $g(\emptyset) = 0$. Thus, the greedy algorithm maximizes the function with respect to a cardinality constraint within an approximation factor of $1 - 1/e$ [226]. However, there are two caveats. First, the greedy algorithm uses a budget of $k - 1$ instead of k (budget of one is spent on identifying x_1^*) and thus Li et al. obtain an approximation factor of $1 - k/((k-1)e)$. Second, and most importantly, observe that the approximation factor is obtained on the function $g(S)$ and not $f(S)$, i.e., they get a set S of size $k - 1$ such that

$$f(\{x_1^*\}) - f(\{x_1^*\} \cup S) \geq \left(1 - \frac{k}{(k-1)e}\right) \cdot (f(\{x_1^*\}) - f(\{x_1^*\} \cup S^*)),$$

where S^* is the optimal set of size $k - 1$ to be added to $\{x_1^*\}$ with the goal of minimizing $f(\cdot)$. We remark that this set is not necessarily related to the set that minimizes $f(\cdot)$ with respect to the cardinality constraint. Clearly, this approach can be applied for the submodular farness function $g_f(\cdot)$ in place of $f(\cdot)$. It can not, however, provide an approximation algorithm for $g_f(\cdot)$ in the usual sense – and furthermore it would not be easily extendable to $g_c(\cdot)$.

Table F.2: Small networks used for group-closeness experiments with ILP solver.

(a) Complex networks

Graph	Type	n	m
dimacs10-celegans_metabolic	U	453	2,025
arenas-meta	U	453	2,025
contact	U	274	2,124
arenas-jazz	U	198	2,742
sociopatterns-infectious	U	410	2,765
dnc-corecipient	U	849	10,384
moreno_oz	D	214	2,658
wiki_talk_lv	D	510	2,783
wiki_talk_eu	D	617	2,811
dnc-temporalGraph	D	520	3,518
dimacs10-celegansneural	D	297	4,296
wiki_talk_bn	D	700	4,316
wiki_talk_eo	D	822	6,076
wiki_talk_gl	D	1,009	7,435

(b) High-diameter networks

Graph	Type	n	m
tuvalu	UU	152	187
niue	UU	461	529
nauru	UU	618	729
dimacs10-netscience	UU	379	914
asoiaf	UU	796	2,823
tuvalu	UW	152	187
niue	UW	461	529
nauru	UW	618	729
tuvalu	DU	152	374
niue	DU	461	1,055
librec-filmtrust-trust	DU	267	1,099
nauru	DU	618	1,427
tuvalu	DW	152	374
niue	DW	461	1,055
nauru	DW	618	1,427

F.2 Instances Statistics

Table F.1: Small networks used for group-harmonic experiments with ILP solver.

(a) Complex networks

Graph	Type	n	m
convote	U	219	586
dimacs10-football	U	115	613
wiki_talk_ht	U	537	787
moreno_innovation	U	241	1,098
dimacs10-celegans_metabolic	U	453	2,025
arenas-meta	U	453	2,025
foodweb-baywet	U	128	2,106
contact	U	275	2,124
foodweb-baydry	U	128	2,137
moreno_oz	U	217	2,672
arenas-jazz	U	198	2,742
sociopatterns-infectious	U	411	2,765
dimacs10-celegansneural	U	297	4,296
radoslaw_email	U	168	5,783
convote	D	219	586
wiki_talk_ht	D	537	787
moreno_innovation	D	241	1,098
foodweb-baywet	D	128	2,106
foodweb-baydry	D	128	2,137
moreno_oz	D	217	2,672
dimacs10-celegansneural	D	297	4,296
radoslaw_email	D	168	5,783

(b) High-diameter networks

Graph	Type	n	m
dbpedia-similar	UU	430	564
niue	UU	461	1,055
tuvalu	UU	436	1,082
librec-filmtrust-trust	UU	874	1,853
niue	UW	461	1,055
tuvalu	UW	436	1,082
niue	DU	461	1,055
tuvalu	DU	436	1,082
librec-filmtrust-trust	DU	874	1,853
niue	DW	461	1,055
tuvalu	DW	436	1,082

Table F.3: Large networks used for group-harmonic experiments.

(a) Complex networks

Graph	Type	n	m
petster-hamster-household	U	874	4,003
petster-hamster-friend	U	1,788	12,476
petster-hamster	U	2,000	16,098
loc-brightkite_edges	U	58,228	214,078
douban	U	154,908	327,162
petster-cat-household	U	105,138	494,858
loc-gowalla_edges	U	196,591	950,327
wikipedia_link_fy	U	65,562	1,071,668
wikipedia_link_ckb	U	60,722	1,176,289
petster-dog-household	U	260,390	2,148,179
livemocha	U	104,103	2,193,083
flickrEdges	U	105,938	2,316,948
petster-friendships-cat	U	149,700	5,448,197
wikipedia_link_mi	D	7,996	116,457
foldoc	D	13,356	120,238
wikipedia_link_so	D	7,439	125,046
wikipedia_link_lo	D	3,811	132,837
wikipedia_link_co	D	8,252	177,420

(b) High-diameter networks

Graph	Type	n	m
marshall-islands	UU	1,080	2,557
micronesia	UU	1,703	3,600
kiribati	UU	1,867	4,412
opsahl-powergrid	UU	4,941	6,594
samoa	UU	6,926	15,217
comores	UU	7,250	17,554
marshall-islands	UW	1,080	2,557
micronesia	UW	1,703	3,600
kiribati	UW	1,867	4,412
DC	UW	9,522	14,807
samoa	UW	6,926	15,217
comores	UW	7,250	17,554
marshall-islands	DU	1,080	2,557
micronesia	DU	1,703	3,600
kiribati	DU	1,867	4,412
samoa	DU	6,926	15,217
comores	DU	7,250	17,554
opsahl-openflights	DU	2,939	30,501
tntp-ChicagoRegional	DU	12,982	39,018
marshall-islands	DW	1,080	2,557
micronesia	DW	1,703	3,600
kiribati	DW	1,867	4,412
samoa	DW	6,926	15,217
comores	DW	7,250	17,554

Table F.4: Large (strongly) connected components of the networks in Table F.3 used for group-closeness experiments.

(a) Complex networks

Graph	Type	n	m
loc-brightkite_edges	U	56,739	212,945
douban	U	154,908	327,162
petster-cat-household	U	68,315	494,562
wikipedia_link_ckb	U	60,257	801,794
wikipedia_link_fy	U	65,512	921,533
livemocha	U	104,103	2,193,083
wikipedia_link_mi	D	3,696	99,237
wikipedia_link_lo	D	1,622	109,577
wikipedia_link_so	D	5,149	114,922
foldoc	D	13,274	119,485
wikipedia_link_co	D	5,150	160,474
web-NotreDame	D	53,968	296,228
slashdot-zoo	D	26,997	333,425
soc-Epinions1	D	32,223	443,506
wikipedia_link_jv	D	39,248	1,059,059

(b) High-diameter networks

Graph	Type	n	m
seychelles	UU	3,907	4,322
comores	UU	3,789	4,630
andorra	UU	4,219	4,933
opsahl-powergrid	UU	4,941	6,594
liechtenstein	UU	6,215	7,002
faroe-islands	UU	12,129	13,165
seychelles	UW	3,907	4,322
comores	UW	3,789	4,630
andorra	UW	4,219	4,933
liechtenstein	UW	6,215	7,002
faroe-islands	UW	12,129	13,165
DC	UW	9,522	14,807
seychelles	DU	3,907	8,225
andorra	DU	4,160	8,288
comores	DU	3,789	8,952
liechtenstein	DU	6,205	13,591
faroe-islands	DU	12,077	25,679
opsahl-openflights	DU	2,868	30,404
tntp-ChicagoRegional	DU	12,978	39,017
seychelles	DW	3,907	8,225
andorra	DW	4,160	8,288
comores	DW	3,789	8,952
liechtenstein	DW	6,205	13,591
faroe-islands	DW	12,077	25,679

F.3 Running Times – Group-Harmonic Maximization

Table F.5: Running times (s) of Greedy-H and Greedy-LS-H on the complex networks of Table F.3a.

(a) Undirected unweighted

Graph	Greedy-H			Greedy-LS-H		
k	5	10	50	5	10	50
petster-hamster-household	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
petster-hamster-friend	<0.1	<0.1	<0.1	<0.1	<0.1	0.1
petster-hamster	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
loc-brightkite_edges	1.1	1.0	1.1	4.3	6.6	25.8
douban	8.1	8.1	8.4	40.3	86.3	303.0
petster-cat-household	0.1	0.2	0.3	19.5	23.8	106.1
loc-gowalla_edges	8.9	8.4	8.7	59.8	97.3	1,064.5
wikipedia_link_fy	3.8	3.8	4.0	13.3	15.7	137.9
wikipedia_link_ckb	7.3	7.3	7.4	12.9	14.6	80.2
petster-dog-household	10.3	10.4	10.7	131.9	212.3	843.8
livemocha	11.2	11.4	11.8	52.5	64.6	277.9
flickrEdges	44.2	45.4	46.4	119.5	128.4	217.6
petster-friendships-cat	2.7	2.8	2.9	35.6	55.1	266.7

(b) Directed unweighted

Graph	Greedy-H			Greedy-LS-H		
k	5	10	50	5	10	50
wikipedia_link_mi	0.3	0.3	0.3	0.6	1.0	3.8
foldoc	0.6	0.6	0.6	1.6	1.7	14.7
wikipedia_link_so	0.1	0.1	0.1	0.3	0.4	2.2
wikipedia_link_lo	0.2	0.2	0.2	0.2	0.2	0.7
wikipedia_link_co	0.2	0.3	0.3	0.5	0.5	2.6

Table F.6: Running times (s) of Greedy-H and Greedy-LS-H on the high-diameter networks of Table F.3b.

(a) Undirected unweighted

Graph	Greedy-H			Greedy-LS-H		
k	5	10	50	5	10	50
marshall-islands	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
micronesia	<0.1	<0.1	<0.1	<0.1	<0.1	0.2
kiribati	<0.1	<0.1	<0.1	<0.1	<0.1	0.3
opsahl-powergrid	0.2	0.2	0.2	1.7	0.9	1.4
samoa	0.8	0.9	0.9	2.6	2.9	5.5
comores	0.5	0.5	0.6	1.1	2.3	8.6

Table (F.7) Undirected weighted

Graph	Greedy-H			Greedy-LS-H		
k	5	10	50	5	10	50
marshall-islands	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
micronesia	<0.1	<0.1	<0.1	<0.1	<0.1	0.2
kiribati	<0.1	<0.1	<0.1	<0.1	<0.1	0.3
opsahl-powergrid	0.2	0.2	0.2	1.7	0.9	1.4
samoa	0.8	0.9	0.9	2.6	2.9	5.5
comores	0.5	0.5	0.6	1.1	2.3	8.6

(a) Directed unweighted

Graph	Greedy-H			Greedy-LS-H		
k	5	10	50	5	10	50
marshall-islands	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
micronesia	<0.1	<0.1	<0.1	<0.1	<0.1	0.2
kiribati	<0.1	<0.1	<0.1	<0.1	<0.1	0.3
samoa	0.8	0.9	0.9	2.6	1.9	5.9
comores	0.5	0.5	0.6	1.1	3.3	10.5
opsahl-openflights	<0.1	<0.1	<0.1	<0.1	<0.1	0.2
tntp-ChicagoRegional	2.7	2.9	3.2	11.5	20.6	85.8

(b) Directed weighted

Graph	Greedy-H			Greedy-LS-H		
k	5	10	50	5	10	50
marshall-islands	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
micronesia	<0.1	<0.1	<0.1	<0.1	<0.1	0.2
kiribati	<0.1	<0.1	<0.1	<0.1	<0.1	0.3
samoa	0.8	0.9	0.9	2.6	1.9	5.9
comores	0.5	0.5	0.6	1.1	3.3	10.5
opsahl-openflights	<0.1	<0.1	<0.1	<0.1	<0.1	0.2
tntp-ChicagoRegional	2.7	2.9	3.2	11.5	20.6	85.8

F.4 Running Times – Group-Closeness Maximization

Table F.8: Running time (s) of GS-LS-C and Greedy-LS-C on the complex networks of Table F.4a.

(a) Undirected unweighted

Graph	GS-LS-C			Greedy-LS-C		
k	5	10	50	5	10	50
loc-brightkite_edges	11.8	22.1	146.4	11.5	20.8	110.9
douban	35.0	59.4	222.0	26.3	43.5	202.5
petster-cat-household	32.7	66.1	363.2	32.2	63.2	341.5
wikipedia_link_fy	100.2	102.5	476.9	27.5	50.3	434.7
wikipedia_link_ckb	19.7	103.2	767.2	19.4	34.1	718.3
livemocha	58.1	86.3	713.0	46.5	58.2	604.9

(b) Directed unweighted

Graph	GS-LS-C			Greedy-LS-C		
k	5	10	50	5	10	50
wikipedia_link_mi	<0.1	0.8	2.9	0.1	0.2	1.8
foldoc	2.3	3.5	0.5	1.7	2.2	5.7
wikipedia_link_so	0.7	1.5	23.7	0.5	0.9	3.3
wikipedia_link_lo	0.8	1.6	26.0	0.5	2.1	13.5
wikipedia_link_co	0.8	1.7	42.9	1.2	1.7	18.5
soc-Epinions1	4.2	6.9	30.1	3.6	6.0	28.2
slashdot-zoo	4.1	6.4	19.9	3.4	7.1	15.4
web-NotreDame	14.9	37.3	1,106.5	14.4	23.4	388.6
wikipedia_link_jv	22.9	97.1	30.0	17.7	14.5	49.9

Table F.9: Running time (s) of GS-LS-C and Greedy-LS-C on the high-diameter networks of Table F.4b.

(a) Undirected unweighted

Graph	GS-LS-C			Greedy-LS-C		
k	5	10	50	5	10	50
opsahl-powergrid	0.9	1.1	13.4	0.7	0.4	3.7
andorra	3.6	8.9	55.1	1.9	3.9	26.7
seychelles	1.6	5.3	26.7	0.9	3.4	25.0
liechtenstein	10.8	21.2	56.3	2.2	16.4	38.2
comores	1.2	5.0	22.9	1.4	4.9	18.0
faroe-islands	33.9	77.2	313.5	25.3	96.4	268.4

(b) Directed unweighted

Graph	GS-LS-C			Greedy-LS-C		
k	5	10	50	5	10	50
andorra	5.2	6.2	5.0	4.2	3.5	26.0
seychelles	1.4	5.1	21.0	0.8	3.3	17.6
liechtenstein	17.7	14.4	59.7	4.0	15.5	41.3
comores	1.7	4.3	7.7	1.2	4.1	19.2
faroe-islands	20.2	77.0	254.1	25.7	44.8	189.4
opsahl-openflights	<0.1	0.1	1.0	<0.1	0.1	0.6
tntp-ChicagoRegional	45.3	151.4	0.3	32.5	68.2	304.3

(c) Undirected weighted

Graph	GS-LS-C			Greedy-LS-C		
k	5	10	50	5	10	50
andorra	20.5	35.5	182.0	4.5	10.9	64.3
seychelles	2.6	13.7	93.1	2.3	3.3	62.6
liechtenstein	3.8	8.6	230.3	4.1	27.0	265.6
DC	7.8	18.9	473.2	9.9	14.0	98.3
comores	2.3	10.1	140.1	2.3	9.6	55.3
faroe-islands	17.1	137.5	907.0	15.6	27.4	411.3

(d) Directed weighted

Graph	GS-LS-C			Greedy-LS-C		
k	5	10	50	5	10	50
andorra	5.9	16.0	129.3	6.4	5.8	52.8
seychelles	2.1	2.8	59.2	2.3	2.7	29.6
liechtenstein	3.7	16.9	227.5	3.8	22.2	167.9
comores	1.9	7.0	90.0	2.2	10.7	28.1
faroe-islands	16.2	148.1	696.2	15.3	27.2	98.5

G Appendix of Chapter 8

G.1 Running Times

Table G.1: Running time (s) of GBC, GCC, GHC, and GED-Walk maximization (both lazy and stochastic algorithms) on 36 cores for groups with size 5 to 100.

	$k = 5$		$k = 10$		$k = 20$		$k = 50$		$k = 100$	
	GBC	GED	GBC	GED	GBC	GED	GBC	GED	GBC	GED
Network	GCC	GED-S	GCC	GED-S	GCC	GED-S	GCC	GED-S	GCC	GED-S
	GHC		GHC		GHC		GHC		GHC	
	36.7	1.2	71.7	1.5	141.1	2.5	351.7	8.3	709.2	16.4
com-dblp	29.4	1.1	30.3	1.4	33.0	2.4	41.0	5.5	48.3	11.4
	24.8		25.0		25.1		25.8		27.4	
	17.7	0.3	35.3	0.5	70.2	0.9	175.1	2.0	354.3	4.6
wikipedia_link_mr	0.9	0.3	1.0	0.6	1.1	0.8	1.5	1.8	2.0	4.0
	0.6		0.6		0.7		0.7		0.9	
	66.3	3.2	131.9	4.1	261.2	7.3	645.5	11.5	1,292.3	28.3
roadNet-PA	3,903.0	3.0	4,150.4	4.1	4,353.8	5.2	4,676.0	13.4	4,911.8	30.0
	5,280.3		5,616.2		5,163.0		3,837.0		5,431.1	
	74.3	1.1	146.4	1.4	290.0	2.6	717.9	5.5	1,440.2	11.8
citeseer	52.5	1.1	56.9	1.5	62.5	2.3	76.5	5.8	87.7	10.1
	53.3		54.3		56.0		58.9		62.4	
	83.9	2.4	165.3	3.4	325.6	4.9	814.9	9.4	1,634.9	28.4
roadNet-TX	4,903.0	2.4	5,230.8	3.2	5,675.7	4.6	6,149.2	14.8	6,485.9	26.4
	4,531.6		4,927.1		6,900.1		5,214.9		5,433.4	
	21.7	0.7	43.1	1.1	88.0	3.2	223.8	6.6	450.6	12.8
web-Stanford	57.8	0.7	59.8	1.1	63.2	2.1	63.0	4.2	66.1	8.1
	10.3		10.9		10.8		11.1		11.7	
	83.2	0.4	164.9	0.5	328.1	0.8	823.5	1.9	1,652.9	4.0
petster-dog-household	17.3	0.4	17.4	0.6	17.8	0.9	19.8	1.8	24.1	3.5
	4.2		4.2		4.3		4.6		5.0	
	41.0	0.7	81.5	1.1	162.7	1.7	406.3	3.9	815.9	8.4
wikipedia_link_bn	6.0	0.7	6.2	1.1	7.2	1.7	8.8	3.6	10.7	6.5
	2.9		2.9		3.1		3.3		3.6	
	95.6	0.5	189.0	0.7	376.2	1.4	935.6	3.2	1,874.0	6.4
petster-catdog-household	61.5	0.6	61.8	0.8	62.3	1.5	64.4	3.2	70.2	6.0
	49.5		50.3		47.8		49.5		50.2	
	46.1	0.7	91.4	1.1	182.5	1.7	461.0	4.2	932.3	9.1
wikipedia_link_uz	20.7	0.8	21.3	1.1	22.0	1.9	25.2	4.3	31.1	9.5
	33.3		33.5		33.7		34.1		34.8	
	21.3	1.0	42.0	1.4	84.2	2.5	207.5	4.4	412.6	6.8
dimacs9-COL	454.0	1.0	480.6	1.2	529.5	1.9	578.7	4.2	616.8	38.0
	570.7		618.3		681.7		683.6		607.2	
	48.8	0.7	95.7	1.0	189.9	1.5	475.0	3.8	952.1	9.1
munmun_twitter_social	20.1	0.7	21.4	0.9	24.9	1.5	36.4	3.4	42.6	6.4
	79.4		80.6		80.7		82.8		85.7	

H Appendix of Chapter 9

H.1 Affected Vertices

Tables H.1–H.4 report the average number of vertices affected by a batch of $b \in \{1, \ldots, 10^4\}$ edge updates on all the considered instances.

