ALGORITHMS FOR THE IDENTIFICATION OF CENTRAL NODES IN LARGE REAL-WORLD NETWORKS

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

ELISABETTA BERGAMINI

Tag der mündlichen Prüfung:	11. December 2017
erster Gutachter:	Prof. Dr. Henning Meyerhenke
zweiter Gutachter:	Prof. Dr. Pierluigi Crescenzi

ERKLÄRUNG

Ich versichere, diese Dissertation selbstständig angefertigt, alle benutzten Hilfsmittel vollständig angegeben, und kenntlich gemacht zu haben, was aus Arbeiten anderer und eigener Veröffentlichungen unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, 2017

Elisabetta Bergamini

ACKNOWLEDGEMENTS

First of all, I thank my advisor Henning Meyerhenke for offering me the opportunity to work in his group and for all the advice and support he gave me during the time of my PhD. Thank you also for your trust, for giving me the chance to develop my own research ideas and for always being willing to meet my needs. I would also like to thank Pierluigi Crescenzi for agreeing to act as a second referee and taking the time to read my thesis and travel to Karlsruhe. I am also very grateful to David Bader and Oded Green for their hospitality and advice during my research stay in Atlanta.

A big thank also goes to my past and present colleagues. In particular, I would like to thank Christian Staudt, Moritz von Looz and Roland Glantz, who accompanied me from the very beginning of my PhD. Thank you for your help and support, and for all the fun moments we had together. Thank you also to Charilaos (Harry/Efca) Tzovas for always being able to put everyone in a good mood (sometimes by threatening to eat them, but I guess the end justifies the means). Thank you also to Eugenio Angriman, Marcel Radermacher, Tamara Mchedlidze and all the other colleagues on the third floor for making lunch breaks always pleasant.

I also wish to thank all the (ex-)students I advised or collaborated with, in particular Kolja Esders, Maximilian Vogel, Michael Wegner, Patrick Bisenius and Arie Slobbe: thank you for all the great work you have done! My thanks also go to Matteo Riondato, for all the precious feedback he gave me.

I am also grateful to my coauthors Michele Borassi, Andrea Marino, Lorenzo Severini, Gianlorenzo D'Angelo and Yllka Velaj: I really enjoyed collaborating with you! Thank you also to all my colleagues from the DFG SPP "Algorithms for BIG DATA", in particular Mark Ortmann, David Veith, Mahyar Behdju, Manuel Penschuck, Michael Hamann, Jan Hackfeld, Miriam Schlöter and Marsha Kleinbauer: thank you for all the interesting discussions and fun we had at conferences, meetings and summer schools.

Special thanks also to Laurette Lauffer, Lilian Beckert and Ralf Kölmel for all their help, without them I would have been lost.

A huge thanks also to my dear friends Julia, Vincenzo, Gabriele, Flavia, Dimitra, Ali, Spiros, Sabrina and Oliver, for always supporting me and being there for me. Thank you also for your warmth and hospitality in the last months, you have been like a second family to me. I also thank Valeria and Martina for the great time we had during their stay in Karlsruhe and Eleonora, Letizia, Sara, Adele and Eric, for being good friends in spite of the distance. Thank you also to Ole for helping me improve my German!

I also wish to thank my parents Ilva and Giuseppe, my siblings and siblings-in-law Chiara, Enrico and Matteo for their support, and my two wonderful nephews Francesco and Riccardo for all the joy they have brought me.

Last but not least, a few words cannot describe how grateful I am to my boyfriend Enrico for his continuous support, for his patience and for his love... thank you!

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ZUSAMMENFASSUNG

Network Science ist ein wachsendes Forschungsfeld mit dem Ziel, nützliche Informationen aus Netzwerkdaten (d.h. Daten zu den Beziehungen zwischen Entitäten) zu gewinnen. Diese Art von Daten – die oft als Netzwerke bezeichnet werden – lasst sich auf natürliche Weise mit Graphen modellieren, sodass zahlreiche Erkenntnisse der Graphentheorie genutzt werden können, um Fragen zu den Daten zu beantworten.

Eine der häufigsten Aufgaben der Network Science ist die Identifizierung der besonders wichtigen Entitäten eines Netzwerks. Zum Beispiel könnte es Knoten oder Verbindungen geben, deren Entfernung zu einer größeren Störung oder deren Einfügung zu einer wesentlichen Änderung des Netzwerkflusses führt.

In den letzten Jahren wurde oft versucht, diesen anwendungsabhängigen Begriff von Wichtigkeit zu formalisieren. Auf diese Weise wurden mehrere *Zentralitätsmaße* (also Funktionen, die den Knoten oder Kanten eines Netzwerks einen Wert zuweisen, der ihre Wichtigkeit widerspiegelt) vorgeschlagen. Zentralitätsmaße kann man – je nachdem, ob sie den gesamten Graphen oder nur einen kleinen Teil den Knoten herum berücksichtigen – als lokal oder global klassifizieren. Lokale Zentralitätsmaße (wie z.B. der Grad eines Knotes) können typischerweise Knoten schlecht charakterisieren und komplexe Mechanismen, die das Netzwerk beinflussen, nicht aufdecken. Anderseits ist die Berechnung von globalen Zentralitätsmaßen in der Regel aufwändiger, insbesondere für viele heutige Netzwerke, die Millionen oder sogar Milliarden von Knoten haben.

Viele Netzwerke, die von Interesse sind – wie z.B. soziale Netzwerke und der Web-Graph – sind nicht nur sehr groß, sondern entwickeln sich auch kontinuierlich im Laufe der Zeit, indem neue Verbindungen geschaffen und alte gekappt werden. Eine einzige Analyse des Netzwerkes ist also möglicherweise nicht ausreichend, um die Dynamik des Systems zu verstehen. Dynamische Algorithmen hingegen erlauben es, die zur Zentralitätsberechnung verwendeten Daten mehrmals zu aktualisieren, und dabei die Laufzeit in Grenzen zu halten.

Hauptziel dieser Arbeit ist es, skalierbare Algorithmen zu entwickeln, mit denen man immer noch relevante Informationen extrahieren kann. Zu den Techniken, die zu diesem Zweck eingesetzt wurden, gehören Näherungsverfahren, die Verwendung effizienter Datenstrukturen und die Entwicklung von Heuristiken, die bestehende Algorithmen effizienter machen.

Im Allgemeinen folgt diese Arbeit dem Paradigma des *Algorithm Engineering*, welches neben dem Entwurf und der theoretischen Analyse auch die Implementierung sowie die systematische experimentelle Evaluation der Verfahren berücksichtigt.

Die Arbeit lässt sich in vier Abschnitte gliedern. Alle betrachteten Probleme beziehen sich auf die effiziente Berechnung von Zentralitätsmaßen in großen und/oder dynamischen Netzwerken, jedoch variieren die angewandten Techniken je nach Zentralitätsmaß und Szenario. Im Folgenden werden kurz der Inhalt und die Beiträge jedes Teils dieser Arbeit zusammengefasst.

DYNAMISCHE ALGORITHMEN FÜR BETWEENNESS CENTRALITY. Die Betweenness Centrality ist ein bekanntes Zentralitätsmaß, das auf dem Zählen kürzester Wege

basiert. Für einen Knoten $v \in V$ wird angenommen, dass er umso zentraler liegt, je mehr er an möglichen Kommunikationswegen zwischen Paaren von Knoten beteiligt ist. Es werden alle kürzesten Wege zwischen Paaren von Knoten genutzt, daher braucht die Berechnung von Betweenness Centrality mindestens eine quadratische Laufzeit in der Anzahl der Knoten. Bei Netzwerken, die sich im Laufe der Zeit ändern, wäre eine Neuberechnung nach jeder Änderung zu teuer für große Eingaben.

Kapitel 5 beschreibt einen dynamischen Algorithmus, der die Betweenness von allen Knoten nach der Einfügung neuer Kanten aktualisiert. Unser Algorithmus iBet ist effizienter als zwei der besten bereits bestehenden Methoden [67, 72]; seine Laufzeit ist durchschnittlich um eine Größenordnung schneller.

Danach (Kapitel 6) wird das Problem der Aktualisierung eines einzelnen Knoten betrachtet. Obwohl ein dynamischer Algorithmus wie iBet für diese Aufgabe genutzt werden könnte, ist die Aktualisierung nur eines einzelnen Knotens auch effizienter möglich. Es wird daher für dieses Szenario ein dynamischer Algorithmus vorgeschlagen und durch Experimente verdeutlicht, dass die gewählte Methode bis zu zwei Größenordnungen schneller als iBet ist. Dieses Ergebnis unterscheidet sich vom statischen Fall, wo die Berechnung des Wertes eines einzelnen Knotens nicht wesentlich schneller ist, als die Berechnung aller Werte.

Die bisher vorgeschlagenen Algorithmen sind jeweils exakt und zielen auf Netzwerke mit bis zu Zehntausenden von Knoten und Kanten. Kapitel 7 beschreibt den ersten dynamischen Algorithmus, der eine Approximation von Betweenness aktualisiert. Unsere neue Methode basiert auf dem statischen Approximationsalgorithmus RK [107] und erbt seine Garantie auf die Approximationsgüte: Nach jeder Aktualisierung garantiert der Algorithmus, dass sich die approximierten Betweenness-Werte höchstens ϵ von den exakten Werten mit Wahrscheinlichkeit $1 - \delta$ oder höher unterscheiden, wobei ϵ und δ beliebig kleine Konstanten sind. Unsere Experimente zeigen, dass unsere Methode um mehrere Größenordnungen schneller als RK ist. Sie zeigen auch ein deutlich verbessertes Skalierungsverhalten im Vergleich zu exakten Ansätzen – z.B. kann unsere Methode in wenigen Sekunden Betweenness in einem Netzwerk mit 36 Millionen Kanten aktualisieren.

Einige der in diesem Teil vorgestellten Ergebnisse wurden als [22], [18], [16] und [17] veröffentlicht.

BERECHNUNG VON KNOTEN MIT HÖCHSTER CLOSENESS CENTRALITY. Die Closeness Centrality ist definiert als die Umkehrung der durchschnittlichen Graphdistanz zwischen einem Knoten und den anderen Knoten des Netzwerks. Um die Berechnung der Closeness für alle Knoten zu durchführen, muss man ein APSP (All-Pairs Shortest Path) lösen. Allerdings braucht man für viele Anwendungen nur die zentralsten Knoten, anstatt die Closeness aller Knoten zu finden.

In diesem Teil schlagen wir zunächst Algorithmen vor, die die k Knoten mit höchster Closeness in Netzwerken mit kleinen bzw. großen Durchmessern effizient finden (Kapitel 9). Die vorgeschlagenen Techniken berechnen verschiedene obere Schranken für die Closeness der Knoten und stoppen die Berechnung, wenn k Knoten gefunden wurden, deren Closeness höher ist als die obere Schranken der anderen Knoten. Mit unserem neuen Ansatz können wir z.B. die Top-10-Knoten mit der höchsten Closeness im gesamten nordamerikanischen Straßennetzwerk (mit 36 Millionen Kanten) in circa einer Stunde finden, während bestehende Methode wie z.B. [97] selbst nach Tagen nicht terminieren würden.

In Kapitel 10 betrachten wir dann das Problem der Aktualisierung der k Knoten mit höchster Closeness in dynamischen Netzwerken. Unsere Algorithmen, die auf den statischen Top-k-Algorithmen von Kapitel 9 basieren, verwenden Informationen von früheren Berechnungen, um unnötige Operationen zu vermeiden. Unsere experimentellen Ergebnisse zeigen, dass in vielen Fällen unsere dynamischen Algorithmen zwei Größenordnungen schneller sind als die statischen Algorithmen von Kapitel 9; bei einigen großen Graphen erreichen wir sogar durchschnittliche Speedups zwischen 10^3 und 10^4.

Kapitel 9 und Kapitel 10 zielen darauf ab, individuell zentrale Knoten zu finden. Allerdings kann es in einigen Anwendungen erforderlich sein, eine Gruppe von Knoten zu finden, die als Ganzes zentral sind. Group Closeness Maximization (GCM) zielt darauf ab, eine Gruppe von k Knoten mit der minimalen durchschnittlichen Distanz zu den anderen Knoten zu finden. In Kapitel 11 schlagen wir Techniken vor, um einen Greedy-Algorithmus [37] für GCM so abzuändern, dass seine Laufzeit skaliert, d.h. auch für große Eingaben vertretbar bleibt. Unsere Methode findet eine Gruppe G, so dass die Closeness $c(G)$ von G mindestens $(1 - 1/e)c(G^\star)$ ist, wobei $G^\star$ das Optimum ist. In Netzwerken mit bis zu hundert Millionen Kanten finden wir eine Lösung in Minuten, während der Greedy-Algorithmus in [37] für Netzwerke mit Hunderttausenden von Kanten mehrere Tage dauern würde.

Einige der in diesem Teil vorgestellten Ergebnisse wurden als [19], [24] und [14] veröffentlicht.

NÄHRUNG DER ELECTRICAL CLOSENESS. In diesem letzten Teil betrachten wir ein Zentralitätsmaß, das – ebenso wie die traditionelle Closeness – die umgekehrte durchschnittliche Distanz zu den anderen Knoten anzeigt. Jedoch bedeutet Distanz in diesem Fall *Resistance Distance* [33]. Die Resistance Distance zwischen zwei Knoten u und v ist die erwarteten Anzahl von Schritten, die ein Random Walk, der in u startet, benötigt, um v zu erreichen und dann zum ersten Mal zu u zurückzukehren (bis auf einen multiplikativen Faktor).

Leider ist die Berechnung der Electrical Closeness teuer: sie erfordert, eine $n \times n$-Matrix zu invertieren, wobei n die Anzahl der Knoten ist. In Kapitel 12 stellen wir zwei Approximationsalgorithmen vor, die auf der Lösung linearer Gleichungssysteme basieren und zu sehr genauen Ergebnissen in der Praxis führen. Mit ihnen können wir nun eine Schätzung der Electrical Closeness eines Knotens in Netzwerken mit zehn Millionen Knoten innerhalb von Sekunden oder wenigen Minuten berechnen. Darüber hinaus zeigen unsere Experimente, dass die Electrical Closeness zwischen den Knoten deutlich besser als die Kürzester-Pfad-Closeness unterscheiden kann (d.h. sie führt zu einem breiteren Intervall der Werte). Sie ist zudem auch wesentlich widerstandsfähiger gegen verrauschte Daten - auf diese Weise zeigen wir, dass zwei bekannte Nachteile der Kürzester-Pfad-Closeness von der elektrischen Variante gemildert werden.

Ergebnisse wurden als [20] veröffentlicht.

ABSTRACT

Network science is a growing field of study whose aim is essentially to discover useful information from network data, i.e. data on relations between entities. This kind of data – often referred to as networks – can be naturally modeled with graphs, making it possible to use the many existing results from graph theory to answer questions on the data.

Among these questions, one of the most natural is whether there are entities in the network that are particularly important. For instance, there might be nodes or connections whose removal leads to a disruption or whose insertion generates a major change in the network flow.

Recent years have seen several attempts to formalize this clearly application-dependent notion of importance. As a result, several *centrality measures* (i.e., functions that associate to the nodes or edges of a network a value indicating their importance) have been introduced. Depending on whether they take the whole graph into account or just a limited subset of it, one can distinguish between local and global centrality measures. Local centrality measures (such as the degree of nodes) are mostly intuitive and easy to compute, but they are typically unable to differentiate well and to reveal insights on more complex mechanisms affecting the network. On the other hand, computing global centrality measures is typically an expensive task, especially for many today's networks having millions or billions of connections.

In addition to being massive in size, many networks of interest – such as social networks and the Web graph – evolve continuously over time, creating new connections and deleting existing ones. A static analysis of a specific snapshot of the network might not be sufficient to understand the dynamics of the underlying system, raising the need for algorithms that can efficiently keep track of centrality in evolving graphs.

Achieving scalability while still being able to extract relevant information is the main goal of this work. Techniques employed to this end include approximation, efficient data structures and the use of heuristics to reduce redundant work in existing algorithms.

In general, the research presented in this work has been carried out following the *algorithm engineering* paradigm. Differently from purely theoretical analysis, algorithm engineering does not only focus on design and theoretical analysis, but also on implementation and systematic experimental evaluation.

This work is divided into three main parts. All considered problems are related to the efficient computation of centrality in large and/or evolving networks, but the used techniques vary considerably, depending on the targeted centrality measures and scenarios. In the following, we briefly summarize the content and contributions presented in each part.

DYNAMIC ALGORITHMS FOR BETWEENNESS CENTRALITY. Betweenness centrality is a popular measure which ranks nodes according to their participation in the shortest paths of the network. A node $v \in V$ is therefore considered to be central when it lies in many shortest paths between other pairs of nodes. Because it requires *all* shortest paths between node pairs, the computation of betweenness is at least quadric in the number of nodes. In networks that change over time, recomputing betweenness from scratch after each update would be too expensive, in particular for large instances.

In Chapter 5, we first propose an algorithm for updating the betweenness centrality of all nodes after edge insertions. Our algorithm builds on two existing dynamic approaches [67, 72] and improves on them by identifying and significantly reducing redundant operations. Our experiments show that the proposed approach, which we call iBet, improves the state of the art by about one order of magnitude on average.

We then consider in Chapter 6 the problem of updating the betweenness centrality of a *single node*. Although a dynamic algorithm such as iBet could be used for this task, we notice that the update of a single node can be carried out much more efficiently. We therefore propose a new dynamic algorithm for this scenario and compare it experimentally against iBet, showing considerably improved running times – of up to two orders of magnitude on the largest tested instances. Interestingly, this is in contrast to the static case, for which computing the betweenness of a single node requires all pairwise distances and is therefore not significantly faster than computing betweenness for all nodes.

Both algorithms proposed so far are exact and therefore target networks with up to tens of thousands of nodes and edges. Chapter 7 describes the first dynamic algorithm that updates an *approximation* of betweenness centrality. Our new method builds on the static approximation algorithm RK [107] and inherits its theoretical guarantee on the approximation quality: After each update, the approximated betweenness values differ by at most ϵ from the exact values with probability at least $1 - \delta$, where ϵ and δ can be arbitrarily small constants. Our experiments show that our new approach is orders of magnitude faster than RK and also has a much improved scaling behavior compared to exact approaches – e.g., we can update betweenness scores in a network with 36 million edges in a few seconds on typical workstation hardware.

Some of the results presented in this part have been published as [22], [18], [16] and [17].

COMPUTATION OF NODES WITH HIGHEST CLOSENESS CENTRALITY. Closeness centrality is defined as the inverse of the average shortest-path distance between one node and the other nodes of the network. Computing closeness for all nodes requires to solve an APSP (All-Pairs Shortest Path). However, since many applications are interested in finding the most-central nodes rather than the score of each node, a natural question is whether the top-k nodes with highest closeness can be found faster than computing closeness for all nodes.

In this part, we first propose algorithms that allow us to efficiently find the k nodes with highest closeness in networks with small and large diameters, respectively (Chapter 9). The proposed techniques compute different upper bounds on the closeness values and stop the computation when k nodes are found whose closeness is higher than the bounds of the other nodes. Using our new approach, for example, we are able to find the top-10 nodes with highest closeness in the whole street network of North America (with 36 millions edges) in about one hour, where existing methods such as [97] would not terminate in days.

We then consider in Chapter 10 the problem of keeping track of the k nodes with highest closeness centrality in dynamic networks. Our algorithms build on the static top-k closeness algorithms presented in Chapter 9 and use information obtained during earlier computations to omit unnecessary work. Our experimental results show that, on many instances, our dynamic algorithms are two orders of magnitude faster than recomputation; on some large graphs, we even reach average speedups between 10^3 and 10^4.

Chapter 9 and Chapter 10 aim at finding nodes that are individually central. However, for some applications it might be required to find a *group* of nodes that is central as a whole. Group Closeness Maximization (GCM) aims at finding a group of k nodes with the minimum average shortest path distance to the remaining nodes. We propose in Chapter 11 techniques for reducing the running time and memory requirements of an existing greedy algorithm [37] for GCM. Our method finds a solution G such that its closeness $c(G)$ is at least $(1 - 1/e)c(G^\star)$, where $G^\star$ is the group with maximum closeness. With our method we can now approximate the group with maximum closeness on networks with up to hundreds of millions of edges in minutes or at most a few hours, whereas a straightforward implementation of the algorithm in [37] would take several days already on networks with hundreds of thousands of edges.

Some of the results presented in this part have been published as [19], [24] and [14].

APPROXIMATION OF ELECTRICAL CLOSENESS. In this last part, we consider a centrality measure which, similarly to traditional closeness, indicates the inverse average distance to the other nodes. However, in this case distance means *resistance distance* [33]. The resistance distance between two nodes u and v is defined as the expected number of steps taken by a random walk starting in u to reach v and then return to u for the first time (up to a multiplicative factor).

Unfortunately, electrical closeness is an expensive measure: an exact computation requires to invert a $n \times n$ matrix, where n is the number of nodes. In Chapter 12, we present two approximation algorithms based on solving linear systems and leading to very accurate results in practice. Using them we are now able to compute an estimation of electrical closeness on networks with tens of millions of nodes within seconds or a few minutes. Also, our experiments indicate that electrical closeness can discriminate among nodes significantly better than traditional shortest-path closeness and is also considerably more resistant to noise – we thus show that two known drawbacks of shortest-path closeness are alleviated by the electrical variant.

The results presented in this part have been published as [20].

Part I

INTRODUCTION

1 MOTIVATION AND CONTRIBUTION

1.1 NETWORK ANALYSIS AND CENTRALITY MEASURES

Most complex systems, both artificial and living, are comprised of interacting components whose behavior is typically much simpler than the one of the overall system. It is often the case that a shift of perspective towards the interconnections between parts and the structure of these interconnections can reveal important insights about the underlying system. For example, a recent study performed on the Facebook social network has shown that a person's romantic partner can be identified with high accuracy, without relying on any data other than the connections between users [7].

Network science is an emerging discipline investigating the properties of complex systems based on their relational structure. Data indicating relations between entities or individuals are often referred to as *networks*. In addition to the romantic partnership example mentioned above, network models have been used to shed light on a variety of complex phenomena, including the causes of obesity, the diffusion of infectious diseases, the neural correlates of Alzheimer's disease and the propagation of viruses through the Internet [39, 99, 128], just to mention a few examples. One of the reasons behind the success of network models can be attributed to their very natural representation as graphs, mathematical objects consisting of a set of nodes and a set of edges connecting node pairs. This implies that the rich existing theory about graphs and their properties can be used to extract insights on the system under study.

Among others, a very common task in network analysis is the identification of the most "important" nodes of a network. For instance, there might be nodes or connections whose removal leads to a disruption or whose insertion generates a major change in the network flow. There might as well be elements from which information can spread quickly over the whole network and others that can be easily reached. The first attempts to formally define this notion of importance date back to the late 1940s, when Alex Bavelas and his research group at MIT conducted some experiments in the context of communication patterns and group collaboration [11]. The results suggested that centrality was related to group efficiency in problem-solving, and agreed with the subjects's perception of leadership.

In the following decades, several *centrality measures* (i.e., functions defined on the nodes or edges of a network indicating their importance) have been introduced and employed in a multitude of contexts. For example, it has been show experimentally in [5] that some centrality measures are most correlated to epidemic spreading, whereas others can better explain rumor dynamics. Other centrality measures have been used to understand the wealth of families in the 15th-century Florence based on marriages and financial transactions [98], and others to explain the dominance of early Moscow based on the network of trade in the 12th and 13th century [100]. In social network analysis, centrality measures are used to identify important or influential actors and search engines employ centrality to rank web pages [32].

Depending on their scope in the graph, one can distinguish between local and global centrality measures. For instance, the number of connections of a node is a local measure, since the information it encodes is only based on a small subset of the network. On the contrary, global measures take the whole graph into account, and a change in a single edge or node could drastically influence them. Local centrality measures are mostly intuitive and easy to compute, but they are typically unable to differentiate well among nodes (or edges) and to reveal insights on more complex mechanisms affecting the network. On the other hand, since the score of each node or edge depends on the whole graph, computing global centrality measures is usually an expensive task, typically at least quadratic in the size of the graph. Phenomena such as the diffusion of social media and the considerably-increased computing power of modern machines made the size of available data increase drastically. As a result, quadratic algorithms are out of reach for many today's networks, often containing millions or even billions of nodes and edges. In most cases, only algorithms with a complexity linear in the size of the graph take a reasonable amount of time to complete. Also, many networks of interest evolve continuously over time, creating new connections and deleting existing ones. For example, every second, several thousands of tweets and millions of emails are sent.[1] For such networks, a static analysis of a specific snapshot might not be sufficient to understand the dynamics of the underlying system, raising the need for scalable algorithms that can efficiently recompute the required information after a change occurs.

1.2 OBJECTIVES AND METHODOLOGY

Achieving *scalability* while still being able to extract relevant information is the main goal of this work. With the term scalability we mean the capability of effectively applying algorithms to the growing volume of data arising in practice. It does not refer in this context to parallel scalability – i.e., using large numbers of processing elements, although in some cases we also use parallelism to further decrease the running time of our algorithms.

All contributions presented in this work aim at improving scalability of existing approaches for different subproblems in the area of centrality computation. Our results are mostly obtained following one of two paths: *algorithmic improvements* to existing algorithms – leading to an exact solution in a shorter time, or *approximation* – yielding inexact but still qualitatively similar results. In the latter case, we make sure that the results returned by the algorithm are meaningful by either comparing them agains exact results on a variety of real-world instances or by proving theoretical guarantees on the approximation quality (or, in most cases, both).

An example of algorithmic improvement is the work presented in Chapter 9, whose goal is to find the k nodes with highest closeness centrality (a centrality measure ranking nodes according to their average shortest-path distance to the other nodes). In this case, we exploit properties of real-world networks to compute quick upper bounds on the centrality of nodes and skip nodes that cannot be among the top k. We show experimentally that the approach is orders of magnitude faster than the textbook algorithm, while still finding exactly the same solution. Also dynamic algorithms (such as the ones presented in Chapter 5 and Chapter 10) return exactly the same result as recomputation on the modified graph. In this case, properties of the graph before the update are used to skip unnecessary work.

1 Source: www.internetlivestats.com

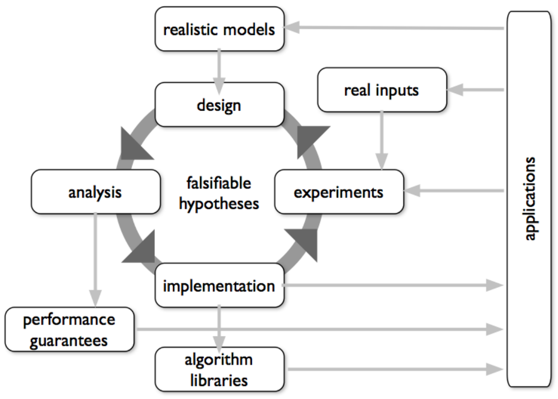


Figure 1: Algorithm engineering cyclic process [111].

We followed the path of approximation in Chapter 12 (among others), where we study the problem of computing electrical closeness (a variant of closeness taking all paths into account and not only the shortest ones). Since an exact computation would be prohibitive on networks with more than a few thousands of nodes, we propose two inexact approaches for which we show experimentally that the returned ranking is very similar to the exact one. Nevertheless, approximation is orders of magnitude faster than an exact approach, computing electrical closeness of a node on networks with millions of nodes in seconds or minutes.

In general, the research presented in this work has been carried out trying to follow the *algorithm engineering* paradigm. Algorithm engineering can be seen as an iterative process cycling through the stages of algorithm design, analysis, implementation and experimental evaluation, guided by real-world computational problems [111] (see Figure 1). In this view, the implementation and experimental study of algorithms are not left to practitioners, but are part of the process of algorithm development. Thus, all algorithms presented in this work have been implemented and thoroughly tested on a variety of real-world instances. Our code is implemented in C++ building on NetworKit [119], an open-source framework for large-scale network analysis. The carefully-chosen NetworKit data structures provide us a convenient basis for writing efficient code. Also, implementing *all* algorithms in NetworKit, we make sure that our comparisons are unbiased.

Another fundamental aspect of algorithm engineering is the reproducibility of results, so that other researchers can either draw the same conclusions by repeating experiments, or disprove previous theories. In this respect, we always use data sets from publicly available collections and make our algorithms available as part of NetworKit (for the most recent results, we plan to make them available in the near future).

1.3 OUTLINE AND CONTRIBUTION

This work is divided into four parts. The first one introduces the targeted problems and the used techniques, and the following three present original contributions in different subtopics. All considered problems are related to the efficient computation of centrality in large and/or evolving networks, but the used techniques vary considerably, depending on the targeted centrality measures and scenarios. The second part focuses on algorithms for updating a popular centrality measure called *betweenness* in dynamic networks, both exactly and using approximation. In the third part, scalable algorithms for finding the k most-central nodes in static and dynamic networks are presented, where here "central" refers to a measure called closeness centrality. Both betweenness and closeness centrality build on the assumption that information flows through the network following shortest paths. Since in some applications it might make sense to also consider alternative slightly-longer paths, in the fourth part we consider an additional measure called electrical closeness. Electrical closeness takes all paths of the network into account (weighted based on their length), but is expensive to compute exactly. Thus, we propose accurate and scalable algorithms for approximating it.

INTRODUCTION. The introduction is composed of three chapters, the first of which motivates the problems considered in this work and outlines our contributions. Chapter 2 presents basic notation used throughout the thesis and gives an overview of some of the most well-known centrality measures. Since several algorithms presented in this thesis update shortest-path based centrality measures in dynamic graphs, in Chapter 3 we first introduce some notation related to dynamic graphs, and then give motivation for developing dynamic algorithms for real-world evolving networks.

DYNAMIC ALGORITHMS FOR BETWEENNESS CENTRALITY. Betweenness centrality is a popular measure which ranks nodes according to their participation in the shortest paths of the network. Nodes with high betweenness can be important in routing, spreading processes and mediation of interactions. More precisely, the betweenness centrality of a node v is defined as the fraction of shortest paths between other node pairs that go through v. Because it requires *all* shortest paths between node pairs, the computation of betweenness is at least quadric in the number of nodes. In particular, with Brandes's algorithm [31], it can be computed in $\Theta(nm)$ time in unweighted graphs and $\Theta(nm + n^2 \log n)$ time in weighted graphs, where n is the number of nodes and m is the number of edges. In networks that change over time, recomputing betweenness from scratch after each update would be too expensive. After an introduction of betweenness and related work in Chapter 4, we propose in Chapter 5 an algorithm for updating the betweenness centrality of all nodes after edge insertions. Our algorithm builds on two of the best existing dynamic approaches [67, 72] and improves on them by identifying and significantly reducing redundant operations. Our experiments show that the proposed approach, which we call iBet, improves the state of the art by about one order of magnitude on average.

We then consider in Chapter 6 the problem of updating the betweenness centrality of a *single node*, motivated – among other applications – by the maximum betweenness improvement problem (MBI) [42]. MBI is based on the assumption that having a high centrality can be beneficial for a node, and is defined as the problem of choosing a set of k edges to be added to the graph such that they maximize the betweenness of a given node v. The best known approximation algorithm for MBI (developed by Crescenzi et al. [42]) requires to repeatedly add edges to the graph and compute the betweenness of v in the modified graphs. Although a dynamic algorithm such as iBet could be used for this task, we notice that the update of a single node can be carried out much more efficiently. We therefore propose a new dynamic algorithm for this scenario and compare it experimentally against iBet, showing considerably improved running times – of up to two orders of magnitude on the largest tested instances. Preliminary results show that the difference is even higher if we compare our new approach for a single node against algorithms existing before iBet [67, 72] – between two and three orders of magnitude on average. Interestingly, this is in contrast to the static case, for which computing the betweenness of a single node requires all pairwise distances and is therefore not significantly faster than computing betweenness for all nodes.