Table H.1: Average number of affected vertices in the road networks of Table 9.1.

Graph	Edge insertions: Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$	Graph	Edge removals: Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
be	1.2	12.0	113.5	1,143.7	12,598.5	be	1.3	12.0	113.6	1,144.9	12,945.1
cz	1.3	11.9	116.0	1,135.4	12,179.6	cz	1.5	11.9	116.2	1,135.9	12,383.4
fi	0.9	10.2	109.9	1,121.2	11,983.8	fi	1.0	10.2	109.9	1,123.1	12,158.1
au	0.8	11.3	116.2	1,156.7	12,236.6	au	1.0	11.3	116.2	1,160.1	12,413.1
ca	1.1	12.3	110.2	1,116.2	11,477.2	ca	1.3	12.3	110.2	1,120.9	11,564.0
po	1.1	11.6	113.7	1,120.6	11,295.4	po	1.3	11.6	113.7	1,121.8	11,388.3
it	1.3	10.8	114.2	1,149.6	11,651.4	it	1.6	10.8	114.2	1,149.2	11,697.5
gb	1.0	11.1	116.0	1,150.1	11,726.8	gb	1.1	11.1	116.0	1,150.2	11,792.9
fr	0.8	11.0	116.8	1,156.5	11,539.0	fr	0.9	11.0	116.9	1,158.3	11,592.1
ru	1.2	11.3	112.0	1,096.9	11,117.6	ru	1.4	11.3	111.9	1,097.1	11,153.0
ge	1.1	11.0	113.1	1,129.5	11,356.4	ge	1.1	11.0	113.1	1,129.5	11,383.1
da	1.1	12.3	114.4	1,132.9	11,421.6	da	1.3	12.3	114.4	1,132.9	11,439.9
af	0.8	10.7	111.3	1,088.6	10,981.8	af	0.9	10.7	111.3	1,088.4	10,998.5
us	0.9	10.6	109.1	1,103.7	11,072.7	us	1.0	10.6	109.1	1,103.2	11,081.2
as	0.9	10.9	111.4	1,096.9	11,039.0	as	1.1	10.9	111.4	1,096.9	11,043.5

Table H.2: Average number of affected vertices in the R-MAT networks of Table 9.2.

(a) Edge weights generated by a normal distribution

Graph	Edge insertions: Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$	Graph	Edge removals: Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.03	0.32	3.16	31.60	317.28	rmat-22	0.04	0.34	3.04	31.23	316.55
rmat-23	0.03	0.31	2.93	28.64	291.35	rmat-23	0.01	0.25	2.84	29.13	289.61
rmat-24	0.02	0.29	2.71	27.60	272.77	rmat-24	0.02	0.27	2.69	27.27	273.35

(b) Edge weights generated by an exponential distribution

Graph	Edge insertions: Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$	Graph	Edge removals: Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.03	0.31	3.32	31.71	318.74	rmat-22	0.03	0.31	3.23	31.63	318.52
rmat-23	0.03	0.26	3.02	28.79	292.93	rmat-23	0.04	0.26	2.87	28.98	292.65
rmat-24	0.02	0.29	2.69	27.20	273.99	rmat-24	0.04	0.24	2.77	27.43	274.55

Table H.3: Average number of affected vertices in the complex networks of Table 9.1.

(a) Edge weights generated by a normal distribution

Edge insertions

Graph	Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.09	1.25	11.60	121.27	1,458.87
cy	0.24	2.57	24.97	252.53	2,745.08
fx	0.03	0.37	3.71	38.66	398.10
yg	0.29	2.42	23.59	238.19	2,437.38
fg	0.05	0.85	8.57	82.59	826.29
ll	0.14	1.65	16.12	157.23	1,587.56
lj	0.10	1.85	16.73	162.72	1,625.15
ol	0.11	0.82	8.17	75.39	761.71
di	0.09	0.74	7.36	75.44	745.72
we	0.01	0.53	4.80	48.77	483.54
tw	0.03	0.25	3.26	32.70	325.47
tm	0.03	0.26	3.13	29.73	297.10
fs	0.07	0.54	5.61	58.07	580.53

Edge removals

Graph	Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.15	1.24	11.96	119.92	1,461.82
cy	0.31	2.70	24.65	249.34	2,734.40
fx	0.04	0.49	3.70	37.78	398.93
yg	0.19	2.52	24.55	238.09	2,431.13
fg	0.06	0.88	7.86	80.81	819.98
ll	0.15	1.43	16.23	156.64	1,586.20
lj	0.10	1.54	16.31	163.42	1,627.85
ol	0.04	0.77	7.42	77.20	758.17
di	0.06	0.71	7.60	76.17	745.27
we	0.00	0.44	5.05	48.32	484.18
tw	0.03	0.38	3.32	32.46	326.62
tm	0.01	0.35	2.82	29.07	301.16
fs	0.01	0.81	6.04	57.13	576.43

(b) Edge weights generated by an exponential distribution

Edge insertions

Graph	Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.08	1.08	12.28	121.90	1,462.90
cy	0.20	2.29	24.98	253.57	2,758.08
fx	0.05	0.41	3.96	38.35	400.75
yg	0.21	2.52	23.97	239.36	2,447.33
fg	0.05	0.78	8.09	81.13	827.56
ll	0.12	1.57	16.35	158.27	1,597.29
lj	0.16	1.99	16.77	162.35	1,637.20
ol	0.06	0.82	8.24	74.27	767.03
di	0.07	0.66	7.43	74.23	754.21
we	0.04	0.41	5.12	48.18	486.44
tw	0.02	0.25	3.34	32.67	329.47
tm	0.04	0.41	3.36	30.91	299.93
fs	0.08	0.74	5.74	58.48	585.62

Edge removals

Graph	Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.10	1.16	11.88	121.57	1,465.79
cy	0.19	2.38	24.98	253.28	2,766.21
fx	0.05	0.34	3.74	38.19	398.68
yg	0.24	2.46	23.80	240.16	2,453.34
fg	0.04	0.85	7.95	81.96	830.43
ll	0.19	1.60	15.72	160.37	1,592.24
lj	0.17	1.81	16.88	162.95	1,636.81
ol	0.05	0.86	7.81	75.83	762.64
di	0.07	0.68	8.00	73.94	751.44
we	0.04	0.34	4.85	48.05	487.52
tw	0.05	0.35	3.19	32.34	327.00
tm	0.08	0.33	3.33	30.86	303.08
fs	0.09	0.69	5.90	59.63	585.80

Table H.4: Average number of affected vertices in the random hyperbolic networks of Table 9.2.

(a) Edge weights generated by a normal distribution

Edge insertions

Graph	Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.22	2.38	24.08	236.89	2,381.54
hyp-23	0.20	2.46	23.43	237.93	2,376.37
hyp-24	0.24	2.24	23.41	236.56	2,369.67

Edge removals

Graph	Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.24	2.28	23.96	235.66	2,383.13
hyp-23	0.25	2.53	23.42	236.70	2,376.14
hyp-24	0.26	2.22	24.05	237.51	2,372.14

(b) Edge weights generated by an exponential distribution

Edge insertions

Graph	Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.21	2.49	24.15	238.28	2,396.69
hyp-23	0.20	2.27	23.98	238.23	2,388.64
hyp-24	0.19	2.38	23.53	238.20	2,387.47

Edge removals

Graph	Average #of affected vertices $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.26	2.52	23.95	237.98	2,400.01
hyp-23	0.22	2.25	24.16	238.31	2,392.24
hyp-24	0.22	2.35	23.50	238.74	2,388.09

H.2 Traversed Edges

Table H.5: Average number of edges traversed to handle a single edge update. Results are averaged over the road networks of Table 9.1 and over 100 edge updates.

(a) Road networks

Edge insertions

Graph	Suitor	DynMWMRandom $\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$
be	0.7	22.6	11.0	8.3
cz	1.1	20.6	11.1	8.1
fi	1.0	17.8	11.2	8.5
au	0.9	16.7	10.1	8.2
ca	0.9	21.2	11.1	8.4
po	0.9	19.2	10.3	8.5
it	0.8	18.7	11.3	8.6
gb	1.1	19.4	11.1	8.9
fr	0.9	18.6	9.6	8.0
ru	1.1	19.1	11.2	8.9
ge	1.2	19.8	11.4	8.0
da	0.9	20.1	10.4	8.5
af	1.0	21.9	11.5	8.2
us	0.8	17.4	10.8	8.8
as	1.1	18.0	11.7	9.2

Edge removals

Graph	Suitor	DynMWMRandom $\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$
be	1.0	40.3	22.5	18.2
cz	1.2	42.2	22.6	18.8
fi	0.9	32.6	22.2	17.6
au	0.8	33.7	23.5	19.2
ca	0.8	37.5	23.1	18.3
po	0.7	39.4	22.4	18.1
it	1.0	41.0	24.4	19.2
gb	1.0	32.2	21.2	17.4
fr	0.8	36.1	23.4	19.6
ru	1.0	40.9	22.9	18.6
ge	0.9	40.5	23.7	18.0
da	0.9	39.1	21.3	16.8
af	0.9	37.8	22.0	17.4
us	0.9	35.5	22.0	18.2
as	0.9	39.4	23.4	18.4

(b) Complex networks

Edge insertions

Graph	Suitor	DynMWMRandom $\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$
hy	0.1	17.8	11.7	10.2
cy	0.3	15.7	12.7	9.7
fx	0.6	18.8	10.8	9.3
yg	0.3	16.5	11.5	9.4
fg	0.1	20.8	10.8	7.4
ll	0.2	21.9	9.1	6.3
lj	0.2	22.7	9.8	7.2
ol	0.0	23.7	7.2	5.3
di	0.8	26.6	7.9	5.7
we	0.5	11.1	4.9	3.5

Edge removals

Graph	Suitor	DynMWMRandom $\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$
hy	0.1	25.2	19.9	15.5
cy	0.7	22.7	18.6	16.1
fx	0.0	27.2	17.2	14.1
yg	0.3	24.1	19.0	15.9
fg	0.6	25.4	17.7	14.8
ll	0.9	39.3	19.1	15.1
lj	0.5	44.8	21.5	16.1
ol	0.6	48.8	16.8	12.3
di	0.7	51.8	19.9	14.7
we	0.3	44.9	20.5	15.1

H.3 Speedups on DynMWMRandom

Table H.6: Geometric mean of the speedups of dynamic Suitor over DynMWMRandom on real-world networks of Table 9.1. Results are averaged over 100 edge updates.

(a) Road networks

Edge insertions

Graph	DynMWMRandom $\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$
be	4.41	2.80	2.30
cz	3.90	2.54	2.32
fi	3.05	2.25	1.89
au	2.96	2.44	2.09
ca	4.42	2.99	2.50
po	2.78	1.90	1.68
it	2.62	1.89	1.68
gb	2.71	1.98	1.69
fr	2.89	2.09	1.74
ru	3.23	2.37	2.07
ge	2.93	2.37	1.97
da	3.29	2.22	1.98
af	3.26	1.75	1.82
us	2.80	2.25	1.91
as	2.43	1.91	1.68
geom. mean	3.13	2.22	1.94

Edge removals

Graph	DynMWMRandom $\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$
be	6.93	4.44	3.75
cz	5.73	3.76	3.26
fi	4.16	3.16	2.79
au	3.74	3.12	2.74
ca	4.64	3.51	2.91
po	4.23	3.06	2.50
it	4.17	3.01	2.58
gb	3.69	2.78	2.43
fr	4.34	3.45	2.98
ru	4.56	3.05	2.71
ge	3.58	2.56	2.16
da	4.62	3.00	2.67
af	4.08	3.62	2.41
us	5.21	2.85	3.36
as	4.60	3.28	2.80
geom. mean	4.48	3.22	2.78

(b) Complex networks

Edge insertions

Graph	DynMWMRandom $\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$
hy	1.08	0.94	0.92
cy	1.74	1.63	1.46
fx	1.89	1.48	1.32
yg	0.40	0.35	0.32
fg	1.40	0.86	0.75
ll	4.23	2.61	2.08
lj	3.65	2.18	1.83
ol	4.40	2.10	1.67
di	4.51	2.13	1.62
we	0.40	0.22	0.17
geom. mean	1.73	1.14	0.97

Edge removals

Graph	DynMWMRandom $\varepsilon = 0.1$	$\varepsilon = 0.5$	$\varepsilon = 1$
hy	2.78	2.55	2.41
cy	1.65	1.50	1.45
fx	3.26	2.61	2.19
yg	0.67	0.63	0.59
fg	2.06	1.61	1.44
ll	5.50	3.61	2.92
lj	5.40	3.33	2.76
ol	7.97	3.73	2.94
di	6.11	4.38	2.75
we	1.62	1.01	0.86
geom. mean	2.94	2.14	1.80

H.4 Running Times Compared to DynMWMRandom

Table H.7: Average running time in seconds to handle a single edge update. Results are averaged over the networks of Table 9.1 and over 100 edge updates. Contrarily to the tables in Appendix H.6, here we also take into account the time spent to update the graph data structures.

(a) Road networks

Edge insertions

Graph	Suitor	DynMWMRandom $\varepsilon = 0.1$	DynMWMRandom $\varepsilon = 0.5$	DynMWMRandom $\varepsilon = 1$
be	$1.0 \cdot 10^{-6}$	$4.5 \cdot 10^{-6}$	$2.9 \cdot 10^{-6}$	$2.4 \cdot 10^{-6}$
cz	$1.1 \cdot 10^{-6}$	$4.1 \cdot 10^{-6}$	$2.7 \cdot 10^{-6}$	$2.5 \cdot 10^{-6}$
fi	$1.2 \cdot 10^{-6}$	$3.6 \cdot 10^{-6}$	$2.7 \cdot 10^{-6}$	$2.2 \cdot 10^{-6}$
au	$1.1 \cdot 10^{-6}$	$3.3 \cdot 10^{-6}$	$2.7 \cdot 10^{-6}$	$2.3 \cdot 10^{-6}$
ca	$9.9 \cdot 10^{-7}$	$4.4 \cdot 10^{-6}$	$3.0 \cdot 10^{-6}$	$2.5 \cdot 10^{-6}$
po	$1.5 \cdot 10^{-6}$	$4.1 \cdot 10^{-6}$	$2.8 \cdot 10^{-6}$	$2.5 \cdot 10^{-6}$
it	$1.5 \cdot 10^{-6}$	$3.9 \cdot 10^{-6}$	$2.8 \cdot 10^{-6}$	$2.5 \cdot 10^{-6}$
gb	$1.5 \cdot 10^{-6}$	$4.1 \cdot 10^{-6}$	$3.0 \cdot 10^{-6}$	$2.6 \cdot 10^{-6}$
fr	$1.4 \cdot 10^{-6}$	$4.2 \cdot 10^{-6}$	$3.0 \cdot 10^{-6}$	$2.5 \cdot 10^{-6}$
ru	$1.3 \cdot 10^{-6}$	$4.1 \cdot 10^{-6}$	$3.0 \cdot 10^{-6}$	$2.7 \cdot 10^{-6}$
ge	$1.5 \cdot 10^{-6}$	$4.3 \cdot 10^{-6}$	$3.5 \cdot 10^{-6}$	$2.9 \cdot 10^{-6}$
da	$1.4 \cdot 10^{-6}$	$4.6 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$	$2.7 \cdot 10^{-6}$
af	$1.8 \cdot 10^{-6}$	$5.8 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$	$3.2 \cdot 10^{-6}$
us	$1.4 \cdot 10^{-6}$	$3.9 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$	$2.6 \cdot 10^{-6}$
as	$1.6 \cdot 10^{-6}$	$4.0 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$	$2.7 \cdot 10^{-6}$

Edge removals

Graph	Suitor	DynMWMRandom $\varepsilon = 0.1$	DynMWMRandom $\varepsilon = 0.5$	DynMWMRandom $\varepsilon = 1$
be	$1.0 \cdot 10^{-6}$	$7.2 \cdot 10^{-6}$	$4.6 \cdot 10^{-6}$	$3.9 \cdot 10^{-6}$
cz	$1.3 \cdot 10^{-6}$	$7.5 \cdot 10^{-6}$	$4.9 \cdot 10^{-6}$	$4.3 \cdot 10^{-6}$
fi	$1.3 \cdot 10^{-6}$	$5.4 \cdot 10^{-6}$	$4.1 \cdot 10^{-6}$	$3.6 \cdot 10^{-6}$
au	$1.5 \cdot 10^{-6}$	$5.6 \cdot 10^{-6}$	$4.7 \cdot 10^{-6}$	$4.1 \cdot 10^{-6}$
ca	$1.5 \cdot 10^{-6}$	$6.7 \cdot 10^{-6}$	$5.1 \cdot 10^{-6}$	$4.2 \cdot 10^{-6}$
po	$1.7 \cdot 10^{-6}$	$7.3 \cdot 10^{-6}$	$5.3 \cdot 10^{-6}$	$4.3 \cdot 10^{-6}$
it	$1.8 \cdot 10^{-6}$	$7.3 \cdot 10^{-6}$	$5.3 \cdot 10^{-6}$	$4.5 \cdot 10^{-6}$
gb	$1.6 \cdot 10^{-6}$	$5.8 \cdot 10^{-6}$	$4.4 \cdot 10^{-6}$	$3.8 \cdot 10^{-6}$
fr	$1.5 \cdot 10^{-6}$	$6.5 \cdot 10^{-6}$	$5.1 \cdot 10^{-6}$	$4.4 \cdot 10^{-6}$
ru	$1.7 \cdot 10^{-6}$	$7.7 \cdot 10^{-6}$	$5.2 \cdot 10^{-6}$	$4.6 \cdot 10^{-6}$
ge	$2.2 \cdot 10^{-6}$	$7.9 \cdot 10^{-6}$	$5.7 \cdot 10^{-6}$	$4.8 \cdot 10^{-6}$
da	$1.7 \cdot 10^{-6}$	$8.0 \cdot 10^{-6}$	$5.2 \cdot 10^{-6}$	$4.7 \cdot 10^{-6}$
af	$1.8 \cdot 10^{-6}$	$7.2 \cdot 10^{-6}$	$6.4 \cdot 10^{-6}$	$4.3 \cdot 10^{-6}$
us	$1.8 \cdot 10^{-6}$	$9.3 \cdot 10^{-6}$	$5.1 \cdot 10^{-6}$	$6.0 \cdot 10^{-6}$
as	$1.6 \cdot 10^{-6}$	$7.5 \cdot 10^{-6}$	$5.3 \cdot 10^{-6}$	$4.5 \cdot 10^{-6}$

(b) Complex networks

Edge insertions

Graph	Suitor	DynMWMRandom $\varepsilon = 0.1$	DynMWMRandom $\varepsilon = 0.5$	DynMWMRandom $\varepsilon = 1$
hy	$2.8 \cdot 10^{-6}$	$3.0 \cdot 10^{-6}$	$2.6 \cdot 10^{-6}$	$2.6 \cdot 10^{-6}$
cy	$1.5 \cdot 10^{-6}$	$2.6 \cdot 10^{-6}$	$2.4 \cdot 10^{-6}$	$2.2 \cdot 10^{-6}$
fx	$1.9 \cdot 10^{-6}$	$3.7 \cdot 10^{-6}$	$2.9 \cdot 10^{-6}$	$2.6 \cdot 10^{-6}$
yg	$6.6 \cdot 10^{-6}$	$2.6 \cdot 10^{-6}$	$2.3 \cdot 10^{-6}$	$2.1 \cdot 10^{-6}$
fg	$3.2 \cdot 10^{-6}$	$4.5 \cdot 10^{-6}$	$2.8 \cdot 10^{-6}$	$2.4 \cdot 10^{-6}$
ll	$1.5 \cdot 10^{-6}$	$6.2 \cdot 10^{-6}$	$3.8 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$
lj	$1.7 \cdot 10^{-6}$	$6.3 \cdot 10^{-6}$	$3.8 \cdot 10^{-6}$	$3.2 \cdot 10^{-6}$
ol	$2.1 \cdot 10^{-6}$	$9.4 \cdot 10^{-6}$	$4.5 \cdot 10^{-6}$	$3.6 \cdot 10^{-6}$
di	$2.3 \cdot 10^{-6}$	$1.0 \cdot 10^{-5}$	$4.8 \cdot 10^{-6}$	$3.7 \cdot 10^{-6}$
we	$1.3 \cdot 10^{-5}$	$5.1 \cdot 10^{-6}$	$2.8 \cdot 10^{-6}$	$2.2 \cdot 10^{-6}$

Edge removals

Graph	Suitor	DynMWMRandom $\varepsilon = 0.1$	DynMWMRandom $\varepsilon = 0.5$	DynMWMRandom $\varepsilon = 1$
hy	$1.4 \cdot 10^{-6}$	$4.0 \cdot 10^{-6}$	$3.6 \cdot 10^{-6}$	$3.4 \cdot 10^{-6}$
cy	$2.1 \cdot 10^{-6}$	$3.4 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$	$3.0 \cdot 10^{-6}$
fx	$1.6 \cdot 10^{-6}$	$5.1 \cdot 10^{-6}$	$4.1 \cdot 10^{-6}$	$3.4 \cdot 10^{-6}$
yg	$5.7 \cdot 10^{-6}$	$3.8 \cdot 10^{-6}$	$3.6 \cdot 10^{-6}$	$3.4 \cdot 10^{-6}$
fg	$2.4 \cdot 10^{-6}$	$5.0 \cdot 10^{-6}$	$3.9 \cdot 10^{-6}$	$3.5 \cdot 10^{-6}$
ll	$1.9 \cdot 10^{-6}$	$1.0 \cdot 10^{-5}$	$6.8 \cdot 10^{-6}$	$5.5 \cdot 10^{-6}$
lj	$2.2 \cdot 10^{-6}$	$1.2 \cdot 10^{-5}$	$7.4 \cdot 10^{-6}$	$6.1 \cdot 10^{-6}$
ol	$2.3 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$8.6 \cdot 10^{-6}$	$6.8 \cdot 10^{-6}$
di	$2.4 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$1.0 \cdot 10^{-5}$	$6.5 \cdot 10^{-6}$
we	$8.9 \cdot 10^{-6}$	$1.4 \cdot 10^{-5}$	$9.0 \cdot 10^{-6}$	$7.6 \cdot 10^{-6}$

H.5 Speedups on the Static Algorithm

Table H.8: Geometric mean of the speedups of the dynamic algorithm over a static recomputation on the road networks of Table 9.1. Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ edge updates.

Edge insertions					
Graph	Speedup				
	$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
be	$1.3 \cdot 10^5$	$2.4 \cdot 10^4$	$3.6 \cdot 10^3$	$4.0 \cdot 10^2$	$3.1 \cdot 10^1$
cz	$1.4 \cdot 10^5$	$3.4 \cdot 10^4$	$5.0 \cdot 10^3$	$5.6 \cdot 10^2$	$4.5 \cdot 10^1$
fi	$1.8 \cdot 10^5$	$4.2 \cdot 10^4$	$5.9 \cdot 10^3$	$6.5 \cdot 10^2$	$5.3 \cdot 10^1$
au	$2.3 \cdot 10^5$	$6.0 \cdot 10^4$	$8.0 \cdot 10^3$	$8.7 \cdot 10^2$	$7.1 \cdot 10^1$
ca	$3.1 \cdot 10^5$	$6.1 \cdot 10^4$	$1.0 \cdot 10^4$	$9.9 \cdot 10^2$	$8.4 \cdot 10^1$
po	$3.8 \cdot 10^5$	$1.1 \cdot 10^5$	$1.7 \cdot 10^4$	$1.8 \cdot 10^3$	$1.5 \cdot 10^2$
it	$4.8 \cdot 10^5$	$1.2 \cdot 10^5$	$1.8 \cdot 10^4$	$1.9 \cdot 10^3$	$1.7 \cdot 10^2$
gb	$6.3 \cdot 10^5$	$1.1 \cdot 10^5$	$2.0 \cdot 10^4$	$2.0 \cdot 10^3$	$1.8 \cdot 10^2$
fr	$9.8 \cdot 10^5$	$1.9 \cdot 10^5$	$2.9 \cdot 10^4$	$3.4 \cdot 10^3$	$3.1 \cdot 10^2$
ru	$6.6 \cdot 10^5$	$1.6 \cdot 10^5$	$2.7 \cdot 10^4$	$3.1 \cdot 10^3$	$2.8 \cdot 10^2$
ge	$1.4 \cdot 10^6$	$2.7 \cdot 10^5$	$4.4 \cdot 10^4$	$5.0 \cdot 10^3$	$4.5 \cdot 10^2$
da	$1.4 \cdot 10^6$	$3.0 \cdot 10^5$	$5.1 \cdot 10^4$	$6.4 \cdot 10^3$	$5.7 \cdot 10^2$
af	$1.6 \cdot 10^6$	$2.8 \cdot 10^5$	$4.7 \cdot 10^4$	$5.5 \cdot 10^3$	$5.1 \cdot 10^2$
us	$2.4 \cdot 10^6$	$4.7 \cdot 10^5$	$7.1 \cdot 10^4$	$9.0 \cdot 10^3$	$8.7 \cdot 10^2$
as	$3.8 \cdot 10^6$	$6.0 \cdot 10^5$	$9.4 \cdot 10^4$	$1.3 \cdot 10^4$	$1.2 \cdot 10^3$
geom. mean	$6.0 \cdot 10^5$	$1.3 \cdot 10^5$	$2.0 \cdot 10^4$	$2.2 \cdot 10^3$	$2.0 \cdot 10^2$

Edge removals					
Graph	Speedup				
	$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
be	$1.5 \cdot 10^5$	$3.0 \cdot 10^4$	$4.2 \cdot 10^3$	$4.9 \cdot 10^2$	$4.3 \cdot 10^1$
cz	$2.1 \cdot 10^5$	$4.0 \cdot 10^4$	$6.1 \cdot 10^3$	$6.9 \cdot 10^2$	$6.5 \cdot 10^1$
fi	$2.4 \cdot 10^5$	$4.9 \cdot 10^4$	$7.6 \cdot 10^3$	$8.4 \cdot 10^2$	$7.7 \cdot 10^1$
au	$3.6 \cdot 10^5$	$7.1 \cdot 10^4$	$9.8 \cdot 10^3$	$1.1 \cdot 10^3$	$1.0 \cdot 10^2$
ca	$3.4 \cdot 10^5$	$7.3 \cdot 10^4$	$1.2 \cdot 10^4$	$1.4 \cdot 10^3$	$1.4 \cdot 10^2$
po	$4.6 \cdot 10^5$	$1.3 \cdot 10^5$	$2.1 \cdot 10^4$	$2.4 \cdot 10^3$	$2.4 \cdot 10^2$
it	$4.3 \cdot 10^5$	$1.5 \cdot 10^5$	$2.3 \cdot 10^4$	$2.7 \cdot 10^3$	$2.6 \cdot 10^2$
gb	$6.6 \cdot 10^5$	$1.7 \cdot 10^5$	$2.6 \cdot 10^4$	$3.0 \cdot 10^3$	$3.0 \cdot 10^2$
fr	$1.5 \cdot 10^6$	$2.6 \cdot 10^5$	$4.2 \cdot 10^4$	$4.7 \cdot 10^3$	$4.8 \cdot 10^2$
ru	$7.1 \cdot 10^5$	$2.4 \cdot 10^5$	$3.7 \cdot 10^4$	$4.6 \cdot 10^3$	$4.6 \cdot 10^2$
ge	$1.4 \cdot 10^6$	$3.6 \cdot 10^5$	$6.5 \cdot 10^4$	$7.4 \cdot 10^3$	$7.4 \cdot 10^2$
da	$1.8 \cdot 10^6$	$5.2 \cdot 10^5$	$7.8 \cdot 10^4$	$9.2 \cdot 10^3$	$9.5 \cdot 10^2$
af	$1.7 \cdot 10^6$	$4.5 \cdot 10^5$	$7.4 \cdot 10^4$	$8.7 \cdot 10^3$	$8.8 \cdot 10^2$
us	$2.9 \cdot 10^6$	$7.7 \cdot 10^5$	$1.3 \cdot 10^5$	$1.5 \cdot 10^4$	$1.5 \cdot 10^3$
as	$3.9 \cdot 10^6$	$10.0 \cdot 10^5$	$1.8 \cdot 10^5$	$2.1 \cdot 10^4$	$2.1 \cdot 10^3$
geom. mean	$7.1 \cdot 10^5$	$1.7 \cdot 10^5$	$2.8 \cdot 10^4$	$3.2 \cdot 10^3$	$3.1 \cdot 10^2$

Table H.9: Geometric mean of the speedups of the dynamic algorithm over a static recomputation on the complex networks of Table 9.1. Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ edge updates.

(a) Edge weights generated by a normal distribution

Graph	Edge insertions: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$	Graph	Edge removals: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	$1.3 \cdot 10^5$	$5.0 \cdot 10^4$	$1.0 \cdot 10^4$	$1.2 \cdot 10^3$	$1.3 \cdot 10^2$	hy	$1.6 \cdot 10^5$	$7.8 \cdot 10^4$	$1.8 \cdot 10^4$	$2.2 \cdot 10^3$	$2.3 \cdot 10^2$
cy	$1.4 \cdot 10^5$	$4.2 \cdot 10^4$	$6.4 \cdot 10^3$	$7.6 \cdot 10^2$	$7.9 \cdot 10^1$	cy	$1.6 \cdot 10^5$	$6.2 \cdot 10^4$	$1.1 \cdot 10^4$	$1.2 \cdot 10^3$	$1.2 \cdot 10^2$
fx	$1.5 \cdot 10^5$	$7.2 \cdot 10^4$	$1.7 \cdot 10^4$	$2.1 \cdot 10^3$	$2.5 \cdot 10^2$	fx	$1.9 \cdot 10^5$	$1.0 \cdot 10^5$	$3.5 \cdot 10^4$	$4.8 \cdot 10^3$	$5.7 \cdot 10^2$
yg	$3.1 \cdot 10^5$	$1.0 \cdot 10^5$	$2.0 \cdot 10^4$	$2.4 \cdot 10^3$	$2.6 \cdot 10^2$	yg	$4.3 \cdot 10^5$	$1.4 \cdot 10^5$	$2.9 \cdot 10^4$	$3.7 \cdot 10^3$	$4.0 \cdot 10^2$
fg	$3.0 \cdot 10^5$	$1.1 \cdot 10^5$	$2.0 \cdot 10^4$	$3.5 \cdot 10^3$	$4.2 \cdot 10^2$	fg	$3.2 \cdot 10^5$	$1.5 \cdot 10^5$	$3.2 \cdot 10^4$	$5.7 \cdot 10^3$	$6.4 \cdot 10^2$
ll	$9.2 \cdot 10^5$	$3.1 \cdot 10^5$	$7.5 \cdot 10^4$	$8.7 \cdot 10^3$	$9.6 \cdot 10^2$	ll	$1.0 \cdot 10^6$	$5.0 \cdot 10^5$	$9.2 \cdot 10^4$	$1.3 \cdot 10^4$	$1.3 \cdot 10^3$
lj	$1.1 \cdot 10^6$	$2.5 \cdot 10^5$	$4.7 \cdot 10^4$	$7.2 \cdot 10^3$	$8.5 \cdot 10^2$	lj	$1.4 \cdot 10^6$	$5.1 \cdot 10^5$	$7.4 \cdot 10^4$	$1.0 \cdot 10^4$	$1.2 \cdot 10^3$
ol	$1.2 \cdot 10^6$	$5.0 \cdot 10^5$	$7.3 \cdot 10^4$	$1.1 \cdot 10^4$	$1.4 \cdot 10^3$	ol	$1.6 \cdot 10^6$	$7.5 \cdot 10^5$	$1.4 \cdot 10^5$	$1.6 \cdot 10^4$	$2.0 \cdot 10^3$
di	$1.5 \cdot 10^6$	$7.3 \cdot 10^5$	$1.7 \cdot 10^5$	$2.1 \cdot 10^4$	$2.2 \cdot 10^3$	di	$2.2 \cdot 10^6$	$1.1 \cdot 10^6$	$2.8 \cdot 10^5$	$3.6 \cdot 10^4$	$3.8 \cdot 10^3$
we	$3.8 \cdot 10^6$	$1.1 \cdot 10^6$	$1.8 \cdot 10^5$	$2.5 \cdot 10^4$	$4.2 \cdot 10^3$	we	$4.6 \cdot 10^6$	$1.7 \cdot 10^6$	$3.1 \cdot 10^5$	$4.1 \cdot 10^4$	$6.4 \cdot 10^3$
tw	$1.4 \cdot 10^7$	$4.5 \cdot 10^6$	$8.2 \cdot 10^5$	$1.5 \cdot 10^5$	$2.1 \cdot 10^4$	tw	$1.8 \cdot 10^7$	$6.1 \cdot 10^6$	$1.4 \cdot 10^6$	$2.2 \cdot 10^5$	$3.5 \cdot 10^4$
tm	$1.7 \cdot 10^7$	$6.0 \cdot 10^6$	$1.1 \cdot 10^6$	$1.5 \cdot 10^5$	$2.6 \cdot 10^4$	tm	$2.3 \cdot 10^7$	$8.2 \cdot 10^6$	$2.0 \cdot 10^6$	$2.6 \cdot 10^5$	$4.0 \cdot 10^4$
fs	$2.3 \cdot 10^7$	$9.4 \cdot 10^6$	$1.5 \cdot 10^6$	$2.7 \cdot 10^5$	$3.1 \cdot 10^4$	fs	$3.0 \cdot 10^7$	$1.3 \cdot 10^7$	$2.0 \cdot 10^6$	$3.5 \cdot 10^5$	$4.5 \cdot 10^4$
geom. mean	$1.2 \cdot 10^6$	$4.2 \cdot 10^5$	$7.9 \cdot 10^4$	$1.1 \cdot 10^4$	$1.4 \cdot 10^3$	geom. mean	$1.5 \cdot 10^6$	$6.3 \cdot 10^5$	$1.3 \cdot 10^5$	$1.8 \cdot 10^4$	$2.1 \cdot 10^3$

(b) Edge weights generated by an exponential distribution

Graph	Edge insertions: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$	Graph	Edge removals: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	$1.2 \cdot 10^5$	$5.1 \cdot 10^4$	$9.5 \cdot 10^3$	$1.2 \cdot 10^3$	$1.3 \cdot 10^2$	hy	$1.7 \cdot 10^5$	$8.1 \cdot 10^4$	$1.7 \cdot 10^4$	$2.2 \cdot 10^3$	$2.3 \cdot 10^2$
cy	$1.4 \cdot 10^5$	$4.1 \cdot 10^4$	$6.6 \cdot 10^3$	$7.7 \cdot 10^2$	$7.9 \cdot 10^1$	cy	$1.8 \cdot 10^5$	$6.6 \cdot 10^4$	$1.0 \cdot 10^4$	$1.2 \cdot 10^3$	$1.2 \cdot 10^2$
fx	$1.5 \cdot 10^5$	$7.0 \cdot 10^4$	$1.7 \cdot 10^4$	$2.1 \cdot 10^3$	$2.5 \cdot 10^2$	fx	$1.9 \cdot 10^5$	$1.2 \cdot 10^5$	$3.5 \cdot 10^4$	$4.6 \cdot 10^3$	$5.7 \cdot 10^2$
yg	$3.2 \cdot 10^5$	$1.0 \cdot 10^5$	$2.0 \cdot 10^4$	$2.4 \cdot 10^3$	$2.6 \cdot 10^2$	yg	$3.9 \cdot 10^5$	$1.4 \cdot 10^5$	$2.9 \cdot 10^4$	$3.7 \cdot 10^3$	$4.0 \cdot 10^2$
fg	$2.7 \cdot 10^5$	$1.1 \cdot 10^5$	$2.0 \cdot 10^4$	$3.4 \cdot 10^3$	$4.2 \cdot 10^2$	fg	$2.9 \cdot 10^5$	$1.5 \cdot 10^5$	$3.1 \cdot 10^4$	$5.4 \cdot 10^3$	$6.3 \cdot 10^2$
ll	$1.0 \cdot 10^6$	$3.1 \cdot 10^5$	$6.6 \cdot 10^4$	$8.9 \cdot 10^3$	$9.6 \cdot 10^2$	ll	$1.1 \cdot 10^6$	$4.8 \cdot 10^5$	$8.8 \cdot 10^4$	$1.2 \cdot 10^4$	$1.3 \cdot 10^3$
lj	$1.0 \cdot 10^6$	$2.4 \cdot 10^5$	$4.7 \cdot 10^4$	$8.4 \cdot 10^3$	$8.5 \cdot 10^2$	lj	$1.3 \cdot 10^6$	$4.9 \cdot 10^5$	$7.3 \cdot 10^4$	$1.0 \cdot 10^4$	$1.2 \cdot 10^3$
ol	$1.3 \cdot 10^6$	$4.8 \cdot 10^5$	$6.8 \cdot 10^4$	$1.1 \cdot 10^4$	$1.4 \cdot 10^3$	ol	$1.5 \cdot 10^6$	$6.9 \cdot 10^5$	$1.4 \cdot 10^5$	$1.8 \cdot 10^4$	$2.0 \cdot 10^3$
di	$1.5 \cdot 10^6$	$7.1 \cdot 10^5$	$1.7 \cdot 10^5$	$2.1 \cdot 10^4$	$2.2 \cdot 10^3$	di	$2.0 \cdot 10^6$	$1.1 \cdot 10^6$	$2.7 \cdot 10^5$	$3.7 \cdot 10^4$	$3.7 \cdot 10^3$
we	$3.6 \cdot 10^6$	$1.1 \cdot 10^6$	$1.9 \cdot 10^5$	$2.5 \cdot 10^4$	$4.2 \cdot 10^3$	we	$4.1 \cdot 10^6$	$1.8 \cdot 10^6$	$3.1 \cdot 10^5$	$4.0 \cdot 10^4$	$6.5 \cdot 10^3$
tw	$1.6 \cdot 10^7$	$4.5 \cdot 10^6$	$8.3 \cdot 10^5$	$1.3 \cdot 10^5$	$2.1 \cdot 10^4$	tw	$1.8 \cdot 10^7$	$6.6 \cdot 10^6$	$1.4 \cdot 10^6$	$2.5 \cdot 10^5$	$3.5 \cdot 10^4$
tm	$1.5 \cdot 10^7$	$6.0 \cdot 10^6$	$1.1 \cdot 10^6$	$1.5 \cdot 10^5$	$2.7 \cdot 10^4$	tm	$2.1 \cdot 10^7$	$8.2 \cdot 10^6$	$1.9 \cdot 10^6$	$2.5 \cdot 10^5$	$4.4 \cdot 10^4$
fs	$2.3 \cdot 10^7$	$8.7 \cdot 10^6$	$1.4 \cdot 10^6$	$2.3 \cdot 10^5$	$3.0 \cdot 10^4$	fs	$2.6 \cdot 10^7$	$1.2 \cdot 10^7$	$2.2 \cdot 10^6$	$4.3 \cdot 10^5$	$3.6 \cdot 10^4$
geom. mean	$1.2 \cdot 10^6$	$4.2 \cdot 10^5$	$7.7 \cdot 10^4$	$1.1 \cdot 10^4$	$1.4 \cdot 10^3$	geom. mean	$1.4 \cdot 10^6$	$6.4 \cdot 10^5$	$1.3 \cdot 10^5$	$1.8 \cdot 10^4$	$2.1 \cdot 10^3$

Table H.10: Geometric mean of the speedups of the dynamic algorithm over a static recomputation on the R-MAT networks of Table 9.2. Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ edge updates.