Both algorithms proposed so far are exact and can target networks with up to tens of thousands of nodes and edges. For larger instances, we propose in Chapter 7 a dynamic algorithm that updates an *approximation* of betweenness centrality. Our approach builds on the static approximation algorithm by Riondato and Kornaropoulos [107], and inherits its theoretical guarantee. After each update, the algorithm guarantees that the approximated betweenness values differ by at most ϵ from the exact values with probability at least $1 - \delta$, where ϵ and δ can be arbitrarily small constants. Our experimental study shows that our algorithms are the first to make in-memory computation of a betweenness ranking practical for large dynamic networks. Using approximation, we achieve a much improved scaling behavior compared to exact approaches, enabling us to update betweenness scores in a network with 36 million edges in a few seconds on typical workstation hardware. Our experiments also show that the measured absolute errors are always lower than the guaranteed ones. For nodes with high betweenness, also the rank of nodes is well preserved, even for relatively large values of ϵ.

The results presented in Chapter 5 have been published as "Faster betweenness centrality updates in evolving networks" (coauthored with Henning Meyerhenke, Mark Ortmann, and Arie Slobbe) at the *Sixteenth International Symposium on Experimental Algorithms* (SEA 2017). Chapter 6 is based on joint work with Pierluigi Crescenzi, Gianlorenzo D'Angelo, Henning Meyerhenke, Lorenzo Severini and Yllka Velaj, and is currently in revision for international journal publication. Part of the results presented in Chapter 7 have ben published as "Approximating betweenness centrality in large evolving networks" (coauthored with Henning Meyerhenke and Christian Staudt) at the *Seventeenth Workshop on Algorithm Engineering and Experiments* (ALENEX 2015). Another part of the results has been published as "Fully-dynamic approximation of betweenness centrality" (coauthored with Henning Meyerhenke) at the *Twenty-third Annual European Symposium on Algorithms* (ESA 2015) and further extended in "Approximating betweenness centrality in fully dynamic networks" (coauthored with Henning Meyerhenke), published in the journal *Internet Mathematics.*

COMPUTATION OF NODES WITH HIGHEST CLOSENESS CENTRALITY. Closeness centrality is defined as the inverse of the average shortest-path distance between one node and the other nodes of the network. The idea behind this definition is that a central node must be efficient in spreading information to other nodes: a node is central if the average number of links needed to reach another node is small. Computing closeness for all nodes requires all pairwise distances. However, differently from betweenness, the closeness of a given node can be computed in $\Theta(n + m)$ times using a Breadth-First Search (or in $\Theta(n \log n + m)$ time using Dijkstra's algorithm in weighted graphs). Since many applications are interested in finding the most central nodes rather than the score of each node, a natural question is whether the top-k nodes with highest closeness can be found faster than computing closeness for all nodes.

In Chapter 9 we propose algorithms that allow us to efficiently find the k nodes with highest closeness in networks with small and large diameters, respectively. The proposed techniques compute different upper bounds on the closeness values and stop the computation when k nodes are found whose closeness is higher than the bounds of the other nodes. For example, using our new approach, we are able to find the top-10 nodes with highest closeness in the whole street network of north America (with 36 millions edges) in about one hour, where exhaustive computation would take years.

We then consider in Chapter 10 the problem of keeping track of the k nodes with highest closeness centrality in dynamic networks. Our algorithms build on the static top-k closeness algorithms presented in Chapter 9 and use information obtained during earlier computations to omit unnecessary work. However, they do not require asymptotically more memory than the static algorithms (i. e., linear in the number of nodes). Our experimental results show that, on many instances, our dynamic algorithms are two orders of magnitude faster than recomputation; on some large graphs, we even reach average speedups between 10^3 and 10^4.

The top-k closeness problem consists in finding nodes that are individually central. However, a number of applications are actually interested in finding a *group* of nodes that is central as a whole. For example, in social networks, retailers might want to select a group of nodes as promoters of their product, in order to maximize the spread among users. In this context, picking the k most central nodes might lead to a large overlap between the sets of influenced nodes, whereas there might be k nodes that are not individually central, but that influence different areas of the network. Group Closeness Maximization (GCM) aims at finding a group of k nodes with the minimum average shortest path distance to the remaining nodes (where the distance between a group and a node v is the minimum among the distances between v and the elements of the group). Recently, Chen et al. [37] showed that GCM is NP-hard and proposed an $(1 - 1/e)$-approximation algorithm. Unfortunately, the algorithm does not scale easily to graphs with more than about 10^4 vertices, since it requires to compute pairwise distances. In Chapter 11 we propose techniques for scaling up the approximation algorithm by Chen et al. [37]. Our method allows us to approximate the group with maximum closeness on networks with up to hundreds of millions of edges in minutes or at most a few hours. To have the same theoretical guarantee, the approach in [37] would take several days already on networks with hundreds of thousands of edges. Our new approach allows us to study properties of the most-central group of nodes in large networks, such as its overlap with the k nodes with individually highest closeness.

The results presented in Chapter 9 have been published as "Computing top-k Closeness Centrality Faster in Unweighted Graphs" (coauthored with Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke) at the *Eighteenth Workshop on Algorithm Engineering and Experiments* (ALENEX 2016), and extended in a journal version, currently in revision. The results presented in Chapter 10 are joint work with Patrick Bisenius, Eugenio Angriman and Henning Meyerhenke and have been accepted for publication at the *Twentieth Workshop on Algorithm Engineering and Experiments* (ALENEX 2018). The results presented in Chapter 11 are joint work with Tanya Gonser and Henning Meyerhenke and have been accepted for publication at the *Twentieth Workshop on Algorithm Engineering and Experiments* (ALENEX 2018).

APPROXIMATION OF ELECTRICAL CLOSENESS. In Chapter 12 we consider a centrality measure which, differently from closeness and betweenness, takes all paths of the graph into account, weighted by their length. Just as traditional closeness, electrical closeness [33] indicates the inverse average distance to the other nodes, but in this case distance means *resistance distance*. The resistance distance between two nodes u and v is a quantity proportional to the expected number of steps taken by a random walk starting in u to reach v and then return to u for the first time (the term “electrical” comes from the fact that the same formulation can be obtained interpreting the network as an electrical circuit). Unfortunately, electrical closeness is an expensive measure: an exact computation requires to invert a $n \times n$ matrix, where n is the number of nodes. We present two approximation algorithms based on solving linear systems and leading to very accurate results in practice. Using them we are now able to compute an estimation of electrical closeness in networks with tens of millions of nodes within seconds or a few minutes. Also, our experiments indicate that electrical closeness can discriminate among nodes significantly better than traditional shortest-path closeness and is also considerably more resistant to noise – we thus show that two known drawbacks of shortest-path closeness are alleviated by the electrical variant.

The results presented in Chapter 12 have been published as “Estimating current-flow closeness centrality with a multigrid laplacian solver” (coauthored with Michael Wegner, Dimitar Lukarski, and Henning Meyerhenke) at the *Seventh SIAM Workshop on Combinatorial Scientific Computing* (CSC 2016).

2 SHORTEST PATHS AND CENTRALITY MEASURES

2.1 GRAPH BASICS

Definition 2.1.1 (Graph (unweighted, undirected))**.** *An undirected unweighted graph is an ordered pair $G = (V, E)$, where V is a set and E is a set of 2-element subsets of V, that is $E \subseteq \{\{u, v\} : u, v \in V\}$.*

The elements of V are usually referred to as nodes or vertices, and the elements of E as edges or links. An unweighted graph is said to be *directed* if the elements of E are ordered pairs of elements of V, namely $E \subseteq \{(u, v) : u, v \in E\}$.

Definition 2.1.2 (Graph (weighted, undirected))**.** *An undirected weighted graph is an ordered triplet $G = (V, E, \omega)$, where (V, E) is an undirected unweighted graph and ω is a function $\omega : E \to \mathbb{R}$.*

We call $\omega(e)$ the *weight* of edge $e \in E$. As for the unweighted case, a weighted graph is directed if $E \subseteq \{(u, v) : u, v \in E\}$. Any unweighted graph can be seen as a weighted graph where $\omega(e) = 1$, $\forall e \in E$. In this work, we only consider graphs with positive edge weights. Whether the considered graphs are directed, undirected, weighted or unweighted will be made clear in each chapter.

Given a graph $G = (V, E)$, graph $G' = (V' \subseteq V, E' \subseteq E \cap (V' \times V'))$ is called a subgraph of G. An edge e containing nodes u and v is said to be incident to u and v and both nodes are adjacent. In an undirected graph, the set of nodes adjacent to u are called neighbors of u. In directed graphs, we distinguish between in-neighbors of u (i.e., $v \in V : (v, u) \in E$) and out-neighbors of u (i.e., $v \in V : (u, v) \in E$).

2.2 DISTANCES IN GRAPHS

In mathematics, a metric or distance is a function defined on each pair of elements of a set S. To be a metric on S, a function $d : S \times S \to \mathbb{R}$ has to satisfy the following four properties:

1. non-negativity: $d(u, v) \geq 0 \quad \forall u, v \in S$
2. identity of indiscernibles: $d(u, v) = 0 \iff u = v$
3. symmetry: $d(u, v) = d(v, u) \quad \forall u, v \in S$
4. triangle inequality: $d(u, w) \leq d(u, v) + d(v, w) \quad \forall u, v, w \in S$

Given a graph $G = (V, E, \omega)$, a *node distance measure* is a function $d : V \times V \to \mathbb{R}$ defined on the node pairs of G that satisfies the four properties listed above.[2] In the following, we describe two well-known graph distance measures, which will be necessary to understand the centrality measures described in Section 2.3.

2 Exceptions are typically made for the symmetry property in directed graphs.

2.2.1 *Shortest-path distance*

One of the most natural definitions of distances between two nodes is the length of the shortest path connecting one to the other.

Definition 2.2.1 (Walk, Path). *Given a graph $G = (V, E, \omega)$ and two nodes $s, t \in V$, a walk p_{st} from s to t is a sequence of nodes $(p_0, \ldots, p_k)$ such that $p_0 = s$, $p_k = t$ and $(p_i, p_{i+1}) \in E$, for $i = 0, \ldots, k-1$. A path is a walk where all nodes are distinct, except possibly s and t.*

In particular, a path p_{st} such that $s = t$ is called a *cycle*. A graph with no cycles is said to be acyclic. We will use the acronym DAG for directed acyclic graphs.

We say that node t is *reachable* from node s if there exists a path between s and t. An undirected graph is said to be *connected* if every node in G is reachable from every other node. We use the term *tree* to indicate an undirected connected acyclic graph. A directed graph for which every node is reachable from every other node is called *strongly connected*.

The length of a walk or a path is defined as the sum of the weights of its edges, i. e. $\omega(p_{st}) = \sum_{i=0}^{k-1} \omega(p_i, p_{i+1})$.[3] The *shortest path* between s and t is the path p_{st} with minimum length (notice that there might be multiple paths satisfying this property). The *shortest-path distance* d is therefore the function that associates to each node pair (s, t) the length of the shortest path(s) between s and t.

Definition 2.2.2 (Shortest-path distance). *Let P_{st} be the set of all paths between node s and node t. The shortest-path distance d is a function $d : V \times V \to \mathbb{R}$ such that $d(s, t) := \min_{p \in P_{st}} \omega(p)$, $\forall s, t \in V$.*

It is easy to see that d is a metric in undirected graphs, whereas it does not satisfy symmetry in directed graphs (the distance from s to t might be different from the distance from t to s).

There are several ways of computing shortest-path distances between nodes. Although it is not clear when this problem was considered for the first time, one can imagine that finding shortest paths was essential even for primitive societies (for example, for reaching food as quickly as possible). Edsger W. Dijkstra published his well known algorithm in 1959, rediscovering previous results by Prim and the Czech mathematician Jarník. Using a Fibonacci Heap-based priority queue, Dijkstra's algorithm finds shortest paths between one node and the remaining ones in time $\Theta(m + n \log n)$, where n and m are the number of nodes and the number of edges, respectively. If the graph is unweighted, the shortest paths from one node to the others can be computed with a Breadth-First Search (BFS) in time $\Theta(n + m)$. This problem is often referred to as *Single-Source Shortest Path* (SSSP). When the shortest paths between all pairs of nodes are needed, one usually refers to the All-Pairs Shortest Path (APSP) problem. Variants of SSSP and APSP that only require to compute distances and not the actual paths are sometimes called Single-Source Distances (SSD) and All-Pairs Distances (APD) (although not all authors make a distinction between APSP and APD). APSP and APD can be solved by running Dijkstra's algorithm or BFS from each node, requiring time $\Theta(n^2 \log n + nm)$ in weighted and $\Theta(nm)$ in unweighted graphs. Alternatively, pairwise distances can be computed using matrix multiplications. Using Seidel's algorithm, this can be done in time $\Theta(MM(n) \log n)$ in undirected unweighted

3 We recall that, in an unweighted graph, every edge is assumed to have weight 1.

graphs, where $MM(n)$ is the cost of multiplying two matrices of size $n \times n$. Currently, the best existing algorithms can do this in time $O(n^{2.373})$ [54, 123]. For directed and weighted graphs, other techniques also based on matrix multiplication have been developed [114, 127]. We refer the reader to [48] for an overview on fast matrix multiplication methods.

For dense graphs, methods based on fast matrix multiplication are asymptotically faster than running a SSSP from each node. However, large hidden constants make the algebraic methods usually much slower in practice (also, notice that most real-world networks are actually sparse). Thus, BFS and Dijkstra's algorithm are usually preferred.

2.2.2 *Resistance distance*

Information does not always follow shortest paths. An example is electricity, which flows according to the well-known Ohm's law. One can regard a graph as an *electrical network* where each edge $\{u, v\}$ corresponds to a resistor with conductance ω_{uv} (the edge weight) or resistance $1/\omega_{uv}$. The conductance can be interpreted as the ease with which an electrical current can flow through the edge. Then, the resistance distance between a pair of nodes u and v is the electrical resistance measured across nodes u and v or, in other words, the potential difference that appears across terminals u and v when a unit current source is applied between them. Intuitively, the resistance distance measures how close u and v are: it is small when there are many short paths between them. We refer the reader to Chapter 12 for a more formal definition of resistance distance.

The resistance distance is related to a matrix called *graph Laplacian*, defined as the difference between the degree matrix and the adjacency matrix of the graph, i. e. $L := D - A$. Naming $L^\dagger$ the Moore-Penrose pseudoinverse of the Laplacian L, one can show that the resistance distance $\rho(u, v)$ between u and v can be computed as $L^\dagger_{uu} - L^\dagger_{uv} - L^\dagger_{vu} + L^\dagger_{uu}$. If G is undirected, then $\rho(u, v) = \rho(v, u)$ and the resistance distance is a metric. Since inverting a matrix is an expensive operation, alternative methods are sometimes used in practice. Naming b_{uv} a vector with all zeros but $+1$ in position u and -1 in position v, $\rho(u, v)$ can be computed as $p_{uv}(u) - p_{uv}(v)$, where p_{uv} is the solution to the linear system $Lp_{uv} = b_{uv}$. Spielman and Teng [117] showed that one can construct iterative algorithms to solve Laplacian systems in nearly-linear running time. Here, nearly-linear refers to a complexity of the form $O(\lambda \log^c \lambda log(1/\tau))$, where λ is the number of nonzero entries in the matrix, c is a positive constant, and τ is the desired accuracy to be reached (tolerance). These results have been further improved and simplified in the last years [68, 73, 74]. So-called *multigrid methods* have an even better *empirical* running time of $O(m \log(1/\tau))$ (where m is the number of edges), and are therefore used in practice [75, 84].

Resistance distance is equivalent, up to a multiplicative factor,[4] to *commute-time distance*, namely the expected time taken by a random walk starting in vertex u to travel to vertex v and then go back to u (for the first time). For further properties of this measure, we refer the reader to [86] and references therein.

4 The multiplicative factor is the square root of the *graph volume*, which is the sum of the weights of all edges.

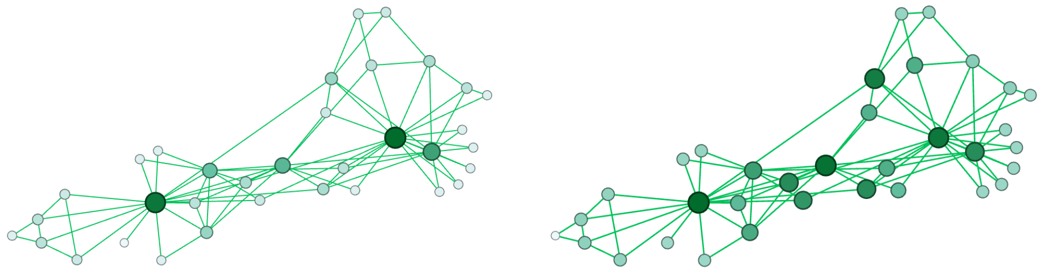


Figure 2: Ranking of nodes in the `karate` graph based on their degree (left) and their closeness centrality (right). The color and size of the nodes indicate their centrality.

2.3 CENTRALITY MEASURES

The concept of centrality answers a natural question: "Given a network, which are the most *important* nodes or edges?". In fact, this question has arisen many times in sociology, psychology and computer science. But what does "important" actually mean? In [53], Freeman suggests as a starting point for defining a notion of centrality the fact that the center of a star graph should be its most important node. However, the center of a star verifies several properties at the same time: it is the node with the highest number of neighbors, the node with minimum average distance to the other nodes, the node through which most shortest paths pass, and many more. Clearly, there cannot be a universal definition of centrality, as this is strongly application-dependent. Over the years, a huge number of definitions of importance have been proposed under the name of *centrality measures*. A centrality measure is a function, defined on either the nodes or the edges of a graph, that attributes to each element a real (usually non-negative) number, representing its importance. Depending on whether it is defined on the nodes or the edges, we distinguish between *node centrality* and *edge centrality* measures.

2.3.1 *Node Centrality Measures*

DEGREE CENTRALITY. Degree centrality is one of the simplest measures and it identifies the structural importance of a node with its number of connections. In an undirected graph, the degree of a node v is defined as follows:

$$\mathsf{degree}(v) := |\{w \in V : \{v, w\} \in E\}| \tag{1}$$

In directed graphs, one can distinguish between *outdegree* of v ($|\{w \in V : (v, w) \in E\}|$) and *indegree* of v ($|\{w \in V : (w, v) \in E\}|$). Differently from other measures, degree is local: it only depends on the immediate neighborhood of a node. Nevertheless, in real-world complex networks, there is often a strong correlation between degree and other global centrality measures.

CLOSENESS CENTRALITY. According to closeness, a node is central when it can reach other nodes quickly. Assuming that information follows shortest paths, this also means that a node with high closeness can spread information to other nodes efficiently. More formally, the closeness centrality of a node v is:

$$c_C(v) := \frac{n-1}{\sum_{w \in V} d(v,w)} \tag{2}$$

where d is the shortest-path distance in G.

The graph in Figure 2 represents the well-known Zachary's karate club, where each node is a member of the club and an edge indicates an interaction between the members. The left part of the figure indicates the degrees of the nodes, whereas the right part shows their closeness (larger and darker nodes are more central). In this graph (as in most real-world complex networks), nodes with high degree also have a high closeness. However, closeness differentiates better among nodes with smaller degree.

Notice that closeness is defined only on connected graphs (or, if G is directed, it has to be strongly connected), since if some node w was not reachable from v, the denominator would be infinity. Hence, alternative measures for disconnected graphs have been proposed. Naming $R(v)$ the set of nodes reachable from v and $r(v)$ its cardinality, closeness can be generalized as follows [97]:

$$c_C(v) := \frac{(r(v)-1)^2}{(n-1)\sum_{w \in R(v)} d(v,w)} \tag{3}$$

In case a vertex v has (out)degree 0, its closeness centrality is set to 0. Another established variant of closeness for disconnected graphs is *harmonic closeness* [25]. Instead of computing the inverse of the sum of the distances, harmonic closeness sums the inverse of the distances to the other nodes.

$$c_H(v) := \sum_{w \in R(v), w \neq v} \frac{1}{d(v,w)} \tag{4}$$

In addition to extend to disconnected graphs in a very natural way, harmonic closeness has been shown to satisfy all axioms presented in [25]. However, notice that harmonic closeness is not exactly equivalent to closeness. For example, consider a small graph as in Figure 3. Node a has distance 1 to nodes b and c, distance 2 to node d and distance 3 to node e. On the other hand, node e has distance 1 to node c and distance 2 to all remaining nodes. Although the sum of their distances are the same (and therefore their closeness values), the harmonic closeness of a is higher than the harmonic closeness of e.

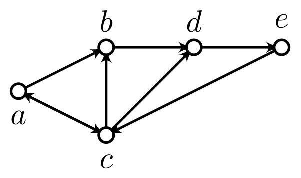


Figure 3: Nodes a and e have the same closeness but different harmonic values.

BETWEENNESS CENTRALITY. Betweenness centrality ranks the nodes according to their participation in the shortest paths of the graph. In other words, a node has high betweenness when it lies in many shortest paths between other pairs of nodes. Naming σ_{st}

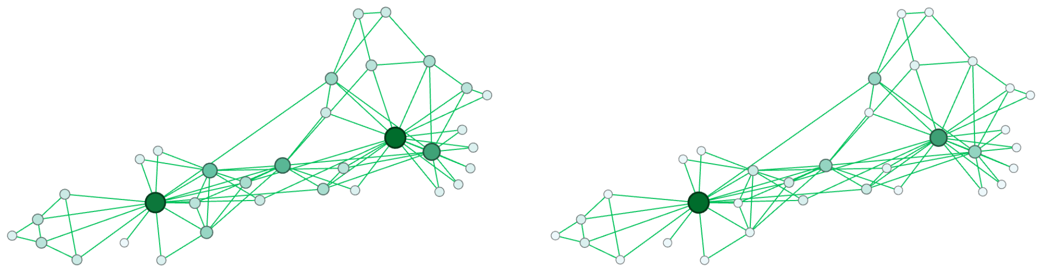


Figure 4: Ranking of nodes in the `karate` graph based on their degree (left) and their betweenness centrality (right). The color and size of the nodes indicate their centrality.

the number of shortest paths between node s and node t and $\sigma_{st}(v)$ the number of these paths that go through v, betweenness is defined as follows:

$$c_B(v) := \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{5}$$

Betweenness is often normalized in order to have all scores between 0 and 1. To do this, it is sufficient to divide by $n(n-1)$ in Eq (5). Figure 4 shows a comparison between degree and betweenness centrality on Zachary's karate club. Although not exactly the same, the rankings given by the two measures on this graph are quite similar. In fact, there is often a high correlation between degree and betweenness in complex networks. The experimental study in Chapter 12 gives some evidence of this fact.

ELECTRICAL CLOSENESS. Introduced by Brandes and Fleischer [33], electrical closeness (also known as current-flow closeness) ranks the nodes based on their average distance to the other nodes, with the difference that here "distance" means resistance distance.

$$c_{EC}(v) := \frac{n-1}{\sum_{w \in V} \rho(v,w)} \tag{6}$$

Electrical closeness is defined only on undirected graphs and, as shortest-path closeness, it is well-defined only if the graph is connected (but could be extended as in Eq. (3) or Eq. (4)). One criticism to shortest-path closeness is that its distribution is often rather flat in complex networks. This is actually not surprising, if we consider that the diameter (and therefore also the average shortest-path distance to other nodes) is typically extremely small. The fact that electrical closeness takes all paths into account (weighted by their length) and not only the shortest one makes it a promising alternative to traditional closeness. The experiments in Chapter 12 support the intuition that electrical closeness can differentiate better among nodes.

ELECTRICAL BETWEENNESS. In an electrical network, the analog of the fraction of shortest paths going through a node is the fraction of current flowing through that node. Resistance distance between two nodes s and t is computed applying a positive supply in

s and a negative supply in t, equal and opposite to the one in s. This supply generates a current $x_{st}(\vec{e})$ on each edge e. Based on the current flowing through the incident edges of a node, Brandes and Fleischer [33] define the throughput $\tau_{st}(v)$ of a node v as:

$$\tau_{st}(v) := \frac{1}{2} \sum_{e:=\{v,w\}\in E} |x_{st}(\vec{e})| \tag{7}$$

Electrical betweenness of node v is then defined as:

$$c_{EB}(v) := \sum_{s\neq v\neq t} \tau_{st}(v) \tag{8}$$

The fact that electrical closeness and betweenness account for all paths and not only the shortest ones makes them very interesting measures. However, a major drawback is that their exact computation is quite expensive: it requires $O(I(n-1)+n)$ time for the former and $O(I(n-1)+mn\log n)$ for the latter, where $I(n-1)$ is the time needed to invert a $(n-1)\times(n-1)$-matrix (which is $\Omega(n^2 \log n)$ [33]). For this reason, an approximation algorithm for electrical betweenness has been proposed in [33], and we present two new approaches for electrical closeness in Chapter 12.

EIGENVECTOR CENTRALITY. Eigenvector centrality is based on the concept that a node is important when it has important neighbors. Thus, connections to highly-central nodes contribute more to the score of a node than connections to nodes with lower centrality. Being A the adjacency matrix of the graph, the eigenvector x of A is the vector solving the equation $Ax = \lambda x$ for some constant λ. In general, there can be many different eigenvalues λ for which a non-zero eigenvector x exists. However, the Perron-Frobenius theorem implies that all entries of x are positive only for the greatest eigenvalue λ_{max} of A. Thus, assuming nodes are indexed from 1 to n, eigenvector centrality is defined as:

$$c_{Eig}(v_i) := x_i \tag{9}$$

where x_i is the i-th component of the eigenvector x solving the equation $Ax = \lambda_{max}x$. Eigenvector centrality requires the graph to be (strongly) connected. If this is not the case, the eigenvector might be equal to zero in correspondence of some of the components.

PAGERANK. Alleged to be used in Google's ranking algorithm, PageRank is one of the best-known centrality indices. Let the nodes be indexed from 1 to n and let α be a constant (called *damping factor*) in the interval $(0,1)$. Then, the PageRank of a node v_i is defined as

$$c_{PR}(v_i) := x_i \tag{10}$$

where x_i satisfies the equation

$$x_i = \alpha \sum_{j=1}^{n} \frac{A_{ji}}{\deg(v_j)} x_j + \frac{1-\alpha}{n} \tag{11}$$

PageRank might be interpreted as the stationary distribution of a random walk over the pages of the Web, following hyperlinks between pages (where pages are nodes and hyperlinks are edges). The damping factor α represents the probability of the user to continue the random walk, whereas $(1-\alpha)$ can be interpreted as the probability of a random jump to another page.

KATZ CENTRALITY. Katz centrality can be seen as an extension of degree centrality that accounts for all walks starting from a node. The length of the walks is taken into account by means of an attenuation factor α. In particular, let ω_v^k be the number of walks of length k starting in v. Then, Katz centrality is the sum – over all possible lengths – of the number of walks of that length, multiplied by the attenuation factor.

$$c_{Katz}(v) := \sum_{k=1}^{+\infty} \omega_v^k \alpha^k \tag{12}$$

Katz centrality can also be expressed in terms of powers of the adjacency matrix A. In this respect, notice that A^k encodes the number of walks of length i between pairs of nodes, i. e. $(A^k)_{ij}$ contains the number of walks of length k between v_i and v_j (assuming, again, that nodes are indexed from 1 to n). Then, Eq. (12) can be rewritten as

$$c_{Katz}(v_i) := \sum_{k=1}^{+\infty} \sum_{j=1}^{n} \alpha^j (A^k)_{ji} \tag{13}$$

For the series to converge, the value of the attenuation factor α has to be smaller than the reciprocal of (the absolute value of) the largest eigenvalue of A.

2.3.2 *Edge Centrality Measures*

As the name suggests, an edge centrality measure is a function that attributes to each edge of the graph a number representing its importance. Just like node centrality measures, a variety of indices have been proposed in the literature for edges as well. In the following, we describe two of them, since they have both received considerable attention in the research community and are of interest for the next chapters.

EDGE BETWEENNESS CENTRALITY. Similarly to node betweenness, edge betweenness indicates the fraction of shortest paths between pairs of nodes going through an edge. Given an edge $e \in E$, let us name σ_{st} the number of shortest paths between node s and node t and $\sigma_{st}(e)$ the number of these paths going through e. Thus, the definition of edge betweenness is analogous to that of node betweenness.

$$c_B(e) := \sum_{s,t \in V} \frac{\sigma_{st}(e)}{\sigma_{st}} \tag{14}$$

In an unweighted graph, edge betweenness is always at least 1, since the shortest path between the two endpoints of the edge is the edge itself.

SPANNING EDGE CENTRALITY. Spanning edge centrality is defined as the fraction of minimum spanning trees containing an edge. A spanning tree of a connected undirected graph G is a connected subgraph of G containing all of its nodes and $n-1$ of its edges. A minimum spanning tree (MST) is a spanning tree whose total weight (the sum of the

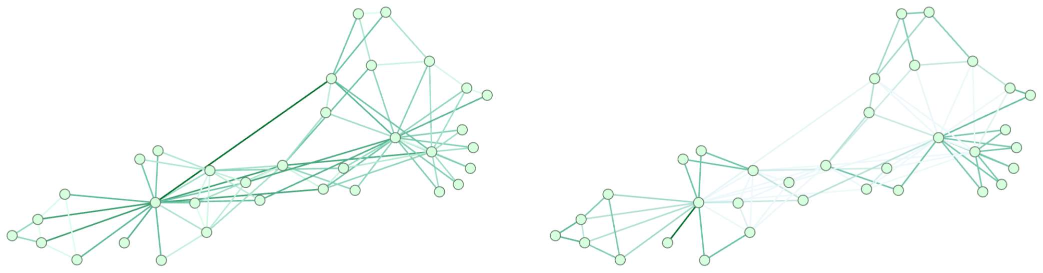


Figure 5: Ranking of edges in the `karate` graph based on their betweenness (left) and their spanning centrality (right). The color of the edges indicate their centrality.

weights of its edges) is minimum among all spanning trees of G. Spanning edge centrality is then defined as follows:

$$c_S(e) := \frac{\tau(e)}{\tau} \tag{15}$$

where τ is the number of MSTs in G and $\tau(e)$ the number of those going through edge e. Initially introduced under the name "spanning edge betweenness" in Teixeira et al. [121], this measure is based on a quite different concept than edge betweenness. Intuitively, spanning edge centrality indicates how important an edge is for the connectivity of the graph. Thus, edges with a degree-1 node as an endpoint have centrality 1 (the maximum possible value), since all spanning trees have to contain them and their removal would disconnect the graph. The difference between spanning edge centrality and edge betweenness can be seen also in Figure 5.

Although this might not be apparent at first sight, the spanning edge centrality of an edge $\{u, v\}$ in an unweighted graph is the resistance distance between u and v. Thus, it can be either computed using the Moore-Penrose pseudoinverse of the Laplacian, or by solving a Laplacian system for each edge.