(a) Edge weights generated by a normal distribution

Graph	Edge insertions: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$	Graph	Edge removals: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	$5.9 \cdot 10^5$	$2.2 \cdot 10^5$	$4.5 \cdot 10^4$	$8.7 \cdot 10^3$	$1.1 \cdot 10^3$	rmat-22	$6.4 \cdot 10^5$	$3.0 \cdot 10^5$	$7.3 \cdot 10^4$	$1.5 \cdot 10^4$	$1.9 \cdot 10^3$
rmat-23	$1.1 \cdot 10^6$	$4.0 \cdot 10^5$	$8.5 \cdot 10^4$	$1.8 \cdot 10^4$	$2.2 \cdot 10^3$	rmat-23	$1.2 \cdot 10^6$	$5.5 \cdot 10^5$	$1.4 \cdot 10^5$	$2.8 \cdot 10^4$	$3.7 \cdot 10^3$
rmat-24	$2.1 \cdot 10^6$	$7.6 \cdot 10^5$	$1.6 \cdot 10^5$	$3.7 \cdot 10^4$	$4.5 \cdot 10^3$	rmat-24	$2.4 \cdot 10^6$	$1.0 \cdot 10^6$	$2.6 \cdot 10^5$	$6.3 \cdot 10^4$	$7.5 \cdot 10^3$
geom. mean	$1.1 \cdot 10^6$	$4.0 \cdot 10^5$	$8.5 \cdot 10^4$	$1.8 \cdot 10^4$	$2.2 \cdot 10^3$	geom. mean	$1.2 \cdot 10^6$	$5.5 \cdot 10^5$	$1.4 \cdot 10^5$	$3.0 \cdot 10^4$	$3.7 \cdot 10^3$

(b) Edge weights generated by an exponential distribution

Graph	Edge insertions: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$	Graph	Edge removals: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	$5.9 \cdot 10^5$	$2.2 \cdot 10^5$	$4.4 \cdot 10^4$	$8.4 \cdot 10^3$	$1.1 \cdot 10^3$	rmat-22	$6.5 \cdot 10^5$	$3.1 \cdot 10^5$	$7.2 \cdot 10^4$	$1.4 \cdot 10^4$	$1.8 \cdot 10^3$
rmat-23	$1.1 \cdot 10^6$	$4.0 \cdot 10^5$	$8.4 \cdot 10^4$	$1.8 \cdot 10^4$	$2.2 \cdot 10^3$	rmat-23	$1.2 \cdot 10^6$	$5.7 \cdot 10^5$	$1.4 \cdot 10^5$	$2.6 \cdot 10^4$	$3.7 \cdot 10^3$
rmat-24	$2.0 \cdot 10^6$	$7.6 \cdot 10^5$	$1.6 \cdot 10^5$	$3.7 \cdot 10^4$	$4.5 \cdot 10^3$	rmat-24	$2.4 \cdot 10^6$	$1.0 \cdot 10^6$	$2.6 \cdot 10^5$	$6.4 \cdot 10^4$	$7.5 \cdot 10^3$
geom. mean	$1.1 \cdot 10^6$	$4.1 \cdot 10^5$	$8.4 \cdot 10^4$	$1.8 \cdot 10^4$	$2.2 \cdot 10^3$	geom. mean	$1.2 \cdot 10^6$	$5.7 \cdot 10^5$	$1.4 \cdot 10^5$	$2.9 \cdot 10^4$	$3.7 \cdot 10^3$

Table H.11: Geometric mean of the speedups of the dynamic algorithm over a static recomputation on the random hyperbolic networks of Table 9.2. Results are averaged over 100 batches with $b \in \{1, \ldots, 10^4\}$ edge updates.

(a) Edge weights generated by a normal distribution

Graph	Edge insertions: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$	Graph	Edge removals: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	$3.4 \cdot 10^5$	$1.2 \cdot 10^5$	$2.5 \cdot 10^4$	$2.8 \cdot 10^3$	$2.8 \cdot 10^2$	hyp-22	$3.8 \cdot 10^5$	$1.9 \cdot 10^5$	$3.8 \cdot 10^4$	$4.4 \cdot 10^3$	$4.2 \cdot 10^2$
hyp-23	$6.3 \cdot 10^5$	$2.0 \cdot 10^5$	$4.6 \cdot 10^4$	$5.2 \cdot 10^3$	$5.5 \cdot 10^2$	hyp-23	$6.5 \cdot 10^5$	$3.3 \cdot 10^5$	$7.2 \cdot 10^4$	$8.6 \cdot 10^3$	$8.2 \cdot 10^2$
hyp-24	$1.3 \cdot 10^6$	$3.9 \cdot 10^5$	$8.9 \cdot 10^4$	$1.0 \cdot 10^4$	$1.1 \cdot 10^3$	hyp-24	$1.3 \cdot 10^6$	$6.7 \cdot 10^5$	$1.4 \cdot 10^5$	$1.7 \cdot 10^4$	$1.6 \cdot 10^3$
geom. mean	$6.6 \cdot 10^5$	$2.1 \cdot 10^5$	$4.7 \cdot 10^4$	$5.3 \cdot 10^3$	$5.5 \cdot 10^2$	geom. mean	$6.9 \cdot 10^5$	$3.5 \cdot 10^5$	$7.3 \cdot 10^4$	$8.5 \cdot 10^3$	$8.1 \cdot 10^2$

(b) Edge weights generated by an exponential distribution

Graph	Edge insertions: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$	Graph	Edge removals: Speedup $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	$3.4 \cdot 10^5$	$1.1 \cdot 10^5$	$2.4 \cdot 10^4$	$2.8 \cdot 10^3$	$2.8 \cdot 10^2$	hyp-22	$3.8 \cdot 10^5$	$1.9 \cdot 10^5$	$3.8 \cdot 10^4$	$4.4 \cdot 10^3$	$4.2 \cdot 10^2$
hyp-23	$6.2 \cdot 10^5$	$2.0 \cdot 10^5$	$4.6 \cdot 10^4$	$5.2 \cdot 10^3$	$5.5 \cdot 10^2$	hyp-23	$6.7 \cdot 10^5$	$3.5 \cdot 10^5$	$7.1 \cdot 10^4$	$8.5 \cdot 10^3$	$8.2 \cdot 10^2$
hyp-24	$1.3 \cdot 10^6$	$3.8 \cdot 10^5$	$9.0 \cdot 10^4$	$1.0 \cdot 10^4$	$1.1 \cdot 10^3$	hyp-24	$1.4 \cdot 10^6$	$6.8 \cdot 10^5$	$1.4 \cdot 10^5$	$1.7 \cdot 10^4$	$1.6 \cdot 10^3$
geom. mean	$6.5 \cdot 10^5$	$2.1 \cdot 10^5$	$4.6 \cdot 10^4$	$5.3 \cdot 10^3$	$5.5 \cdot 10^2$	geom. mean	$7.1 \cdot 10^5$	$3.5 \cdot 10^5$	$7.3 \cdot 10^4$	$8.5 \cdot 10^3$	$8.1 \cdot 10^2$

H.6 Running Times Compared to the Static Algorithm

Table H.12: Average running times in seconds of the static and the dynamic Suitor algorithms for 100 batches of $b \in \{1, \ldots, 10^4\}$ edge updates on the road networks of Table 9.1. The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively.

Edge insertions

Graph	Static	Dynamic $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
be	0.07	$5.5 \cdot 10^{-7}$	$3.0 \cdot 10^{-6}$	$2.1 \cdot 10^{-5}$	$1.9 \cdot 10^{-4}$	$2.0 \cdot 10^{-3}$
cz	0.10	$7.2 \cdot 10^{-7}$	$3.1 \cdot 10^{-6}$	$2.1 \cdot 10^{-5}$	$1.9 \cdot 10^{-4}$	$2.1 \cdot 10^{-3}$
fi	0.11	$6.2 \cdot 10^{-7}$	$2.7 \cdot 10^{-6}$	$1.9 \cdot 10^{-5}$	$1.8 \cdot 10^{-4}$	$2.0 \cdot 10^{-3}$
au	0.16	$6.9 \cdot 10^{-7}$	$2.7 \cdot 10^{-6}$	$2.0 \cdot 10^{-5}$	$1.9 \cdot 10^{-4}$	$2.2 \cdot 10^{-3}$
ca	0.20	$6.5 \cdot 10^{-7}$	$3.3 \cdot 10^{-6}$	$2.0 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$	$2.4 \cdot 10^{-3}$
po	0.36	$9.8 \cdot 10^{-7}$	$3.4 \cdot 10^{-6}$	$2.2 \cdot 10^{-5}$	$2.0 \cdot 10^{-4}$	$2.3 \cdot 10^{-3}$
it	0.41	$8.6 \cdot 10^{-7}$	$3.3 \cdot 10^{-6}$	$2.2 \cdot 10^{-5}$	$2.2 \cdot 10^{-4}$	$2.3 \cdot 10^{-3}$
gb	0.45	$7.2 \cdot 10^{-7}$	$4.3 \cdot 10^{-6}$	$2.3 \cdot 10^{-5}$	$2.3 \cdot 10^{-4}$	$2.4 \cdot 10^{-3}$
fr	0.72	$7.7 \cdot 10^{-7}$	$3.9 \cdot 10^{-6}$	$2.6 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$	$2.1 \cdot 10^{-3}$
ru	0.67	$1.0 \cdot 10^{-6}$	$4.4 \cdot 10^{-6}$	$2.5 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$	$2.3 \cdot 10^{-3}$
ge	1.17	$8.8 \cdot 10^{-7}$	$4.5 \cdot 10^{-6}$	$2.7 \cdot 10^{-5}$	$2.3 \cdot 10^{-4}$	$2.5 \cdot 10^{-3}$
da	1.46	$1.1 \cdot 10^{-6}$	$4.6 \cdot 10^{-6}$	$3.0 \cdot 10^{-5}$	$2.3 \cdot 10^{-4}$	$2.5 \cdot 10^{-3}$
af	1.27	$8.4 \cdot 10^{-7}$	$4.7 \cdot 10^{-6}$	$2.6 \cdot 10^{-5}$	$2.3 \cdot 10^{-4}$	$2.5 \cdot 10^{-3}$
us	2.29	$9.8 \cdot 10^{-7}$	$5.0 \cdot 10^{-6}$	$3.3 \cdot 10^{-5}$	$2.5 \cdot 10^{-4}$	$2.6 \cdot 10^{-3}$
as	3.20	$8.6 \cdot 10^{-7}$	$5.5 \cdot 10^{-6}$	$3.4 \cdot 10^{-5}$	$2.5 \cdot 10^{-4}$	$2.5 \cdot 10^{-3}$

Edge removals

Graph	Static	Dynamic $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
be	0.07	$4.7 \cdot 10^{-7}$	$2.5 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.6 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
cz	0.10	$5.0 \cdot 10^{-7}$	$2.6 \cdot 10^{-6}$	$1.7 \cdot 10^{-5}$	$1.6 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
fi	0.11	$4.7 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$1.4 \cdot 10^{-4}$	$1.4 \cdot 10^{-3}$
au	0.16	$4.5 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$1.6 \cdot 10^{-5}$	$1.6 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
ca	0.20	$5.9 \cdot 10^{-7}$	$2.7 \cdot 10^{-6}$	$1.6 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
po	0.36	$8.1 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
it	0.41	$9.6 \cdot 10^{-7}$	$2.8 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
gb	0.45	$6.9 \cdot 10^{-7}$	$2.8 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.4 \cdot 10^{-3}$
fr	0.74	$5.0 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
ru	0.67	$9.6 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.4 \cdot 10^{-4}$	$1.4 \cdot 10^{-3}$
ge	1.17	$8.7 \cdot 10^{-7}$	$3.3 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
da	1.49	$8.5 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.9 \cdot 10^{-5}$	$1.6 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
af	1.26	$7.8 \cdot 10^{-7}$	$2.7 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.4 \cdot 10^{-4}$	$1.4 \cdot 10^{-3}$
us	2.30	$8.1 \cdot 10^{-7}$	$3.0 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$
as	3.19	$8.5 \cdot 10^{-7}$	$3.3 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	$1.5 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$

Table H.13: Average running times in seconds of the static and the dynamic Suitor algorithms for 100 batches of $b \in \{1, \ldots, 10^4\}$ edge updates on the R-MAT networks of Table 9.2. The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively.

(a) Edge weights generated by a normal distribution

Edge insertions

Graph	Static	Dynamic $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.41	$7.1 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$9.3 \cdot 10^{-6}$	$4.8 \cdot 10^{-5}$	$3.5 \cdot 10^{-4}$
rmat-23	0.84	$8.1 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$1.0 \cdot 10^{-5}$	$4.8 \cdot 10^{-5}$	$3.6 \cdot 10^{-4}$
rmat-24	1.73	$8.3 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$4.7 \cdot 10^{-5}$	$3.6 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.41	$6.6 \cdot 10^{-7}$	$1.4 \cdot 10^{-6}$	$5.7 \cdot 10^{-6}$	$2.8 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$
rmat-23	0.85	$6.9 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$6.2 \cdot 10^{-6}$	$3.1 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$
rmat-24	1.72	$7.2 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$6.7 \cdot 10^{-6}$	$2.8 \cdot 10^{-5}$	$2.2 \cdot 10^{-4}$

(b) Edge weights generated by an exponential distribution

Edge insertions

Graph	Static	Dynamic $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.41	$7.1 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$9.4 \cdot 10^{-6}$	$5.0 \cdot 10^{-5}$	$3.5 \cdot 10^{-4}$
rmat-23	0.85	$7.9 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$1.0 \cdot 10^{-5}$	$4.9 \cdot 10^{-5}$	$3.6 \cdot 10^{-4}$
rmat-24	1.73	$8.7 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$4.8 \cdot 10^{-5}$	$3.6 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
rmat-22	0.41	$6.4 \cdot 10^{-7}$	$1.4 \cdot 10^{-6}$	$5.8 \cdot 10^{-6}$	$3.0 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$
rmat-23	0.85	$7.4 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$6.2 \cdot 10^{-6}$	$3.3 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$
rmat-24	1.73	$7.4 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$6.7 \cdot 10^{-6}$	$2.7 \cdot 10^{-5}$	$2.2 \cdot 10^{-4}$

Table H.14: Average running times in seconds of the static and the dynamic Suitor algorithms for 100 batches of $b \in \{1, \ldots, 10^4\}$ edge updates on the complex networks of Table 9.1. The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively.

(a) Edge weights generated by a normal distribution

Edge insertions

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.08	$6.0 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$7.8 \cdot 10^{-6}$	$6.4 \cdot 10^{-5}$	$5.6 \cdot 10^{-4}$
cy	0.06	$4.5 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$9.9 \cdot 10^{-6}$	$8.3 \cdot 10^{-5}$	$7.3 \cdot 10^{-4}$
fx	0.11	$7.5 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$6.8 \cdot 10^{-6}$	$5.1 \cdot 10^{-5}$	$3.9 \cdot 10^{-4}$
yg	0.21	$7.1 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$9.0 \cdot 10^{-5}$	$7.9 \cdot 10^{-4}$
fg	0.19	$6.4 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$9.6 \cdot 10^{-6}$	$5.3 \cdot 10^{-5}$	$4.1 \cdot 10^{-4}$
ll	0.69	$7.6 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$9.4 \cdot 10^{-6}$	$8.0 \cdot 10^{-5}$	$6.7 \cdot 10^{-4}$
lj	0.68	$6.5 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$9.7 \cdot 10^{-5}$	$6.6 \cdot 10^{-4}$
ol	0.80	$7.0 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$7.4 \cdot 10^{-5}$	$5.2 \cdot 10^{-4}$
di	1.19	$7.9 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$7.1 \cdot 10^{-6}$	$5.6 \cdot 10^{-5}$	$5.0 \cdot 10^{-4}$
we	2.15	$5.9 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$1.2 \cdot 10^{-5}$	$8.9 \cdot 10^{-5}$	$4.4 \cdot 10^{-4}$
tw	11.23	$8.9 \cdot 10^{-7}$	$2.5 \cdot 10^{-6}$	$1.4 \cdot 10^{-5}$	$7.6 \cdot 10^{-5}$	$4.7 \cdot 10^{-4}$
tm	14.45	$8.9 \cdot 10^{-7}$	$2.5 \cdot 10^{-6}$	$1.3 \cdot 10^{-5}$	$1.0 \cdot 10^{-4}$	$5.0 \cdot 10^{-4}$
fs	21.51	$1.0 \cdot 10^{-6}$	$2.5 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$7.6 \cdot 10^{-5}$	$6.0 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.08	$5.1 \cdot 10^{-7}$	$1.0 \cdot 10^{-6}$	$4.6 \cdot 10^{-6}$	$3.5 \cdot 10^{-5}$	$3.2 \cdot 10^{-4}$
cy	0.06	$4.1 \cdot 10^{-7}$	$1.0 \cdot 10^{-6}$	$6.0 \cdot 10^{-6}$	$5.4 \cdot 10^{-5}$	$4.9 \cdot 10^{-4}$
fx	0.11	$5.9 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$3.3 \cdot 10^{-6}$	$2.3 \cdot 10^{-5}$	$1.8 \cdot 10^{-4}$
yg	0.22	$5.1 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$7.6 \cdot 10^{-6}$	$5.8 \cdot 10^{-5}$	$5.2 \cdot 10^{-4}$
fg	0.19	$6.0 \cdot 10^{-7}$	$1.2 \cdot 10^{-6}$	$6.0 \cdot 10^{-6}$	$3.3 \cdot 10^{-5}$	$2.7 \cdot 10^{-4}$
ll	0.69	$6.9 \cdot 10^{-7}$	$1.4 \cdot 10^{-6}$	$7.6 \cdot 10^{-6}$	$5.5 \cdot 10^{-5}$	$4.8 \cdot 10^{-4}$
lj	0.68	$5.2 \cdot 10^{-7}$	$1.4 \cdot 10^{-6}$	$9.6 \cdot 10^{-6}$	$6.8 \cdot 10^{-5}$	$4.7 \cdot 10^{-4}$
ol	0.80	$5.2 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$5.7 \cdot 10^{-6}$	$5.2 \cdot 10^{-5}$	$3.7 \cdot 10^{-4}$
di	1.19	$5.6 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$4.3 \cdot 10^{-6}$	$3.4 \cdot 10^{-5}$	$3.0 \cdot 10^{-4}$
we	2.15	$4.9 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$7.3 \cdot 10^{-6}$	$5.4 \cdot 10^{-5}$	$2.8 \cdot 10^{-4}$
tw	11.46	$7.0 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$8.0 \cdot 10^{-6}$	$5.3 \cdot 10^{-5}$	$2.9 \cdot 10^{-4}$
tm	14.47	$6.5 \cdot 10^{-7}$	$1.8 \cdot 10^{-6}$	$7.4 \cdot 10^{-6}$	$5.5 \cdot 10^{-5}$	$3.3 \cdot 10^{-4}$
fs	20.99	$7.3 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$5.8 \cdot 10^{-5}$	$4.2 \cdot 10^{-4}$