3 DYNAMIC SHORTEST-PATH ALGORITHMS

3.1 PRELIMINARIES

Dynamic graphs are graphs that evolve over time. Depending on their nature, different kinds of graph updates are defined. In all following definitions, we assume Γ to be the set of all possible graphs with a finite number of nodes and positive edge weights.

Definition 3.1.1 (Edge insertion). *Let $G = (V, E, \omega) \in \Gamma$ be a graph, $u, v \in V$ be two nodes such that $(u, v) \notin E$ and let $\alpha \in \mathbb{R}^+$ be a positive number. An edge insertion is a function $\phi_{(u,v,\alpha)} : \Gamma \to \Gamma$ such that $\phi_{(u,v,\alpha)}(G) = G' := (V, E', \omega')$, where $E' = E \cup \{(u, v)\}$, $\omega'(u, v) = \alpha$ and $\omega'(e) = \omega(e) \quad \forall e \in E$.*

Definition 3.1.2 (Edge deletion). *Let $G = (V, E, \omega) \in \Gamma$ be a graph and $u, v \in V$ be two nodes such that $(u, v) \in E$. An edge deletion is a function $\phi_{(u,v)} : \Gamma \to \Gamma$ such that $\phi_{(u,v)}(G) = G' := (V, E', \omega')$, where $E' = E \setminus \{(u, v)\}$ and $\omega'(e) = \omega(e) \quad \forall e \in E'$.*

Definition 3.1.3 (Edge-weight decrease). *Let $G = (V, E, \omega) \in \Gamma$ be a graph, $u, v \in V$ be two nodes such that $(u, v) \in E$ and let $\alpha \in \mathbb{R}^+$ be a positive number, $\alpha < \omega(u, v)$. An edge weight decrease is a function $\phi_{(u,v,\alpha)} : \Gamma \to \Gamma$ such that $\phi_{(u,v,\alpha)}(G) = G' := (V, E, \omega')$, where $\omega'(u, v) = \alpha$ and $\omega'(e) = \omega(e) \quad \forall e \in E \setminus \{(u, v)\}$.*

Definition 3.1.4 (Edge-weight increase). *Let $G = (V, E, \omega) \in \Gamma$ be a graph, $u, v \in V$ be two nodes such that $(u, v) \in E$ and let $\alpha \in \mathbb{R}^+$ be a positive number, $\alpha > \omega(u, v)$. An edge weight decrease is a function $\phi_{(u,v,\alpha)} : \Gamma \to \Gamma$ such that $\phi_{(u,v,\alpha)}(G) = G' := (V, E, \omega')$, where $\omega'(u, v) = \alpha$ and $\omega'(e) = \omega(e) \quad \forall e \in E \setminus \{(u, v)\}$.*

Edge insertions and edge-weight decreases are named *incremental* updates, whereas edge deletions and edge-weight increases are referred to as *decremental* updates.

A dynamic graph algorithm is an algorithm that takes as input the new graph G', the edge update $\phi_{u,v,\alpha}$ and a set $\mathcal{P}$ of properties of G and returns the updated set of properties $\mathcal{P}'$ for G'. A straightforward dynamic algorithm is the static algorithm that recomputes $\mathcal{P}'$ on G' without using any information about $\mathcal{P}$. Depending on what kind of updates they can handle, algorithms for dynamic graphs are also divided into incremental, decremental and fully-dynamic (algorithms that can handle both incremental and decremental updates).

In addition to single-edge updates, algorithms for *batch updates* exist. A batch update is a sequence of edge updates that are applied to the graph one after another. More formally:

Definition 3.1.5 (Batch update). *Let $G = (V, E, \omega) \in \Gamma$ be a graph and let $\beta = (\phi^{(1)}_{(u_1,v_1,\alpha_1)}, \dots, \phi^{(k)}_{(u_k,v_k,\alpha_k)})$ be a sequence of edge updates. A batch update ϕ_β is the composition of functions $\phi^{(k)}_{(u_k,v_k,\alpha_k)} \circ \dots \circ \phi^{(1)}_{(u_1,v_1,\alpha_1)} : \Gamma \to \Gamma$.*

A dynamic batch algorithm updates a set of properties after a batch of updates is applied to G. Depending on the kind of updates composing the batch, incremental, decremental and fully-dynamic batch algorithms are defined.

In addition to edge updates, we can define *node updates*, divided into node insertions and node deletions.

Definition 3.1.6 (Node insertion)**.** *Let $G = (V, E, \omega) \in \Gamma$ be a graph, $x \notin V$ be a new node and let $\beta = (\phi^{(1)}_{(u_1,v_1,\alpha_1)}, \ldots, \phi^{(k)}_{(u_k,v_k,\alpha_k)})$ be a sequence of edge insertions such that either $u_i = x$ or $v_i = x$, for all $i \in \{1, \ldots, k\}$. A node insertion is defined as $\phi_\beta \circ \psi$, where $\psi(G) = G' := (V \cup \{x\}, E, \omega)$.*

Definition 3.1.7 (Node deletion)**.** *Let $G = (V, E, \omega) \in \Gamma$ be a graph, $x \in V$ be a node of G and let $\beta = (\phi^{(1)}_{(u_1,v_1)}, \ldots, \phi^{(k)}_{(u_k,v_k)})$ be a sequence of edge deletions such that $\phi^{(i)}_{(u_i,v_i)} \in \beta \iff (u_i, v_i) \in E \wedge x \in \{u_i, v_i\}$. A node deletion is defined as $\psi \circ \phi_\beta$, where $\psi(G) = G' := (V \setminus \{x\}, E, \omega)$.*

For simplicity of notation, in the following we will use (u, v, α) to indicate the edge update $\phi_{(u,v,\alpha)}$ and $\beta = \{(u_1, v_1, \alpha_1), \ldots, (u_k, v_k, \alpha_k)\}$ to indicate the corresponding batch update. If the graph is unweighted, we will simply write (u, v) and $\beta = \{(u_1, v_1), \ldots, (u_k, v_k)\}$.

3.2 DYNAMIC ALGORITHMS: A DATA-DRIVEN MOTIVATION

3.2.1 *Affected Nodes*

Many networks of interest – such as social networks and the Web graph – are dynamic in nature and some of them evolve over time at a very quick pace. For such networks, a natural idea is to develop algorithms that can reuse information from the initial graph and update it efficiently after a change occurs. It is as well reasonable, though, to ask ourselves whether the gain (in terms of running time) would be worth the effort of spending time developing a new algorithm and implementing additional code. For example, consider the problem of updating pairwise distances in the graph depicted in Figure 6. The insertion of edge (u, v) connects the component of u with that of v affecting the distance of about half of all possible node pairs. Similarly, connecting any node in one component with any node in the other component would lead to the same result. In such a case, the improvement of any dynamic algorithm on static recomputation would be very limited. Thus, when developing a dynamic algorithm for this specific instance, we should be aware that our algorithm cannot be more than (roughly) two times faster than recomputation.

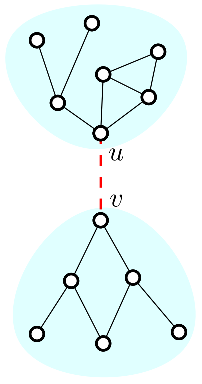


Figure 6: Edge insertion connecting two components.

Fortunately, real-world networks usually look quite different from the one in Figure 6. In this section, we try to give evidence for the fact that shortest-path distances in a complex network are resilient to most edge changes, motivating efforts to develop dynamic algorithms and providing a reference for their empirical evaluation. The reason why we specifically focus on shortest-path distances is that the dynamic algorithms we propose in the next chapters recompute centrality measures based on shortest paths. Other distances (such as resistance distance) might behave differently, but are not considered in this chapter. For this study, we consider a set of real-world networks with real edge dynamics, taken from the KONECT dataset [76] and summarized in Table 1. All edges of these graphs are characterized by a time of arrival. In case of multiple edges, we ignore additional edges between nodes that were already connected by an edge with a smaller timestamp.

Graph	Nodes	Edges	Description	Type
`digg`	30398	85155	Digg users replies	Directed
`slashdot`	51083	116573	Slashdot threads	Directed
`linux`	63399	159996	Linux kernel mailing list replies	Directed
`facebook`	46952	183412	Facebook wall posts	Directed
`enron`	87273	297456	Emails between Enron employees	Directed
`facebookFriends`	63731	817035	Facebook friendships	Undirected
`ca-HepPh`	28093	3148447	Coauthorship	Undirected
`wikipedia`	1870709	36532531	Hyperlinks of the English Wikipedia	Directed

Table 1: Networks with real edge dynamics.

In our experiments, we consider each graph without its last 1000 edges (the ones with the highest timestamp) and we insert these edges back into the graph in increasing order of timestamp. We call *affected* the node pairs that change their distance after the insertion of an edge and we indicate their set with A. Computing the set of affected pairs would require to compute all pairwise distances, at least on the initial graph. Since this would be too expensive for the networks of Table 1, we actually compute an upper bound on the number of affected pairs. Given an edge insertion (u, v), we call $A(u) := \{t \in V : d(u,t) \neq d'(u,t)\}$ and $A(v) := \{v \in V : d(s,v) \neq d'(s,v)\}$, where d and d' indicate the shortest-path distances in the original and the modified graph, respectively. If $(s,t) \in A$, then necessarily $(s,v) \in A(v)$ and $(u,t) \in A(u)$ (if G is undirected, we assume s to be closer to u than t, w.l.o.g.). Intuitively, this is due to the fact that all new shortest paths have to go through (u, v). A more formal proof can be found, among other sources, in [104]. Therefore, $|\overline{A}| := |A(u)| \cdot |A(v)|$ (or $|\overline{A}| := 2 \cdot |A(u)| \cdot |A(v)|$ in undirected graphs) is an upper bound on the number of affected node pairs $|A|$. Notice that $A(u)$ (respectively, $A(v)$) can be easily computed with one SSSP from u (respectively, to v) on the initial graph and one on the modified graph.

Figure 9 shows the distribution of the percentage of affected node pairs, i. e. $\frac{|\overline{A}|}{n(n-1)} \cdot 100$. For simplicity, the figure denotes this percentage with $|A|$ (although we recall that this is actually an upper bound on the number of affected nodes). Table 2 reports average, maximum and standard deviation of the percentage of affected pairs. Although there is some clear variation among the networks and among the updates (the standard deviation is often higher than the mean), it is interesting to notice that no edge insertion affects more than 0.5% of all possible node pairs. On `ca-HepPh` and `wikipedia`, no insertion affects more than 0.005% of node pairs. This means that an optimal dynamic algorithm could achieve a speedup of 20000 (i. e. 1/0.00005) on static recomputation for the worst-case instances of these two graphs[5].

The very small number of affected pairs could be due to one of two factors, or a combination of both. On the one hand, it might be that the structure of complex networks is such that *any* edge insertion would not affect a large number of pairs, and that random edge insertions would lead to a similar distribution of $|\overline{A}|$. On the other hand, the reason for these small numbers of affected pairs might be attributed to the specific insertions we

5 This is based on the assumption that the running time of the static algorithm is proportional to the total number of node pairs, which is actually the case for networks where $m = O(n)$, and the running time of the dynamic algorithm is proportional to the number of affected pairs.

Graph	Nodes	Edges	Average $\lvert\overline{A}\rvert$	Maximum $\lvert\overline{A}\rvert$	Std. Dev. $\lvert\overline{A}\rvert$
`digg`	30398	85155	0.01013%	0.05282%	0.00661%
`slashdot`	51083	116573	0.00291%	0.04370%	0.00376%
`linux`	63399	159996	0.00011%	0.00853%	0.00041%
`facebook`	46952	183412	0.03686%	0.44824%	0.05455%
`enron`	87273	297456	0.00083%	0.08258%	0.00387%
`facebookFriends`	63731	817035	0.00268%	0.09858%	0.00592%
`ca-HepPh`	28093	3148447	0.00024%	0.00355%	0.00082%
`wikipedia`	1870709	36532531	0.00001%	0.00149%	0.00005%

Table 2: Upper bounds $\lvert\overline{A}\rvert$ on the percentage of affected node pairs in networks with real dynamics. The fourth column contains the average value of $\lvert\overline{A}\rvert$ over the 1000 edge insertions, the fifth column the maximum, and the last column the standard deviation.

Graph	Nodes	Edges	Average $\lvert\overline{A}\rvert$	Maximum $\lvert\overline{A}\rvert$	Std. Dev. $\lvert\overline{A}\rvert$
`digg`	30398	86155	0.01206%	0.05627%	0.00646%
`slashdot`	51083	117573	0.00500%	0.04639%	0.00539%
`linux`	63399	160996	0.00262%	0.30255%	0.01561%
`facebook`	46952	184412	0.12777%	1.15197%	0.14379%
`enron`	87273	298456	0.04543%	0.95641%	0.08232%
`facebookFriends`	63731	818035	0.02006%	0.23578%	0.02613%
`ca-HepPh`	28093	3149447	0.00252%	0.07355%	0.00539%
`wikipedia`	1870709	36533531	0.00015%	0.01506%	0.00073%

Table 3: Upper bounds $\lvert\overline{A}\rvert$ on the percentage of affected node pairs under random insertions. The fourth column contains the average value of $\lvert\overline{A}\rvert$ over the 1000 edge updates, the fifth column the maximum, and the last column the standard deviation.

considered, which might only involve peripheral nodes or nodes that were already close to each other before the insertion. In the following, we try to answer this question by comparing the distribution of affected pairs generated by real updates and random updates, respectively.

3.2.2 *Real Edge Dynamics vs Random Updates*

Figure 9 and Table 2 depict the distribution of the number of node pairs affected by 1000 real edge insertions. Understanding whether this distribution would be similar under different kinds of dynamics (e. g. random edge insertions) might not only give us insights on the properties of shortest paths in complex networks, but also possibly provide us with a realistic way of testing dynamic algorithms on networks for which we have no temporal information. In fact, in the literature it is quite common to find papers that test dynamic algorithms based on random edge insertions and deletions (see for example [58, 72, 112]) without, however, discussing whether such updates would be realistic. Notice that here by “realistic” we simply mean “generating a distribution of affected pairs similar to that of real edge insertions”. Simulating the evolution of a graph given a static snapshot is a very interesting, but also complex, research question exceeding the scope of this work.

We first consider two simple ways of generating edge insertions, and then we provide

Graph	Nodes	Edges	Average $\lvert\overline{A}\rvert$	Maximum $\lvert\overline{A}\rvert$	Std. Dev. $\lvert\overline{A}\rvert$
`digg`	30398	84155	0.01029%	0.03528%	0.00612%
`slashdot`	51083	115573	0.00305%	0.03337%	0.00320%
`linux`	63399	158996	0.00097%	0.11179%	0.00648%
`facebook`	46952	182412	0.02125%	0.43805%	0.04184%
`enron`	87273	296456	0.00090%	0.15963%	0.00671%
`facebookFriends`	63731	816035	0.00127%	0.03750%	0.00313%
`ca-HepPh`	28093	3147447	0.00001%	0.00133%	0.00006%
`wikipedia`	1870709	36531531	0.00001%	0.00096%	0.00003%

Table 4: Upper bounds $\lvert\overline{A}\rvert$ on the percentage of affected node pairs under random deletions. The fourth column contains the average value of $\lvert\overline{A}\rvert$ over the 1000 edge updates, the fifth column the maximum, and the last column the standard deviation.

results suggesting that more sophisticated techniques are unlikely to lead to much more realistic results. The first method we consider are *random edge insertions*. Each node pair (u, v) such that $(u, v) \notin E$ has the same probability of being chosen for a new insertion. The affected pairs are therefore the ones for which their distance in G without edge (u, v) is larger than their distance in $G' := (V, E \cup \{(u, v)\})$. The second method are *random edge deletions*. Here an existing edge $e \in E$ is chosen uniformly at random among the edges in E and deleted from the graph. Although at first sight the first method is suitable only to test incremental algorithms (i. e. that update distances after an insertion) and the second one to test decremental algorithms (i. e. that update distances after a deletion), this is in fact not true. Indeed, when using random insertions, we could initially compute distances in the graph $G' := (V, E \cup \{(u, v)\})$, delete (u, v), and update distances using a decremental algorithm. Similarly, after choosing an edge e at random, one can delete it from G, compute distances on the obtained graph, insert it back into G and update distances with an incremental algorithm[6].

We consider 1000 random edge insertions and 1000 random edge deletions on the networks of Table 1 and compute the distribution of affected node pairs. Figure 10 and Table 3 show the affected pairs for random insertions, whereas Figure 11 and Table 4 show the affected pairs for random deletions. Interestingly, the distribution under real edge insertions (Figure 9 and Table 2) is much more similar to the one obtained with random deletions than the one obtained with random insertions. Consider, for example, the `facebook` graph. Under random insertions, the percentage of affected pairs is below 0.05% only 12.1% of the times, whereas this is true 47.0% of the times under random deletions and 41.6% of the times under real insertions. Also, under random insertions, 3.1% of updates lead to a percentage of affected pairs between 0.5% and 2%, whereas this is never the case for real insertions and random deletions, where the percentage of affected pairs is always below 0.5%. In general, random insertions lead to a significantly higher number of affected pairs, indicating a tendency of real insertions to preserve shortest-path distances (compared to random ones). On the other hand, the percentage of affected pairs is always below 2% for all types of updates, meaning that the structure itself of the tested networks makes shortest paths resilient to most edge modifications.

6 Again, we emphasize that we do not aim at simulating the evolution of a network, but just at providing ways of testing dynamic algorithms that would lead to a similar performance as with real edge dynamics.

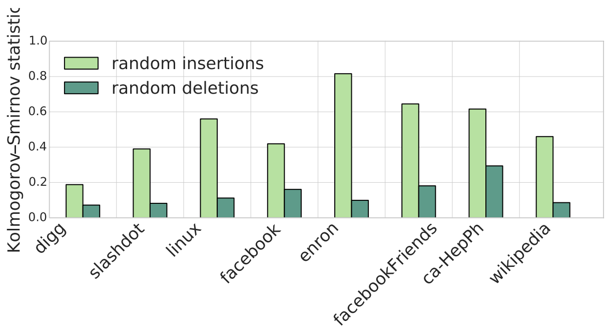


Figure 7: Kolmogorov-Smirnov statistic for the distribution of affected node pairs generated by real insertions and the distribution generated by random insertions and random deletions, respectively.

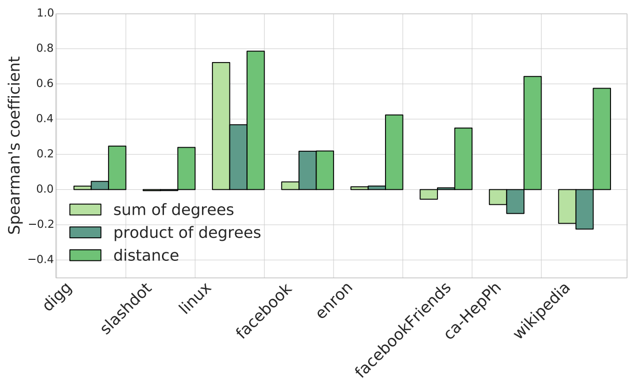


Figure 8: Correlation between number of affected node pairs and sum of the degrees of the two endpoints, product of the degrees of the two endpoints and distance between the endpoints before the insertion.

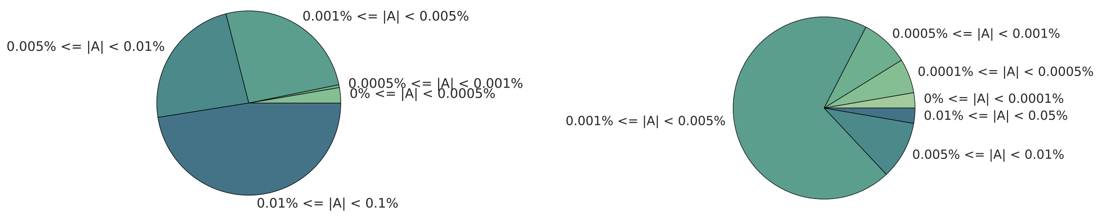
To quantify the similarity of the distribution of affected pairs generated by real insertions, random insertions and random deletions, we use the two-sample Kolmogorov-Smirnov statistic. The Kolmogorov-Smirnov (KS) statistic is defined as the supremum of the absolute difference between the empirical distributions of two samples. This means that it is equal to 0 when the two empirical distributions are exactly the same, and the closer it is to 1, the more unlikely it is that that the samples have been generated by the same probability distribution. Figure 7 shows the KS statistic for the distribution of affected pairs generated by 1000 real insertions and the distribution generated by 1000 random insertions and 1000 random deletions, respectively. It is apparent that real insertions are much closer to random deletions than to random insertions. In particular, the difference between the distribution of real insertions and random deletions is always below 0.2 (except for `ca-HepTh`, where it is 0.29) and on 4 out of 8 graphs (`digg`, `slashdot`, `enron` and `wikipedia`) it is below 0.1. On the contrary, the difference between real insertions and random insertions is over 0.8 for `enron` and above 0.4 for most of the graphs.

In terms of number of affected pairs, random deletions are therefore a better candidate for approximating real insertions than random insertions. Still, there might be other ways of choosing the edges to update (based on properties of the sampled edges) that better approximate real insertion (again, we recall that our "approximation" is in terms of number of affected pairs). To investigate this, we need to identify properties of the newly-inserted edges that influence the value of $|\overline{A}|$. For example, it may be natural to think that, if the two endpoints are far away from each other before the insertion, then the number of affected pairs will be higher. Or that nodes with a higher degree might affect a larger portion of the graph, compared to nodes with only a few neighbors. Our results in this regard are quite counterintuitive. Figure 8 shows the correlation between $|\overline{A}|$ and the distance between the endpoints of the inserted edge, the product of their degrees and the sum of their degrees, respectively. In particular, each column represents the Spearman's correlation coefficient between $|\overline{A}|$ and each one of the three properties. We recall that Spearman's correlation coefficient is 1.0 when two variables are monotone functions of one another and 0 when they are completely uncorrelated.

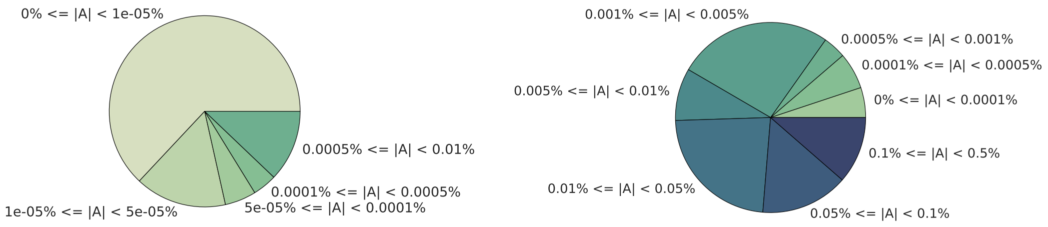
The correlation between $|\overline{A}|$ and degrees is mostly very close to 0, both for the sum and the product. Only for the `linux` graph, the correlation with the sum is above 0.6. However, for all other tested networks, this correlation is below 0.1 – and it is even negative for the `facebookFriends`, `ca-HepPh` and `wikipedia` graphs. Therefore, it seems unlikely that we can predict the number of affected pairs based on the degree of the endpoints of the updated edge, and consequently also quite unlikely that sampling edges based on degree can generate a distribution of affected pairs similar to that of real edge insertions.

Even the correlation with distance, although positive, is mostly quite small. It is never above 0.8 and for almost all graphs it is below 0.6. Also, creating new edges with a probability based on the distance requires to compute all-pairs shortest paths, which is impractical for large networks. Other properties, such as the betweenness or spanning edge centrality of the edges, might have a higher correlation with the number of affected pairs, but would be too expensive to compute on the networks of Table 1.

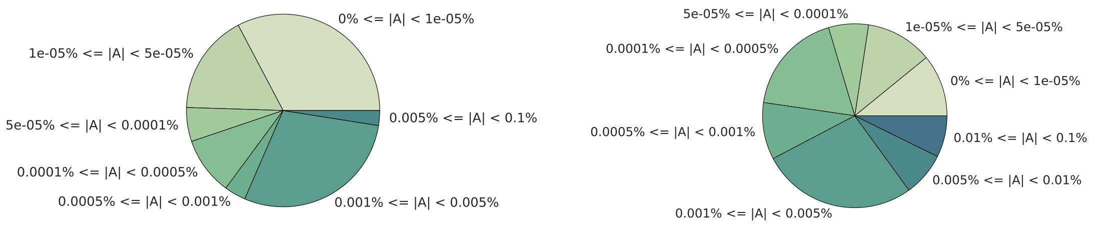
To summarize, our results suggest that random deletions generate a distribution of affected pairs that is much more similar to that of real insertions than the one generated by random insertions. Also, simple properties such as degree of the nodes incident to an edge and their distance before the insertion do not seem to have a strong correlation with the number of affected pairs and it is therefore unlikely that a more sophisticated way of sampling edges based on these properties can lead to much better results than random deletions. Since random deletions are easy to compute and their distribution of affected nodes does not differ much from that of real edge insertions, in the rest of this work we use random deletions whenever we need to test dynamic algorithms, and networks with real edge dynamics are not appropriate or sufficient for our tests.



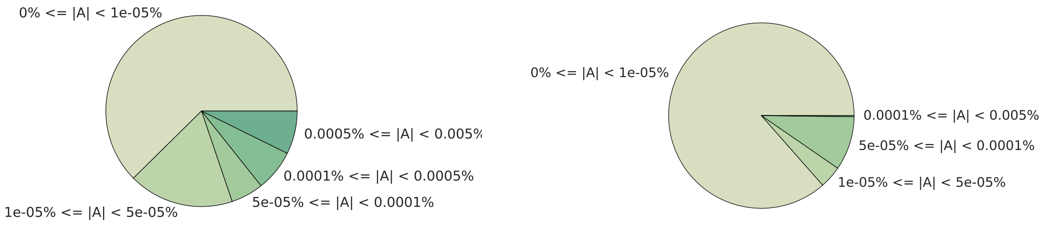
(a) Left: `digg` graph. Right: `slashdot` graph.



(b) Left: `linux` graph. Right: `facebook` graph.

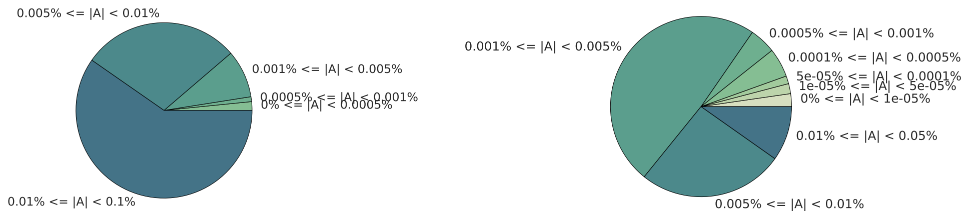


(c) Left: `enron` graph. Right: `facebookFriends` graph.

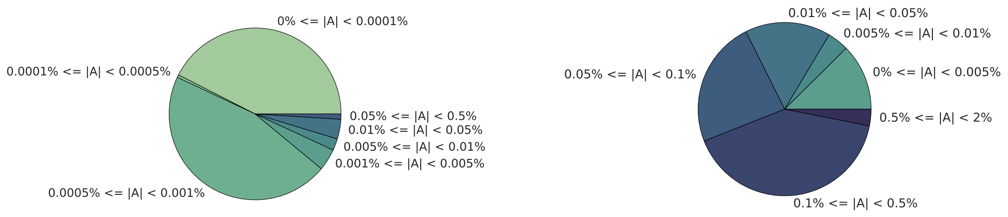


(d) Left: `ca-HepPh` graph. Right: `wikipedia` graph.

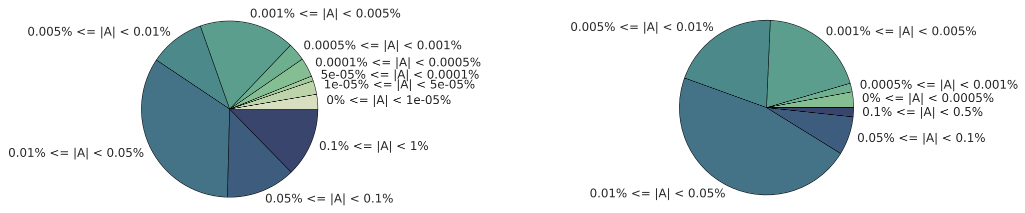
Figure 9: Upper bound on the percentage of affected node pairs (i.e. pairs that change their distance) over 1000 real edge insertions, for the networks of Table 1.



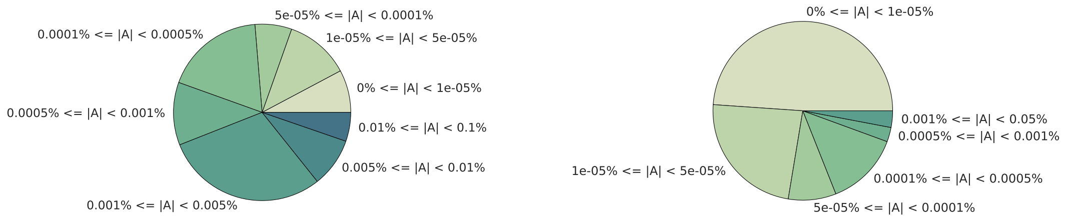
(a) Left: `digg` graph. Right: `slashdot` graph.



(b) Left: `linux` graph. Right: `facebook` graph.

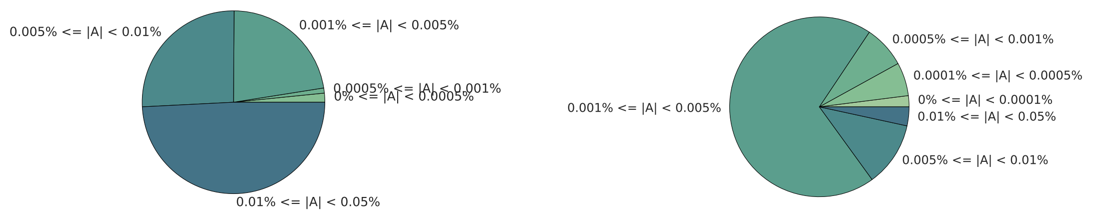


(c) Left: `enron` graph. Right: `facebookFriends` graph.

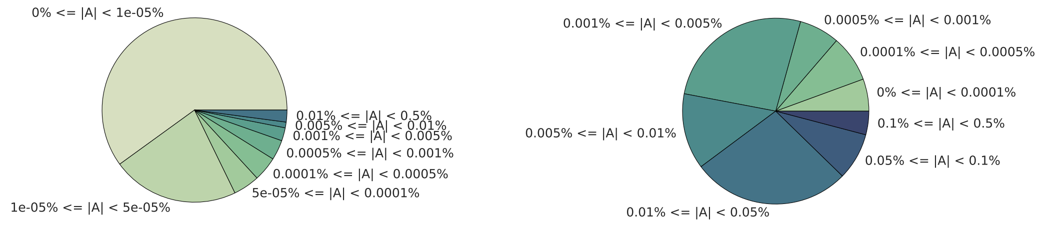


(d) Left: `ca-HepPh` graph. Right: `wikipedia` graph.