(b) Edge weights generated by an exponential distribution

Edge insertions

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.08	$6.5 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$8.3 \cdot 10^{-6}$	$6.4 \cdot 10^{-5}$	$5.7 \cdot 10^{-4}$
cy	0.06	$4.6 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$9.7 \cdot 10^{-6}$	$8.3 \cdot 10^{-5}$	$7.4 \cdot 10^{-4}$
fx	0.11	$7.3 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$6.7 \cdot 10^{-6}$	$5.0 \cdot 10^{-5}$	$3.9 \cdot 10^{-4}$
yg	0.21	$6.8 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$8.8 \cdot 10^{-5}$	$7.9 \cdot 10^{-4}$
fg	0.19	$7.1 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$9.5 \cdot 10^{-6}$	$5.5 \cdot 10^{-5}$	$4.1 \cdot 10^{-4}$
ll	0.69	$7.0 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$	$7.7 \cdot 10^{-5}$	$6.7 \cdot 10^{-4}$
lj	0.68	$7.1 \cdot 10^{-7}$	$2.9 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$	$8.2 \cdot 10^{-5}$	$6.6 \cdot 10^{-4}$
ol	0.80	$6.5 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$1.2 \cdot 10^{-5}$	$7.4 \cdot 10^{-5}$	$5.2 \cdot 10^{-4}$
di	1.18	$8.0 \cdot 10^{-7}$	$1.7 \cdot 10^{-6}$	$7.1 \cdot 10^{-6}$	$5.6 \cdot 10^{-5}$	$5.0 \cdot 10^{-4}$
we	2.14	$6.3 \cdot 10^{-7}$	$2.0 \cdot 10^{-6}$	$1.2 \cdot 10^{-5}$	$8.9 \cdot 10^{-5}$	$4.4 \cdot 10^{-4}$
tw	11.41	$7.9 \cdot 10^{-7}$	$2.6 \cdot 10^{-6}$	$1.4 \cdot 10^{-5}$	$9.1 \cdot 10^{-5}$	$4.6 \cdot 10^{-4}$
tm	14.66	$1.0 \cdot 10^{-6}$	$2.5 \cdot 10^{-6}$	$1.3 \cdot 10^{-5}$	$1.1 \cdot 10^{-4}$	$4.8 \cdot 10^{-4}$
fs	21.69	$1.0 \cdot 10^{-6}$	$2.6 \cdot 10^{-6}$	$1.6 \cdot 10^{-5}$	$9.3 \cdot 10^{-5}$	$6.5 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic				
		$b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hy	0.08	$4.7 \cdot 10^{-7}$	$9.8 \cdot 10^{-7}$	$4.5 \cdot 10^{-6}$	$3.5 \cdot 10^{-5}$	$3.2 \cdot 10^{-4}$
cy	0.06	$3.7 \cdot 10^{-7}$	$9.7 \cdot 10^{-7}$	$6.2 \cdot 10^{-6}$	$5.4 \cdot 10^{-5}$	$4.9 \cdot 10^{-4}$
fx	0.11	$6.0 \cdot 10^{-7}$	$9.6 \cdot 10^{-7}$	$3.2 \cdot 10^{-6}$	$2.3 \cdot 10^{-5}$	$1.8 \cdot 10^{-4}$
yg	0.22	$5.6 \cdot 10^{-7}$	$1.6 \cdot 10^{-6}$	$7.6 \cdot 10^{-6}$	$5.9 \cdot 10^{-5}$	$5.2 \cdot 10^{-4}$
fg	0.19	$6.5 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$6.1 \cdot 10^{-6}$	$3.5 \cdot 10^{-5}$	$2.7 \cdot 10^{-4}$
ll	0.69	$6.2 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$8.1 \cdot 10^{-6}$	$5.8 \cdot 10^{-5}$	$4.8 \cdot 10^{-4}$
lj	0.68	$5.6 \cdot 10^{-7}$	$1.5 \cdot 10^{-6}$	$9.8 \cdot 10^{-6}$	$6.9 \cdot 10^{-5}$	$4.7 \cdot 10^{-4}$
ol	0.80	$5.5 \cdot 10^{-7}$	$1.2 \cdot 10^{-6}$	$6.0 \cdot 10^{-6}$	$4.6 \cdot 10^{-5}$	$3.7 \cdot 10^{-4}$
di	1.18	$6.1 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$4.5 \cdot 10^{-6}$	$3.3 \cdot 10^{-5}$	$3.0 \cdot 10^{-4}$
we	2.16	$5.6 \cdot 10^{-7}$	$1.2 \cdot 10^{-6}$	$7.2 \cdot 10^{-6}$	$5.5 \cdot 10^{-5}$	$2.9 \cdot 10^{-4}$
tw	11.47	$7.2 \cdot 10^{-7}$	$1.8 \cdot 10^{-6}$	$7.9 \cdot 10^{-6}$	$4.7 \cdot 10^{-5}$	$2.8 \cdot 10^{-4}$
tm	14.62	$7.5 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$7.9 \cdot 10^{-6}$	$5.7 \cdot 10^{-5}$	$2.9 \cdot 10^{-4}$
fs	21.26	$8.2 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$1.0 \cdot 10^{-5}$	$5.0 \cdot 10^{-5}$	$5.0 \cdot 10^{-4}$

Table H.15: Average running times in seconds of the static and the dynamic Suitor algorithms for 100 batches of $b \in \{1, \ldots, 10^4\}$ edge updates on the random hyperbolic networks of Table 9.2. The columns "Static" and "Dynamic" report the average time (in seconds) for the static and for the dynamic algorithm, respectively.

(a) Edge weights generated by a normal distribution

Edge insertions

Graph	Static	Dynamic $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.21	$6.4 \cdot 10^{-7}$	$1.8 \cdot 10^{-6}$	$8.8 \cdot 10^{-6}$	$7.7 \cdot 10^{-5}$	$7.5 \cdot 10^{-4}$
hyp-23	0.43	$6.8 \cdot 10^{-7}$	$2.2 \cdot 10^{-6}$	$9.3 \cdot 10^{-6}$	$8.2 \cdot 10^{-5}$	$7.5 \cdot 10^{-4}$
hyp-24	0.85	$6.5 \cdot 10^{-7}$	$2.2 \cdot 10^{-6}$	$9.7 \cdot 10^{-6}$	$8.3 \cdot 10^{-5}$	$7.6 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.22	$5.7 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$5.7 \cdot 10^{-6}$	$4.9 \cdot 10^{-5}$	$5.0 \cdot 10^{-4}$
hyp-23	0.43	$6.6 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$5.9 \cdot 10^{-6}$	$5.0 \cdot 10^{-5}$	$5.1 \cdot 10^{-4}$
hyp-24	0.85	$6.7 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$6.1 \cdot 10^{-6}$	$5.2 \cdot 10^{-5}$	$5.3 \cdot 10^{-4}$

(b) Edge weights generated by an exponential distribution

Edge insertions

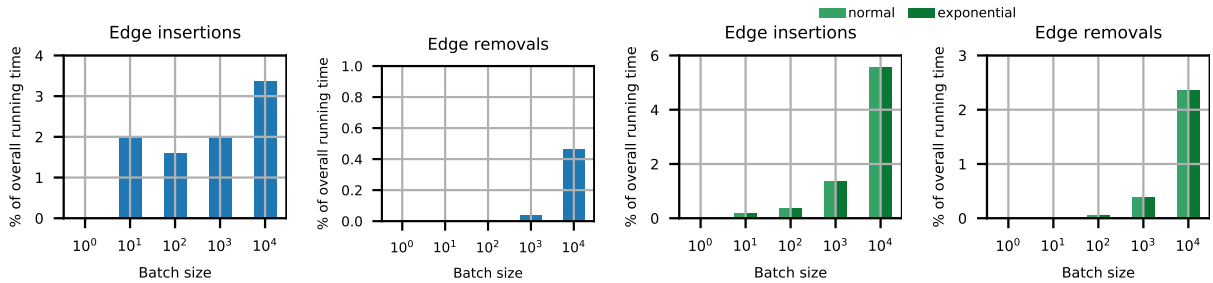
Graph	Static	Dynamic $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.21	$6.3 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	$8.8 \cdot 10^{-6}$	$7.7 \cdot 10^{-5}$	$7.5 \cdot 10^{-4}$
hyp-23	0.43	$7.0 \cdot 10^{-7}$	$2.1 \cdot 10^{-6}$	$9.4 \cdot 10^{-6}$	$8.2 \cdot 10^{-5}$	$7.6 \cdot 10^{-4}$
hyp-24	0.85	$6.6 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	$9.6 \cdot 10^{-6}$	$8.3 \cdot 10^{-5}$	$7.6 \cdot 10^{-4}$

Edge removals

Graph	Static	Dynamic $b = 1$	$b = 10^1$	$b = 10^2$	$b = 10^3$	$b = 10^4$
hyp-22	0.22	$5.7 \cdot 10^{-7}$	$1.1 \cdot 10^{-6}$	$5.7 \cdot 10^{-6}$	$4.9 \cdot 10^{-5}$	$5.0 \cdot 10^{-4}$
hyp-23	0.43	$6.5 \cdot 10^{-7}$	$1.2 \cdot 10^{-6}$	$6.1 \cdot 10^{-6}$	$5.0 \cdot 10^{-5}$	$5.1 \cdot 10^{-4}$
hyp-24	0.85	$6.2 \cdot 10^{-7}$	$1.3 \cdot 10^{-6}$	$5.9 \cdot 10^{-6}$	$5.2 \cdot 10^{-5}$	$5.3 \cdot 10^{-4}$

H.7 Preprocessing Time

Figure H.1 shows the percentage of running time spent by the static Suitor algorithm in sorting the adjacency lists after a batch of edge updates w.r.t. the overall running time.



(a) Road networks (b) Complex networks

Figure H.1: Percentage of time spent by the static Suitor algorithm for the preprocessing step (i.e., sorting the adjacency lists after a batch of edge updates) w.r.t. the overall running time of the algorithm.

Acronyms

ADS	Adaptive Sampling
APSP	All-Pairs Shortest Path
BFS	Breadth-First Search
CC	Connected Component
CG	Conjugate Gradient
DAG	Directed Acyclic Graph
DFS	Depth-First Search
JLT	Johnson-Lindenstrauss Transform
KT	Kendall's Tau
MCM	Maximum Cardinality Matching
MWM	Maximum Weighted Matching
SCC	Strongly Connected Component
SF	State Frame (sampling state of the parallel ADS framework described in Chapter 4)
SSSP	Single-Source Shortest Path
UST	Uniform Spanning Tree
WCC	Weakly Connected Component

Glossary

$G_{\star,\alpha}$	Augmented graph.
$c_b(u)$	Betweenness centrality of u.
$c_{\text{nrwb}}(u)$	Normalized random walk betweenness of u.
$c_c(u)$	Closeness centrality of u.
$c_e(u)$	Electrical closeness of u.
$c_{f,\alpha}(u)$	Forest closeness of u.
$c_{\text{ED}}(u)$	ED-Walk centrality of u.
$c_{\text{eig}}(u)$	Eigenvector centrality of u.
$c_h(u)$	Harmonic centrality of u.
$c_{\text{K}}(u)$	Katz centrality of u.
$c_{\text{PR}}(u)$	PageRank score of u.
$t_c(u,v)$	Commute time between u and v.
$c(u,v)$	Conductance between u and v.
$\deg(u)$	Degree of u.
$\deg_{\text{in}}(u)$	In-degree of u.
$\deg_{\text{out}}(u)$	Out-degree of u.
$\text{diam}(G)$	Diameter of a graph G.
$\zeta_\alpha(u,v)$	Forest distance between u and v.
$\rho(u,v)$	Resistance distance between u and v.
$d(u,v)$	Shortest-path distance between u and v.
$\text{ecc}(u)$	Eccentricity of u.
E	Set of edges in a graph.
$c_{\text{eff}}(u,v)$	Effective conductance between u and v.
$r_{\text{eff}}(u,v)$	Effective resistance (or resistance distance) between u and v.
$i(u,v)$	Electrical current flowing from u to v.
$\mathbf{E}[X]$	Expected value of a random variable X.
$f(u)$	Farness of u.
$f_{f,\alpha}(u)$	Forest farness of u.
$g_c(S)$	Group-closeness of S.
$g_d(S)$	Group-degree of S.
$g_f(S)$	Group-farness of S.
$g_{ff,\alpha}(S)$	Group forest farness of S.
$g_{fc,\alpha}(S)$	Group forest closeness of S.
$g_{\text{ED}}(S)$	GED-Walk centrality of S.

$\phi_i(S)$	Number of walks of length i that contain at least one vertex in $S \subseteq V$.
$g_h(S)$	Group-harmonic of S.
$t_h(u, v)$	Hitting time from u to v.
$\mathcal{K}(G)$	Kirchhoff index of graph G.
$\mathbf{L}_\star^\dagger$	Moore-Penrose pseudoinverse of the Laplacian matrix $\mathbf{L}_\star$.
$\mathbf{A}$	Adjacency matrix.
$\mathbf{D}$	Degree matrix.
$\mathrm{diag}(\mathbf{M})$	Diagonal of a matrix $\mathbf{M}$.
$\mathbf{\Omega}_\alpha$	Forest matrix.
$\mathbf{I}$	Identity matrix.
$\mathbf{L}$	Laplacian matrix.
$\mathbf{L}_\star$	Laplacian matrix of the augmented graph $G_{\star,\alpha}$.
$\mathbf{L}^\dagger$	Moore-Penrose pseudoinverse of the Laplacian matrix $\mathbf{L}$.
ω	Matrix multiplication exponent.
$\mathbf{J}$	Matrix where every element is one.
$\mathrm{tr}(\mathbf{M})$	Trace of a matrix $\mathbf{M}$.
$\mathbf{e}_u$	Canonical unit vector for u.
$\mathbf{j}$	All-ones vector.
$N(u)$	Neighbors of u.
$N(S)$	Set of vertices that have at least one neighbor in $S \subseteq V$.
$N_{\text{in}}(u)$	In-neighbors of u.
$N_{\text{out}}(u)$	Out-neighbors of u.
m	Number of edges in a graph.
n	Number of vertices in a graph.
$p(u)$	Electric potential of vertex u.
$p(u, v)$	Electric potential difference between u and v: $p(u) - p(v)$.
$R(u)$	Vertices reachable from u.
$r(u)$	Number of vertices reachable from u.
$r(u, v)$	Resistance of edge $\{u, v\}$.
$\sigma_{x,y}$	Number of shortest paths from x to y.
$\sigma_{x,y}(u)$	Number of shortest paths from x to y that cross u.
V	Set of vertices in a graph.
$\mathrm{vol}(G)$	Volume of a graph G.
$w(u, v)$	Weight of the edge $\{u, v\}$ (or (u, v) if the graph is directed).

Bibliography

1. A. Abboud, F. Grandoni, and V. V. Williams. "Subcubic Equivalences Between Graph Centrality Problems, APSP and Diameter". In: *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*. SIAM, 2015, pp. 1681–1697. DOI: 10.1137/1.9781611973730.112.
2. A. Abboud, V. V. Williams, and J. R. Wang. "Approximation and Fixed Parameter Subquadratic Algorithms for Radius and Diameter in Sparse Graphs". In: *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*. SIAM, 2016, pp. 377–391. DOI: 10.1137/1.9781611974331.ch28.
3. R. Action. "the Rise of the Medici". *American Journal of Sociology* 98, 1993, pp. 1259–1319.
4. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
5. R. Albert and A. Barabási. "Statistical mechanics of complex networks". *CoRR* cond-mat/0106096, 2001.
6. R. Albert, H. Jeong, and A. Barabási. "The diameter of the world wide web". *CoRR* cond-mat/9907038, 1999.
7. D. J. Aldous. "The Random Walk Construction of Uniform Spanning Trees and Uniform Labelled Trees". *SIAM J. Discret. Math.* 3:4, 1990, pp. 450–465. DOI: 10.1137/0403039.
8. V. L. Alev, N. Anari, L. C. Lau, and S. O. Gharan. "Graph Clustering using Effective Resistance". In: *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 41:1–41:16. DOI: 10.4230/LIPIcs.ITCS.2018.41.
9. J. Alman and V. V. Williams. "A Refined Laser Method and Faster Matrix Multiplication". In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*. SIAM, 2021, pp. 522–539. DOI: 10.1137/1.9781611976465.32.
10. P. Amestoy, I. S. Duff, J. L'Excellent, and F. Rouet. "Parallel Computation of Entries of A^{-1}". *SIAM J. Sci. Comput.* 37:2, 2015. DOI: 10.1137/120902616.
11. A. Anand, S. Baswana, M. Gupta, and S. Sen. "Maintaining Approximate Maximum Weighted Matching in Fully Dynamic Graphs". In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, pp. 257–266. DOI: 10.4230/LIPIcs.FSTTCS.2012.257.

12. E. Angriman, R. Becker, G. D'Angelo, H. Gilbert, A. van der Grinten, and H. Meyerhenke. "Group-Harmonic and Group-Closeness Maximization - Approximation and Engineering". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*. SIAM, 2021, pp. 154–168. DOI: 10.1137/1.9781611976472.12.

13. E. Angriman, A. van der Grinten, A. Bojchevski, D. Zügner, S. Günnemann, and H. Meyerhenke. "Group Centrality Maximization for Large-scale Graphs". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*. SIAM, 2020, pp. 56–69. DOI: 10.1137/1.9781611976007.5.

14. E. Angriman, A. van der Grinten, M. von Looz, H. Meyerhenke, M. Nöllenburg, M. Predari, and C. Tzovas. "Guidelines for Experimental Algorithmics: A Case Study in Network Analysis". *Algorithms* 12:7, 2019, p. 127. DOI: 10.3390/a12070127.

15. E. Angriman, A. van der Grinten, and H. Meyerhenke. "Local Search for Group Closeness Maximization on Big Graphs". In: *2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019*. IEEE, 2019, pp. 711–720. DOI: 10.1109/BigData47090.2019.9006206.

16. E. Angriman, H. Meyerhenke, C. Schulz, and B. Uçar. "Fully-dynamic Weighted Matching Approximation in Practice". In: *Proceedings of the 2021 SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. 2021, pp. 32–44. DOI: 10.1137/1.9781611976830.4.

17. E. Angriman, M. Predari, A. van der Grinten, and H. Meyerhenke. "Approximation of the Diagonal of a Laplacian's Pseudoinverse for Complex Network Analysis". In: *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 6:1–6:24. DOI: 10.4230/LIPIcs.ESA.2020.6.

18. M. Arar, S. Chechik, S. Cohen, C. Stein, and D. Wajc. "Dynamic Matching: Reducing Integral Algorithms to Approximately-Maximal Fractional Algorithms". In: *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 7:1–7:16. DOI: 10.4230/LIPIcs.ICALP.2018.7.

19. M. Arbel and H. Attiya. "Concurrent updates with RCU: search tree as an example". In: *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*. ACM, 2014, pp. 196–205. DOI: 10.1145/2611462.2611471.

20. V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit. "Local Search Heuristics for k-Median and Facility Location Problems". *SIAM J. Comput.* 33:3, 2004, pp. 544–562. DOI: 10.1137/S0097539702416402.

21. M. Ashtiani, A. Salehzadeh-Yazdi, Z. Razaghi-Moghadam, H. Hennig, O. Wolkenhauer, M. Mirzaie, and M. Jafari. "A systematic survey of centrality measures for protein-protein interaction networks". *BMC Syst. Biol.* 12:1, 2018, 80:1–80:17. DOI: 10.1186/s12918-018-0598-2.

22. D. Avis. "A survey of heuristics for the weighted matching problem". *Networks* 13:4, 1983, pp. 475–493. DOI: 10.1002/net.3230130404.

23. K. Avrachenkov, P. Gonçalves, and M. Sokol. "On the Choice of Kernel and Labelled Data in Semi-supervised Learning Methods". In: *Algorithms and Models for the Web Graph - 10th International Workshop, WAW 2013, Cambridge, MA, USA, December 14-15, 2013, Proceedings*. Springer, 2013, pp. 56–67. DOI: 10.1007/978-3-319-03536-9_5.

24. H. Avron and S. Toledo. "Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix". *J. ACM* 58:2, 2011, 8:1–8:34. DOI: 10.1145/1944345.1944349.

25. L. Backstrom and J. M. Kleinberg. "Romantic partnerships and the dispersion of social ties: a network analysis of relationship status on facebook". In: *Computer Supported Cooperative Work, CSCW '14, Baltimore, MD, USA, February 15-19, 2014*. ACM, 2014, pp. 831–841. DOI: 10.1145/2531602.2531642.