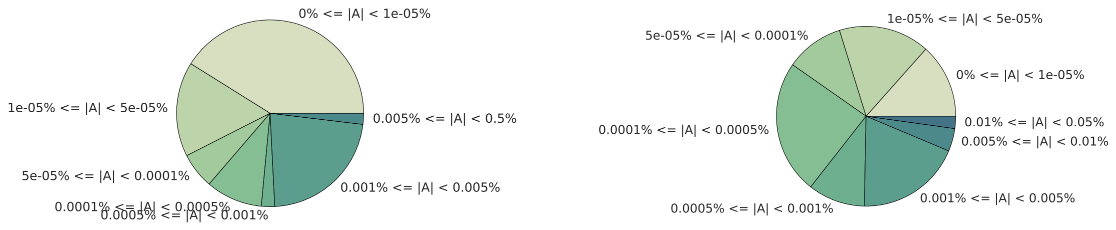
Figure 10: Upper bound on the percentage of affected node pairs (i.e. pairs that change their distance) over 1000 random edge insertions, for the networks of Table 1.



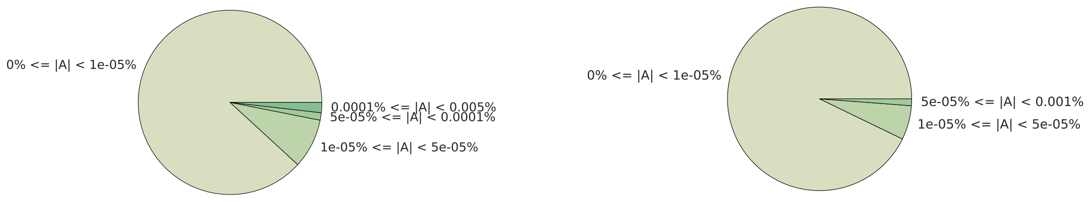
(a) Left: `digg` graph. Right: `slashdot` graph.



(b) Left: `linux` graph. Right: `facebook` graph.



(c) Left: `enron` graph. Right: `facebookFriends` graph.



(d) Left: `ca-HepPh` graph. Right: `wikipedia` graph.

Figure 11: Upper bound on the percentage of affected node pairs (i.e. pairs that change their distance) over 1000 random edge deletions, for the networks of Table 1.

Part II

DYNAMIC ALGORITHMS FOR BETWEENNESS CENTRALITY

4 OVERVIEW OF ALGORITHMS FOR BETWEENNESS CENTRALITY

4.1 INTRODUCTION

Betweenness centrality is a well-known measure which ranks nodes according to their participation in the shortest paths of the network. Nodes with high betweenness can be important in routing, spreading processes and mediation of interactions. Depending on the context, this can mean, for example, finding the most influential persons in a social network, the key infrastructure nodes in the internet, or super spreaders of a disease. Formally, the betweenness of a node v is defined as $c_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where σ_{st} is the number of shortest paths between two nodes s and t and $\sigma_{st}(v)$ is the number of these paths that go through node v. Because it requires *all* shortest paths between node pairs, the computation of betweenness is at least quadric in the number of nodes. In particular, the (asymptotically) fastest existing algorithm is due to Brandes [31] and requires $\Theta(nm)$ time in unweighted graphs and $\Theta(nm + n^2 \log n)$ time in weighted graphs, where n is the number of nodes and m is the number of edges. Some heuristics have been proposed to speed up Brandes's algorithm in practice [51, 102]. In particular, the authors of [51] report speedups of up to a factor 75 compared to Brandes's algorithm on some networks. However, even these methods cannot scale to networks with more than a few hundreds of thousands of edges, whereas nowadays several networks of interest have millions or even billions of nodes and edges. Thus, recent years have seen the publication of approximation algorithms and heuristics that aim to reduce the computational effort, while finding betweenness values that are as close as possible to the exact ones. Good results have been obtained in this regard; in particular, recent algorithms [30, 107, 108] give probabilistic guarantees on the quality of the approximation.

In addition, most real-world networks, such as the Web graph or social networks, continuously undergo changes. Since an update in the graph might affect only a small fraction of nodes, recomputing betweenness with Brandes's algorithm (or even with a static approximation algorithm) after each update would be very inefficient. For this reason, another line of research has focused on developing algorithms that can efficiently update betweenness after a change in a graph. In this chapter, we provide an overview of existing algorithms for betweenness centrality, both exact and approximate, and for both static and dynamic graphs.

4.2 BRANDES'S ALGORITHM (BA)

Betweenness centrality can be easily computed in time $\Theta(n^3)$ by simply applying its definition. In 2001, Brandes proposed an algorithm (BA) [31] which requires time $\Theta(nm)$ for unweighted and $\Theta(n(m + n \log n))$ for weighted graphs, i.e. the time of computing n single-source shortest paths (SSSPs). The algorithm is composed of two parts: the *augmented APSP* computation phase based on n SSSPs and the *dependency accumulation* phase. As dynamic algorithms based on BA build on these two steps as well, we explain them now in more detail.

AUGMENTED APSP. In this first part, BA needs to perform an *augmented* APSP, meaning that instead of simply computing distances between all node pairs (s,t), it also finds the number of shortest paths σ_{st} and the set of predecessors $P_s(t)$. This can be done while computing an SSSP from each node s (i.e. BFS for unweighted and Dijkstra for weighted graphs). When a node w is extracted from the SSSP (priority) queue, BA computes $P_s(w)$ as $\{v : (v,w) \in E \land d(s,w) = d(s,v) + \omega(v,w)\}$, by looping over the incoming edges, and σ_{sw} as $\sum_{v \in P_s(w)} \sigma_{sv}$. Notice that the time complexity of this phase is the same as computing a "normal" SSSP from each node, since computing predecessors and number of shortest paths only takes $\Theta(m)$ additional time for each SSSP.

DEPENDENCY ACCUMULATION. Brandes defines the *one-side dependency* of a node s on a node v as $\delta_{s\bullet}(v) := \sum_{t \neq v} \sigma_{st}(v)/\sigma_{st}$. Then, the betweenness centrality $c_B(v)$ of a node v can be expressed as $\sum_{s \neq v} \delta_{s\bullet}(v)$. In [31], Brandes proves the following theorem:

Theorem 4.2.1. *[31] The one-side dependency of node s on node v, $s \neq v$, can be rewritten as:*

$$\delta_{s\bullet}(v) = \sum_{w: v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s\bullet}(w)), \qquad \forall s, v \in V$$

Intuitively, the term $\delta_{s\bullet}(w)$ in the equation represents the contribution of the sub-DAG (of the SSSP DAG of s) rooted in w to the betweenness of v, whereas the term 1 is the contribution of w itself. For all nodes v such that $\{w : v \in P_s(w)\} = \emptyset$ (i.e. the nodes that have no successors), we know that $\delta_{s\bullet}(v) = 0$. Indeed, if v has no successors, there cannot be other nodes whose shortest paths from s go through v. Starting from these nodes, we can compute $\delta_{s\bullet}(v)$ $\forall v \in V$ by "walking up" the SSSP DAG rooted in s, using Theorem (4.2.1). Notice that it is fundamental that we process the nodes in order of decreasing distance from s, because to correctly compute $\delta_{s\bullet}(v)$, we need to know $\delta_{s\bullet}(w)$ for all successors of v. This can be done by inserting the nodes into a stack as soon as they are extracted from the SSSP (priority) queue in the first step. This procedure is repeated for all $s \in V$ and the betweenness of v is then simply computed as $\sum_{s \neq v} \delta_{s\bullet}(v)$.

In this second step, for each source s, each node has to iterate over the predecessors in the shortest paths from s. Since this is a subset of the incoming edges, the cost of this part is $O(m)$ for each source and thus $O(nm)$ in total.

4.3 STATIC APPROXIMATION ALGORITHMS

Most existing approximation algorithms are based on sampling shortest paths according to some criterion and extrapolating the betweenness of nodes based on the computed paths. The algorithm by Brandes and Pich [34] is the first approximation algorithm to have been proposed. The algorithm basically works like Brandes's static algorithm, with the difference that SSSPs are computed only from a *subset* of nodes – called *pivots* – instead of all nodes. For each pivot s, the one-side dependencies $\delta_{s\bullet}(v)$ are computed for all $v \in V$ and the betweenness of each v is approximated as $\frac{n}{|S|} \sum_{s \in S} \delta_{s\bullet}(v)$, where S is the set of pivots. Selecting the pivots uniformly at random, the approximation can be proven to be an unbiased estimator for $c_B(v)$ (i.e. its expectation is equal to $c_B(v)$).

In a subsequent work, Geisberger et al. [56] notice that the approach by Brandes and Pich can overestimate betweenness scores of nodes close to the pivots. To limit this bias, they introduce a *scaling function* which gives less importance to contributions from pivots that are close to the node. In particular, the authors propose two possibilities: a linear scaling function, where the effect of the contribution of a pivot on a node v increases linearly with the distance between the pivot and v, and a bisection scaling function, which considers only contributions of pivots that are "far enough" from v and ignores contributions from other pivots.

In contrast to the two approaches described so far – which consider the problem of estimating the betweenness of *all* nodes in the graph, Bader et al. [8] approximate the betweenness of a specific node only, based on an adaptive sampling technique that reduces the number of pivots for nodes with high centrality. Chehreghani [36] proposes alternative sampling techniques that try to minimize the number of samples, when the betweenness of a single node has to be estimated.

Different from previous approaches is the approximation algorithm by Riondato and Kornaropoulos [107], which samples a *single* random shortest path at each iteration. This approach does not compute one-side dependencies, but simply approximates the betweenness of each node as the fraction of sampled paths the node is part of. This allows for a theoretical guarantee on the quality of approximation: if the number r of sampled paths is large enough, one can prove that $\Pr(\exists v \in V : |c_B(v) - \tilde{c}_B(v)| > \epsilon) < \delta$, where ϵ and δ are two arbitrarily-chosen constants in $(0, 1)$. The same guarantee is offered by ABRA [108], a recent approximation algorithm based on progressive random sampling, which leverages on Rademacher averages and pseudodimension. In their experimental evaluation, the authors of ABRA show that their new approach requires significantly less samples than the one by Riondato and Kornaropoulos [107]. Also, they propose an adaptation of their algorithm which has a multiplicative error guarantee for the top-k nodes with highest betweenness. ABRA has then been further improved by Borassi and Natale in [30]. Their algorithm, named KADABRA, makes use of balanced bidirectional SSSPs and achieves average speedups on ABRA of up to two orders of magnitude in their experimental evaluation.

4.4 DYNAMIC ALGORITHMS

The basic idea of dynamic betweenness algorithms is to keep track of the old betweenness scores (and additional data structures) and efficiently update this information after some modification in the graph. Based on the type of updates they can handle, dynamic algorithms are classified as *incremental* (only edge insertions and weight decreases), *decremental* (only edge deletions and weight increases) or *fully-dynamic* (all kinds of edge updates). Although using different techniques, one commonality of all these approaches is that they build on the two phases of Brandes's static algorithm BA [31].

The approach proposed by Green et al. [58] for unweighted graphs maintains all previously calculated betweenness values and additional information, namely pairwise distances, number of shortest paths and lists of predecessors of each node in the shortest paths from each source node $s \in V$. Using this information, the algorithm tries to limit the recomputation of BA to the nodes whose betweenness has been affected by the edge insertion. Kourtellis et al. [72] modify the approach by Green et al. [58] in order to reduce the memory requirements from $O(nm)$ to $O(n^2)$. Instead of being stored, the predecessors are recomputed every time the algorithm requires them. The authors show that not only using less memory allows them to scale to larger graphs, but their approach (which we refer to as KDB, from the authors's initials) turns out to be also faster than the one by Green et al. [58] in practice (most likely because of the cost of maintaining the data structure of the algorithm by Green et al.).

Kas et al. [67] extend an existing algorithm for the dynamic all-pairs shortest paths (APSP) problem by Ramalingam and Reps [104] to also update betweenness scores. Differently from the previous two approaches, this algorithm can handle also weighted graphs. Although good speedups have been reported for this approach, no experimental evaluation compares its performance with that of the approaches by Green et al. [58] and Kourtellis et al. [72]. We refer to this algorithm as KWCC, from the authors's initials.

Nasre et al. [94] compare the distances between each node pair before and after the update and then recompute the dependencies from scratch as in BA (see Section 4.2). Although this algorithm is faster than recomputation on some graph classes (i.e. when only edge insertions are allowed and the graph is sparse and weighted), it is shown in [18] that its practical performance is much worse than that of the algorithm proposed by Green et al. [58]. This is quite intuitive, since recomputing all dependencies requires $\Omega(n^2)$ time independently of the number of nodes that are actually affected by the insertion.

Pontecorvi and Ramachandran [101] extend existing fully-dynamic APSP algorithms with new data structures to update *all* shortest paths and then recompute dependencies as in BA. To our knowledge, this algorithm has never been implemented, probably because of the quite complicated data structures it requires. Also, since it recomputes dependencies from scratch as Nasre et al. [94], we expect its practical performance to be similar.

Differently from the other algorithms, the approach by Lee et al. [79] is not based on dynamic APSP algorithms. The idea is to decompose the graph into its biconnected components and then recompute the betweenness values from scratch only for the nodes in the component affected by the update. Although this allows for a smaller memory requirement ($\Theta(m)$ versus $\Omega(n^2)$ needed by the other approaches), the speedups on recomputation reported in [79] are significantly worse than those reported for example by Kourtellis et al. [72].

The ones described so far are all dynamic *exact* algorithms, meaning that they update exact betweenness scores. The algorithm presented in Chapter 7 is the first dynamic algorithm to update an *approximation* of betweenness centrality. After it was published, another dynamic approximation algorithm was proposed by Hayashi at al. [61]. The authors represent the shortest paths in a data structure named *hypergraph sketch*, which allows them to further speedup the algorithm in Chapter 7 of up to one order of magnitude, according to their experimental evaluation.

5 FASTER INCREMENTAL BETWEENNESS CENTRALITY

5.1 INTRODUCTION

As mentioned in Section 4.4, several algorithms for updating betweenness centrality in dynamic networks have been proposed. Nevertheless, an exhaustive experimental comparison between the different approaches is still missing in the literature. In this chapter, we describe and experimentally evaluate two of the approaches for which the best speedups have been reported, namely KDB [72] and KWCC [67] (named after the initials of the authors). These approaches are based on the two steps of Brandes's algorithm (BA): the augmented APSP and the dependency accumulation step. While describing them, we identify redundant or unnecessary operations, whenever present. Thus, we propose two new solutions – one for the augmented APSP and one for the dependency recomputation – that significantly reduce the amount of work done by KDB and KWCC.

In particular, in the APSP update, we notice that both KWCC and KDB repeatedly iterate over the neighboring edges of affected nodes, whereas we show that this operation can be done only once, based on properties of the newly-created shortest paths. Compared to KWCC and KDB, our algorithm for the augmented APSP update is asymptotically faster for dense graphs: $O(n^2)$ in the worst case versus $O(nm)$.

Also, we show that dependencies can be recomputed by accumulating values in a fashion similar to that of BA. However, differently from BA, our method only processes nodes that lie in shortest paths between affected pairs. This improves over KDB, where also several non-affected nodes might be processed, and KWCC, which instead of accumulating dependency changes, updates the betweenness scores for each affected node pair separately. Our dependency update works also for weighted graphs (whereas KDB does not) and it is asymptotically faster than the dependency update of KWCC for sparse graphs ($O(nm + n \log n)$ in the worst case versus $O(v^3)$).

In the following, we describe how KDB and KWCC perform the augmented APSP (Section 5.2) and the dependency accumulation (Section 5.3). After describing existing algorithms and their limitations, we introduce our new approaches in Section 5.2.3 and Section 5.3.3.

5.2 DYNAMIC AUGMENTED APSP

As mentioned earlier, dynamic algorithms based on BA build on its two steps. In the following, we will see how KDB [72] and KWCC [67] update the augmented APSP data structures (i.e. distances and number of shortest paths) after an edge insertion or a weight decrease. One difference between these two approaches is that KDB does not store the predecessors explicitly, whereas KWCC does. However, since in [72] it was shown that keeping track of the predecessors only introduces overhead, we report a slightly-modified version of KWCC that recomputes them "on the fly" when needed (we will also use this version in our experiments in Section 5.5). We will then introduce our new approach in Section 5.2.3.

5.2.1 *Algorithm by Kourtellis et al. (KDB)*

Let (u, v) be the new edge inserted into G (we recall from Section 4.4 that KDB works only on unweighted graphs, so edge weight modifications are not supported). For each source node $s \in V$, there are three possibilities: (i) $d(s, u) = d(s, v)$, (ii) $|d(s, u) - d(s, v)| = 1$ and (iii) $|d(s, u) - d(s, v)| > 1$ (in case (ii) and (iii), let us assume that $d(s, u) < d(s, v)$ without loss of generality). We recall that d is the distance *before* the edge insertion.

In the first case, it is easy to see that the insertion does not affect any shortest path rooted in s, and therefore nothing needs to be updated for s.

In case (ii), the distance between s and the other nodes is not affected, since there already existed an alternative shortest-path from s to v. However, the insertion creates new shortest paths from s to to v and consequently to all the nodes t in the sub-DAG (of the SSSP DAG from s) rooted in v. To account for this, for each of these nodes t, we add $\sigma_{su} \cdot \sigma_{vt}$ to the old value of σ_{st} (where $\sigma_{su} \cdot \sigma_{vt}$ is the number of new shortest paths between s and t going through (u, v)).

Finally, in case (iii), a part of the sub-DAG rooted in v might get closer to s. This case is handled with a BFS traversal rooted in v. In the traversal, all neighbors y of nodes x extracted from the BFS queue are examined and all the ones such that $d(s, y) \geq d'(s, x)$ are also enqueued. For each traversed node y, the new distance $d'(s, y)$ is computed as $\min_{z:(z,y)\in E} d'(s, z) + 1$ and the number of shortest paths σ'_{sy} as $\sum_{z\in P'_s(y)} \sigma_{sz}$.

5.2.2 *Algorithm by Kas et al. (KWCC)*

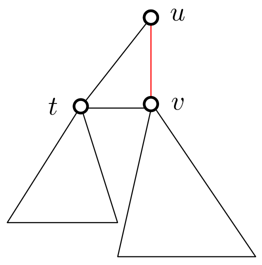


Figure 12: Insertion of (u, v).

KWCC updates the augmented APSP based on a dynamic APSP algorithm by Ramalingam and Reps [104]. Instead of checking for each source s whether the new edge (or the weight decrease) changes the SSSP DAG rooted in s, KWCC first identifies the *affected sources* $S = \{s : d(s, v) \geq d(s, u) + \omega'(u, v)\}$. These are exactly the nodes for which there is some change in the SSSP DAG. The affected sources are identified by running a pruned BFS rooted in u on G transposed (i.e. the graph obtained by reversing the direction of edges in G). For each node s traversed in the BFS, KWCC checks whether the neighbors of s are also affected sources and, if not, it does not continue the traversal from them. Notice that even on weighted graphs, a (pruned) BFS is sufficient since we already know all distances to v and we can basically sidestep the use of a priority queue.

Once all affected sources s are identified, KWCC starts a pruned BFS rooted in v for each of them. In the pruned BFS, only nodes t such that $d(s, t) \geq d(s, u) + \omega'(u, v) + d(v, t)$ are traversed (the *affected targets* of s). The new distance $d'(s, t)$ is set to $d(s, u) + \omega'(u, v) + d(v, t)$ and the new number of shortest paths $\sigma'(s, t)$ is set to $\sum_{z\in P'_s(t)} \sigma_{sz}$ as in KDB. Compared to KDB, the augmented APSP update of KWCC requires fewer operations. First, it efficiently identifies the affected sources instead of checking all nodes. Second, in case (iii), KDB might traverse more nodes than KWCC. For example, assume (u, v) is a new edge and the resulting SSSP DAG of u is as in Figure 12. Then, KWCC will prune the BFS in t, since $d(u, t) < d(u, v) + d(v, t)$, skipping all the SSSP DAGs rooted in t. On the contrary, KDB will traverse the whole subtree rooted in t, although neither the distances nor the number of shortest paths from u to those nodes are affected. The reason for this will be made clearer in Section 5.3.1.

5.2.3 *Faster augmented APSP update*

To explain our idea for improving the APSP update step, let us start with an example, shown in Figure 13. The insertion of (u, v) decreases the distance from nodes x_1, x_2, u to all the nodes shown in green. KWCC would first identify the affected sources $S = \{x_1, x_2, u\}$ and, for each of them, run a pruned BFS rooted in v. This means we are repeating almost exactly the same procedure for each of the affected sources. We clearly have to update the distances and number of shortest paths between each affected source and the affected targets (and this cannot be avoided). However, KWCC also goes through the outgoing edges of each affected target multiple times, leading to a worst-case running time of $O(mn)$. Notice that this is true also for KDB, with the difference that KDB starts a BFS from each node instead of first identifying the affected sources and that it also visits additional nodes. Our basic idea is to avoid this redundancy and is based on the following proposition (a similar result was proven also in [82]).

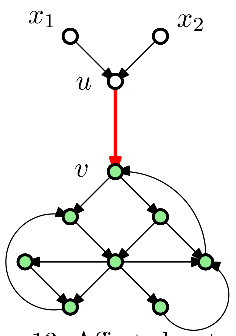


Figure 13: Affected targets (in green) and affected sources (x_1, x_2, u).

Proposition 5.2.1. *Let $t \in V$ and $y \in P_v(t)$ be given. Then, $S(t) \subseteq S(y)$.*

Proof. Let s be any node in $S(t)$, i.e. either $d'(s,t) = d(s,t)$ and $\sigma'_{st} \neq \sigma_{st}$ (case (i)), or $d'(s,t) < d(s,t)$ (case (ii)). We want to show that $s \in S(y)$.

Before proving this, we show that y has to be in $P'_s(t)$. In fact, if $s \in S(t)$, there have to be shortest paths between s and t going through (u, v), i.e. $d'(s,t) = d(s,u) + \omega'(u,v) + d(v,t)$. On the other hand, we know $y \in P_v(t)$ and thus

$$d'(s,t) = d(s,u) + \omega'(u,v) + d(v,y) + \omega(y,t). \tag{16}$$

Now, $d(s,u) + \omega'(u,v) + d(v,y)$ cannot be larger than $d'(s,y)$, or this would mean that $d'(s,t) > d'(s,v) + \omega(y,t)$, which contradicts the triangle inequality. Also, $d(s,u) + \omega'(u,v) + d(v,y)$ cannot be smaller than $d'(s,y)$ by definition of distance. Thus, $d'(s,y) = d(s,u) + \omega'(u,v) + d(v,y)$. If we substitute this in Eq. (16), we obtain $d'(s,t) = d'(s,y) + \omega(y,t)$, which means $y \in P'_s(t)$.

Now, let us consider case (i). We have two options: either y was a predecessor of t from s also before the edge update, i.e. $y \in P_s(t)$, or it was not. If it was not, it means $d(s,y) + \omega(y,t) > d(s,t) = d'(s,t) = d'(s,y) + \omega(y,t)$, which implies $d(s,y) > d'(s,y)$ and thus $s \in S(y)$. If it was, we can similarly show that $d(s,y) = d'(s,y)$. Since we have seen before that $d'(s,y) = d(s,u) + \omega'(u,v) + d(v,y)$, there has to be at least one new shortest path from s to y in G' going through (u,v), which means $\sigma'_{sy} > \sigma_{sy}$ and therefore $s \in S(y)$.

Case (ii) can be easily proven by contradiction. We know $d(s,t) \leq d(s,y) + \omega(y,t)$ (by the triangle inequality) and that $\omega'(y,t) = \omega(y,t)$. Thus, if it were true that $d(s,y) = d'(s,y)$ then

$$d(s,t) \leq d(s,y) + \omega(y,t) = d'(s,y) + \omega(y,t) = d'(s,t), \tag{17}$$

which contradicts our hypothesis that $d'(s,t) < d(s,t)$ (case (ii)). Thus, $d(s,y) \neq d'(s,y)$. Since pairwise distances in G' can only be equal to or shorter than pairwise distances in G, $d(s,y) \neq d'(s,y)$ implies $d(s,y) > d'(s,y)$ and thus $s \in S(y)$.

□

In particular, this implies that $S(t) \subseteq S(v)$ for each $t \in T(u)$. Consequently, it is sufficient to compute $S(v)$ and $T(u)$ once via two pruned BFSs. Our approach is described in Algorithm 1. The pruned BFS to compute $S(v)$ is performed in Line 3. Then, a pruned BFS from v is executed, whereby for each $t \in T(u)$ we store one of its predecessors $p(t)$ in the BFS (Line 27).

Let $d^\star(s,t)$ be the length of a shortest path between s and t going through (u,v), i.e. $d^\star(s,t) := d(s,u) + \omega'(u,v) + d(v,t)$. To finally compute $S(t)$ all that is left to do is to test whether $d^\star(s,t) \leq d(s,t)$ for each $s \in S(p(t))$ once we remove t from the queue (Lines 11 - 22). Note that this implies that $S(p(t))$ was already computed. In case $d^\star(s,t) < d(s,t)$, the path from s to t via edge (u,v) is shorter than before and therefore we set $d'(s,t)$ to $d^\star(s,t)$ and σ'_{st} to $\sigma_{su} \cdot \sigma_{vt}$, since all new shortest paths now go through (u,v)). Also in case of equality ($d^\star(s,t) = d(s,t)$), s is in $S(t)$, since its number of shortest paths has changed. Consequently we set σ'_{st} to $\sigma_{st} + \sigma_{su} \cdot \sigma_{vt}$ (since in this case also old shortest paths are still valid). If $d^\star(s,t) > d(s,t)$, the edge (u,v) does not lie on any shortest path from s to t, hence $s \notin S(t)$ (and s is not added to $S(t)$ in Lines 18 - 20).

5.3 DYNAMIC DEPENDENCY ACCUMULATION

After updating distances and number of shortest paths, dynamic algorithms need to update the betweenness scores. This means increasing the score of all nodes that lie in new shortest paths, but also decreasing that of nodes that used to be in old shortest paths between affected nodes. Again, we will first see how KDB and KWCC update the dependencies and then we will present our new approach in Section 5.3.3.

5.3.1 *Algorithm by Kourtellis et al. (KDB)*

In addition to d and σ, KDB keeps track of the old dependencies $\delta_{s\bullet}(v)$ $\forall s, v \in V$. The dependency update is done in a way similar to BA (see Section 4.2). Also in this case, nodes v are processed in decreasing order of their new distance $d'(s,v)$ from s (otherwise it would not be possible to apply Theorem (4.2.1)). However, in this case we would only like to process nodes for which the dependency has actually changed. To do this, while still making sure that the nodes are processed in the right order, KDB replaces the stack used in BA with a bucket list. Every node that is traversed during the APSP update is inserted into the bucket list in a position equal to its new distance from s. Then, nodes are extracted from the bucket list starting from the ones with maximum distance. Every time a node v is extracted, we compute its new dependency as $\delta'_{s\bullet}(v) = \sum_{w: v \in P'_s(w)} \frac{\sigma'_{sv}}{\sigma'_{sw}}(1 + \delta'_{s\bullet}(w))$. Since

Algorithm 1: Augmented APSP update

```
Input  : Graph $G = (V, E)$, edge insertion/weight decrease $(u, v, \omega'(u, v))$, $d(s, t)$,
         $\sigma_{st},\ \forall (s, t) \in V^2$
Output : Updated $d'(s, t)$, $\sigma'_{st},\ \forall (s, t) \in V^2$
Assume : Initially $d'(s, t) = d(s, t)$ and $\sigma'_{st} = \sigma_{st}\ \ \forall (s, t) \in V^2$
$vis(v) \leftarrow$ false $\forall v \in V$;
if $\omega'(u, v) \leq d(u, v)$ then
    $S(v) \leftarrow$ findAffectedSources$(G, (u, v, \omega'(u, v)))$;
    $d(u, v) \leftarrow \omega'(u, v)$;
    $Q \leftarrow \emptyset$;
    $p(v) \leftarrow v$;
    $Q.push(v)$;
    $vis(v) \leftarrow$ true;
    while $Q.length() > 0$ do
        $t = Q.front()$;
        foreach $s \in S(p(t))$ do
            if $d(s, t) \geq d(s, u) + \omega'(u, v) + d(v, t)$ then
                if $d(s, t) > d(s, u) + \omega'(u, v) + d(v, t)$ then
                    $d'(s, t) \leftarrow d(s, u) + \omega'(u, v) + d(v, t)$;
                    $\sigma'_{st} \leftarrow 0$;
                end
                $\sigma'_{st} \leftarrow \sigma'_{st} + \sigma_{su} \cdot \sigma_{vt}$;
                if $t \neq v$ then
                    $S(t).insert(s)$;
                end
            end
        end
        foreach $w$ s.t. $(t, w) \in E$ do
            if not $vis(w)$ and $d(u, w) \geq \omega'(u, v) + d(v, w)$ then
                $Q.push(w)$;
                $vis(w) \leftarrow$ true;
                $p(w) \leftarrow t$;
            end
        end
    end
end
```

we are processing the nodes in order of decreasing new distance, we can be sure that $\delta'_{s\bullet}(v)$ is computed correctly. The score of v is then updated by adding the new dependency $\delta'_{s\bullet}(v)$ and subtracting the old $\delta_{s\bullet}(v)$, which was previously stored. Also, all neighbors $y \in P'_s(v)$ that are not in the bucket list yet are inserted at level $d'(s, y) = d'(s, v) - 1$. Notice that, in the example in Figure 12, all the nodes in the sub-DAG of t are necessary to compute the new dependency of t, although they have not been affected by the insertion. This is why they are traversed during the APSP update.

5.3.2 *Algorithm by Kas et al. (KWCC)*

KWCC does not store dependencies. On the contrary, for every node pair (s, t) for which either $d(s, t)$ or σ_{st} has been affected by the insertion, all the nodes in the new shortest paths and the ones in the old shortest paths between s and t are processed. More specifically, starting from t, all the nodes $y \in P'_s(t)$ are inserted into a queue. When a node y is extracted, we increase its betweenness by $\sigma'(s, y) \cdot \sigma'(y, t)/\sigma'(s, t)$ (i.e. the fraction of shortest paths between s and t going through y). Then, also y enqueues all nodes in $P'_s(y)$

and the process is repeated until we reach s. Decreasing the betweenness of nodes in the old paths is done in a similar fashion, with the only difference that nodes in $P_s(y)$ are enqueued (instead of nodes in $P'_s(y)$) and that $\sigma(s,y) \cdot \sigma(y,t)/\sigma(s,t)$ is subtracted from the scores of processed nodes. Notice that the worst-case complexity of this approach is $O(n^3)$, whereas that of KDB is $O(nm)$. This cubic running time is due to the fact that, for each affected node pair (s,t) (at most $\Theta(n^2)$), there could be up to $\Theta(n)$ nodes lying in either one of the old or new shortest paths between s and t. (In the running time analysis of [72], this is represented by the term $|\sigma_{old}|I$.) This means that, if many nodes are affected, KWCC can even be slower than recomputation with BA. On the other hand, we have seen in Section 5.2.2 that KDB also processes nodes for which the betweenness has not changed (see Figure 12 and its explaination), which in some cases might result in a higher running time than KWCC.

5.3.3 *Faster betweenness update*

We propose a new approach for updating the betweenness scores. As KWCC, we do not store the old dependencies (resulting in a lower memory requirement) and we only process the nodes whose betweenness has actually been affected. However, we do this by accumulating contributions of nodes only once for each affected source, in a fashion similar to KDB. For an affected source $s \in S$ and for any node $v \in V$, let us define $\Delta_{s,\bullet}(v)$ as $\sum_{t \in T(s)} \sigma_{st}(v)/\sigma_{st}$. This is the contribution of nodes whose old shortest paths from s went through v, but which have been affected by the edge insertion. Analogously, we can define $\Delta'_{s,\bullet}(v)$ as $\sum_{t \in T(s)} \sigma'_{st}(v)/\sigma'_{st}$. Then, the new dependency $\delta'_{s,\bullet}(v)$ can be expressed as:

$$\delta'_{s,\bullet}(v) = \delta_{s,\bullet}(v) - \Delta_{s,\bullet}(v) + \Delta'_{s,\bullet}(v) \tag{18}$$

Notice that for all nodes $t \notin T(s)$, $\sigma'_{st} = \sigma_{st}$ and $\sigma'_{st}(v) = \sigma_{st}(v)$, therefore their contribution to $\delta_{s,\bullet}(v)$ is not affected by the edge update. The new betweenness $c'_B(v)$ can then be computed as $c_B(v) - \sum_{s \in S} \Delta_{s,\bullet}(v) + \sum_{s \in S} \Delta'_{s,\bullet}(v)$. The following theorem allows us to compute $\Delta_{s,\bullet}(v)$ and $\Delta'_{s,\bullet}(v)$ efficiently.

Theorem 5.3.1. *For any $s \in T, v \in V$:*

$$\Delta_{s,\bullet}(v) = \sum_{w: v \in P_s(w) \wedge w \in T(s)} \sigma_{sv}/\sigma_{sw}(1 + \Delta_{s,\bullet}(w)) + \sum_{w: v \in P_s(w) \wedge w \notin T(s)} \sigma_{sv}/\sigma_{sw} \cdot \Delta_{s,\bullet}(w) \ .$$

Similarly:

$$\Delta'_{s,\bullet}(v) = \sum_{w: v \in P'_s(w) \wedge w \in T(s)} \sigma'_{sv}/\sigma'_{sw}(1 + \Delta'_{s,\bullet}(w)) + \sum_{w: v \in P'_s(w) \wedge w \notin T(s)} \sigma'_{sv}/\sigma'_{sw} \cdot \Delta'_{s,\bullet}(w) \ .$$

Proof. We prove only the equation for $\Delta_{s,\bullet}(v)$, the one for $\Delta'_{s,\bullet}(v)$ can be proven analogously. Let t be any node in $T(s)$, $t \neq v$. Then, the term $\sigma_{st}(v)/\sigma_{st}$ can be rewritten as $\sum_{w: v \in P_s(w)} \sigma_{st}(v,w)/\sigma_{st}$, where $\sigma_{st}(v,w)$ is the number of shortest paths between s and t going through both v and w. Then:

$$\Delta_{s,\bullet}(v) = \sum_{t \in T(s)} \sigma_{st}(v)/\sigma_{st} = \sum_{t \in T(s)} \sum_{w: v \in P_s(w)} \sigma_{st}(v,w)/\sigma_{st} = \sum_{w: v \in P_s(w)} \sum_{t \in T(s)} \sigma_{st}(v,w)/\sigma_{st} \ .$$

Now, of the σ_{sw} paths from s to w, there are σ_{sv} many that also go through v. Therefore, for $t \neq w$, there are $\frac{\sigma_{sv}}{\sigma_{sw}} \cdot \sigma_{st}(w)$ shortest paths from s to t containing both v and w, i.e. $\sigma_{st}(v, w) = \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \sigma_{st}(w)$. On the other hand, if $t = w$, $\sigma_{st}(v, w)$ is simply σ_{sv}. Therefore, we can rewrite the equation above as:

$$\begin{aligned}
&\sum_{w: v \in P_s(w) \wedge w \in T(s)} \left\{ \frac{\sigma_{sv}}{\sigma_{sw}} + \sum_{t \in T(s) - \{w\}} \frac{\sigma_{st}(v,w)}{\sigma_{st}} \right\} + \sum_{w: v \in P_s(w) \wedge w \notin T(s)} \sum_{t \in T(s)} \frac{\sigma_{st}(v,w)}{\sigma_{st}} \\
&= \sum_{w: v \in P_s(w) \wedge w \in T(s)} \frac{\sigma_{sv}}{\sigma_{sw}} \left(1 + \sum_{t \in T(s) - \{w\}} \frac{\sigma_{st}(w)}{\sigma_{st}} \right) + \sum_{w: v \in P_s(w) \wedge w \notin T(s)} \frac{\sigma_{sv}}{\sigma_{sw}} \sum_{t \in T(s)} \frac{\sigma_{st}(w)}{\sigma_{st}} \\
&= \sum_{w: v \in P_s(w) \wedge w \in T(s)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \Delta_{s,\bullet}(w)) + \sum_{w: v \in P_s(w) \wedge w \notin T(s)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \Delta_{s,\bullet}(w) \ .
\end{aligned}$$

□

Theorem 5.3.1 allows us to accumulate the dependency changes in a way similar to BA. To compute $\Delta_{s,\bullet}$, we need to process nodes in decreasing order of $d(s, \cdot)$, whereas to compute $\Delta'_{s,\bullet}$ we need to process them in decreasing order of $d'(s, \cdot)$. To do this, we use two priority queues PQ_s and PQ'_s (if the graph is unweighted, we can use bucket lists as the ones used in KDB). Notice that nodes w such that $\sigma_{st}(w) = 0 \wedge \sigma'_{st}(w) = 0 \ \forall t \in T(s)$ do not need to be added to the queue. PQ_s and PQ'_s are filled with all nodes in $T(s)$ during the APSP update in Algorithm 1. In PQ_s, nodes w are inserted with priority $d(s, w)$ and PQ'_s with priority $d'(s, w)$. Algorithm 2 shows how we decrease betweenness of nodes that lied in old shortest paths from s (notice that this is repeated for each $s \in S(v)$). In Lines 7 - 12, Theorem 5.3.1 is applied to compute $\Delta_{s,\bullet}(y)$ for each predecessor y of w. Then, y is also enqueued and this is repeated until PQ_s is empty (i.e. when we reach s). The betweenness update of nodes in the new shortest paths works in a very similar way. The only difference is that PQ'_s is used instead of PQ, that d' and σ' are used instead of d and σ and that $\Delta'_{s,\bullet}$ is added to c_B and not subtracted in Line 4. At the end of the update, σ is set to σ' and d is set to d'.

In undirected graphs, we can notice that $\sum_{s \in S(w)} \Delta_{s,\bullet}(w) = \sum_{t \in T(w)} \Delta_{t,\bullet}(w)$. Thus, to account also for the changes in the shortest paths between w and the nodes in $T(w)$, $2\Delta_{s,\bullet}$ is subtracted from $c_B(w)$ in Line 4 (and analogously $2\Delta'_{s,\bullet}$ is added in the update of nodes in the new shortest paths).

5.4 TIME COMPLEXITY

Let us study the complexity of our two new algorithms for updating APSP and betweenness scores described in Section 5.2.3 and Section 5.3.3, respectively. We define the *extended size* $||A||$ of a set of nodes A as the sum of the number of nodes in A and the number of edges that have a node of A as their endpoint. Then, the following holds.

Theorem 5.4.1. *The running time required by Algorithm 1 to update the augmented APSP after an edge insertion (or weight decrease)* $(u, v, \omega'(u, v))$ *is* $\Theta(||S(v)|| + ||T(u)|| + \sum_{y \in T(u)} |S(p(y))|)$, *where* $p(y)$ *can be any node in* $P_u(y)$.

Proof. The function `findAffectedSources` in Line 3 identifies the set of affected sources starting a BFS in v and visiting only the nodes $s \in S(v)$. This takes $\Theta(||S(v)||)$, since

Algorithm 2: Betweenness update for nodes in old shortest paths

```
1  $\Delta_{s,\bullet}(u) \leftarrow 0\ \forall u \in V$;
2  while $PQ_s \neq \emptyset$ do
3      $w \leftarrow PQ_s$.extractMax();
4      $c_B(w) \leftarrow c_B(w) - \Delta_{s,\bullet}(w)$;
5      foreach $y$ s.t. $(y, w) \in E$ do
6          if $y \neq s$ and $d(s, w) = d(s, y) + \omega(y, w)$ then
7              if $w \in T(s)$ then
8                  $c \leftarrow \frac{\sigma_{sy}}{\sigma_{sw}} \cdot (1 + \Delta_{s,\bullet}(w))$;
9              end
10             else
11                 $c \leftarrow \frac{\sigma_{sy}}{\sigma_{sw}} \cdot \Delta_{s,\bullet}(w)$;
12             end
13             if $y \notin PQ_s$ then
14                 Insert $y$ into $PQ_s$ with priority $d(s, y)$;
15             end
16             $\Delta_{s,\bullet}(y) \leftarrow \Delta_{s,\bullet}(y) + c$;
17         end
18     end
19 end
```

this pruned BFS visits all nodes in $S(v)$ and their incident edges. Then, the while loop of Lines 9 - 30 identifies all the affected targets $T(u)$ with a pruned BFS. This part (excluding Lines 11 - 22) requires $\Theta(||T(u)||)$ operations, since all affected targets and their incident edges are visited. In Lines 11 - 22, for each affected node $t \in T(u)$, all the affected sources of the predecessor $p(y)$ of y are scanned. This part requires in total $\Theta(\sum_{t \in T(u)} |S(p(y))|)$ operations. □

Notice that, since $|S(p(y))|$ is $O(n)$ and both $||T(u)||$ and $||S(v)||$ are $O(n+m)$, the worst-case complexity of Algorithm 1 is $O(n^2)$. To show the complexity of the dependency update described in Algorithm 2, let us introduce, for a given source node s, the set $\tau(s) := T(s) \cup \{w \in V : \Delta_{s,\bullet}(w) > 0\}$. Then, the following theorem holds.

Theorem 5.4.2. *The running time of Algorithm 2 is* $\Theta(||\tau(s)|| + |\tau(s)| \log |\tau(s)|)$ *for weighted graphs and* $\Theta(||\tau(s)||)$ *for unweighted graphs.*

Proof. In the following, we assume a binary heap priority queue for weighted graphs and a bucket list priority queue for unweighted graphs. Then, the `extractMax()` operation in Line 3 requires constant time for unweighted and logarithmic time for weighted graphs. Also, for each node extracted from PQ, all neighbors are visited in Lines 5 - 18. Therefore, it is sufficient to prove that the set of nodes inserted into (and therefore extracted from) PQ is exactly $\tau(s)$. As we said in the description of Algorithm 2, PQ is initially populated with the nodes in $T(s)$. Then, all nodes y inserted into PQ in Line 14 are nodes that lied in at least one shortest path between s and a node in $T(s)$ before the insertion. This means that there is at least one $t \in T(s)$ such that $\sigma_{st}(y) > 0$, which implies that $\Delta_{s,\bullet}(y) > 0$, by definition of $\Delta_{s,\bullet}(y)$. □

The running time necessary to increase the betweenness score of nodes such that $\Delta'_{s,\bullet} > 0$ can be computed analogously, defining $\tau'(s) = T(s) \cup \{w \in V : \Delta'_{s,\bullet}(w) > 0\}$. Overall, the running time of the betweenness update score described in Section 5.3.3 is $\Theta(\sum_{s \in S} ||\tau(s)|| + ||\tau'(s)||)$ for unweighted and $\Theta(\sum_{s \in S} ||\tau(s)|| + ||\tau'(s)|| + |\tau(s)| \log |\tau(s)| + |\tau'(s)| \log |\tau'(s)|)$

for weighted graphs. Consequently, in the worst case, this is $O(nm)$ for unweighted and $O(n(m+n\log n))$ for weighted graphs, which matches the running time of BA. For sparse graphs, this is asymptotically faster than KWCC, which requires $\Theta(n^3)$ operations in the worst case.

5.5 EXPERIMENTAL RESULTS

IMPLEMENTATION AND SETTINGS For our experiments, we implemented BA, KDB, KWCC, and our new approach, which we refer to as iBet (from Incremental Betweenness). All the algorithms were implemented in C++, building on the open-source *NetworKit* framework [119]. All codes are sequential; they were executed on a 64bit machine with 2 x 8 Intel(R) Xeon(R) E5-2680 cores at 2.7 GHz with 256 GB RAM with a single thread on a single CPU.

DATA SETS AND EXPERIMENTAL DESIGN For our experiments, we consider a set of unweighted undirected real-world networks belonging to different domains, taken from SNAP [81], KONECT [76], and LASAGNE (`piluc.dsi.unifi.it/lasagne`). We also create a weighted version of each networks, by assigning random weights to edges with mean 1.0 and standard deviation 0.1. Since KDB cannot handle weighted graphs, only KWCC and iBet are tested on them . The networks are reported in Table 5. Due to the time required by the static algorithm and the memory constraints of all dynamic algorithms ($\Theta(n^2)$), we only considered networks with up to about 30000 nodes.

To simulate real edge insertions, we remove an existing edge from the graph (chosen uniformly at random), compute betweenness on the graph without the edge and then re-insert the edge, updating betweenness with the incremental algorithms (and recomputing it with BA). A motivation for this technique can be found in Section 3.2.2. For all networks, we consider 100 edge insertions and report the average over these 100 runs.

Graph	Nodes	Edges	Type	Time BA [s]
`maldives`	1 736	2 290	street	0.57
`HC-BIOGRID`	4 039	10 321	biological network	5.42
`Mus-musculus`	4 610	5 747	biological network	2.83
`Caenor-elegans`	4 723	9 842	metabolic	5.05
`ca-GrQc`	5 241	14 484	coauthorship	4.29
`advogato`	7 418	42 892	social	14.62
`hprd-pp`	9 465	37 039	biological network	30.23
`ca-HepTh`	9 877	25 973	coauthorship	22.19
`dr-melanogaster`	10 625	40 781	biological network	41.31
`oregon1-010526`	11 174	23 409	autonomous systems	24.72
`oregon2-010526`	11 461	32 730	autonomous systems	29.70
`Homo-sapiens`	13 690	61 130	biological network	69.81
`GoogleNw`	15 763	148 585	hyperlinks	91.70
`as-caida20071105`	26 475	53 381	autonomous systems	153.62
`faroe-islands`	31 097	31 974	street	98.18

Table 5: Graphs used in the experiments and running time of the static algorithm BA.

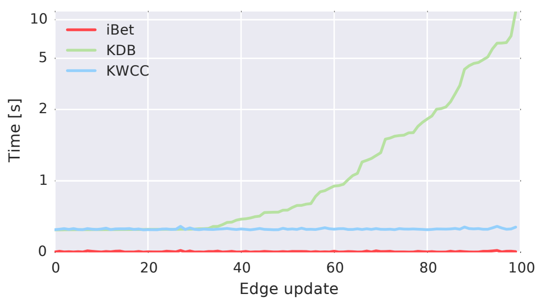
	Average Speedup			Maximum Speedup			Minimum Speedup		
Graph	iBet	KDB	KWCC	iBet	KDB	KWCC	iBet	KDB	KWCC
`maldives`	**38.2**	3.5	4.2	**2105**	26	67	**3.3**	1.6	0.1
`HC-BIOGRID`	**76.9**	14.7	10.9	**1815**	20	81	**8.9**	5.7	1.3
`Mus-musculus`	**245.3**	22.4	7.9	**56997**	9078	54	**17.0**	2.6	0.5
`Caenor-elegans`	**112.2**	15.5	8.4	**83725**	14131	91	**18.9**	1.8	1.1
`ca-GrQc`	**1282.0**	75.7	30.2	**78549**	10720	63	**29.0**	1.9	0.7
`advogato`	**72.0**	18.4	13.3	**206532**	23565	108	**9.7**	1.2	1.9
`hprd-pp`	**188.4**	16.7	28.5	**395032**	52012	131	**22.2**	1.1	2.8
`ca-HepTh`	**616.8**	25.0	32.9	**274505**	24027	90	**38.1**	1.2	0.3
`dr-melanogaster`	**117.3**	14.1	16.4	**309409**	25927	92	**23.9**	1.5	1.7
`oregon1-010526`	**72.5**	10.5	10.4	**9045**	419	67	**11.3**	1.3	0.8
`oregon2-010526`	**68.5**	10.9	14.6	**320208**	2774	86	**20.5**	1.2	1.7
`Homo-sapiens`	**106.3**	18.3	15.2	**417723**	34666	98	**11.4**	1.3	1.8
`GoogleNw`	**276.9**	152.2	26.5	**87397**	3506	137	**6.3**	0.6	1.4
`as-caida20071105`	**41.0**	9.0	6.2	**3023**	38	34	**12.2**	1.0	0.5
`faroe-islands`	**153.8**	6.8	1.6	**225266**	6003	38	**1.7**	1.5	0.0005

Table 6: The table shows the average, maximum and 75-th percentile of the speedups of the incremental algorithms on BA. The best result is shown in bold font.

EXPERIMENTAL RESULTS Table 5 shows the running times of BA for each graph and Table 6 reports the speedups of the three incremental algorithms on BA. Notice that with the term *speedup* we mean the ratio between the running times of two algorithms and not the parallel speedup (all implementations are sequential). Our new method iBet clearly outperforms the other two approaches and is always faster than both of them. On average, iBet is faster than BA by a factor 179.1, whereas KDB by a factor 13.0 and KWCC by a factor 22.9.

Figure 14 compares the APSP update (on the left) and dependency update (on the right) steps for the `oregon1-010526` graph (a similar behavior was observed also for the other graphs of Table 5. On the left, the running time of the APSP update phase of the three incremental algorithms on 100 edge insertions are reported, sorted by the running time taken by KDB. It is clear that the APSP update of iBet is always faster than the competitors. This is due to the fact that iBet processes the edges between the affected targets only once instead of doing it once for each affected source as both KDB and KWCC. Also, the running time of the APSP update of KDB varies significantly. On about one third of the updates, it is basically as fast as KWCC. This means that in these cases, KDB only visits a small amount of nodes in addition to the affected ones (see Figure 12 and its explanation). However, in other cases KDB can be much slower, as shown in the figure.

On the right of Figure 14, the running times of the dependency update step are reported. Also for this step, iBet is faster than both KDB and KWCC. However, for this part there is not a clear winner between KWCC and KDB. In fact, in some cases KDB needs to process additional nodes in order to recompute dependencies, whereas KWCC only processes nodes in the shortest paths between affected nodes. However, KDB processes each node at most once for each source node s, whereas KWCC might process the same node several times if it lies in several shortest paths between s and other nodes (we recall that the worst-case



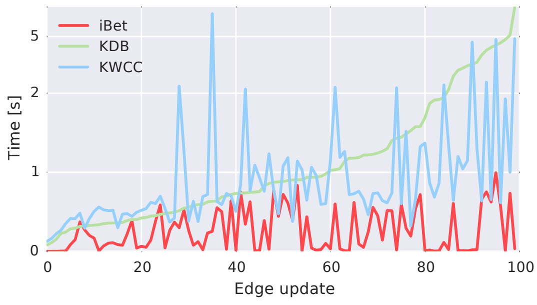
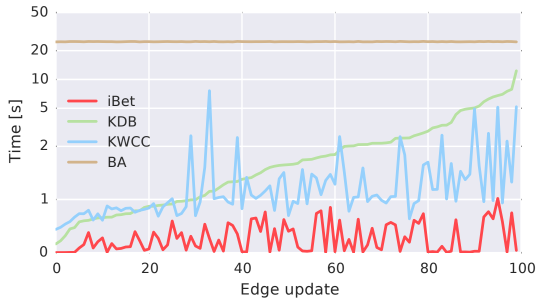


Figure 14: Running times of iBet, KDB and KWCC for 100 edge updates on `oregon1-010526`. Left: times for the APSP update step. Right: times for the dependency update step.



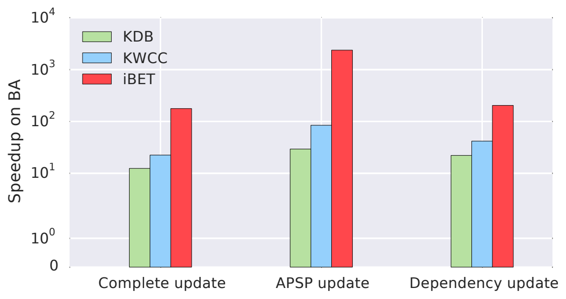


Figure 15: Left: Running times of iBet, KDB, KWCC and BA on the `oregon1-010526` graph for 100 edge updates. Right: Average speedups on recomputation with BA (geometric mean) over all networks of Table 5 for the three incremental algorithms. The column on the left shows the speedup of the complete update, the one in the middle the speedup of the APSP update only and the one on the right the speedup of the dependency update only.

running time of KWCC is $O(n^3)$, whereas that of KDB is $O(nm)$). Notice also that in some rare cases KDB is slightly faster than iBet in the dependency update. This is probably due to the fact that our implementation of iBet is based on a priority queue, whereas KDB on a bucket list.

Figure 15 on the left reports the total running times of iBet, KDB, KWCC and BA on `oregon1-010526`. Although the running times vary significantly among the updates, iBet is always the fastest among all algorithms. On the contrary, there is not always a clear winner between KDB and KWCC. On the right, Figure 15 shows the geometric mean of the speedups on recomputation for the three incremental algorithms, considering the complete update, the APSP update step only and the dependency update step only, respectively. iBet is the method with the highest speedup both overall and on the APSP update and dependency update steps separately, meaning that each of the improvements described in Section 5.2.3 and Section 5.3.3 contribute to the final speedup. On average, iBet is a factor 82.7 faster than KDB and a factor 28.5 faster than KWCC on the APSP update step and it is a factor 9.4 faster than KDB and a factor 4.9 faster than KWCC on the dependency update step. Overall, the speedup of iBet on KDB ranges from 6.6 to 29.7 and is on average (geometric mean of the speedups) 14.7 times faster. The average speedup on KWCC is 7.4, ranging from a factor 4.1 to a factor 16.0.

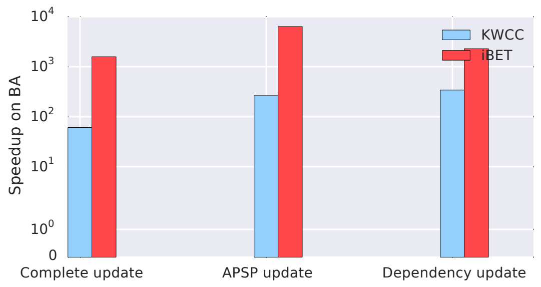


Figure 16: Average speedups on recomputation with BA (geometric mean) over all networks of Table 5, with random weights. The column on the left shows the speedup of the complete update, the one in the middle the speedup of the APSP update only and the one on the right the speedup of the dependency update only.

The results for weighted networks (reported in Figure 16) are even better: the average speedup of iBet on recomputation is 1582, whereas that of KWCC is 61, meaning that iBet is about 26 times faster than KWCC on average (we recall that KDB does not work for weighted graph).

BIBLIOGRAPHIC NOTES

The results presented in this chapter have been published as “Faster betweenness centrality updates in evolving networks” (coauthored with Henning Meyerhenke, Mark Ortmann, and Arie Slobbe) at the *Sixteenth International Symposium on Experimental Algorithms* (SEA 2017).

6

DYNAMIC SINGLE-NODE BETWEENNESS CENTRALITY

6.1 INTRODUCTION

The dynamic algorithms described in Section 4.4, as well as the new algorithm iBet presented in Chapter 5, update the betweenness centrality scores of *all* nodes after an edge insertion. However, there might be cases in which only the betweenness centrality of a specific node needs to be computed (and updated, in a dynamic scenario).

As a motivation, we consider the problem of Maximum Betweenness Improvement (MBI), initially introduced by Crescenzi et al. [42]. MBI can be described as follows: assuming that a node v in an unweighted graph G can connect itself with k other nodes, how should these nodes be chosen in order to maximize the betweenness centrality of v? In other terms, we want to add a set of k edges to the graph (all incident to v), such that the betweenness of v in the new graph is as high as possible. This problem can be of interest in all contexts in which having a high betweenness can be beneficial for a node. For example, in the field of transportation network analysis, betweenness centrality was shown to be positively related to the efficiency of an airport [88]. In the context of social networks, the authors of [87] show experimentally that nodes with high betweenness are also very effective in spreading influence to other nodes. Therefore, it might be interesting for a user to create new links with other users or pages in order to increase his own influence spread.

Crescenzi et al. [42] propose a greedy algorithm (Greedy), guaranteed to find a solution that is a $(1 - \frac{1}{e})$-approximation of the optimum. For k times, the algorithm tries to insert into G all possible edges incident to v. For each edge, Greedy computes the betweenness score of v in the new graph containing the edge, and chooses the edge that maximizes it. Algorithm 6 shows the pseudocode of Greedy. Although very accurate, this algorithm needs to compute betweenness centrality $O(k \cdot n)$ times (the number of iterations times the number of potential edges to be added to v), leading to an overall running time of $O(k \cdot n^2 m)$ – clearly too expensive for large networks.

A simple idea would be to use iBet, the dynamic algorithm presented in Chapter 5, to update betweenness after each insertion. However, Greedy requires only the betweenness of v to be updated, so iBet might perform some unnecessary work. In this chapter we investigate whether other techniques can be even faster in the context of updating the betweenness of a single node, and we apply them to speed up the greedy algorithm for MBI by Crescenzi et al. [42]. However, we would like to point out that the algorithm we propose can be applied to more general problems than updating betweenness within Greedy. Indeed, in MBI all new edges are incident to one specific node, whereas our algorithm can handle insertions of edges anywhere in the graph. In addition, the algorithm also works for edge weight decreases in weighted graphs, whereas Greedy can only deal with edge insertions in unweighted graphs.

6.2 DYNAMIC BETWEENNESS FOR A SINGLE NODE

To understand why an algorithm specifically designed to update the betweenness of a single node might be more efficient than algorithms that update betweenness of all nodes, consider the example shown in Figure 17. The insertion of an edge (u, v) does not only affect the betweenness of the nodes lying in the new shortest paths, but also that of the nodes lying in the old shortest paths between affected sources and affected targets (represented in red). Indeed, the fraction of shortest paths going through these nodes – and therefore their betweenness – has decreased as a consequence of the new insertion. Therefore, algorithms for updating the betweenness of all nodes have to walk over each old shortest path between node pairs whose distance (or number of shortest paths) has changed. However, if we are only interested in the betweenness of one particular node x, we can simply update the distances (and number of shortest paths) and check which of these updates affect the betweenness of x.

In the following, we assume the reader to be familiar with the algorithm iBet described in Chapter 5 and we describe how this can be adapted to update the betweenness of a single node. Also, we assume that the graph G is unweighted and connected, but the algorithm can be easily extended to weighted and disconnected graphs, similarly to iBet. Our algorithm can be divided in two phases: an *initialization* phase, where pairwise distances, the betweenness of a given node x and additional information are computed and stored, and an *update* phase, where the data structures are updated after the edge insertion.

6.2.1 *Initialization*

We recall from Chapter 5 that iBet stores the pairwise distances $d(s, t)$ and the number of shortest paths σ_{st} for each $s, t \in V$. In addition to this, in this case we also store the number $\sigma_{st}(x)$ of shortest paths between s and t that go through x. Then, we can compute betweenness by using its definition, i. e. $c_B(x) = \sum_{s \neq x \neq t} \frac{\sigma_{st}(x)}{\sigma_{st}}$.

The initialization can be easily done by running an augmented Single-Source Shortest Path (SSSP) from each node, as in the first phase of Brandes's algorithm for betweenness centrality [31] (see Chapter 5 for a brief description of Brandes's algorithm). While computing distances from a source node s to any other node t, we set the number σ_{st} of shortest paths between s and t to the sum $\sum \sigma_{sp}$ over all predecessors p in the shortest

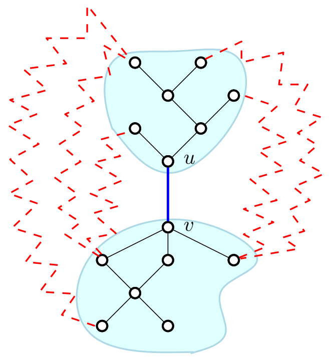


Figure 17: Insertion of (u, v) affects the betweenness of nodes lying in the old shortest paths (red).

paths from s (and we set $\sigma_{ss} = 1$). This can be done for a node s in $O(m)$ in unweighted graphs and in $O(m + n \log n)$ in weighted graphs (the cost of running a BFS or Dijkstra, respectively). Instead of discarding this information after each SSSP as in Brandes's algorithm, we store the distances $d(s,t)$ and the numbers of shortest paths σ_{st} $\forall s,t \in V$ in two matrices. Then, we can compute the number $\sigma_{st}(x)$ of shortest paths going through x. For each node pair (s,t), $\sigma_{st}(x)$ is equal to $\sigma_{sx} \cdot \sigma_{xt}$ if $d(s,t) = d(s,x) + d(x,t)$, and to 0 otherwise. The betweenness $c_B(x)$ of x can then be computed using the definition. This second part can be done in $O(n^2)$ time by looping over all node pairs. Therefore the total running time of the initialization is $O(nm)$ for unweighted graphs and $O(n(m + n \log n))$ for weighted graphs, and the memory requirement is $O(n^2)$, since we need to store three matrices of size $n \times n$ each.

6.2.2 *Update*

The update works in a way similar to iBet (see Chapter 5), with a few differences. Let us assume that an edge (u,v) with weight ω'_{uv} has been inserted into the graph, or that its old weight ω_{uv} has been decreased to a new value ω'_{uv} (for simplicity, from now on we will refer to "edge insertions", but the results apply to edge weight decreases as well). Algorithm 3 gives an overview of the algorithm for betweenness update for a single node x, whereas Algorithm 4 and Algorithm 5 describe the update of σ_{st} and $\sigma_{st}(x)$ when $d(s,t) > d(s,u) + \omega'_{uv} + d(v,t)$ and when $d(s,t) = d(s,u) + \omega'_{uv} + d(v,t)$, respectively.

Algorithm 3 shares its structure with iBet. In Lines 2-8, after setting the new distance between u and v, also σ_{uv} and $\sigma_{uv}(x)$ are updated with either updateSigmaGR or updateSigmaEQ. Then, just like in iBet, the affected sources are identified with a pruned BFS rooted in u on G transposed (function findAffectedSources).

Then, a (pruned) BFS rooted in v is started to find the affected targets for u (Lines 14-40). In Lines 36-40, the neighbors w of the affected target t are visited and, if they are also affected (i.e. $d(u,w) \geq \omega'_{uv} + d(v,w)$), they are enqueued. Also, t is stored as the predecessor of w (Line 40). In Lines 17-32, for each affected node pair (s,t), we first subtract the old contribution $\sigma_{st}(x)/\sigma_{st}$ from the betweenness of x, then we recompute $d(s,t)$, σ_{st} and $\sigma_{st}(x)$ with either updateSigmaGR or updateSigmaEQ, and finally we add the new contribution $\sigma'_{st}(x)/\sigma'_{st}$ to $c_B(x)$. Notice that, if x did not lie in any shortest path between s and t before the edge insertion, $\sigma_{st}(x) = 0$ and therefore $c_B(x)$ is not decreased in Line 19. Analogously, if x is not part of a shortest path between s and t after the insertion, $c_B(x)$ is not increased in Line 29.