26. D. A. Bader, G. Cong, and J. Feo. "On the Architectural Requirements for Efficient Execution of Graph Algorithms". In: *34th International Conference on Parallel Processing (ICPP 2005), 14-17 June 2005, Oslo, Norway*. IEEE Computer Society, 2005, pp. 547–556. DOI: 10.1109/ICPP.2005.55.

27. D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. "Approximating Betweenness Centrality". In: *Algorithms and Models for the Web-Graph, 5th International Workshop, WAW 2007, San Diego, CA, USA, December 11-12, 2007, Proceedings*. Springer, 2007, pp. 124–137. DOI: 10.1007/978-3-540-77004-6_10.

28. A.-L. Barabási and R. Albert. "Emergence of scaling in random networks". *science* 286:5439, 1999, pp. 509–512.

29. L. Barenboim and T. Maimon. "Fully Dynamic Graph Algorithms Inspired by Distributed Computing: Deterministic Maximal Matching and Edge Coloring in Sublinear Update-Time". *ACM J. Exp. Algorithmics* 24:1, 2019, 1.14:1–1.14:24. DOI: 10.1145/3338529.

30. S. Barthelmé, N. Tremblay, A. Gaudilliere, L. Avena, and P.-O. Amblard. "Estimating the inverse trace using random forests on graphs". *arXiv preprint arXiv:1905.02086*, 2019.

31. M. Bastian, S. Heymann, and M. Jacomy. "Gephi: An Open Source Software for Exploring and Manipulating Networks". In: *Proceedings of the Third International Conference on Weblogs and Social Media, ICWSM 2009, San Jose, California, USA, May 17-20, 2009*. The AAAI Press, 2009.

32. S. Baswana, M. Gupta, and S. Sen. "Fully Dynamic Maximal Matching in O(log n) Update Time (Corrected Version)". *SIAM J. Comput.* 47:3, 2018, pp. 617–650. DOI: 10.1137/16M1106158.

33. A. Bavelas. "A mathematical model for group structures". *Applied anthropology* 7:3, 1948, pp. 16–30.

34. A. Bavelas. "Communication patterns in task-oriented groups". *The journal of the acoustical society of America* 22:6, 1950, pp. 725–730.

35. M. A. Beauchamp. "An improved index of centrality". *Behavioral science* 10:2, 1965, pp. 161–163.

36. C. Bekas, E. Kokiopoulou, and Y. Saad. "An estimator for the diagonal of a matrix". *Applied numerical mathematics* 57:11-12, 2007, pp. 1214–1229.

37. E. Bergamini, M. Borassi, P. Crescenzi, A. Marino, and H. Meyerhenke. "Computing top-k Closeness Centrality Faster in Unweighted Graphs". *ACM Trans. Knowl. Discov. Data* 13:5, 2019, 53:1–53:40. DOI: 10.1145/3344719.

38. E. Bergamini, T. Gonser, and H. Meyerhenke. "Scaling up Group Closeness Maximization". In: *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018, New Orleans, LA, USA, January 7-8, 2018*. SIAM, 2018, pp. 209–222. DOI: 10.1137/1.9781611975055.18.

39. E. Bergamini and H. Meyerhenke. "Approximating Betweenness Centrality in Fully Dynamic Networks". *Internet Math.* 12:5, 2016, pp. 281–314. DOI: 10.1080/15427951.2016.1177802.

40. E. Bergamini, H. Meyerhenke, M. Ortmann, and A. Slobbe. "Faster Betweenness Centrality Updates in Evolving Networks". In: *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 23:1–23:16. DOI: 10.4230/LIPIcs.SEA.2017.23.

41. E. Bergamini, M. Wegner, D. Lukarski, and H. Meyerhenke. "Estimating Current-Flow Closeness Centrality with a Multigrid Laplacian Solver". In: *2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing, CSC 2016, Albuquerque, New Mexico, USA, October 10-12, 2016*. SIAM, 2016, pp. 1–12. DOI: 10.1137/1.9781611974690.ch1.

42. A. Berman and R. J. Plemmons. *Nonnegative Matrices in the Mathematical Sciences*. SIAM, 1994. DOI: 10.1137/1.9781611971262.

43. H. R. Bernard. "The Development of Social Network Analysis: A Study in the Sociology of Science, Linton C. Freeman. Empirical Press, Vancouver, BC (2004)". *Soc. Networks* 27:4, 2005, pp. 377–384. DOI: 10.1016/j.socnet.2005.06.004.

44. A. Bernstein and C. Stein. "Faster Fully Dynamic Matchings with Small Approximation Ratios". In: *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*. SIAM, 2016, pp. 692–711. DOI: 10.1137/1.9781611974331.ch50.

45. S. Bhattacharya, M. Henzinger, and G. F. Italiano. "Deterministic Fully Dynamic Data Structures for Vertex Cover and Matching". *SIAM J. Comput.* 47:3, 2018, pp. 859–887. DOI: 10.1137/140998925.

46. S. Bhattacharya, M. Henzinger, and D. Nanongkai. "New deterministic approximation algorithms for fully dynamic matching". In: *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*. ACM, 2016, pp. 398–411. DOI: 10.1145/2897518.2897568.

47. M. Birn, V. Osipov, P. Sanders, C. Schulz, and N. Sitchinava. "Efficient Parallel and External Matching". In: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*. Springer, 2013, pp. 659–670. DOI: 10.1007/978-3-642-40047-6_66.

48. P. Bisenius, E. Bergamini, E. Angriman, and H. Meyerhenke. "Computing Top-k Closeness Centrality in Fully-dynamic Graphs". In: *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018, New Orleans, LA, USA, January 7-8, 2018*. SIAM, 2018, pp. 21–35. DOI: 10.1137/1.9781611975055.3.

49. R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP*. Oxford University Press, USA, 2020.

50. R. E. Bixby. "Solving Real-World Linear Programs: A Decade and More of Progress". *Oper. Res.* 50:1, 2002, pp. 3–15. DOI: 10.1287/opre.50.1.3.17780.

51. G. E. Blelloch, A. Gupta, I. Koutis, G. L. Miller, R. Peng, and K. Tangwongsan. "Near linear-work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs". In: *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*. ACM, 2011, pp. 13–22. DOI: 10.1145/1989493.1989496.

52. P. Boldi, M. Rosa, and S. Vigna. "Robustness of social and web graphs to node removal". *Soc. Netw. Anal. Min.* 3:4, 2013, pp. 829–842. DOI: 10.1007/s13278-013-0096-x.

53. P. Boldi and S. Vigna. "Axioms for Centrality". *Internet Math.* 10:3-4, 2014, pp. 222–262. DOI: 10.1080/15427951.2013.865686.

54. B. Bollobás. *Modern Graph Theory*. Springer, 2002. DOI: 10.1007/978-1-4612-0619-4.

55. M. Borassi, P. Crescenzi, and M. Habib. "Into the Square: On the Complexity of Some Quadratic-time Solvable Problems". *Electron. Notes Theor. Comput. Sci.* 322, 2016, pp. 51–67. DOI: 10.1016/j.entcs.2016.03.005.

56. M. Borassi and E. Natale. "KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation". In: *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 20:1–20:18. DOI: 10.4230/LIPIcs.ESA.2016.20.

57. S. P. Borgatti. "Centrality and network flow". *Soc. Networks* 27:1, 2005, pp. 55–71. DOI: 10.1016/j.socnet.2004.11.008.

58. K. M. Borgwardt, C. S. Ong, S. Schönauer, S. V. N. Vishwanathan, A. J. Smola, and H. Kriegel. "Protein function prediction via graph kernels". In: *Proceedings Thirteenth International Conference on Intelligent Systems for Molecular Biology 2005, Detroit, MI, USA, 25-29 June 2005*. 2005, pp. 47–56. DOI: 10.1093/bioinformatics/bti1007.

59. S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. T. Morris, and N. Zeldovich. "An Analysis of Linux Scalability to Many Cores". In: *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. USENIX Association, 2010, pp. 1–16.

60. E. Bozzo and M. Franceschet. "Resistance distance, closeness, and betweenness". *Soc. Networks* 35:3, 2013, pp. 460–469. DOI: 10.1016/j.socnet.2013.05.003.

61. U. Brandes. "A faster algorithm for betweenness centrality". *Journal of mathematical sociology* 25:2, 2001, pp. 163–177.

62. U. Brandes and D. Fleischer. "Centrality Measures Based on Current Flow". In: *STACS 2005, 22nd Annual Symposium on Theoretical Aspects of Computer Science, Stuttgart, Germany, February 24-26, 2005, Proceedings*. Springer, 2005, pp. 533–544. DOI: 10.1007/978-3-540-31856-9_44.

63. U. Brandes and C. Pich. "Centrality Estimation in Large Networks". *Int. J. Bifurc. Chaos* 17:7, 2007, pp. 2303–2318. DOI: 10.1142/S0218127407018403.

64. A. Z. Broder. "Generating Random Spanning Trees". In: *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*. IEEE Computer Society, 1989, pp. 442–447. DOI: 10.1109/SFCS.1989.63516.

65. D. Chakrabarti, Y. Zhan, and C. Faloutsos. "R-MAT: A Recursive Model for Graph Mining". In: *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*. SIAM, 2004, pp. 442–446. DOI: 10.1137/1.9781611972740.43.

66. O. Chapelle, B. Scholkopf, and A. Zien. "Semi-supervised learning (chapelle, o. et al., eds.; 2006) [book reviews]". *IEEE Transactions on Neural Networks* 20:3, 2009, pp. 542–542.

67. O. Chapelle, J. Weston, and B. Schölkopf. "Cluster Kernels for Semi-Supervised Learning". In: *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS 2002, December 9-14, 2002, Vancouver, British Columbia, Canada]*. MIT Press, 2002, pp. 585–592.

68. M. Charikar and S. Solomon. "Fully Dynamic Almost-Maximal Matching: Breaking the Polynomial Barrier for Worst-Case Time Bounds". *CoRR* abs/1711.06883, 2017.

69. P. Y. Chebotarev and E. Shamis. "The forest metrics of a graph and their properties". *AUTOMATION AND REMOTE CONTROL C/C OF AVTOMATIKA I TELEMEKHANIKA* 61:8; ISSU 2, 2000, pp. 1364–1373.

70. P. Chebotarev and E. Shamis. "On Proximity Measures for Graph Vertices". *CoRR* abs/math/0602073, 2006.

71. P. Chebotarev and E. Shamis. "The Forest Metrics for Graph Vertices". *Electron. Notes Discret. Math.* 11, 2002, pp. 98–107. DOI: 10.1016/S1571-0653(04)00058-7.

72. P. Chebotarev and E. Shamis. "The Matrix-Forest Theorem and Measuring Relations in Small Social Groups". *CoRR* abs/math/0602070, 2006.

73. S. Chechik, E. Cohen, and H. Kaplan. "Average Distance Queries through Weighted Samples in Graphs and Metric Spaces: High Scalability with Tight Statistical Guarantees". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2015, August 24-26, 2015, Princeton, NJ, USA*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 659–679. DOI: 10.4230/LIPIcs.APPROX-RANDOM.2015.659.

74. M. H. Chehreghani, A. Bifet, and T. Abdessalem. "An In-depth Comparison of Group Betweenness Centrality Estimation Algorithms". In: *IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10-13, 2018*. IEEE, 2018, pp. 2104–2113. DOI: 10.1109/BigData.2018.8622133.

75. M. H. Chehreghani, A. Bifet, and T. Abdessalem. "Novel Adaptive Algorithms for Estimating Betweenness, Coverage and k-path Centralities". *CoRR* abs/1810.10094, 2018.

76. C. Chen, W. Wang, and X. Wang. "Efficient Maximum Closeness Centrality Group Identification". In: *Databases Theory and Applications - 27th Australasian Database Conference, ADC 2016, Sydney, NSW, Australia, September 28-29, 2016, Proceedings*. Springer, 2016, pp. 43–55. DOI: 10.1007/978-3-319-46922-5_4.

77. A. T. Clements, M. F. Kaashoek, and N. Zeldovich. "Scalable address spaces using RCU balanced trees". In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*. ACM, 2012, pp. 199–210. DOI: 10.1145/2150976.2150998.

78. E. Cohen. "Size-Estimation Framework with Applications to Transitive Closure and Reachability". *J. Comput. Syst. Sci.* 55:3, 1997, pp. 441–453. DOI: 10.1006/jcss.1997.1534.

79. E. Cohen, D. Delling, T. Pajor, and R. F. Werneck. "Computing classic closeness centrality, at scale". In: *Proceedings of the second ACM conference on Online social networks, COSN 2014, Dublin, Ireland, October 1-2, 2014*. ACM, 2014, pp. 37–50. DOI: 10.1145/2660460.2660465.

80. M. B. Cohen, R. Kyng, G. L. Miller, J. W. Pachocki, R. Peng, A. B. Rao, and S. C. Xu. "Solving SDD linear systems in nearly $m\log^{1/2}n$ time". In: *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*. ACM, 2014, pp. 343–352. DOI: 10.1145/2591796.2591833.

81. B. S. Cohn and M. Marriott. "Networks and centres of integration in Indian civilization". *Journal of social Research* 1:1, 1958, pp. 1–9.

82. D. Conte, P. Foggia, C. Sansone, and M. Vento. "Thirty Years Of Graph Matching In Pattern Recognition". *Int. J. Pattern Recognit. Artif. Intell.* 18:3, 2004, pp. 265–298. DOI: 10.1142/S0218001404003228.

83. O. contributors. *Planet dump retrieved from https://planet.osm.org*. 2017.

84. M. S. Crouch and D. M. Stubbs. "Improved Streaming Algorithms for Weighted Matching, via Unweighted Matching". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2014, September 4-6, 2014, Barcelona, Spain*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014, pp. 96–104. DOI: 10.4230/LIPIcs.APPROX-RANDOM.2014.96.

85. T. David, R. Guerraoui, and V. Trigonakis. "Everything you always wanted to know about synchronization but were afraid to ask". In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 2013, pp. 33–48. DOI: 10.1145/2517349.2522714.

86. A. Dekker. "Conceptual Distance in Social Network Analysis". *J. Soc. Struct.* 6, 2005.

87. C. Demetrescu, A. V. Goldberg, and D. S. Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. American Mathematical Soc., 2009.

88. C. Demetrescu and G. F. Italiano. "A new approach to dynamic all pairs shortest paths". *J. ACM* 51:6, 2004, pp. 968–992. DOI: 10.1145/1039488.1039492.

89. J. Dibbelt, B. Strasser, and D. Wagner. "Customizable Contraction Hierarchies". *ACM J. Exp. Algorithmics* 21:1, 2016, 1.5:1–1.5:49. DOI: 10.1145/2886843.

90. P. D. Dobson and A. J. Doig. "Distinguishing enzyme structures from non-enzymes without alignments". *Journal of molecular biology* 330:4, 2003, pp. 771–783.

91. S. Dolev, Y. Elovici, R. Puzis, and P. Zilberman. "Incremental deployment of network monitors based on Group Betweenness Centrality". *Inf. Process. Lett.* 109:20, 2009, pp. 1172–1176. DOI: 10.1016/j.ipl.2009.07.019.

92. K. Dong, A. R. Benson, and D. Bindel. "Network Density of States". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. ACM, 2019, pp. 1152–1161. DOI: 10.1145/3292500.3330891.

93. M. Doostmohammadian, H. R. Rabiee, and U. A. Khan. "Centrality-based epidemic control in complex social networks". *Soc. Netw. Anal. Min.* 10:1, 2020, p. 32. DOI: 10.1007/s13278-020-00638-7.

94. D. E. Drake and S. Hougardy. "A simple approximation algorithm for the weighted matching problem". *Inf. Process. Lett.* 85:4, 2003, pp. 211–213. DOI: 10.1016/S0020-0190(02)00393-9.

95. D. E. Drake and S. Hougardy. "Linear Time Local Improvements for Weighted Matchings in Graphs". In: *Experimental and Efficient Algorithms, Second International Workshop, WEA 2003, Ascona, Switzerland, May 26-28, 2003, Proceedings*. Springer, 2003, pp. 107–119. DOI: 10.1007/3-540-44867-5_9.

96. A. Droschinsky, P. Mutzel, and E. Thordsen. "Shrinking Trees not Blossoms: A Recursive Maximum Matching Approach". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*. SIAM, 2020, pp. 146–160. DOI: 10.1137/1.9781611976007.12.

97. R. Duan, S. Pettie, and H. Su. "Scaling Algorithms for Weighted Matching in General Graphs". *ACM Trans. Algorithms* 14:1, 2018, 8:1–8:35. DOI: 10.1145/3155301.

98. D. Ediger, J. Riedy, D. A. Bader, and H. Meyerhenke. "Computational graph analytics for massive streaming data". *Large Scale Network-Centric Distributed Systems*, 2013, pp. 619–648.

99. J. Edmonds. "Paths, trees, and flowers". *Canadian Journal of mathematics* 17, 1965, pp. 449–467.

100. W. Ellens, F. Spieksma, P Van Mieghem, A Jamakovic, and R. Kooij. "Effective graph resistance". *Linear algebra and its applications* 435:10, 2011, pp. 2491–2506.

101. D. Eppstein, Z. Galil, and G. F. Italiano. "Dynamic Graph Algorithms". In: *Algorithms and Theory of Computation Handbook*. Ed. by M. J. Atallah. Chapman & Hall/CRC Applied Algorithms and Data Structures series. CRC Press, 1999.

102. D. Eppstein and J. Wang. "Fast Approximation of Centrality". *J. Graph Algorithms Appl.* 8, 2004, pp. 39–45. DOI: 10.7155/jgaa.00081.

103. L. Epstein, A. Levin, J. Mestre, and D. Segev. "Improved Approximation Guarantees for Weighted Matching in the Semi-streaming Model". *SIAM J. Discret. Math.* 25:3, 2011, pp. 1251–1265. DOI: 10.1137/100801901.

104. M. G. Everett and S. P. Borgatti. "The centrality of groups and classes". *The Journal of mathematical sociology* 23:3, 1999, pp. 181–201.

105. E. Eyal and D. Halperin. "Dynamic maintenance of molecular surfaces under conformational changes". In: *Proceedings of the 21st ACM Symposium on Computational Geometry, Pisa, Italy, June 6-8, 2005*. ACM, 2005, pp. 45–54. DOI: 10.1145/1064092.1064102.

106. U. Feige, V. S. Mirrokni, and J. Vondrák. "Maximizing Non-monotone Submodular Functions". *SIAM J. Comput.* 40:4, 2011, pp. 1133–1153. DOI: 10.1137/090779346.

107. J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. "On graph problems in a semi-streaming model". *Theor. Comput. Sci.* 348:2-3, 2005, pp. 207–216. DOI: 10.1016/j.tcs.2005.09.013.

108. L. C. Freeman. "A set of measures of centrality based on betweenness". *Sociometry*, 1977, pp. 35–41.

109. L. C. Freeman. "Centrality in social networks conceptual clarification". *Social networks* 1:3, 1978, pp. 215–239.

110. G. Frobenius, F. G. Frobenius, F. G. Frobenius, F. G. Frobenius, and G. Mathematician. "Über Matrizen aus nicht negativen Elementen", 1912.

111. T. M. J. Fruchterman and E. M. Reingold. "Graph Drawing by Force-directed Placement". *Softw. Pract. Exp.* 21:11, 1991, pp. 1129–1164. DOI: 10.1002/spe.4380211102.

112. T. Fushimi, K. Saito, T. Ikeda, and K. Kazama. "A New Group Centrality Measure for Maximizing the Connectedness of Network Under Uncertain Connectivity". In: *Complex Networks and Their Applications VII - Volume 1 Proceedings The 7th International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2018, Cambridge, UK, December 11-13, 2018*. Springer, 2018, pp. 3–14. DOI: 10.1007/978-3-030-05411-3_1.