In the following, we consider updateSigmaGR and updateSigmaEQ separately.

6.2.2.1 *UpdateSigmaGR*

Let us consider the case $d(s,t) > d(s,u) + \omega'_{uv} + d(v,t)$. In this case, all the old shortest paths between affected pairs are discarded, as they are not shortest paths any longer, and all the new shortest paths go through edge (u,v). Therefore, we can set the new number σ'_{st} of shortest paths between s and t to $\sigma_{su} \cdot \sigma_{vt}$. Since all old shortest paths should be discarded, also $\sigma'_{st}(x)$ depends only on the new shortest paths and not on whether x used to lie in some shortest paths between s and t before the edge insertion. Depending on the position of x with respect to the new shortest paths, we can define three cases, depicted in Figure 18. In Case (a) (left), x lies in one of the shortest paths between s

and u. This means that it also lies in some shortest paths between s and t. In particular, the number of these paths $\sigma'_{st}(x)$ is equal to $\sigma_{su}(x) \cdot \sigma_{vt}$. Notice that no shortest paths between s and u can be affected (see Chapter 5) and therefore $\sigma_{su}(x) = \sigma'_{su}(x)$. In Case (b) (center), x lies in one of the shortest paths between v and t. Analogously to Case 1, the new number of shortest paths between s and t going through x is $\sigma'_{st}(x) = \sigma_{su} \cdot \sigma_{vt}(x)$. Notice that Case (a) and Case (b) cannot both be true at the same time. In fact, if $d(s,u) = d(s,x) + d(x,u)$ and $d(v,t) = d(v,x) + d(x,t)$, we would have that $d'_{st} = d(s,u) + \omega'_{uv} + d(v,t) = d(s,x) + d(x,u) + \omega'_{uv} + d(v,x) + d(x,t) > d(s,x) + d(x,t)$, which is impossible, since d'_{st} is the shortest-path distance between s and t. Therefore, at least one among $\sigma_{su}(x)$ and $\sigma_{vt}(x)$ must be equal to 0. Finally, in Case (c) (right), $\sigma_{su}(x)$ and $\sigma_{vt}(x)$ are both equal to 0, meaning that x does not lie on any new shortest path between s and t. Once again, this is independent on whether x lied in an old shortest path between s and t or not. Algorithm 4 shows the computation of σ'_{st} and $\sigma'_{st}(x)$. Notice that, in the computation of $\sigma'_{st}(x)$, the first addend is greater than zero only in Case (a) and the second only in Case (b).

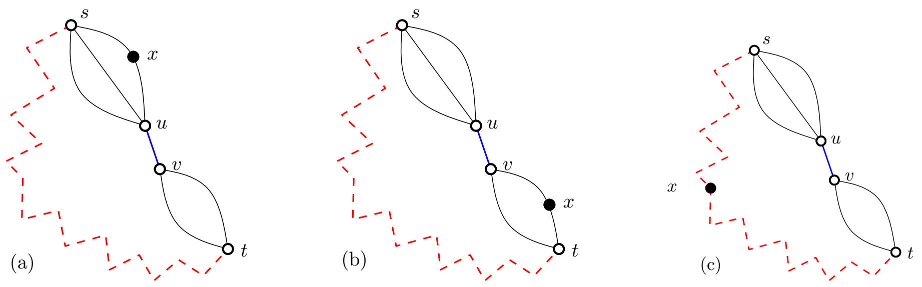


Figure 18: Possible positions of x with respect to the new shortest paths after the insertion of edge (u,v). On the left, x lies between the source s and u. In the center, x lies between v and the target t. On the right, x does not lie on any new shortest path between s and t.

6.2.2.2 *UpdateSigmaEQ*

Let us now consider the case $d(s,t) = d(s,u) + \omega'_{uv} + d(v,t)$. Here all the old shortest paths between s and t are still valid and, in addition to them, new shortest paths going through (u,v) have been created. Therefore, the new number of shortest paths σ'_{st} is simply $\sigma_{st} + \sigma_{su} \cdot \sigma_{vt}$. Notice that we never count the same path multiple times, since all new paths go through (u,v) and none of the old paths does. Also all old shortest paths between s and t through x are still valid, therefore $\sigma'_{st}(x)$ is given by the old $\sigma_{st}(x)$ plus the number of new shortest paths going through both x and (u,v). This number can be computed as described for updateSigmaGR according to the cases of Figure 18. Algorithm 5 shows the computation of σ'_{st} and $\sigma'_{st}(x)$.

6.2.3 *Time complexities*

6.2.3.1 *Dynamic betweenness algorithm*

Let us define the *extended size* $||A||$ of a set of nodes A as the sum of the number of nodes in A and the number of edges that have a node of A as their endpoint. Then, the following proposition holds.

Proposition 6.2.1. *The running time of Algorithm 3 for updating the betweenness of a single node after an edge insertion* (u, v) *is* $\Theta(||S(v)|| + ||T(u)|| + \sum_{y \in T(u)} |S(P(y))|)$.

Proof. The function `findAffectedSources` in Line 9 identifies the set of affected sources starting a BFS in v and visiting only the nodes s such that $d(s, u) + \omega'_{uv} + d(v, t) \leq d(s, t)$. This takes $\Theta(||S(v)||)$, since this partial BFS visits all nodes in $S(v)$ and their incident edges. Then, the while loop of Lines 14 - 40 (excluding the part in Lines 16 - 32) identifies all the affected targets $T(u)$ with a partial BFS. This part requires $\Theta(||T(u)||)$ operations, since all affected targets and their incident edges are visited. In Lines 16 - 32, for each affected node $t \in T(u)$, all the affected sources of the predecessor $P(t)$ of t are scanned. This part requires in total $\Theta(\sum_{t \in T(u)} |S(P(t))|)$ operations, since for each node in $S(P(t))$, Lines 17 - 32 require constant time. □

Notice that, since $S(P(y))$ is $O(n)$ and both $||T(u)||$ and $||S(v)||$ are $O(n + m)$, the worst-case complexity of Algorithm 3 is $O(n^2)$ (assuming $m = \Omega(n)$). This matches the worst-case running time of the augmented APSP update of iBet. However, notice that iBet needs a second step to update the betweenness of all nodes, which is more expensive and requires $\Theta(nm)$ operations in the worst case. Also, this introduces a contrast between the static and the incremental case: Whereas the static computation of one node's betweenness has the same complexity as computing it for all nodes (at least no algorithm for computing it for one node faster than computing it for all nodes exists so far), in the incremental case the betweenness update of a single node can be done in $O(n^2)$, whereas there is no algorithm faster than $O(nm)$ for the update of all nodes.

6.2.3.2 *Greedy algorithm for betweenness maximization*

Using the dynamic algorithm, we can improve the running time of the Greedy algorithm (Algorithm 6) for the MBI problem. In fact, in Line 4 of Algorithm 6, each edge $(x, v) \notin E$ is added to the graph and, for each inserted edge, betweenness is computed in the new graph. Computing betweenness with BA, this step requires $O(nm)$, leading to an overall complexity of $O(kn^2m)$. In Line 4, instead of computing betweenness on the new graph from scratch, we can use Algorithm 3. As we proved previously, its worst-case complexity is $O(n^2)$. This leads to an overall worst-case complexity of $O(kn^3)$ for Greedy. However, our experiments in Section 6.3 show that Greedy is actually much faster in practice.

Algorithm 3: Update of $c_B(x)$ after an edge insertion

```
Algorithm: Incremental betweenness
Input    : Graph $G = (V, E)$, edge update $(u, v, \omega'_{uv})$, pairwise distances $d$, numbers $\sigma$ of shortest
           paths, numbers $\sigma_x$ of shortest paths through $x$, betweenness value $c_B(x)$ of $x$
Output   : Updated $d$, $\sigma$, $\sigma_x$ and $c_B(x)$
Assume   : boolean $vis(v)$ is false, $\forall v \in V$
if $d(u, v) \geq \omega'_{uv}$ then
    if $d(u, v) > \omega'_{uv}$ then
        $d(u, v) \leftarrow \omega'_{uv}$;
        $\sigma_{uv}, \sigma_{uv}(x) \leftarrow$updateSigmaGR($G, (u, v), d, \sigma, \sigma_x$);
    end
    else
        $\sigma_{uv}, \sigma_{uv}(x) \leftarrow$updateSigmaEQ($G, (u, v), d, \sigma, \sigma_x$);
    end
    $S(v) \leftarrow$ findAffectedSources($G, (u, v), d$);
    $Q \leftarrow \emptyset$;
    $P(v) \leftarrow v$;
    $Q.push(v)$;
    $vis(v) \leftarrow$ true;
    while $Q.length() > 0$ do
        $t = Q.front()$;
        foreach $s \in S(P(t))$ do
            if $d(s, t) \geq d(s, u) + \omega'_{uv} + d(v, t)$ then
                if $x \neq s$ and $x \neq t$ then
                    $c_B(x) \leftarrow c_B(x) - \sigma_{st}(x)/\sigma_{st}$;
                end
                if $d(s, t) > d(s, u) + \omega'_{uv} + d(v, t)$ then
                    $\sigma_{st}, \sigma_{st}(x) \leftarrow$updateSigmaGR($G, (u, v), d, \sigma, \sigma_x$);
                    $d(s, t) \leftarrow d(s, u) + \omega'_{uv} + d(v, t)$;
                end
                else
                    $\sigma_{st}, \sigma_{st}(x) \leftarrow$updateSigmaEQ($G, (u, v), d, \sigma, \sigma_x$);
                end
                if $x \neq s$ and $x \neq t$ then
                    $c_B(x) \leftarrow c_B(x) + \sigma_{st}(x)/\sigma_{st}$;
                end
                if $t \neq v$ then
                    $S(t).insert(s)$;
                end
            end
        end
        foreach $w$ s.t. $(t, w) \in E$ do
            if not $vis(w)$ and $d(u, w) \geq \omega'_{uv} + d(v, w)$ then
                $Q.push(w)$;
                $vis(w) \leftarrow$ true;
                $P(w) \leftarrow t$;
            end
        end
    end
end
```

Algorithm 4: Update of σ_{st} and $\sigma_{st}(x)$ when (u, v) creates new shortest paths of length smaller than $d(s, t)$

```
Algorithm: UpdateSigmaGR
Input  : Graph $G = (V, E)$, edge insertion $(u, v)$, pairwise distances $d$, numbers $\sigma$ of shortest paths,
         numbers $\sigma_x$ of shortest paths through $x$
Output : Updated $\sigma'_{st}$, $\sigma'_{st}(x)$
1 $\sigma'_{st} \leftarrow \sigma_{su} \cdot \sigma_{vt}$;
2 $\sigma'_{st}(x) \leftarrow \sigma_{su}(x) \cdot \sigma_{vt} + \sigma_{su} \cdot \sigma_{vt}(x)$;
3 return $\sigma'_{st}$, $\sigma'_{st}(x)$;
```

Algorithm 5: Update of σ_{st} and $\sigma_{st}(x)$ when (u, v) creates new shortest paths of length equal to $d(s, t)$

```
Algorithm: UpdateSigmaEQ
Input  : Graph $G = (V, E)$, edge insertion $(u, v)$, pairwise distances $d$, numbers $\sigma$ of shortest paths,
         numbers $\sigma_x$ of shortest paths through $x$
Output : Updated $\sigma'_{st}$, $\sigma'_{st}(x)$
1 $\sigma'_{st} \leftarrow \sigma_{st} + \sigma_{su} \cdot \sigma_{vt}$;
2 $\sigma'_{st}(x) \leftarrow \sigma_{st}(x) + \sigma_{su}(x) \cdot \sigma_{vt} + \sigma_{su} \cdot \sigma_{vt}(x)$;
3 return $\sigma'_{st}$, $\sigma'_{st}(x)$;
```

Algorithm 6: Greedy algorithm for MBI [42].

```
Input  : A directed graph $G = (V, E)$; a vertex $x \in V$; and an integer $k \in \mathbb{N}$
Output : Set of edges $S \subseteq \{(x, v) \notin E\}$ such that $|S| \leq k$
1 $S \leftarrow \emptyset$;
2 for $i = 1, 2, \ldots, k$ do
3     foreach $v \in V : (x, v) \notin E$ do
4         Compute $c_B(x)$ in $G_v := (V, E \cup S \cup \{(x, v)\})$;
5     end
6     $v_{\max} \leftarrow \arg\max\{c_B(x)$ in $G_v\}$ over $v \in V : (x, v) \notin E$;
7     $S \leftarrow S \cup \{(x, v_{\max})\}$;
8 end
9 return $S$;
```

6.3 EXPERIMENTAL EVALUATION

In our experiments, we evaluate the running time of the dynamic algorithm for updating the betweenness of a node and of the version of Greedy based on it. All algorithms compared in our experiments are implemented in C++, building on the open-source NetworKit [119] framework. The experiments were done on a machine equipped with 256 GB RAM and a 2.7 GHz Intel Xeon CPU E5-2680 having 2 sockets with 8 cores each. To make the comparison with previous work more meaningful, we use only one of the 16 cores. The machine runs 64 bit SUSE Linux and we compiled our code with g++-4.8.1 and OpenMP 3.1.

For our experiments, we consider a set of real-world networks belonging to different domains, taken from SNAP [81], KONECT [76], and the 10th DIMACS Implementation Challenge [9]. The properties of the networks are reported in Table 7 (directed graphs) and in Table 8 (undirected graphs).

6.3.1 *Running times of the dynamic algorithm for the betweenness of one node*

In the following, we refer to our incremental algorithm for the update of the betweenness of a single node as SI (Single-node Incremental). Since there are no other algorithms specifically designed to compute or update the betweenness of a single node, we use the static algorithm by Brandes [31] and the algorithm iBet from Chapter 5 for a comparison. Indeed, the algorithm by Brandes (which here we refer to as Stat, from Static) is the best known algorithm for static computation of betweenness and iBet (which here we name AI, from All-nodes Incremental) has been shown to outperform other dynamic algorithms (see Chapter 5).

To compare the running times of the algorithms for betweenness centrality, we choose a node x at random and we assume we want to compute the betweenness of x. Then, we add an edge to the graph, also chosen uniformly at random among the node pairs (u, v) such that $(u, v) \notin E$. After the insertion, we use the three algorithms to update the betweenness centrality of x and compare their running times. We recall that Stat is a static algorithm, which means that we can only run it from scratch on the graph after the edge insertion. On each graph, we repeat this 100 times and report the average running time obtained by each of the algorithms.

Table 7 and Table 8 show the running times for directed and undirected graphs, respectively. As expected, both dynamic algorithms AI and SI are faster than the static approach and SI is the fastest among all algorithms. This is expected, since SI is the one that performs the smallest number of operations. Also, notice that the standard deviation of the running times of both AI and SI is very high, sometimes even higher than the average. This is actually not surprising, since different edge insertions might affect portions of the graph of very different sizes. Figure 19 and Figure 20 report the running times of AI and SI as a function of the number of affected node pairs for two directed and undirected graphs, respectively (similar results can be observed for the other tested graphs). As expected, the running time of both algorithms (as well as the difference between the running time of AI and that of SI) mostly increases as the number of affected pairs increases. However, AI presents a much larger deviation than SI. This is due to the fact that its running time also depends on the number of nodes that used to lie in old shortest paths between the affected pairs. Indeed, the number of nodes whose betweenness gets affected does not only depend on the number of affected pairs (which we recall to be the ones for which the edge

Graph	Nodes	Edges	Time Stat [s]	Time AI [s]	Time SI [s]	STD AI [s]	STD SI [s]
`subelj-jung`	6 120	50 535	1.25	0.0019	**0.0002**	0.0036	0.0005
`wiki-Vote`	7 115	100 762	8.18	0.0529	**0.0015**	0.0635	0.0038
`elec`	7 118	103 617	8.67	0.0615	**0.0019**	0.0858	0.0053
`freeassoc`	10 617	63 788	14.96	0.1118	**0.0034**	0.1532	0.0036
`dblp-cite`	12 591	49 728	5.04	0.1726	**0.0071**	0.7905	0.0451
`subelj-cora`	23 166	91 500	34.08	0.3026	**0.0327**	1.1598	0.1575
`ego-twitter`	23 370	33 101	8.47	0.0062	**0.0001**	0.0576	0.0003
`ego-gplus`	23 628	39 242	10.01	0.0024	**0.0001**	0.0026	0.0000
`munmun-digg`	30 398	85 247	78.09	0.2703	**0.0073**	0.2539	0.0099
`linux`	30 837	213 424	34.75	0.0692	**0.0108**	0.3019	0.0637

Table 7: Average running times of the betweenness algorithms on directed real-world graphs. The last two columns report the standard deviation of the running times of AI and SI over the 100 edge insertions.

Graph	Nodes	Edges	Time Stat [s]	Time AI [s]	Time SI [s]	SD AI [s]	SD SI [s]
`Mus-musculus`	4 610	5 747	2.87	0.0337	**0.0037**	0.0261	0.0024
`HC-BIOGRID`	4 039	10 321	5.32	0.1400	**0.0083**	0.1450	0.0119
`Caenor-eleg`	4 723	9 842	4.75	0.0506	**0.0025**	0.0406	0.0014
`ca-GrQc`	5 241	14 484	4.15	0.0377	**0.0033**	0.0245	0.0017
`advogato`	7 418	42 892	12.65	0.1820	**0.0024**	0.1549	0.0008
`hprd-pp`	9 465	37 039	29.19	0.2674	**0.0053**	0.1873	0.0021
`ca-HepTh`	9 877	25 973	21.57	0.1404	**0.0095**	0.1108	0.0053
`dr-melanog`	10 625	40 781	38.18	0.2687	**0.0067**	0.2212	0.0029
`oregon1`	11 174	23 409	23.77	0.5676	**0.0037**	0.5197	0.0020
`oregon2`	11 461	32 730	27.98	0.5655	**0.0039**	0.5551	0.0026
`Homo-sapiens`	13 690	61 130	68.06	0.5920	**0.0079**	0.4203	0.0035
`GoogleNw`	15 763	148 585	76.17	2.4744	**0.0044**	4.1075	0.0045
`CA-CondMat`	21 363	91 342	168.44	1.1375	**0.0486**	0.7485	0.0358

Table 8: Average running times of the betweenness algorithms on undirected real-world graphs. The last two columns report the standard deviation of the running times of AI and SI over the 100 edge insertions.

insertion creates a shortcut or new shortest paths), but also on how many shortest paths there used to be between the affected pairs before the insertion and how long these paths were.

Table 9 and Table 10 show the speedups of SI on AI and those of SI on Stat, for directed and undirected graphs, respectively. Although the speedups vary considerably among the networks and the edge insertions, SI is always at least as fast as AI and up to 1560 times faster (maximum speedup for `GoogleNw`). On average (geometric mean of the average speedups over the tested networks), SI is 29 times faster than AI for undirected graphs and 18 times faster for directed graphs. The high speedups on the dynamic algorithm for all nodes is due to the fact that, when focusing on a single node, we do not need to update the scores of all the nodes that lie in some shortest path affected by the edge insertion. On the contrary, for each affected source node s, AI has to recompute the change in dependencies by iterating over all nodes that lie in either a new or an old shortest path from s. As a result, SI is extremely fast: on all tested instances, its running time is always smaller than 0.05 seconds, whereas AI can take up to seconds to update betweenness.

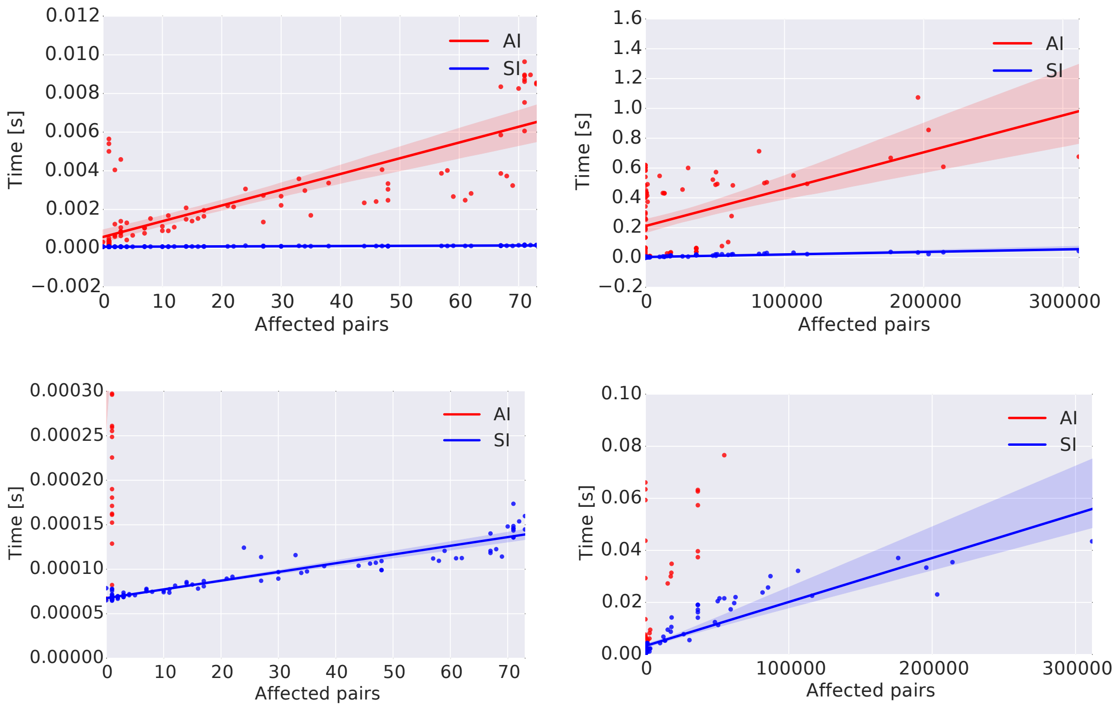


Figure 19: Top: Running time of AI and SI as a function of the number of affected node pairs for two directed graphs (left: `ego-gplus`, right: `munmun-digg`). Bottom: Same as the two plots above, but zoomed on the running times of SI. The points are the computed running times, the lines are the results of a linear regression and the area around the lines is a 95% confidence interval for the regression.

Compared to recomputation, SI is on average about 4200 times faster than Stat on directed and about 33000 times on undirected graphs (geometric means of the speedups). Since SI has shown to outperform other approaches in the context of updating the betweenness centrality of a single node after an edge insertion, we use it to update betweenness in the greedy algorithm for the Maximum Betweenness Improvement problem. Therefore, in all the following experiments, what we refer to as Greedy is Algorithm 6 where we recompute betweenness after each edge insertion with SI.

6.3.2 *Running times of the greedy algorithm for betweenness maximization*

In this section, we report the running times of Greedy, using SI to recompute betweenness. Table 11 and Table 12 show the results on directed and undirected graphs, respectively. For each value of k, the tables show the running time required by Greedy when k edges are added to the graph. Notice that this is not the running time of the kth iteration, but the total running time of Greedy for a certain value of k. Since on directed graphs the betweenness of x is a submodular function of the solutions for MBI (see [42]), we speed up the computation for $k > 1$ similarly to what has been done in [42].

Although the standard deviation is quite high, we can clearly see that exploiting submodularity has significant effects on the running times: for all graphs in Table 11, we see that the difference in running time between computing the solution for $k = 1$ and $k = 10$

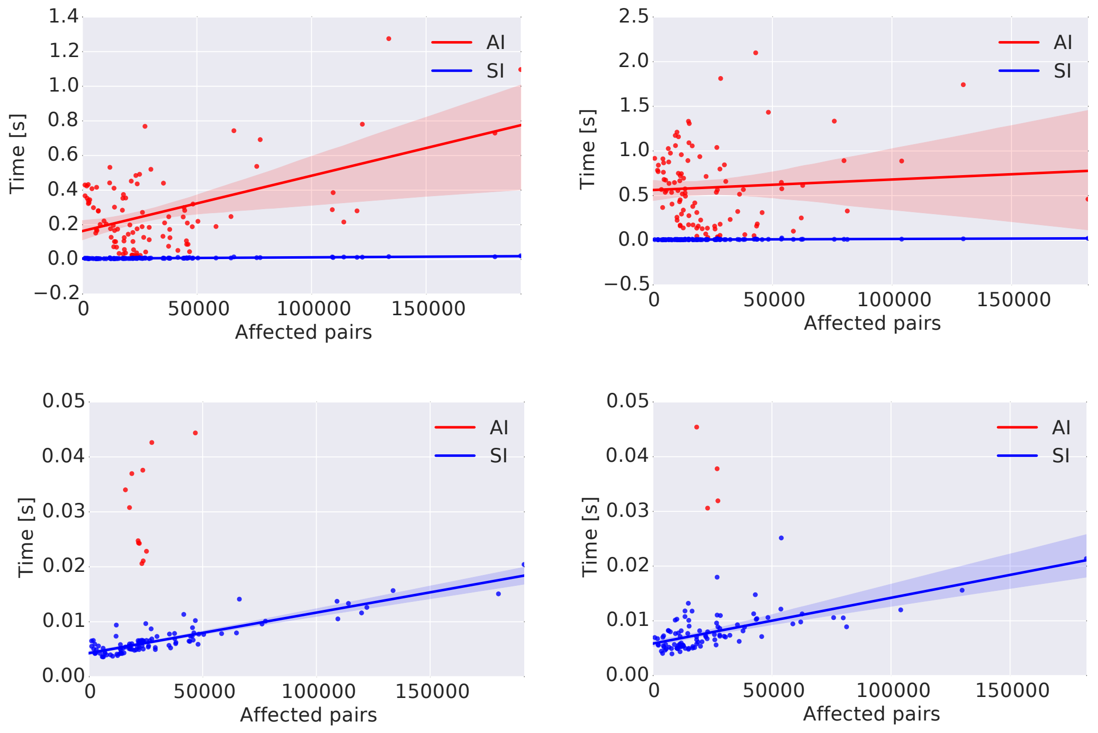


Figure 20: Top: Running time of AI and SI as a function of the number of affected node pairs for two undirected graphs (left: `dr-melanog`, right: `Homo-sapiens`). Bottom: same as the two plots above, but zoomed on the running times of SI. The points are the computed running times, the lines are the results of a linear regression and the area around the lines is a 95% confidence interval for the regression.

is at most a few seconds. Also, for all graphs the computation never takes more than a few minutes.

Unfortunately, submodularity does not hold for undirected graphs, therefore for each k we need to apply SI to all possible new edges between x and other nodes. Nevertheless, apart from the `CA-CondMat` graph (where, on average, it takes about 10 hours for $k = 10$) and `ca-HepTh` (where it takes about 1.5 hours), Greedy never requires more than 1 hour for $k = 10$. For $k = 1$, it takes at most a few minutes. Quite surprisingly, the running time of the first iteration is often smaller than that of the following ones, in particular if we consider that the first iteration also includes the initialization of SI. This might be due to the fact that, initially, the pivots are not very central and therefore many edge insertions between the pivots and other nodes affect only a few shortest paths. Since the running time of IA is proportional to the number of affected node pairs, this makes it very fast during the first iteration. On the other hand, at each iteration the pivot x gets more and more central, affecting a greater number of nodes when a new shortcut going through x is created.

To summarize, our experimental results show that the new incremental algorithm for the betweenness of one node is much faster than the algorihm iBet proposed in Chapter 5 (which is, in turn, faster than other existing incremental algorithms for the betweenness of all nodes), taking always fractions of seconds even when the others take seconds. The

	Speedups on Stat			Speedups on AI		
Graph	Geometric mean	Maximum	Minimum	Geometric mean	Maximum	Minimum
`subelj-jung`	24668.3	67477.6	342.9	10.0	63.7	1.1
`wiki-Vote`	23779.8	381357.7	275.6	39.3	310.7	1.0
`elec`	21560.5	408629.3	175.6	32.1	285.1	1.1
`freeassoc`	6783.2	330333.4	707.5	13.0	94.0	1.1
`dblp-cite`	24745.7	140950.0	11.4	13.3	314.5	1.1
`subelj-cora`	18936.5	543630.2	22.9	32.4	257.5	1.0
`ego-twitter`	111597.2	134716.0	3169.2	4.0	216.3	1.0
`ego-gplus`	115936.2	154869.5	57650.6	14.6	74.5	1.1
`munmun-digg`	34299.5	998564.0	1796.8	30.7	188.3	1.2
`linux`	103469.6	433745.5	59.5	32.1	94.6	1.5

Table 9: Speedups on the static algorithm and on the dynamic algorithm for all nodes on directed networks. For both Stat and AI, the first column reports the geometric mean of the speedups over the 100 insertions, the second column reports the maximum speedups and the third column the minimum speedup.

combination of it with the greedy approach for the MBI problem allows us to maximize betweenness of graphs with hundreds of thousands of edges in reasonable time.

BIBLIOGRAPHIC NOTES

This chapter is based on joint work with Pierluigi Crescenzi, Gianlorenzo D'Angelo, Henning Meyerhenke, Lorenzo Severini and Yllka Velaj, and is currently in revision for international journal publication. A preprint containing the results presented in this chapter and additional results related to the MBI problem can be found in [23]. In particular, in [23] we show that MBI cannot be approximated in polynomial-time within a factor $(1 - \frac{1}{2e})$ and that the Maximum Ranking Improvement problem (similar to MBI, with the difference that the ranking of a given node should be maximized, and not its betweenness score) does not admit any polynomial-time constant factor approximation algorithm, both unless $P = NP$. We also compare experimentally the greedy algorithm against several baselines and show that it outperforms them in terms of accuracy.

	Speedups on Stat			Speedups on AI		
Graph	Geometric mean	Maximum	Minimum	Geometric mean	Maximum	Minimum
`Mus-musculus`	1031.2	174166.8	191.4	7.7	21.9	1.7
`HC-BIOGRID`	962.3	4060.6	56.7	17.0	51.1	4.5
`Caenor-eleg`	2152.2	293172.8	474.6	15.0	49.4	1.3
`ca-GrQc`	1517.5	220289.2	351.8	10.0	22.8	2.1
`advogato`	5819.0	698406.6	2860.5	43.4	192.7	1.9
`hprd-pp`	5846.6	10852.3	1696.6	39.2	119.5	3.2
`ca-HepTh`	2642.6	432794.2	549.2	12.3	35.7	2.8
`dr-melanog`	6105.9	10589.7	1869.7	29.9	88.3	3.1
`oregon1`	7407.4	733008.3	1562.3	72.6	493.4	2.4
`oregon2`	9192.5	617710.0	1595.8	68.8	470.5	2.5
`Homo-sapiens`	9216.4	17177.4	2706.2	57.0	165.2	3.4
`GoogleNw`	34967.2	505509.9	3799.3	137.3	1560.3	2.9
`CA-CondMat`	4073.9	10690.6	537.8	20.8	69.4	2.8

Table 10: Speedups on the static algorithm and on the dynamic algorithm for all nodes on undirected networks. For both Stat and AI, the first column reports the geometric mean of the speedups over the 100 insertions, the second column reports the maximum speedups and the third column the minimum speedup.

	Running time Greedy				STD. DEV. Greedy			
Graph	$k = 1$	$k = 2$	$k = 5$	$k = 10$	$k = 1$	$k = 2$	$k = 5$	$k = 10$
`subelj-jung`	1.79	1.91	1.99	2.10	0.56	0.58	0.61	0.68
`wiki-Vote`	14.32	14.44	14.74	15.19	10.75	10.81	11.04	11.46
`elec`	12.47	12.57	12.81	13.13	7.80	7.83	7.99	8.16
`freeassoc`	81.52	83.01	87.00	96.60	66.27	67.88	70.84	82.01
`dblp-cite`	584.90	694.19	710.90	729.73	1060.50	1268.18	1296.99	1328.83
`subelj-cora`	1473.04	1504.96	1600.68	1688.39	1491.48	1526.95	1657.98	1784.74
`ego-twitter`	164.43	179.13	217.19	229.39	200.10	211.52	259.85	275.22
`ego-gplus`	211.39	225.58	230.26	240.29	195.22	186.00	188.82	196.78
`munmun-digg`	736.13	739.82	749.74	759.58	313.45	313.50	313.66	316.35
`linux`	1145.94	1239.16	1271.74	1311.28	822.06	917.50	933.02	951.61

Table 11: The left part of the table reports the running times (in seconds) of Greedy on directed real-world graphs for different values of k. The right part shows the standard deviations.

	Running time Greedy				STD. DEV. Greedy			
Graph	$k = 1$	$k = 2$	$k = 5$	$k = 10$	$k = 1$	$k = 2$	$k = 5$	$k = 10$
`Mus-musculus`	27.06	87.80	394.30	1155.46	15.53	36.35	176.05	630.80
`HC-BIOGRID.`	34.54	85.98	289.84	701.50	9.63	25.63	100.04	217.76
`Caenor-elegans`	11.17	25.47	94.94	320.85	3.23	10.23	23.66	55.19
`ca-GrQc.`	19.76	43.01	149.43	438.98	8.64	20.65	53.63	96.31
`advogato`	12.42	28.07	81.79	299.05	1.56	13.96	28.23	147.66
`hprd-pp`	47.08	111.85	460.31	1561.82	12.84	29.65	59.01	439.32
`ca-HepTh`	100.34	464.66	2069.34	5926.75	42.83	282.09	604.61	1320.20
`dr-melanog`	71.43	160.89	614.92	2084.71	18.01	31.55	46.88	333.84
`oregon1`	30.66	69.06	191.63	441.09	4.87	9.41	23.99	76.51
`oregon2`	36.44	73.35	233.28	594.53	9.63	16.92	25.26	44.3 7
`Homo-sapiens`	99.82	276.09	1155.97	3554.53	20.30	54.42	258.89	673.55
`GoogleNw`	68.33	102.35	220.32	451.29	11.71	17.18	36.19	76.37
`CA-CondMat`	1506.68	3402.10	12177.24	36000.24	381.00	927.40	2178.47	17964.74

Table 12: The left part of the table reports the running times (in seconds) of Greedy on undirected real-world graphs for different values of k. The right part shows the standard deviations.

7 FULLY-DYNAMIC BETWEENNESS APPROXIMATION

7.1 INTRODUCTION

As mentioned in Chapter 4, computing betweenness centrality exactly can be rather onerous for networks with more than a few tens of thousands of nodes. For large networks, approximation algorithms are therefore used in practice. On the other hand, if these large networks evolve over time, even rerunning an approximation algorithm after each modification or set of modifications might be too expensive, in particular for networks that change at a very quick pace. In these scenarios, dynamic exact algorithms such as the ones presented in Chapter 5 and Chapter 6 are also out of reach, both because of their quadratic memory requirement, and because their initialization requires to run an expensive static exact algorithm.

In this chapter, as our main contribution, we present the first approximation algorithms for betweenness in graphs that change over time. Such graphs may be directed or undirected, weighted or unweighted. We consider two dynamic scenarios, an incremental one (i. e.only edge insertions or weight decreases are allowed) and a fully-dynamic one, which also handles edge deletions or weight increase operations. In order to account for quickly-changing networks, we process the edge modifications in *batches*: our algorithms recompute betweenness after a sequence of updates is applied to the graph.

Based on the static approximation algorithm RK by Riondato and Kornaropoulos [107], our dynamic algorithms assert the same guarantee: the approximated betweenness values differ by at most ϵ from the exact values with probability at least $1 - \delta$, where ϵ and δ can be arbitrarily small constants. Running time and memory required depend on how tightly the error should be bounded. Notice that the same guarantee is offered by two newer algorithms (ABRA [108] and KADABRA [30]), which have both been shown to perform better than RK in practice. However, the work presented in this chapter is antecedent these two algorithms.

Besides resampling as few shortest paths as possible, several new intermediate algorithmic results contribute to the speed of the respective new approximation algorithms: (i) In Section 7.3 we propose new upper bounds on the vertex diameter VD (i. e.number of nodes in the shortest path(s) with the maximum number of nodes). These bounds vary depending on the graph type (weighted vs unweighted, directed vs undirected). Their usefulness stems from the fact that the new bounds can often improve the one used in the RK algorithm [107] and thereby significantly reduce the number of samples necessary for the error guarantee. (ii) In Section 7.4, besides detailing the betweenness approximation algorithms, we also present the first fully-dynamic algorithms for updating an approximation of VD in undirected graphs. (iii) As part of the betweenness approximation algorithms, we propose an algorithm with lower time complexity for updating single-source shortest paths in unweighted graphs after a batch of edge updates.

Our experimental study shows that our algorithms are the first to make in-memory computation of a betweenness ranking practical for large dynamic networks. With approximation we achieve a much improved scaling behavior compared to exact approaches,

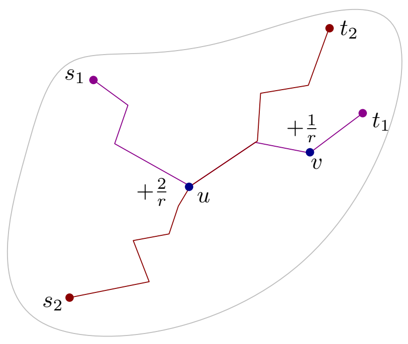


Figure 21: Sampled paths and score update in the RK algorithm

enabling us to update betweenness scores in a network with 36 million edges in a few seconds on typical workstation hardware. Moreover, processing batches of edges, our algorithms yield significant speedups (several orders of magnitude) compared to restarting the approximation with RK. Regarding accuracy, our experiments show that the estimated absolute errors are always lower than the guaranteed ones. For nodes with high betweenness, also the rank of nodes is well preserved, even when relatively few shortest paths are sampled.

Since RK is the building block of our dynamic algorithms, we will describe it in Section 7.2. Following the notation used in RK, in the following we use the *normalized* definition of betweenness centrality, namely $c_B(v) := \frac{1}{n(n-1)} \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$.

7.2 RK ALGORITHM

In this section we briefly describe the static betweenness approximation algorithm by Riondato and Kornaropoulos (RK) [107], the foundation for our incremental approach. The idea of RK is to sample a set $S = \{p_{(1)}, ..., p_{(r)}\}$ of r shortest paths between randomly sampled source-target pairs (s, t). Then, RK computes the approximate betweenness centrality $\tilde{c}_B(v)$ of a node v as the fraction of sampled paths $p_{(k)} \in S$ that v is internal to, by adding $\frac{1}{r}$ to the node's score for each of these paths. Figure 21 illustrates an example where the sampling of two shortest paths leads to $\frac{2}{r}$ and $\frac{1}{r}$ being added to the score of u and v, respectively. Each possible shortest path p_{st} has the following probability of being sampled in each of the r iterations: $\pi_G(p_{st}) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma_{st}}$ (Lemma 7 of [107]). The number r of samples required to approximate betweenness scores with the given error guarantee is calculated as

$$r = \frac{c}{\epsilon^2} \left(\lfloor \log_2 (VD - 2) \rfloor + 1 + \ln \frac{1}{\delta} \right), \tag{19}$$

where ϵ and δ are constants in $(0, 1)$, $c \approx 0.5$ and VD is the vertex diameter of G, i.e. the number of nodes in the shortest path of G with maximum number of nodes. In unweighted graphs VD coincides with diam+1, where diam is the number of edges in the longest shortest path. In weighted graphs VD and the (weighted) diameter diam (i. e.the length of the longest shortest path) are unrelated quantities. The following error guarantee holds:

Lemma 7.2.1. [107] *If r shortest paths are sampled according to the above-defined probability distribution π_G, then with probability at least $1-\delta$ the approximations $\tilde{c}_B(v)$ of the betweenness centralities are within ϵ from their exact value:*

$$\Pr(\exists v \in V \ s.t. \ |c_B(v) - \tilde{c}_B(v)| > \epsilon) < \delta.$$

To sample the shortest paths according to π_G, RK first chooses a node pair (s,t) uniformly at random and performs an SSSP search from s, keeping also track of the number σ_{sv} of shortest paths from s to v and of the list of predecessors $P_s(v)$ for any node v. Then one shortest path is selected: Starting from t, a predecessor $z \in P_s(t)$ is selected with probability $\sigma_{sz}/\sum_{w \in P_s(t)} \sigma_{sw} = \sigma_{sz}/\sigma_{st}$. The sampling is repeated iteratively until node s is reached. Algorithm 7 is the pseudocode for RK. Function `computeAugmentedSSSP` is an SSSP algorithm that keeps track of the number of shortest paths and of the list of predecessors while computing distances, as in BA [31] (see Chapter 4.2). Since we are only interested in the paths from s to t, we can stop the computation of the SSSP once t is reached.

Algorithm 7: RK algorithm

```
Input  : Graph $G = (V, E), \epsilon, \delta \in (0, 1)$
Output : Approximated betweenness values $\forall v \in V$
1  foreach node $v \in V$ do
2      $\tilde{c}_B(v) \leftarrow 0$;
3  end
4  VD $(G) \leftarrow$ getVertexDiameter($G$);
5  $r \leftarrow (c/\epsilon^2)(\lfloor \log_2(\mathrm{VD}(G) - 2) \rfloor + \ln(1/\delta))$;
6  for $i \leftarrow 1$ to $r$ do
7      $(s_i, t_i) \leftarrow$ sampleUniformNodePair($V$);
8      $(d_{s_i}, \sigma_{s_i}, P_{s_i}) \leftarrow$ computeAugmentedSSSP($G, s_i$);
       // Now one path from $s_i$ to $t_i$ is sampled uniformly at random
9      $v \leftarrow t_i$;
10     $p_{(i)} \leftarrow$ empty list;
11     while $P_{s_i}(v) \neq \{s_i\}$ do
12         sample $z \in P_{s_i}(v)$ with $\Pr = \sigma_{s_i}(z)/\sigma_{s_i}(v)$;
13         $\tilde{c}_B(z) \leftarrow \tilde{c}_B(z) + 1/r$;
14         add $z \to p_{(i)}$; $v \leftarrow z$;
15     end
16 end
17 return $\{(v, \tilde{c}_B(v)),\ v \in V\}$
```

APPROXIMATING THE VERTEX DIAMETER. RK uses two upper bounds on VD that can be both computed in $O(n+m)$. For unweighted undirected graphs, it samples a source node s_i for each connected component of G, computes a BFS from each s_i and sums the two shortest paths with maximum length starting in s_i. The VD approximation is the maximum of these sums over all components. For directed or weighted graphs, RK approximates VD with the size of the largest weakly connected component, which can be a significant overestimation for complex networks, possibly of orders of magnitude. In this chapter, we present new approximations for directed and for weighted graphs, described in Section 7.3.

7.3 NEW UPPER BOUNDS ON THE VERTEX DIAMETER

7.3.1 *Directed unweighted graphs.*

Let G be a directed unweighted graph. For now, let us assume G is strongly connected. Let s be any node in G and let u be the node with maximum forward distance from s (i. e.$d(s,u) \geq d(s,x)\ \forall x \in V$). Analogously, let v be the node with maximum backward distance (i. e.$d(v,s) \geq d(x,s)\ \forall x \in V$). Then, naming $\tilde{VD}_{\text{SC}}$ the sum $d(s,u) + d(v,s) + 1$:

Lemma 7.3.1. *$VD \leq \tilde{VD}_{SC} < 2\,VD$.*

Proof. Let x and y be two nodes such that the number of nodes in the shortest path from x to y is equal to VD. Due to the triangle inequality, $d(x,y) \leq d(x,s) + d(s,y)$. Therefore, $d(x,y) \leq d(v,s) + d(s,u)$. Since in unweighted graphs $d(x,y) = VD - 1$, the first inequality holds. By definition of VD, $2 \cdot VD \geq (d(v,s)+1) + (d(s,u)+1) > \tilde{VD}_{\text{SC}}$. □

The upper bound $\tilde{VD}_{\text{SC}}$ can be computed in $O(n+m)$, simply by running a forward and a backward BFS from any source node s.

Let us now consider any directed unweighted graph G. We can define the directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of strongly connected components (SCCs) (sometimes referred to as *shrunken graph* in the literature) similarly to Borassi et al. [29]. In $\mathcal{G}$, the set of vertices is the set of SCCs of G and there is an edge from $C \in \mathcal{V}$ to $C' \in \mathcal{V}$ if and only if there is an edge in E from a node in C to a node in C'. (Notice that this also means that all nodes in C' are reachable from the nodes in C.) More in general, the set of nodes reachable from any node in C is the set of nodes in the SCCs reachable from C in $\mathcal{G}$.

We can now define an algorithm that computes an upper bound on VD for G. For each C in $\mathcal{V}$, we compute an upper bound $\tilde{VD}_{\text{SC}}(C)$ on VD in C (i. e., considering only paths that are contained in C) as described before for strongly connected graphs. This can be done in linear time by running a backward and a forward BFS from a random source node s for each SCC and stopping the search when a visited node belongs to a different SCC from that of s. For each C_i, we know that the nodes reachable from nodes in C_i are only those in the SCCs C_j such that there is a path in $\mathcal{G}$ from C_i to C_j. We can compute a topological sorting $\{C_1, \ldots, C_k\}$ of $\mathcal{V}$ (that is, $(C_i, C_j) \in \mathcal{E} \Rightarrow i < j$). Let C_k be the last component in the topological ordering. Then, we know that no path from a node in C_k to any node that is not in C_k exists, which means that the node C_k in $\mathcal{G}$ has outdegree 0.

We call $\tilde{VD}_{\text{DIR}}(C)$ the upper bound on VD restricted only to nodes that *start* in C (but may end somewhere else). For C_k, $\tilde{VD}_{\text{DIR}}(C_k)$ is equal to $\tilde{VD}_{\text{SC}}(C_k)$. For the other components, we can compute it in the following way: Starting from C_k, we process all the components in reverse topological ordering and set

$$\tilde{VD}_{\text{DIR}}(C) = \max_{(C,C') \in \mathcal{E}} \tilde{VD}_{\text{DIR}}(C') + \tilde{VD}_{\text{SC}}(C).$$

To get an upper bound on the whole graph, we can take the maximum over all upper bounds $\tilde{VD}_{\text{DIR}}(C)$, i. e. we set $\tilde{VD}_{\text{DIR}} := \max_{C \in \mathcal{V}} \tilde{VD}_{\text{DIR}}(C)$. In other words:

$$\tilde{VD}_{\text{DIR}} = \max_{p \in \mathcal{P}(\mathcal{G})} \sum_{C_i \in p} \tilde{VD}_{\text{SC}}(C_i)$$

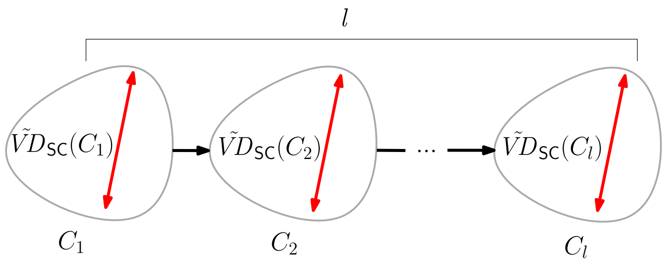

l
$\tilde{VD}_{\mathsf{SC}}(C_1)$
$\tilde{VD}_{\mathsf{SC}}(C_2)$
$\tilde{VD}_{\mathsf{SC}}(C_l)$
C_1
C_2
C_l

Figure 22: Path $(C_1, ..., C_l)$ of the DAG $\mathcal{G}$ of SCCs. Each SCC C_i has its own upper bound $\tilde{VD}_{\text{SC}}(C_i)$ and $\tilde{VD}_{\text{DIR}}$ is computed as $\sum_{i=1,..,l} \tilde{VD}_{\text{SC}}(C_i)$.

where $\mathcal{P}(G)$ is the set of paths in $\mathcal{G}$.

Proposition 7.3.1. *$VD \leq \tilde{VD}_{DIR} < 2 \cdot VD^2$.*

Proof. Let us prove the first inequality. Let $p = (u_1, \dots, u_{VD})$ be a shortest path in G whose number $|p|$ of nodes is equal to VD. Say p traverses l SCCs $(C_1, ..., C_l)$. Then p can be partitioned in l subpaths p_i, $i = 1, .., l$, such that $p_i \subseteq C_i$ and p_i is a shortest path in C_i. By Lemma 7.3.1, $|p_i| \leq \tilde{VD}_{\text{SC}}(C_i)$, $i = 1, ..., l$. Therefore, $|p| = \sum_{i=1,...,l} |p_i| \leq \sum_{i=1,...,l} \tilde{VD}_{\text{SC}}(C_i) \leq \tilde{VD}_{\text{DIR}}$ (this last inequality holds by definition of $\tilde{VD}_{\text{DIR}}$).

How "bad" can $\tilde{VD}_{\text{DIR}}$ be in the worst case? Let now $(C_1, ..., C_l)$ denote the path in $\mathcal{G}$ such that $\tilde{VD}_{\text{DIR}} = \sum_{i=1,..,l} \tilde{VD}_{\text{SC}}(C_i)$. Let l be the number of components in this path. How large can l be? Since there is at least one node of G in each C_i, there must be at least a shortest path of size l in G that goes through the components $C_1, ..., C_l$. Therefore, $l \leq VD$. Also, let $k = \max_{C \in \mathcal{V}} \tilde{VD}_{\text{SC}}(C)$. By Lemma 7.3.1, $k < 2 \cdot VD(C_k)$, where C_k is the SCC whose upper bound is equal to k. Clearly, $k < 2 \cdot VD$. Then, by definition, $\tilde{VD}_{\text{DIR}} = \sum_{i=1,..,l} \tilde{VD}_{\text{SC}}(C_i) \leq l \cdot k < 2 \cdot VD^2$.

□

The upper bound can be computed in $O(n + m)$. Indeed, $\mathcal{G}$ can be computed in $O(n + m)$, by finding the SCCs of G and scanning the edges in E. Then, the topological sorting and the accumulation of the $\tilde{VD}_{\text{DIR}}(C)$ of the different components can be done in $O(|\mathcal{V}| + |\mathcal{E}|) = O(n + m)$. Notice that our new upper bound is never larger than the size of the largest weakly connected component, the previous bound used in RK. Also, although the upper bound can be as bad as $2 \cdot VD^2$ in theory, our experimental results on real-world complex networks show that it is within a factor 4 from the exact VD and several orders of magnitude smaller than the size of the largest weakly connected components.

7.3.2 *Undirected weighted graphs*

Let G be an undirected graph. For simplicity, let G be connected for now. If it is not, we compute an approximation for each connected component and take the maximum over all the approximations. Let $T \subseteq G$ be an SSSP tree from any source node $s \in V$. Let p_{xy} denote a shortest path between x and y in G and let p^T_{xy} denote a shortest path between x and y in T. Let $|p_{xy}|$ be the number of nodes in p_{xy} and $d(x, y)$ be the distance between x and y in G, and analogously for $|p^T_{xy}|$ and $d^T(x, y)$. Let $\overline{\omega}$ and $\underline{\omega}$ be the maximum and minimum edge weights in G, respectively. Let u and v be the nodes with maximum distance from s, i. e.$d(s,u) \geq d(s,v) \geq d(s,x)\ \forall x \in V, x \neq u$. We define the VD approximation $\tilde{VD}_{\mathrm{W}} := 1 + \frac{d(s,u)+d(s,v)}{\underline{\omega}}$. Then:

Proposition 7.3.2. *$VD \leq \tilde{VD}_W < 2 \cdot VD \cdot \overline{\omega}/\underline{\omega}$.*

Proof. To prove the first inequality, we can notice that $d^T(x,y) \geq d(x,y)$ for all $x, y \in V$, since all the edges of T are contained in those of G. Also, since every edge has weight at least $\underline{\omega}$, $d(x,y) \geq (|p_{xy}| - 1) \cdot \underline{\omega}$. Therefore, $d^T(x,y) \geq (|p_{xy}| - 1) \cdot \underline{\omega}$, which can be rewritten as $|p_{xy}| \leq 1 + \frac{d^T(x,y)}{\underline{\omega}}$, for all $x, y \in V$. Thus, $VD = \max_{x,y} |p_{xy}| \leq 1 + (\max_{x,y} d^T(x,y))/\underline{\omega} \leq 1 + \frac{d^T(s,u)+d^T(s,v)}{\underline{\omega}} = 1 + \frac{d(s,u)+d(s,v)}{\underline{\omega}}$, where the last expression equals $\tilde{VD}_{\mathrm{W}}$ by definition.

To prove the second inequality, we first notice that $d(s,u) \leq (|p_{su}| - 1) \cdot \overline{\omega}$, and analogously $d(s,v) \leq (|p_{sv}| - 1) \cdot \overline{\omega}$. Consequently, $\tilde{VD}_{\mathrm{W}} \leq 1 + (|p_{su}| + |p_{sv}| - 2) \cdot \overline{\omega}/\underline{\omega} < 2 \cdot |p_{su}| \cdot \overline{\omega}/\underline{\omega}$, supposing that $|p_{su}| \geq |p_{sv}|$ without loss of generality. By definition of VD, $|p_{su}| \leq VD$. Therefore, $\tilde{VD}_{\mathrm{W}} < 2 \cdot VD \cdot \overline{\omega}/\underline{\omega}$.

□

To obtain the upper bound $\tilde{VD}_{\mathrm{W}}$, we can simply compute an SSSP search from any node s, find the two nodes with maximum distance and perform the remaining calculations. Notice that $\tilde{VD}_{\mathrm{W}}$ extends the upper bound proposed for RK [107] for unweighted graphs: When the graph is unweighted and thus $\underline{\omega} = \overline{\omega}$, $\tilde{VD}_{\mathrm{W}}$ becomes equal to the approximation used by RK. Complex networks are often characterized by a small diameter and in networks like coauthorship, friendship, communication networks, VD and $\overline{\omega}/\underline{\omega}$ can be several order of magnitude smaller than the size of the largest component, though. In this case the new bound translates into a substantially improved VD approximation.

7.3.3 *Directed weighted graphs.*

The upper bound for directed weighted graphs can be easily derived from those described in Sections 7.3.1 and 7.3.2. If G is strongly connected, we can define $\tilde{VD}_{\mathrm{SCW}} := 1 + \frac{d(s,u)+d(v,s)}{\underline{\omega}}$, where s is any node, u is the node with maximum forward distance from s, v is the node with maximum backward distance and $\underline{\omega}$ is the minimum edge weight. It can be easily proved that Proposition 7.3.2 holds also in this case, considering a forward SSSP tree from s (where distances are bounded by $d(s,u)$) and a backward SSSP tree (where distances are bounded by $d(v,s)$). For general directed weighted graphs, we can use the algorithm described in Section 7.3.1 using $\tilde{VD}_{\mathrm{SCW}}(C) := 1 + \frac{d(s,u)+d(v,s)}{\underline{\omega}_c}$ as an upper bound for each SCC C (where $\underline{\omega}_c$ is the minimum edge weight in C). It is easy to prove that the resulting bound is an upper bound on VD and that it is always smaller than $2 \cdot \max_{C \in \mathcal{G}} \frac{\overline{\omega}_C}{\underline{\omega}_C} \cdot VD^2$, using Propositions 7.3.1 and 7.3.2. Since it requires to compute an SSSP tree for each SCC, the complexity of computing the bound is $O(m + n \log n)$ (the other operations can be done in linear time, as described in Section 7.3.1).

7.4 FULLY-DYNAMIC APPROXIMATION ALGORITHMS

7.4.1 *Path subsitution.*

Our algorithms for dynamic betweenness approximation are composed of two phases: an *initialization* phase, which executes RK on the initial graph, and an *update* phase, which

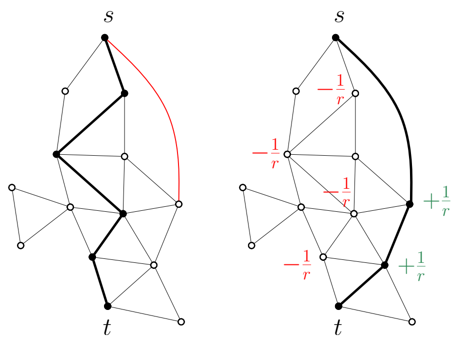


Figure 23: Updating shortest paths and betweenness scores

recomputes the approximated betweenness scores after a sequence of edge updates. Let us consider a batch $\beta = \{e_1, ..., e_k\}$ of edge updates $e_i = \{u_i, v_i, \omega(u_i, v_i)\}$ applied to a graph G. Also, let us assume for the moment that β is composed of **edge insertions only** (or weight decreases) and β **does not increase** the vertex diameter of G and therefore also the number r of samples required by RK for the maximum error guarantee. We will discuss the general case in Section 7.4.2. Intuitively, our basic idea is to keep the old sampled paths and update them only when necessary, instead of re-computing r shortest paths from scratch. Figure 23 shows an example to illustrate this idea: Assume several shortest paths between s and t exist, of which one has been sampled (with black nodes). An edge insertion (represented in red) shortens the distance between s and t, creating a new shorter path. Therefore, we simply subtract $1/r$ from each node in the old shortest path and add $1/r$ to each node in the new one.

From this point on, we give a formal description and only consider edge insertions. We suppose the graph is undirected, but in this restricted semi-dynamic setting our results can be easily extended to weight decreases and directed graphs. Let $G' = (V, E \cup \beta)$ be the new graph, let $d'_s(t)$ denote the new distance between any node pair (s,t) and let σ'_{st} be the new number of shortest paths between s and t. Let $\mathcal{S}_{st}$ and $\mathcal{S}'_{st}$ be the old and the new set of shortest paths between s and t, respectively. A new set $S' = \{p'_{(1)}, ..., p'_{(r)}\}$ of shortest paths has to be sampled now in order to let Lemma 7.2.1 hold for the new configuration; in particular, the probability $\Pr(p'_{(k)} = p'_{st})$ of each shortest path p'_{st} to be sampled must be equal to $\pi_{G'}(p'_{st}) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma'_{st}}$. Clearly, one could rerun RK on the new graph, but we can be more efficient: We recall that we are assuming that the *VD* and therefore also the number of samples for G' is smaller than or equal to that of G. Given any old sampled path p_{st}, we can *keep* p_{st} if the set of shortest paths between s and t has not changed, and *replace* it with a new path between s and t otherwise. Then, the following lemma holds:

Lemma 7.4.1. *Let S be a set of shortest paths of G sampled according to π_G. Let $\mathcal{P}$ be the procedure that creates S' by substituting each path $p_{st} \in S$ with a path p'_{st} according to the following rules: 1. $p'_{st} = p_{st}$ if $d'_s(t) = d_s(t)$ and $\sigma'_{st} = \sigma_{st}$. 2. p'_{st} selected uniformly at random among $\mathcal{S}'_{st}$ otherwise. Then, p'_{st} is a shortest path of G' and the probability of any shortest path p'_{xy} of G' to be sampled at each iteration is $\pi_{G'}(p'_{xy})$, i.e.* $\Pr(p'_{(k)} = p'_{xy}) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma'_{xy}}, k = 1, ..., r.$

Proof. To see that p'_{st} is a shortest path of G', it is sufficient to notice that, if $d'_s(t) = d_s(t)$ and $\sigma'_s(t) = \sigma_s(t)$, then all the shortest paths between s and t in G are shortest paths also in G'.

Let p'_{xy} be a shortest path of G' between nodes x and y. Basically, there are two possibilities for p'_{xy} to be the k-th sample. Naming e_1 the event $\{\mathcal{S}_{xy} = \mathcal{S}'_{xy}\}$ (the set of shortest paths between x and y does not change after the edge insertion) and e_2 the complementary event of e_1, we can write $\Pr(p'_{(k)} = p'_{xy})$ as $\Pr(p'_{(k)} = p'_{xy} \cap e_1) + \Pr(p'_{(k)} = p'_{xy} \cap e_2)$.

Using conditional probability, the first addend can be rewritten as $\Pr(p'_{(k)} = p'_{xy} \cap e_1) = \Pr(p'_{(k)} = p'_{xy} | e_1)\Pr(e_1)$. As the procedure $\mathcal{P}$ keeps the old shortest path when e_1 occurs, then $\Pr(p'_{(k)} = p'_{xy}|e_1) = \Pr(p_{(k)} = p'_{xy}|e_1) = \frac{1}{n(n-1)}\frac{1}{\sigma_x(y)}$, which is also equal to $\frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)}$, since $\sigma_x(y) = \sigma'_x(y)$ when we condition on e_1. Therefore, $\Pr(p'_{(k)} = p'_{xy} \cap e_1) = \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)} \cdot \Pr(e_1)$.

Analogously, $\Pr(p'_{(k)} = p'_{xy} \cap e_2) = \Pr(p'_{(k)} = p'_{xy}|e_2)\Pr(e_2)$. In this case, $\Pr(p'_{(k)} = p'_{xy}|e_2) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma'_x(y)}$, since this is the probability of the node pair (x, y) to be the k-th sample in the initial sampling and of p'_{xy} to be selected among other paths in $\mathcal{S}'_{xy}$. Then, $\Pr(p'_{(k)} = p'_{xy} \cap e_2) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma'_x(y)} \cdot \Pr(e_2) = \frac{1}{n(n-1)} \cdot \frac{1}{\sigma'_x(y)} \cdot (1 - \Pr(e_1))$. The probability $\Pr(p'_{(k)} = p'_{xy})$ can therefore be rewritten as:

$$\Pr(p'_{(k)} = p'_{xy}) = \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)} \cdot \Pr(e_1) + \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)} \cdot (1\text{-}\Pr(e_1)) = \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)}.$$

□

Since the set of paths is constructed according to $\pi_{G'}$, Theorem 7.4.1 follows directly from Lemma 7.2.1.

Theorem 7.4.1. *Let $G = (V, E)$ be a graph and let $G' = (V, E \cup \beta)$ be the modified graph after the the batch β. Let $VD(G) \geq VD(G')$. Let S be a set of r shortest paths of G sampled according to π_G and $r = \frac{c}{\epsilon^2}\left(\lfloor \log_2 (VD(G) - 2) \rfloor + 1 + \ln \frac{1}{\delta}\right)$ for some constants $\epsilon, \delta \in (0, 1)$. Then, if a new set S' of shortest paths of G' is built according to procedure $\mathcal{P}$ and the approximated values of betweenness centrality $\tilde{c}'_B(v)$ of each node v are computed as the fraction of paths of S' that v is internal to, then*

$$\Pr(\exists v \in V \ s.t. \ |c'_B(v) - \tilde{c}'_B(v)| > \epsilon) < \delta,$$

where $c'_B(v)$ is the new exact value of betweenness centrality of v after the edge insertion.