113. H. N. Gabow. "Data Structures for Weighted Matching and Nearest Common Ancestors with Linking". In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*. SIAM, 1990, pp. 434–443.

114. H. N. Gabow and R. E. Tarjan. "Faster Scaling Algorithms for General Graph-Matching Problems". *J. ACM* 38:4, 1991, pp. 815–853. DOI: 10.1145/115234.115366.

115. D. Gale and L. S. Shapley. "College Admissions and the Stability of Marriage". *Am. Math. Mon.* 120:5, 2013, pp. 386–391. DOI: 10.4169/amer.math.monthly.120.05.386.

116. Z. Galil. "Efficient algorithms for finding maximum matching in graphs". *ACM Computing Surveys (CSUR)* 18:1, 1986, pp. 23–38.

117. A. Galland and M. Lelarge. "Invariant embedding for graph classification". In: *ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Data*. 2019.

118. G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, et al. "The scip optimization suite 7.0", 2020.

119. M. Ghaffari and D. Wajc. "Simplified and Space-Optimal Semi-Streaming (2+epsilon)-Approximate Matching". In: *2nd Symposium on Simplicity in Algorithms, SOSA 2019, January 8-9, 2019, San Diego, CA, USA*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 13:1–13:8. DOI: 10.4230/OASIcs.SOSA.2019.13.

120. A. Ghosh, S. P. Boyd, and A. Saberi. "Minimizing Effective Resistance of a Graph". *SIAM Rev.* 50:1, 2008, pp. 37–66. DOI: 10.1137/050645452.

121. C. Gkantsidis, M. Mihail, and A. Saberi. "Random walks in peer-to-peer networks: Algorithms and evaluation". *Perform. Evaluation* 63:3, 2006, pp. 241–263. DOI: 10.1016/j.peva.2005.01.002.

122. D. F. Gleich. "PageRank Beyond the Web". *SIAM Rev.* 57:3, 2015, pp. 321–363. DOI: 10.1137/140976649.

123. C. D. Godsil and G. F. Royle. *Algebraic Graph Theory*. Springer, 2001. DOI: 10.1007/978-1-4613-0163-9.

124. 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, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*. SIAM, 2005, pp. 156–165.

125. A. V. Goldberg and A. V. Karzanov. "Maximum skew-symmetric flows and matchings". *Math. Program.* 100:3, 2004, pp. 537–568. DOI: 10.1007/s10107-004-0505-z.

126. G. H. Golub and C. F. V. Loan. *Matrix Computations, Fourth Edition*. Johns Hopkins University Press, 2013.

127. T. F. Gonzalez, ed. *Handbook of Approximation Algorithms and Metaheuristics, Second Edition, Volume 1: Methologies and Traditional Applications*. Chapman and Hall/CRC, 2018. DOI: 10.1201/9781351236423.

128. F. Grandoni, S. Leonardi, P. Sankowski, C. Schwiegelshohn, and S. Solomon. "$(1+\varepsilon)$-Approximate Incremental Matching in Constant Deterministic Amortized Time". In: *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*. SIAM, 2019, pp. 1886–1898. DOI: 10.1137/1.9781611975482.114.

129. O. Green, R. McColl, and D. A. Bader. "A Fast Algorithm for Streaming Betweenness Centrality". In: *2012 International Conference on Privacy, Security, Risk and Trust, PASSAT 2012, and 2012 International Confernece on Social Computing, SocialCom 2012, Amsterdam, Netherlands, September 3-5, 2012*. IEEE Computer Society, 2012, pp. 11–20. DOI: 10.1109/SocialCom-PASSAT.2012.37.

130. K. D. Gremban. "Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems". PhD thesis. 1996.

131. A. van der Grinten, E. Angriman, and H. Meyerhenke. "Parallel Adaptive Sampling with Almost No Synchronization". In: *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*. Springer, 2019, pp. 434–447. DOI: 10.1007/978-3-030-29400-7_31.

132. A. van der Grinten, E. Angriman, and H. Meyerhenke. "Scaling up network centrality computations - A brief overview". *it Inf. Technol.* 62:3-4, 2020, pp. 189–204. DOI: 10.1515/itit-2019-0032.

133. A. van der Grinten, E. Angriman, M. Predari, and H. Meyerhenke. "New Approximation Algorithms for Forest Closeness Centrality - for Individual Vertices and Vertex Groups". In: *Proceedings of the 2021 SIAM International Conference on Data Mining, SDM 2021, Virtual Event, April 29 - May 1, 2021*. SIAM, 2021, pp. 136–144. DOI: 10.1137/1.9781611976700.16.

134. A. van der Grinten, E. Bergamini, O. Green, D. A. Bader, and H. Meyerhenke. "Scalable Katz Ranking Computation in Large Static and Dynamic Graphs". In: *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 42:1–42:14. DOI: 10.4230/LIPIcs.ESA.2018.42.

135. G. Guennebaud, B. Jacob, et al. *Eigen v3*. 2010.

136. R. Guimera, S. Mossa, A. Turtschi, and L. N. Amaral. "The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles". *Proceedings of the National Academy of Sciences* 102:22, 2005, pp. 7794–7799.

137. M. Gupta and R. Peng. "Fully Dynamic (1+ e)-Approximate Matchings". In: *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*. IEEE Computer Society, 2013, pp. 548–557. DOI: 10.1109/FOCS.2013.65.

138. M. M. Halldórsson, S. Köhler, B. Patt-Shamir, and D. Rawitz. "Distributed backup placement in networks". *Distributed Comput.* 31:2, 2018, pp. 83–98. DOI: 10.1007/s00446-017-0299-x.

139. T. Hayashi, T. Akiba, and Y. Yoshida. "Efficient Algorithms for Spanning Tree Centrality". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*. IJCAI/AAAI Press, 2016, pp. 3733–3739.

140. M. Henzinger, S. Khan, R. Paul, and C. Schulz. "Dynamic Matching Algorithms in Practice". In: *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 58:1–58:20. DOI: 10.4230/LIPIcs.ESA.2020.58.

141. J. Hoepman. "Simple Distributed Weighted Matchings". *CoRR* cs.DC/0410047, 2004.

142. P. Holme and G. Ghoshal. "Dynamics of networking agents competing for high centrality and low degree". *Physical review letters* 96:9, 2006, p. 098701.

143. H. Hu, J. Zhang, X. Zheng, Y. Yang, and P. Wu. "Self-configuration and self-optimization for LTE networks". *IEEE Commun. Mag.* 48:2, 2010, pp. 94–100. DOI: 10.1109/MCOM.2010.5402670.

144. M. F. Hutchinson. "A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines". *Communications in Statistics-Simulation and Computation* 18:3, 1989, pp. 1059–1076.

145. R. Impagliazzo, R. Paturi, and F. Zane. "Which Problems Have Strongly Exponential Complexity?" *J. Comput. Syst. Sci.* 63:4, 2001, pp. 512–530. DOI: 10.1006/jcss.2001.1774.

146. V. Ishakian, D. Erdös, E. Terzi, and A. Bestavros. "A Framework for the Evaluation and Management of Network Centrality". In: *Proceedings of the Twelfth SIAM International Conference on Data Mining, Anaheim, California, USA, April 26-28, 2012*. SIAM / Omnipress, 2012, pp. 427–438. DOI: 10.1137/1.9781611972825.37.

147. ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2012.

148. Z. Ivkovic and E. L. Lloyd. "Fully Dynamic Maintenance of Vertex Cover". In: *Graph-Theoretic Concepts in Computer Science, 19th International Workshop, WG '93, Utrecht, The Netherlands, June 16-18, 1993, Proceedings*. Springer, 1993, pp. 99–111. DOI: 10.1007/3-540-57899-4_44.

149. K. Iwama, S. Miyazaki, and K. Okamoto. "Stable roommates problem with triple rooms". In: *Proc. 10th KOREA-JAPAN joint workshop on algorithms and computation (WAAC 2007)*. 2007, pp. 105–112.

150. N. S. Izmailian, R. Kenna, and F. Wu. "The two-point resistance of a resistor network: a new formulation and application to the cobweb network". *Journal of Physics A: Mathematical and Theoretical* 47:3, 2013, p. 035003.

151. M. Jacquelin, L. Lin, and C. Yang. "PSelInv - A distributed memory parallel algorithm for selected inversion: The non-symmetric case". *Parallel Comput.* 74, 2018, pp. 84–98. DOI: 10.1016/j.parco.2017.11.009.

152. H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai. "Lethality and centrality in protein networks". *Nature* 411:6833, 2001, pp. 41–42.

153. Y. Jin, Q. Bao, and Z. Zhang. "Forest Distance Closeness Centrality in Disconnected Graphs". In: *2019 IEEE International Conference on Data Mining, ICDM 2019, Beijing, China, November 8-11, 2019*. IEEE, 2019, pp. 339–348. DOI: 10.1109/ICDM.2019.00044.

154. R. Johnson. "The stable marriage problem: Structure and algorithms, by Dan Gusfield and Robert Irving, The MIT Press, Cambridge, MA, 1989, 240 pp., $27.50". *Networks* 24:2, 1994, pp. 129–130. DOI: 10.1002/net.3230240219.

155. W. B. Johnson and J. Lindenstrauss. "Extensions of Lipschitz mappings into a Hilbert space 26". *Contemporary mathematics* 26, 1984.

156. U Kang, S. Papadimitriou, J. Sun, and H. Tong. "Centralities in Large Networks: Algorithms and Observations". In: *Proceedings of the Eleventh SIAM International Conference on Data Mining, SDM 2011, April 28-30, 2011, Mesa, Arizona, USA*. SIAM / Omnipress, 2011, pp. 119–130. DOI: 10.1137/1.9781611972818.11.

157. M. Kas, K. M. Carley, and L. R. Carley. "Incremental closeness centrality for dynamically changing social networks". In: *Advances in Social Networks Analysis and Mining 2013, ASONAM '13, Niagara, ON, Canada - August 25 - 29, 2013*. ACM, 2013, pp. 1250–1258. DOI: 10.1145/2492517.2500270.

158. M. J. Kashyop and N. S. Narayanaswamy. "Lazy or eager dynamic matching may not be fast". *Inf. Process. Lett.* 162, 2020, p. 105982. DOI: 10.1016/j.ipl.2020.105982.

159. L. Katz. "A new status index derived from sociometric analysis". *Psychometrika* 18:1, 1953, pp. 39–43.

160. J. Kazius, R. McGuire, and R. Bursi. "Derivation and validation of toxicophores for mutagenicity prediction". *Journal of medicinal chemistry* 48:1, 2005, pp. 312–320.

161. J. A. Kelner, L. Orecchia, A. Sidford, and Z. A. Zhu. "A simple, combinatorial algorithm for solving SDD systems in nearly-linear time". In: *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*. ACM, 2013, pp. 911–920. DOI: 10.1145/2488608.2488724.

162. D. Kempe, J. M. Kleinberg, and É. Tardos. "Maximizing the Spread of Influence through a Social Network". *Theory Comput.* 11, 2015, pp. 105–147. DOI: 10.4086/toc.2015.v011a004.

163. A. M. Khan, A. Pothen, M. M. A. Patwary, N. R. Satish, N. Sundaram, F. Manne, M. Halappanavar, and P. Dubey. "Efficient Approximation Algorithms for Weighted b-Matching". *SIAM J. Sci. Comput.* 38:5, 2016. DOI: 10.1137/15M1026304.

164. S. S. Khopkar, R. Nagi, A. G. Nikolaev, and V. Bhembre. "Efficient algorithms for incremental all pairs shortest paths, closeness and betweenness in social network analysis". *Soc. Netw. Anal. Min.* 4:1, 2014, p. 220. DOI: 10.1007/s13278-014-0220-6.

165. F. Khorasani, R. Gupta, and L. N. Bhuyan. "Scalable SIMD-Efficient Graph Processing on GPUs". In: *2015 International Conference on Parallel Architectures and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*. IEEE Computer Society, 2015, pp. 39–50. DOI: 10.1109/PACT.2015.15.

166. T. N. Kipf and M. Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

167. D. J. Klein and M. Randić. "Resistance distance". *Journal of mathematical chemistry* 12:1, 1993, pp. 81–95.

168. V. Korenwein, A. Nichterlein, R. Niedermeier, and P. Zschoche. "Data Reduction for Maximum Matching on Real-World Graphs: Theory and Experiments". In: *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 53:1–53:13. DOI: 10.4230/LIPIcs.ESA.2018.53.

169. I. Koutis, G. L. Miller, and R. Peng. "A Nearly-m log n Time Solver for SDD Linear Systems". In: *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*. IEEE Computer Society, 2011, pp. 590–598. DOI: 10.1109/FOCS.2011.85.

170. I. Koutis, G. L. Miller, and R. Peng. "Approaching Optimality for Solving SDD Linear Systems". *SIAM J. Comput.* 43:1, 2014, pp. 337–354. DOI: 10.1137/110845914.

171. I. Koutis, G. L. Miller, and D. Tolliver. "Combinatorial Preconditioners and Multilevel Solvers for Problems in Computer Vision and Image Processing". In: *Advances in Visual Computing, 5th International Symposium, ISVC 2009, Las Vegas, NV, USA, November 30 - December 2, 2009, Proceedings, Part I*. Springer, 2009, pp. 1067–1078. DOI: 10.1007/978-3-642-10331-5_99.

172. E. Kujansuu, T. Lindberg, and E. Mäkinen. "The stable roommates problem and chess tournament pairings". *Divulgaciones Matemáticas* 7:1, 1999, pp. 19–28.

173. J. Kunegis. "Konect: the koblenz network collection". In: *Proceedings of the 22nd international conference on world wide web*. 2013, pp. 1343–1350.

174. R. Kyng, Y. T. Lee, R. Peng, S. Sachdeva, and D. A. Spielman. "Sparsified Cholesky and multigrid solvers for connection laplacians". In: *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*. ACM, 2016, pp. 842–850. DOI: 10.1145/2897518.2897640.

175. R. Kyng and S. Sachdeva. "Approximate Gaussian Elimination for Laplacians - Fast, Sparse, and Simple". In: *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*. IEEE Computer Society, 2016, pp. 573–582. DOI: 10.1109/FOCS.2016.68.

176. C. Laitang, K. Pinel-Sauvagnat, and M. Boughanem. "DTD Based Costs for Tree-Edit Distance in Structured Information Retrieval". In: *Advances in Information Retrieval - 35th European Conference on IR Research, ECIR 2013, Moscow, Russia, March 24-27, 2013. Proceedings*. Springer, 2013, pp. 158–170. DOI: 10.1007/978-3-642-36973-5_14.

177. R. Lambiotte, V. D. Blondel, C. De Kerchove, E. Huens, C. Prieur, Z. Smoreda, and P. Van Dooren. "Geographical dispersal of mobile communication networks". *Physica A: Statistical Mechanics and its Applications* 387:21, 2008, pp. 5317–5325.

178. E. L. Lawler. *Combinatorial optimization: networks and matroids*. Courier Corporation, 2001.

179. J. Lee, V. S. Mirrokni, V. Nagarajan, and M. Sviridenko. "Maximizing Nonmonotone Submodular Functions under Matroid or Knapsack Constraints". *SIAM J. Discret. Math.* 23:4, 2010, pp. 2053–2078. DOI: 10.1137/090750020.

180. M. Lee and C. Chung. "Finding k-highest betweenness centrality vertices in graphs". In: *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*. ACM, 2014, pp. 339–340. DOI: 10.1145/2567948.2577358.

181. J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. 2014.

182. L. Leydesdorff. "Betweenness centrality as an indicator of the interdisciplinarity of scientific journals". *J. Assoc. Inf. Sci. Technol.* 58:9, 2007, pp. 1303–1319. DOI: 10.1002/asi.20614.

183. H. Li, R. Peng, L. Shan, Y. Yi, and Z. Zhang. "Current Flow Group Closeness Centrality for Complex Networks?" In: *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 2019, pp. 961–971. DOI: 10.1145/3308558.3313490.

184. H. Li and Z. Zhang. "Kirchhoff Index as a Measure of Edge Centrality in Weighted Networks: Nearly Linear Time Algorithms". In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. SIAM, 2018, pp. 2377–2396. DOI: 10.1137/1.9781611975031.153.

185. Y. Li, J. Luo, C. Chow, K. Chan, Y. Ding, and F. Zhang. "Growing the charging station network for electric vehicles with trajectory data analytics". In: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1376–1387. DOI: 10.1109/ICDE.2015.7113384.

186. Y.-s. Lim, D. S. Menasché, B. Ribeiro, D. Towsley, and P. Basu. "Online estimating the k central nodes of a network". In: *2011 IEEE Network Science Workshop*. 2011, pp. 118–122.

187. L. Lin, C. Yang, J. C. Meza, J. Lu, L. Ying, and W. E. "SelInv - An Algorithm for Selected Inversion of a Sparse Symmetric Matrix". *ACM Trans. Math. Softw.* 37:4, 2011, 40:1–40:19. DOI: 10.1145/1916461.1916464.

188. N. Lin. *Foundations of social research*. New York: McGraw-Hill, 1976.

189. R. J. Lipton and J. F. Naughton. "Estimating the Size of Generalized Transitive Closures". In: *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*. Morgan Kaufmann, 1989, pp. 165–171.

190. O. E. Livne and A. Brandt. "Lean Algebraic Multigrid (LAMG): Fast Graph Laplacian Linear Solver". *SIAM J. Sci. Comput.* 34:4, 2012. DOI: 10.1137/110843563.

191. M. von Looz, M. S. Özdayi, S. Laue, and H. Meyerhenke. "Generating massive complex networks with hyperbolic geometry faster in practice". In: *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*. IEEE, 2016, pp. 1–6. DOI: 10.1109/HPEC.2016.7761644.

192. L. Lovász. "Random walks on graphs". *Combinatorics, Paul erdos is eighty* 2:1-46, 1993, p. 4.

193. L. Lovász. "Submodular functions and convexity". In: *Mathematical Programming The State of the Art, XIth International Symposium on Mathematical Programming, Bonn, Germany, August 23-27, 1982*. Springer, 1982, pp. 235–257. DOI: 10.1007/978-3-642-68874-4_10.

194. L. Lovász and M. D. Plummer. *Matching theory*. American Mathematical Soc., 2009.

195. A. Lumsdaine, D. P. Gregor, B. Hendrickson, and J. W. Berry. "Challenges in Parallel Graph Processing". *Parallel Process. Lett.* 17:1, 2007, pp. 5–20. DOI: 10.1142/S0129626407002843.

196. Y. Luo, J. Wang, J. Chen, and S. Chen. "A Distributed Algorithm based on Probability for Refining Energy-Efficiency of Multicast Trees in Ad Hoc Networks". In: *30th Annual IEEE Conference on Local Computer Networks (LCN 2005), 15-17 November 2005, Sydney, Australia, Proceedings*. IEEE Computer Society, 2005, pp. 482–483. DOI: 10.1109/LCN.2005.2.

197. K. D. Mackenzie. "Structural centrality in communications networks". *Psychometrika* 31:1, 1966, pp. 17–25.

198. K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarría-Miranda. "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets". In: *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. IEEE, 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5161100.

199. C. Magnien, M. Latapy, and M. Habib. "Fast computation of empirically tight bounds for the diameter of massive graphs". *ACM J. Exp. Algorithmics* 13, 2008. DOI: 10.1145/1412228.1455266.

200. A. Mahmoody, C. E. Tsourakakis, and E. Upfal. "Scalable Betweenness Centrality Maximization via Sampling". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. ACM, 2016, pp. 1765–1773. DOI: 10.1145/2939672.2939869.

201. D. F. Manlove. "Algorithmics of Matching Under Preferences". *Bull. EATCS* 112, 2014.

202. F. Manne and R. H. Bisseling. "A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem". In: *Parallel Processing and Applied Mathematics, 7th International Conference, PPAM 2007, Gdansk, Poland, September 9-12, 2007, Revised Selected Papers*. Springer, 2007, pp. 708–717. DOI: 10.1007/978-3-540-68111-3_74.

203. F. Manne and M. Halappanavar. "New Effective Multithreaded Matching Algorithms". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. IEEE Computer Society, 2014, pp. 519–528. DOI: 10.1109/IPDPS.2014.61.

204. M. Marchiori and V. Latora. "Harmony in the small-world". *Physica A: Statistical Mechanics and its Applications* 285:3-4, 2000, pp. 539–546.

205. J. Matta, G. Ercal, and K. Sinha. "Comparing the speed and accuracy of approaches to betweenness centrality approximation". *Computational Social Networks* 6:1, 2019, pp. 1–30.

206. J. Maue and P. Sanders. "Engineering Algorithms for Approximate Weighted Matching". In: *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*. Springer, 2007, pp. 242–255. DOI: 10.1007/978-3-540-72845-0_19.

207. C. Mavroforakis, R. Garcia-Lebron, I. Koutis, and E. Terzi. "Spanning Edge Centrality: Large-scale Computation and Applications". In: *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*. ACM, 2015, pp. 732–742. DOI: 10.1145/2736277.2741125.

208. A. McGregor. "Finding Graph Matchings in Data Streams". In: *Approximation, Randomization and Combinatorial Optimization, Algorithms and Techniques, 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2005 and 9th InternationalWorkshop on Randomization and Computation, RANDOM 2005, Berkeley, CA, USA, August 22-24, 2005, Proceedings*. Springer, 2005, pp. 170–181. DOI: 10.1007/11538462_15.

209. P. E. McKenney and J. D. Slingwine. "Read-copy update: Using execution history to solve concurrency problems". In: *Parallel and Distributed Computing and Systems*. 1998.

210. A. McLaughlin and D. A. Bader. "Scalable and High Performance Betweenness Centrality on the GPU". In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*. IEEE Computer Society, 2014, pp. 572–583. DOI: 10.1109/SC.2014.52.

211. A. Mehta, A. Saberi, U. V. Vazirani, and V. V. Vazirani. "AdWords and Generalized On-line Matching". In: *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*. IEEE Computer Society, 2005, pp. 264–273. DOI: 10.1109/SFCS.2005.12.

212. E. L. Merrer, N. L. Scouarnec, and G. Trédan. "Heuristical top-k: fast estimation of centralities in complex networks". *Inf. Process. Lett.* 114:8, 2014, pp. 432–436. DOI: 10.1016/j.ipl.2014.03.006.

213. R. Merris. "Doubly stochastic graph matrices". *Publikacije Elektrotehničkog fakulteta. Serija Matematika* 8, 1997, pp. 64–71.

214. R. Merris. "Doubly stochastic graph matrices, II". *Linear and Multilinear Algebra* 45:2-3, 1998, pp. 275–285.

215. S. Micali and V. V. Vazirani. "An O(sqrt(|v|) |E|) Algorithm for Finding Maximum Matching in General Graphs". In: *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*. IEEE Computer Society, 1980, pp. 17–27. DOI: 10.1109/SFCS.1980.12.

216. M. M. Michael. "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects". *IEEE Trans. Parallel Distributed Syst.* 15:6, 2004, pp. 491–504. DOI: 10.1109/TPDS.2004.8.

217. M. Minoux. "Accelerated greedy algorithms for maximizing submodular set functions". In: *Optimization techniques*. Springer, 1978, pp. 234–243.

218. B. Mirzasoleiman, A. Badanidiyuru, A. Karbasi, J. Vondrák, and A. Krause. "Lazier Than Lazy Greedy". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. AAAI Press, 2015, pp. 1812–1818.

219. B. Monien, R. Preis, and R. Diekmann. "Quality matching and local improvement for multilevel graph-partitioning". *Parallel Comput.* 26:12, 2000, pp. 1609–1634. DOI: 10.1016/S0167-8191(00)00049-1.

220. J. L. Moreno. "Who shall survive?: A new approach to the problem of human interrelations.", 1934.

221. M. Mucha and P. Sankowski. "Maximum Matchings in Planar Graphs via Gaussian Elimination". *Algorithmica* 45:1, 2006, pp. 3–20. DOI: 10.1007/s00453-005-1187-5.

222. S. Mumtaz and X. Wang. "Identifying Top-K Influential Nodes in Networks". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*. ACM, 2017, pp. 2219–2222. DOI: 10.1145/3132847.3133126.

223. R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. "Introducing the graph 500". *Cray Users Group (CUG)* 19, 2010, pp. 45–74.

224. O. Narayan and I. Saniee. "Scaling of Random Walk Betweenness in Networks". In: *Complex Networks and Their Applications VII - Volume 1 Proceedings The 7th International Conference on Complex Networks and Their Applications COMPLEX NETWORKS 2018, Cambridge, UK, December 11-13, 2018*. Springer, 2018, pp. 41–51. DOI: 10.1007/978-3-030-05411-3_4.

225. O. Neiman and S. Solomon. "Simple Deterministic Algorithms for Fully Dynamic Maximal Matching". *ACM Trans. Algorithms* 12:1, 2016, 7:1–7:15. DOI: 10.1145/2700206.

226. G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. "An analysis of approximations for maximizing submodular set functions - I". *Math. Program.* 14:1, 1978, pp. 265–294. DOI: 10.1007/BF01588971.

227. M. Newman. *Networks*. Oxford university press, 2018.

228. P. Ni, M. Hanai, W. J. Tan, and W. Cai. "Efficient closeness centrality computation in time-evolving graphs". In: *ASONAM '19: International Conference on Advances in Social Networks Analysis and Mining, Vancouver, British Columbia, Canada, 27-30 August, 2019*. ACM, 2019, pp. 378–385. DOI: 10.1145/3341161.3342865.

229. K. Okamoto, W. Chen, and X. Li. "Ranking of Closeness Centrality for Large-Scale Social Networks". In: *Frontiers in Algorithmics, Second Annual International Workshop, FAW 2008, Changsha, China, June 19-21, 2008, Proceeedings*. Springer, 2008, pp. 186–195. DOI: 10.1007/978-3-540-69311-6_21.

230. H. Oktay, A. S. Balkir, I. Foster, and D. D. Jensen. "Distance estimation for very large networks using mapreduce and network structure indices". In: *Workshop on Information Networks*. 2011.

231. P. W. Olsen, A. G. Labouseur, and J. Hwang. "Efficient top-k closeness centrality search". In: *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. IEEE Computer Society, 2014, pp. 196–207. DOI: 10.1109/ICDE.2014.6816651.

232. K. Onak and R. Rubinfeld. "Maintaining a large matching and a small vertex cover". In: *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*. ACM, 2010, pp. 457–464. DOI: 10.1145/1806689.1806753.

233. E. Otte and R. Rousseau. "Social network analysis: a powerful strategy, also for the information sciences". *J. Inf. Sci.* 28:6, 2002, pp. 441–453. DOI: 10.1177/016555150202800601.

234. M. E. O'Neill. "PCG: A family of simple fast space-efficient statistically good algorithms for random number generation". *ACM Transactions on Mathematical Software*, 2014.

235. L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank citation ranking: Bringing order to the web.* Technical report. Stanford InfoLab, 1999.

236. R. Pastor-Satorras and A. Vespignani. "Immunization of complex networks". *Physical review E* 65:3, 2002, p. 036104.

237. B. Patt-Shamir, D. Rawitz, and G. Scalosub. "Distributed approximation of cellular coverage". *J. Parallel Distributed Comput.* 72:3, 2012, pp. 402–408. DOI: 10.1016/j.jpdc.2011.12.003.

238. A. Paz and G. Schwartzman. "A $(2+\varepsilon)$-Approximation for Maximum Weight Matching in the Semi-streaming Model". *ACM Trans. Algorithms* 15:2, 2019, 18:1–18:15. DOI: 10.1145/3274668.

239. R. Peng and D. A. Spielman. "An efficient parallel solver for SDD linear systems". In: *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*. ACM, 2014, pp. 333–342. DOI: 10.1145/2591796.2591832.

240. O. Perron. "Zur theorie der matrices". *Mathematische Annalen* 64:2, 1907, pp. 248–263.

241. F. R. Pitts. "A graph theoretic approach to historical geography". *The professional geographer* 17:5, 1965, pp. 15–20.

242. R. Preis. "Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs". In: *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings*. Springer, 1999, pp. 259–269. DOI: 10.1007/3-540-49116-3_24.

243. F. J. Provost, D. D. Jensen, and T. Oates. "Efficient Progressive Sampling". In: *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 15-18, 1999*. ACM, 1999, pp. 23–32. DOI: 10.1145/312129.312188.

244. C. C. Pugh and C. Pugh. *Real mathematical analysis*. Springer, 2002.

245. K. L. Putman, H. D. Boekhout, and F. W. Takes. "Fast incremental computation of harmonic closeness centrality in directed weighted networks". In: *ASONAM '19: International Conference on Advances in Social Networks Analysis and Mining, Vancouver, British Columbia, Canada, 27-30 August, 2019*. ACM, 2019, pp. 1018–1025. DOI: 10.1145/3341161.3344829.

246. R Puzis, Y Elovici, and S Dolev. "Fast algorithm for successive group betweenness centrality computation". *Physical Review E* 76, 2007, p. 056709.

247. R. Puzis, Y. Altshuler, Y. Elovici, S. Bekhor, Y. Shiftan, and A. Pentland. "Augmented Betweenness Centrality for Environmentally Aware Traffic Monitoring in Transportation Networks". *J. Intell. Transp. Syst.* 17:1, 2013, pp. 91–105. DOI: 10.1080/15472450.2012.716663.

248. G. Ramalingam and T. W. Reps. "On the Computational Complexity of Dynamic Graph Problems". *Theor. Comput. Sci.* 158:1&2, 1996, pp. 233–277. DOI: 10.1016/0304-3975(95)00079-8.

249. G. Ranjan, Z. Zhang, and D. Boley. "Incremental Computation of Pseudo-Inverse of Laplacian". In: *Combinatorial Optimization and Applications - 8th International Conference, COCOA 2014, Wailea, Maui, HI, USA, December 19-21, 2014, Proceedings*. Springer, 2014, pp. 729–749. DOI: 10.1007/978-3-319-12691-3_54.

250. K. Riesen and H. Bunke. "IAM Graph Database Repository for Graph Based Pattern Recognition and Machine Learning". In: *Structural, Syntactic, and Statistical Pattern Recognition, Joint IAPR International Workshop, SSPR & SPR 2008, Orlando, USA, December 4-6, 2008. Proceedings*. Springer, 2008, pp. 287–297. DOI: 10.1007/978-3-540-89689-0_33.

251. M. Riondato and E. M. Kornaropoulos. "Fast approximation of betweenness centrality through sampling". *Data Min. Knowl. Discov.* 30:2, 2016, pp. 438–475. DOI: 10.1007/s10618-015-0423-0.

252. M. Riondato and E. Upfal. "ABRA: Approximating Betweenness Centrality in Static and Dynamic Graphs with Rademacher Averages". *ACM Trans. Knowl. Discov. Data* 12:5, 2018, 61:1–61:38. DOI: 10.1145/3208351.

253. Y. Rochat. *Closeness centrality extended to unconnected graphs: The harmonic centrality index*. Technical report. 2009.

254. R. A. Rossi and N. K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. AAAI Press, 2015, pp. 4292–4293.

255. P. Sanders. "Algorithm Engineering - An Attempt at a Definition". In: *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*. Springer, 2009, pp. 321–340. DOI: 10.1007/978-3-642-03456-5_22.

256. P. Sankowski. "Faster dynamic matchings and vertex connectivity". In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*. SIAM, 2007, pp. 118–126.

257. P. Sankowski. "Maximum weight bipartite matching in matrix multiplication time". *Theor. Comput. Sci.* 410:44, 2009, pp. 4480–4488. DOI: 10.1016/j.tcs.2009.07.028.

258. E. E. Santos, J. Korah, V. Murugappan, and S. Subramanian. "Efficient Anytime Anywhere Algorithms for Closeness Centrality in Large and Dynamic Graphs". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016, pp. 1821–1830. DOI: 10.1109/IPDPSW.2016.215.

259. A. E. Sariyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek. "Incremental algorithms for closeness centrality". In: *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*. IEEE Computer Society, 2013, pp. 487–492. DOI: 10.1109/BigData.2013.6691611.

260. A. E. Sariyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek. "Incremental Algorithms for Network Management and Analysis based on Closeness Centrality". *CoRR* abs/1303.0422, 2013.

261. A. E. Sariyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. "Incremental closeness centrality in distributed memory". *Parallel Comput.* 47, 2015, pp. 3–18. DOI: 10.1016/j.parco.2015.01.003.

262. A. Schild. "An almost-linear time algorithm for uniform random spanning tree generation". In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*. ACM, 2018, pp. 214–227. DOI: 10.1145/3188745.3188852.

263. I. Schomburg, A. Chang, C. Ebeling, M. Gremse, C. Heldt, G. Huhn, and D. Schomburg. "BRENDA, the enzyme database: updates and major new developments". *Nucleic Acids Res.* 32:Database-Issue, 2004, pp. 431–433. DOI: 10.1093/nar/gkh081.

264. P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad. "Collective classification in network data". *AI magazine* 29:3, 2008, pp. 93–93.

265. O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann. "Pitfalls of Graph Neural Network Evaluation". *CoRR* abs/1811.05868, 2018.

266. J. Sherman and W. J. Morrison. "Adjustment of an inverse matrix corresponding to a change in one element of a given matrix". *The Annals of Mathematical Statistics* 21:1, 1950, pp. 124–127.

267. A. Shoshan and U. Zwick. "All Pairs Shortest Paths in Undirected Graphs with Integer Weights". In: *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. IEEE Computer Society, 1999, pp. 605–615. DOI: 10.1109/SFFCS.1999.814635.

268. R. B. Sidje and Y. Saad. "Rational approximation to the Fermi-Dirac function with applications in density functional theory". *Numer. Algorithms* 56:3, 2011, pp. 455–479. DOI: 10.1007/s11075-010-9397-6.

269. R. R. Singh. *Centrality Measures: A Tool to Identify Key Actors in Social Networks*. Ed. by A. Biswas, R. Patgiri, and B. Biswas. Springer Singapore, 2022. DOI: 10.1007/978-981-16-3398-0_1.

270. S. Solomon. "Fully Dynamic Maximal Matching in Constant Update Time". In: *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*. IEEE Computer Society, 2016, pp. 325–334. DOI: 10.1109/FOCS.2016.43.

271. C. Song, S. Havlin, and H. A. Makse. "Self-similarity of complex networks". *Nature* 433:7024, 2005, pp. 392–395.

272. D. A. Spielman and N. Srivastava. "Graph Sparsification by Effective Resistances". *SIAM J. Comput.* 40:6, 2011, pp. 1913–1926. DOI: 10.1137/080734029.

273. C. L. Staudt, A. Sazonovs, and H. Meyerhenke. "NetworKit: A tool suite for large-scale complex network analysis". *Netw. Sci.* 4:4, 2016, pp. 508–530. DOI: 10.1017/nws.2016.20.

274. K. Stephenson and M. Zelen. "Rethinking centrality: Methods and examples". *Social networks* 11:1, 1989, pp. 1–37.

275. D. Stubbs and V. V. Williams. "Metatheorems for Dynamic Weighted Matching". In: *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 58:1–58:14. DOI: 10.4230/LIPIcs.ITCS.2017.58.

276. L. Subelj and M. Bajec. "Robust network community detection using balanced propagation". *CoRR* abs/1106.5524, 2011.

277. J. Tang, T. Lou, and J. Kleinberg. "Inferring social ties across heterogenous networks". In: *Proceedings of the fifth ACM international conference on Web search and data mining*. 2012, pp. 743–752.

278. P. Thomas. "Semi-Supervised Learning by Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien (Review)". *IEEE Trans. Neural Networks* 20:3, 2009, p. 542. DOI: 10.1109/TNN.2009.2015974.

279. R. C. Trahair. *From Aristotelian to Reaganomics: A dictionary of eponyms with biographies in the social sciences*. Greenwood Publishing Group, 1994.

280. T. W. Valente, K. Coronges, C. Lakon, and E. Costenbader. "How correlated are network centrality measures?" *Connections (Toronto, Ont.)* 28:1, 2008, p. 16.

281. P. Van Mieghem, K. Devriendt, and H Cetinay. "Pseudoinverse of the Laplacian and best spreader node in a network". *Physical Review E* 96:3, 2017, p. 032311.

282. J. Vondrák. "Symmetry and Approximability of Submodular Maximization Problems". In: *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*. IEEE Computer Society, 2009, pp. 651–670. DOI: 10.1109/FOCS.2009.24.

283. Y. Wang, F. Makedon, J. Ford, and H. Huang. "A bipartite graph matching framework for finding correspondences between structural elements in two proteins". In: *The 26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. 2004, pp. 2972–2975.

284. D. B. Wilson. "Generating Random Spanning Trees More Quickly than the Cover Time". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*. ACM, 1996, pp. 296–303. DOI: 10.1145/237814.237880.

285. G. Yan, T. Zhou, B. Hu, Z.-Q. Fu, and B.-H. Wang. "Efficient routing on complex networks". *Physical Review E* 73:4, 2006, p. 046108.

286. P. Yanardag and S. V. N. Vishwanathan. "Deep Graph Kernels". In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*. ACM, 2015, pp. 1365–1374. DOI: 10.1145/2783258.2783417.

287. C. Yen, M. Yeh, and M. Chen. "An Efficient Approach to Updating Closeness Centrality and Average Path Length in Dynamic Networks". In: *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*. IEEE Computer Society, 2013, pp. 867–876. DOI: 10.1109/ICDM.2013.135.

288. Y. Yoshida. "Almost linear-time algorithms for adaptive betweenness centrality using hypergraph sketches". In: *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*. ACM, 2014, pp. 1416–1425. DOI: 10.1145/2623330.2623626.

289. W. W. Zachary. "An information flow model for conflict and fission in small groups". *Journal of anthropological research* 4, 1977, pp. 452–473.

290. J. Zhao, J. C. S. Lui, D. Towsley, and X. Guan. "Measuring and maximizing group closeness centrality over disk-resident graphs". In: *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*. ACM, 2014, pp. 689–694. DOI: 10.1145/2567948.2579356.

291. J. Zhao, P. Wang, J. C. S. Lui, D. Towsley, and X. Guan. "I/O-efficient calculation of H-group closeness centrality over disk-resident graphs". *Inf. Sci.* 400, 2017, pp. 105–128. DOI: 10.1016/j.ins.2017.03.017.

292. D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf. "Learning with Local and Global Consistency". In: *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*. MIT Press, 2003, pp. 321–328.

293. T. Zhu, B. Wang, B. Wu, and C. Zhu. "Maximizing the spread of influence ranking in social networks". *Inf. Sci.* 278, 2014, pp. 535–544. DOI: 10.1016/j.ins.2014.03.070.

294. H. Zhuang, Y. Sun, J. Tang, J. Zhang, and X. Sun. "Influence Maximization in Dynamic Social Networks". In: *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*. IEEE Computer Society, 2013, pp. 1313–1318. DOI: 10.1109/ICDM.2013.145.

295. Y. Zuo and K. Zhang. "Using Structural Features to Characterize Social Ties". In: *IEEE First International Conference on Data Science in Cyberspace, DSC 2016, Changsha, China, June 13-16, 2016*. IEEE Computer Society, 2016, pp. 235–242. DOI: 10.1109/DSC.2016.63.

296. U. Zwick. "All Pairs Lightest Shortest Paths". In: *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*. ACM, 1999, pp. 61–69. DOI: 10.1145/301250.301271.

297. U. Zwick. "All Pairs Shortest Paths in Weighted Directed Graphs ¾ Exact and Almost Exact Algorithms". In: *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*. IEEE Computer Society, 1998, pp. 310–319. DOI: 10.1109/SFCS.1998.743464.