Notice that our guarantee is only probabilistic and it is possible (although with probability smaller than δ) that at some time step the approximated betweenness of a node diverges from the exact one for more than ϵ. Since there is dependency between the approximated values at different time steps, it is possible that this error propagates over the following time steps. However, it is also possible that the error decreases as a consequence of some modifications in the graph. Theorem 7.4.1 tells us that, *at each single time step*, the absolute error on the betweenness of each node is smaller than ϵ, with probability at least $1 - \delta$. We would also like to point out that, although cases where the maximum error exceeds ϵ are theoretically possible, this actually never happened in our experiments, where the measured errors were often orders of magnitude smaller than ϵ (see Section 7.5). Algorithm 8 shows the update procedure based on Theorem 7.4.1. For each sampled node pair (s_i, t_i), $i = 1, ..., r$, we first update the SSSP from s_i, a step which will be discussed in Sections 7.4.3 and 7.4.4. In case the distance or the number of shortest paths between s_i and t_i has changed, a new shortest path is sampled uniformly as in RK. This means that $\frac{1}{r}$ is subtracted from the score of each node in the old shortest path and the same quantity is added to the nodes in the new shortest path. On the other hand, if both distances and number of shortest paths between s_i and t_i have not changed, nothing needs to be updated. Considering edges in a batch allows us to recompute the betweenness scores only once instead of doing it after each single edge update. Moreover, this gives us the possibility to use specific batch algorithms for the update of the SSSP DAGs, which process the nodes affected by multiple edges of β only once, instead of for each single edge.

Differently from RK, in our dynamic algorithm we scan the neighbors every time we need the predecessors instead of storing them (Line 11). This allows us to use $\Theta(n)$ memory per sample (i. e., $\Theta(r \cdot n)$ in total) instead of $\Theta(m)$ per sample, while our experiments show that the running time is hardly influenced. The number of samples depends on ϵ, so in theory this can be as large as $|V|$. However, our experiments show that relatively large values of ϵ (e. g. $\epsilon = 0.05$) lead to a good ranking of nodes with high betweenness and for such values the number of samples is typically much smaller than $|V|$, making the memory requirements of our algorithms significantly less demanding than those of the dynamic exact algorithms ($\Omega(n^2)$) for many applications.

Algorithm 8: BC update after a batch β of edge insertions/weight decreases

```
Input  : Graph $G = (V, E)$, source node $s$, number of iterations $r$, batch $\beta$ of edge insertions/weight
         decreases
Output : New approximated betweenness values $\forall v \in V$
1  for $i \leftarrow 1$ to $r$ do
2      $d_i^{old} \leftarrow d_{s_i}(t_i)$;
3      $\sigma_i^{old} \leftarrow \sigma_{s_i}(t_i)$;
4      $(d_{s_i}, \sigma_{s_i}) \leftarrow$UpdateSSSP$(G, d_{s_i}, \sigma_{s_i}, \beta)$;
5      if $d_{s_i}(t_i) < d_i^{old}$ or $\sigma_{s_i}(t_i) \neq \sigma_i^{old}$ then
6          foreach $w \in p_{(i)}$ do
7              $\tilde{c}_B(w) \leftarrow \tilde{c}_B(w) - 1/r$;
8          end
9          $v \leftarrow t_i$;
10         $p_{(i)} \leftarrow$ empty list;
11         $P_{s_i}(v) \leftarrow \{u \in V | (u, v) \in E \wedge d_{s_i}(u) + \omega(u, v) = d_{s_i}(v)\}$;
12         while $P_{s_i}(v) \neq \{s_i\}$ do
13             sample $z \in P_{s_i}(v)$ with $\Pr = \sigma_{s_i}(z)/\sigma_{s_i}(v)$;
14             $\tilde{c}_B(z) \leftarrow \tilde{c}_B(z) + 1/r$;
15             add $z$ to $p_{(i)}$;
16             $v \leftarrow z$;
17         end
18     end
19 end
20 return $\{(v, \tilde{c}_B(v)),\ v \in V\}$
```

7.4.2 *Sampling new paths.*

In the previous section, we assumed that $VD(G) \geq VD(G')$. Although many real-world networks exhibit a shrinking-diameter behavior [80], it is clearly possible that VD increases as a consequence of edge insertions/deletions or weight updates. If this happens, we can still update the old paths as described in Section 7.4.1, but we also need to sample new additional paths, according to the probability distribution π'_G. The general algorithm to update the betweenness scores after a batch β could be described as follows: First, we update the old shortest paths as described in Section 7.4.1. Then, we recompute an upper bound on $VD(G')$ in linear time, using the algorithms described in Section 7.3. Using $VD(G')$, we compute the number of samples $r(G')$ defined in Eq. (19). If $r(G') > r(G)$, we sample new $r(G') - r(G)$ additional paths using RK. For undirected graphs, we also propose two fully-dynamic algorithms (one for weighted and one for unweighted graphs) to keep track of an upper bound on VD over time (Section 7.4.5). This saves additional time, allowing for a quick recomputation of the upper bound after the batch instead of recomputing it from scratch.

Notice that, if edge deletions are allowed, it is not sufficient to check whether the distance and the number of shortest paths between two nodes s and t has not changed (Line 5 of Algorithm 8), since they might remain unchanged even if the set of shortest paths is actually different. In this case, we always replace the old shortest path with a new one (we basically remove the *if* statement in Line 5).

In the following, we present the fully-dynamic algorithms (for weighted and unweighted graphs) needed to update the shortest paths (updateSSSP in Algorithm 8) and the fully-dynamic algorithm that recomputes an upper bound on VD for undirected graphs. Finally, we show how these algorithms can be combined to obtain an even faster algorithm (than the one described in this section) for dynamic betweenness approximation in undirected graphs (Section 7.4.6).

7.4.3 *SSSP update in weighted graphs.*

Our SSSP update is based on T-SWSF [10], which recomputes distances from a source node s after a batch β of weight updates (or edge insertions/deletions). For our purposes, we need two extensions of T-SWSF: an algorithm that also recomputes the number of shortest paths between s and the other nodes (updateSSSP-W) and one that also updates a VD approximation for the connected component of s (updateApprVD-W) (the latter is used in the fully-dynamic VD approximation for undirected graphs, described in Section 7.4.5). The VD approximation is computed as described in Sections 7.2 and 7.3.2. Thus, updateApprVD-W keeps track of the two maximum distances d' and d'' from s and the minimum edge weight $\underline{\omega}$.

We call *affected nodes* the nodes whose distance (or also whose number of shortest paths, in updateSSSP-W) from s has changed as a consequence of β. Basically, the idea is to put the set A of affected nodes w into a priority queue Q with priority $p(w)$ equal to the candidate distance of w. When w is extracted, if there is actually a path of length $p(w)$ from s to w, the new distance of w is set to $p(w)$, otherwise w is reinserted into Q with a higher candidate distance. In both cases, the affected neighbors of w are inserted into Q.

Algorithm 9 describes the SSSP update for weighted undirected graphs. The extension to directed graphs is trivial. The pseudocode updates both the VD approximation for the connected component of s and the number of shortest paths from s, so it basically includes both updateSSSP-W and updateApprVD-W. Initially, we scan the edges $e = \{u, v\}$ in β and, for each e, we insert the endpoint with greater distance from s into Q (w.l.o.g., let v be such endpoint). The priority $p(v)$ of v represents the *candidate* new distance of v. This is the minimum between the $d(v)$ and $d(u)$ plus the weight of the edge $\{u, v\}$. Notice that we use the expression "insert v into Q" for simplicity, but this can also mean update $p(v)$ if v is already in Q and the new priority is smaller than $p(v)$. When we extract a node w from Q, we have two possibilities: (i) there is a path of length $p(w)$ and $p(w)$ is actually the new distance or (ii) there is no path of length $p(w)$ and the new distance is greater than $p(w)$. In the first case (Lines 9 - 23), we set $d(w)$ to $p(w)$ and insert the neighbors z of w such that $d(z) > d(w) + \omega(\{w, z\})$ into Q (to check if new shorter paths to z that go through w exist). In the second case (Lines 24 - 40), there is no shortest path between s and w known anymore, so that we set $d(w)$ to ∞. We compute $p(w)$ as $\min_{\{v,w\}\in E} d(v) + \omega(v, w)$ (the new candidate distance for w) and insert w into Q. Also its neighbors could have lost one (or all of) their old shortest paths, so we insert them into Q as well. The update of $\underline{\omega}$ can be done while scanning the batch and of d' and d'' when we update $d(w)$.

The pseudocode also updates a global variable $vis(w)$ that keeps track, for each node w, of the number of source nodes from which w is reachable. This is necessary for the fully-dynamic VD approximation and will be explained in Section 7.4.5. In particular, we decrease $vis(w)$ when updating $d(w)$ in case the old $d(w)$ was equal to ∞ (i. e.w has become reachable) and we decrease $vis(w)$ when we set $d(w)$ to ∞ (i. e.w has become unreachable). We update the number of shortest paths after updating $d(w)$, as the sum of the shortest paths of the predecessors of w (Lines 16 - 18). In updateApprVD-W, d' and d'' are recomputed while updating the distances (Line 10) and $\underline{\omega}$ is updated while scanning β (Line 5). Let $|\beta|$ represent the cardinality of β and let $||A||$ represent the sum of the nodes in A and of the edges that have at least one endpoint in A. Then, the following complexity derives from feeding Q with the batch and inserting into/extracting from Q the affected nodes and their neighbors.

Lemma 7.4.2. *The time required by updateApprVD-W (updateSSSP-W) to update the distances and $\tilde{VD}$ (the number of shortest paths) is $O(|\beta|\log|\beta| + ||A||\log||A||)$.*

Proof. In the initial scan of the batch (Lines 2-4), we scan the nodes of the batch and insert the affected nodes into Q (or update their value). This requires at most one heap operation (insert or decrease-key) for each element of β, therefore $O(|\beta|\log|\beta|)$ time. When we extract a node w from Q, we have two possibilities: (i) $con(w) = p(w)$ (Lines 9 - 23) or (ii) $con(w) > p(w)$ (Lines 24 - 40). In the first case, we scan the neighbors of w and perform at most one heap operation for each of them (Lines 19 - 21). In the second case, we scan only if $d(w) \neq \infty$. Therefore, we can perform up to one heap operation per incident edge of w, for each extraction of w in which $d(w) \neq \infty$ or $con(w) = p(w)$. How many times can an affected node w be extracted from Q with $d(w) \neq \infty$ or $con(w) = p(w)$? If the first time we extract w, $con(w)$ is equal to $p(w)$ (case (i)), then the final value of $d(w)$ is reached and w is not inserted into Q anymore. If the first time we extract w, $con(w)$ is greater than $p(w)$ (case (ii)), then w can be inserted into the queue again. However, its distance is set to ∞ and therefore no additional operations are performed, until $d(w)$ becomes less than ∞. But this can happen only in case (i), after which $d(w)$ reaches its final value. To summarize, each affected node w can be extracted from Q with $d(w) \neq \infty$ or $con(w) = p(w)$ at most twice and, every time this happens, at most one heap operation per incident edge of w is performed. The complexity is therefore $O(|\beta|\log|\beta| + ||A||\log||A||)$. □

Algorithm 9: SSSP update for weighted graphs (updateSSSP-W)

```
Input  : Graph $G = (V, E)$, vector $d$ of distances from $s$, vector $\sigma$ of number of shortest paths from
         $s$, batch $\beta$
Output : New values of $d(v)$ and $\sigma(v)$ $\forall v \in V$
$Q \leftarrow$ empty priority queue;
foreach $e = \{u, v\} \in \beta, d(u) < d(v)$ do
    $Q \leftarrow$ insertOrDecreaseKey($v, p(v) = \min\{d(u) + \omega(\{u, v\}), d(w)\}$);
end
$\underline{\omega} \leftarrow \min\{\underline{\omega},\ \omega(e) : e \in \beta\}$;
while there are nodes in $Q$ do
    $\{w, p(w)\} \leftarrow$ extractMin($Q$);
    $con(w) \leftarrow \min_{z:(z,w)\in E} d(z) + \omega(z, w)$;
    if $con(w) = p(w)$ then
        update $d'$ and $d''$;
        if $d(w) = \infty$ then
            $vis(w) \leftarrow vis(w) + 1$;
        end
        $d(w) \leftarrow p(w)$; $\sigma(w) \leftarrow 0$;
        foreach incident edge $(z, w)$ do
            if $d(w) = d(z) + \omega(z, w)$ then
                $\sigma(w) \leftarrow \sigma(w) + \sigma(z)$;
            end
            if $d(z) \geq d(w) + \omega(z, w)$ then
                $Q \leftarrow$ insertOrDecreaseKey($z, p(z) = d(w) + \omega(z, w)$);
            end
        end
    end
    else
        if $d(w) \neq \infty$ then
            $vis(w) \leftarrow vis(w) - 1$;
            if $vis(w)=0$ then
                insert $w$ into $U$;
            end
            if $con(w) \neq \infty$ then
                $Q \leftarrow$insertOrDecreaseKey($w, p(w) = con(w)$);
                foreach incident edge $(z, w)$ do
                    if $d(z) = d(w) + \omega(w, z)$ then
                        $Q \leftarrow$insertOrDecreaseKey($z, p(z) = d(w) + \omega(z, w)$);
                    end
                end
                $d(w) \leftarrow \infty$;
            end
        end
    end
end
```

7.4.4 *SSSP update in unweighted graphs.*

For unweighted graphs, we basically replace the priority queue Q of updateApprVD-W and updateSSSP-W with a list of queues. Each queue represents a *level* (distance from s) from 0 (which only the source belongs to) to the maximum distance d'. The levels replace the priorities and also in this case represent the candidate distances for the nodes. Algorithm 10 describes the pseudocode for unweighted graphs. As in Algorithm 9, we first scan the batch (Lines 3 - 5) and insert the nodes in the queues. Then (Lines 6 - 44), we scan the queues in order of increasing distance from s, in a fashion similar to that of a priority queue. In order not to insert a node in the queues multiple times, we use colors: Initially we set all the nodes to white and then we set a node w to black only when we find the final distance of w (i. e.when we set $d(w)$ to k) (Line 15). Black nodes extracted from a queue are then skipped (Line 10). At the end we reset all nodes to white. The replacement of the priority queue with the list of queues decreases the complexity of the SSSP update algorithms for unweighted graphs, which we call updateApprVD-U and updateSSSP-U, in analogy with the ones for weighted graphs.

Lemma 7.4.3. *The time required by updateApprVD-U (updateSSSP-U) to update the distances and $\tilde{VD}$ (the number of shortest paths) is $O(|\beta| + ||A|| + d_{\max})$, where $d_{\max}$ is the maximum distance from s reached during the update.*

Proof. The complexity of the initialization (Lines 3 - 5) of Algorithm 10 is $O(|\beta|)$, as we have to scan the batch. In the main loop (Lines 6 - 44), we scan all the list of queues, whose final size is $d_{\max}$. Every time we extract a node w whose color is not black, we scan all the incident edges, therefore this operation is linear in the number of neighbors of w. If the first time we extract w (say at level k) $con(w)$ is equal to k, then w will be set to black and will not be scanned anymore. If the first time we extract w, $con(w)$ is instead greater than k, w will be inserted into the queue at level $con(w)$ (if $con(w) < \infty$). Also, other inconsistent neighbors of w might insert w in one of the queues. However, after the first time w is extracted, its distance is set to ∞, so its neighbors will not be scanned unless $con(w) = k$, in which case they will be scanned again, but for the last time, since w will be set to black. To summarize, each affected node and its neighbors can be scanned at most twice. The complexity of the algorithm is therefore $O(|\beta| + \|A\| + d_{\max})$. □

7.4.5 *Fully-dynamic VD approximation.*

The algorithm keeps track of a *VD* approximation for an undirected graph G, i. e.for each connected component of G. It is composed of two phases. In the initialization, we compute an SSSP from a source node s_i for each connected component C_i. During the SSSP search from s_i, we also compute a *VD* approximation $\tilde{VD}_i$ for C_i, as described in Sections 7.2 and 7.3.2. In the update, we recompute the SSSPs and the *VD* approximations with updateApprVD-W (or updateApprVD-U). Since components might split or merge, we might need to compute new approximations, in addition to update the old ones. To do this, for each node, we keep track of the number $vis(v)$ of times it has been visited. This way we discard source nodes that have already been visited and compute a new approximation for components that have become unvisited. Algorithm 11 describes the initialization. Initially, we put all the nodes in a queue and compute an SSSP from the nodes we extract (Line 18). During the SSSP search, we increase the number of visits $vis(v)$ for all the nodes

Algorithm 10: SSSP update for unweighted graphs (updateSSSP-U)

```
Input   : Graph $G = (V, E)$, vector $d$ of distances from $s$, vector $\sigma$ of number of shortest paths from
          $s$, batch $\beta$
Output  : New values of $d(v)$ and $\sigma(v)$ $\forall v \in V$
Assumption: $color(w) = white \quad \forall w \in V$;
$Q[] \leftarrow$ array of empty queues;
foreach $e = \{u, v\} \in \beta, d(u) < d(v)$ do
    $k \leftarrow d(v) + 1$; enqueue $v \to Q[k]$;
end
$k \leftarrow 1$;
while there are nodes in $Q[j], j \geq k$ do
    while $Q[k] \neq \emptyset$ do
        dequeue $w \leftarrow Q[k]$;
        if $color(w) = black$ then continue;
        $con(w) \leftarrow \min_{z:(z,w) \in E} d(z) + 1$;
        if $con(w) = k$ then
            update $d'$ and $d''$;
            if $d(w) = \infty$ then $vis(w) \leftarrow vis(w) + 1$;
            $d(w) \leftarrow k$; $\sigma(w) \leftarrow 0$; $color(w) \leftarrow black$;
            foreach incident edge $(z, w)$ do
                if $d(w) = d(z) + 1$ then
                    $\sigma(w) \leftarrow \sigma(w) + \sigma(z)$;
                end
                if $d(z) > k$ then
                    enqueue $z \to Q[k + 1]$;
                end
            end
        end
        else
            if $d(w) \neq \infty$ then
                $d(w) \leftarrow \infty$;
                $vis(w) \leftarrow vis(w) - 1$;
                if $vis(w)=0$ then
                    insert $w$ into $U$;
                end
                if $con(w) \neq \infty$ then
                    enqueue $w \to Q[con(w)]$;
                    foreach incident edge $(z, w)$ do
                        if $d(z) > k$ then
                            enqueue $z \to Q[k + 1]$;
                        end
                    end
                end
            end
        end
    end
    $k \leftarrow k + 1$;
end
Set to white all the nodes that have been in $Q$;
```

we traverse (Line 24). When extracting the nodes, we skip those that have already been visited (Line 8): this avoids computing multiple approximations for the same component.

In the update (Algorithm 12), we recompute the SSSPs and the VD approximations with updateApprVD-W (or updateApprVD-U) (Line 7). Since components might split, we might need to add VD approximations for some new subcomponents, in addition to recomputing the old ones. Also, if components merge, we can discard the superfluous approximations. To do this, we update $vis(v)$ for each node within updateApprVD-W (or updateApprVD-U). Before the update, all the nodes are visited exactly once. While updating an SSSP from s_i, we increase (decrease) by one $vis(v)$ of the nodes v that become reachable (unreachable) from s_i. This way we can skip the update of the SSSPs from nodes that have already been visited (Line 8). After the update, for all nodes v that have become unvisited ($vis(v) = 0$), we compute a new VD approximation from scratch (Lines 11 - 18). The complexity of the update of the VD approximation derives from the $\tilde{VD}$ update in the single components, using updateApprVD-W and updateApprVD-U.

Theorem 7.4.2. *The time required to update the VD approximation is* $O(n_c \cdot |\beta| \log |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| \log ||A^{(i)}||)$ *in weighted graphs and* $O(n_c \cdot |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| + d^{(i)}_{max})$ *in unweighted graphs, where* n_c *is the number of components in* G *before the update and* $A^{(i)}$ *is the sum of affected nodes in* C_i *and their incident edges.*

Proof. In the first part (Lines 2 - 9 of Algorithm 12), we update an SSSP with updateApprVD-W or updateApprVD-U for each source node s_i such that $vis(s_i)$ is not greater than 1. Therefore the complexity of the first part is $O(n_c \cdot |\beta| \log |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| \log ||A^{(i)}||)$ in weighted graphs and $O(n_c \cdot |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| + d^{(i)}_{max})$ in unweighted, by Lemmas 7.4.2 and 7.4.3. Only some of the affected nodes (those whose distance from a source node becomes ∞) are inserted into the queue U. Therefore the cost of scanning U in Lines 11 - 18 is $O(\sum_{i=1}^{n_c} ||A^{(i)}||)$. New SSSP searches are computed for new components that are not covered by the existing source nodes anymore. However, also such searches involve only the affected nodes and each affected node (and its incident edges) is scanned at most once during the search. Therefore, the total cost is $O(n_c \cdot |\beta| \log |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| \log ||A^{(i)}||)$ for weighted graphs and $O(n_c \cdot |\beta| + \sum_{i=1}^{n_c} ||A^{(i)}|| + d^{(i)}_{max})$ for unweighted graphs. □

Lemma 7.4.4. *At the end of Algorithm 11,* $vis(v) = 1$ *for all* $v \in V$*, and exactly one VD approximation is computed for each connected component of* G*.*

Proof. Let v be any node. Then v must be scanned by *at least* one source node s_i in the while loop (Lines 6 - 13): In fact, either v is visited by some s_i before v is extracted from U, or $vis(v) = 0$ at the moment of the extraction and v becomes a source node itself. This implies that $vis(v) \geq 1$, $\forall v \in V$. On the other hand, $vis(v)$ cannot be greater than 1. In fact, let us assume by contradiction that $vis(v) > 1$. This means that there are at least two source nodes s_i and s_j ($i < j$, w.l.o.g.) that are in the same connected component as v. Then also s_i and s_j are in the same connected component and s_j is visited during the SSSP search from s_i. Then $vis(s_j) = 1$ before s_j is extracted from U and s_j cannot be a source node. Therefore, $vis(v)$ is exactly equal to 1 for each $v \in V$, which means that exactly one VD approximation is computed for each connected component of G.

□

Proposition 7.4.1. *Let* $\mathcal{C}' = \{C'_1, ..., C'_{n'_c}\}$ *be the set of connected components of* G *after the update. Algorithm 12 updates or computes exactly one VD approximation for each* $C'_i \in \mathcal{C}'$*.*

Proof. Let $\mathcal{C} = \{C_1, ..., C_{n_c}\}$ be the set of connected components before the update. Let us consider three basic cases (then it is straightforward to see that the proof holds also for combinations of these cases): (i) $C_i \in \mathcal{C}$ is also a component of $\mathcal{C}'$, (ii) $C_i \in \mathcal{C}$ and $C_j \in \mathcal{C}$ merge into one component C'_k of $\mathcal{C}'$, (iii) $C_i \in \mathcal{C}$ splits into two components C'_j and C'_k of $\mathcal{C}'$. In case (i), the *VD* approximation of C_i is updated exactly once in the for loop (Lines 2 - 9). In case (ii), (assuming $i < j$, w.l.o.g.) the *VD* approximation of C'_k is updated in the for loop from the source node $s_i \in C_i$. In its SSSP search, s_i visits also $s_j \in C_j$, increasing $vis(s_j)$. Therefore, s_j is skipped and exactly one *VD* approximation is computed for C'_k. In case (iii), the source node $s_i \in C_i$ belongs to one of the components (say C'_j) after the update. During the for loop, the *VD* approximation is computed for C'_j via s_i. Also, for all the nodes v in C'_k, $vis(v)$ is set to 0 and v is inserted into U. Then some source node $s'_k \in C'_k$ must be extracted from U in Line 12 and a *VD* approximation is computed for C'_k. Since all the nodes in C'_k are set to visited during the search, no other *VD* approximations are computed for C'_k.

□

Algorithm 11: Dynamic *VD* approximation (initialization)

```
   Input  : Graph $G = (V, E)$
   Output : Upper bound on $VD$
1  $U \leftarrow []$;
2  foreach node $v \in V$ do
3      $vis(v) \leftarrow 0$; insert $v$ into $U$;
4  end
5  $i \leftarrow 1$;
6  while $U \neq \emptyset$ do
7      extract $s$ from $U$;
8      if $vis(s) = 0$ then
9          $s_i \leftarrow s$;
10         $\tilde{VD}_i \leftarrow$ initApprVD($G, s_i$);
11         $i \leftarrow i + 1$;
12     end
13 end
14 $n_C \leftarrow i - 1$;
15 $\tilde{VD} \leftarrow \max_{i=1,...,n_C} \tilde{VD}_i$;
16 return $\tilde{VD}$;
   // initApprVD computes $\tilde{VD}$ and adds 1 to $vis(v)$ of the nodes it visits
17 Function initApprVD($G, s$)
18     SSSP($G, s$);
19     $d' \leftarrow \max\{d(s,u) | u \in V, d(u,v) \neq \infty\}$;
20     $d'' \leftarrow \max\{d(s,v) | v \in V, v \neq u, d(s,v) \neq \infty\}$;
21     $\underline{\omega} \leftarrow \min\{\omega(x,y) | (x,y) \in E\}$;
22     $\tilde{VD} \leftarrow 1 + \frac{d'+d''}{\underline{\omega}}$;
23     foreach node $w \in V$ s.t. $d(s,w) \neq \infty$ do
24         $vis(w) \leftarrow vis(w) + 1$;
25     end
26     return $\tilde{VD}$;
```

7.4.6 *Combined dynamic betweenness approximation.*

Let G be an undirected graph with n_c connected components. In Section 7.4.2, we described an algorithm to update the betweenness approximations in fully-dynamic graphs. If the

Algorithm 12: Dynamic *VD* approximation (updateApprVD)

```
Input   : Graph $G = (V, E)$, vector $vis$
Output  : New $VD$ approximation
1  $U \leftarrow []$;
2  foreach $s_i$ do
3      if $vis(s_i) > 1$ then
4          remove $s_i$ and $\tilde{VD}_i$; decrease $n_C$;
5      end
6      else
           // updateApprVD updates $vis$, inserts all $v$ for which $vis(v) = 0$ into $U$ and
              recomputes a $VD$ approximation $\tilde{VD}_i$
7          $\tilde{VD}_i \leftarrow$ updateApprVD($G, s_i$) ;
8      end
9  end
10 $i \leftarrow n_C$ ;
11 while $U \neq \emptyset$ do
12     extract $s'$ from $U$;
13     if $vis(s') = 0$ then
14         $s'_i \leftarrow s'$;
15         $\tilde{VD}_i \leftarrow$ initApprVD($G, s'_i$);
16         $i \leftarrow i + 1$; $n_C \leftarrow n_C + 1$;
17     end
18 end
19 reset $vis(v)$ to 1 for nodes $v$ such that $vis(v) > 1$;
20 $\tilde{VD} \leftarrow \max_{i=1,\ldots,n_C} \tilde{VD}_i$;
21 return $\tilde{VD}$
```

graph is undirected, we can use the fully dynamic *VD* approximation to recompute $\tilde{VD}$ after a batch, instead of recomputing it from scratch. Then, we could update the r sampled paths with updateSSSP and, if $\tilde{VD}$ (and therefore r) increases, we could sample new paths. However, since updateSSSP and updateApprVD share most of the operations, we can "merge" them and update at the same time the shortest paths from a source node s and the *VD* approximation for the component of s. We call this hybrid function updateSSSPVD. Instead of storing and updating n_c SSSPs for the *VD* approximation and r SSSPs for the betweenness scores, we recompute a *VD* approximation for each of the r samples while recomputing the shortest paths with updateSSSPVD. This way we do not need to compute an additional SSSP for the components covered by the r sampled paths (i. e.the components in which the paths lie), saving time and memory. Only for components that are not covered by any of them (if they exist), we compute and store a separate *VD* approximation. We refer to such components as R' (and to $|R'|$ as r').

In the initialization (Algorithm 13), we first compute the r SSSP, like in RK (Lines 4 - 18). However, we also check which nodes have been visited, as in Algorithm 11. While we compute the r SSSPs, in addition to the distances and number of shortest paths, we also compute a *VD* approximation for each of the r source nodes and increase $vis(v)$ of all the nodes we visit during the sources with initApprVD (Line 8). Since it is possible that the r shortest paths do not cover all the components of G, we compute an additional VD approximation for nodes in the unvisited components, like in Algorithm 11 (Lines 21 - 28). Basically we can divide the SSSPs into two sets: the set R of SSSPs used to compute the r shortest paths and the set R' of SSSPs used for a *VD* approximation in the components that were not scanned by the initial R SSSPs. We call r' the number of the SSSPs in R'.

The betweenness update after a batch is described in Algorithm 14. First (Lines 2 - 19), we recompute the shortest paths like in Algorithm 8: we update the SSSPs from each source node s in R and we replace the old shortest path with a new one (subtracting $1/r$ from the nodes in the old shortest path and adding $1/r$ to those in the new shortest path). To update the SSSPs, we use the fully-dynamic updateSSSPVD that updates also the VD approximation and keeps track of the nodes that become unvisited. Then (Lines 24 - 31), we add a new SSSP to R' for each component that has become unvisited (by both R and R'). After this, we have at least a VD approximation for each component of G. We take the maximum over all these approximations and recompute the number of samples r (Lines 32 - 33). If r has increased, we need to sample new paths and therefore new SSSPs to add to R. Finally, we normalize the betweenness scores, i. e.we multiply them by the old value of r divided by the new value of r (Line 37). We refer to the algorithm for unweighted graphs as DA and the one for weighted as DAW. The difference between DA and DAW is the way the SSSPs and the VD approximation are updated: in DA we use updateApprVD-U and in DAW updateApprVD-W.

Theorem 7.4.3. *Algorithm 14 preserves the guarantee on the maximum absolute error, i. e.naming $c'_B(v)$ and $\tilde{c}'_B(v)$ the new exact and approximated betweenness values, respectively,* $\Pr(\exists v \in V\ s.t.\ |c'_B(v) - \tilde{c}'_B(v)| > \epsilon) < \delta$.

Proof. Let G be the old graph and G' the modified graph after the batch of edge updates. Let p'_{xy} be a shortest path of G' between nodes x and y. To prove the theoretical guarantee, we need to prove that the probability of any sampled path $p'_{(i)}$ is equal to p'_{xy} (i. e.that the algorithms adds $1/r'$ to the nodes in p'_{xy}) is $\frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)}$. Algorithm 14 replaces the first r shortest paths with other shortest paths $p'_{(1)}, ..., p'_{(r)}$ between the same node pairs (Lines 12 - 18) using Algorithm 8, for which we already proved that $\Pr(p'_{(k)} = p'_{xy}) = \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)}$ (Lemma 7.4.1). The additional Δr shortest paths (Line 35) are recomputed from scratch with RK, therefore also in this case $\Pr(p'_{(k)} = p'_{xy}) = \frac{1}{n(n-1)}\frac{1}{\sigma'_x(y)}$ by Lemma 7 of [107].

□

7.4.7 *Complexity of the dynamic betweenness algorithms.*

In this section we presented different algorithms for updating betweenness approximations after batches of edge updates. Algorithm 8 can be used on graphs for which we can be sure that the vertex diameter cannot increase after a batch of edge updates. This includes, for example, unweighted connected graphs on which only edge insertions are allowed. We refer to the unweighted version of this algorithm as IA (incremental approximation) and to the weighted version as IAW (incremental approximation weighted). On general directed graphs, we can use the algorithms described in Section 7.4.2. We name the unweighted version DAD (dynamic approximation directed) and the weighted one DADW. Finally, for undirected graphs, we can use the optimized algorithms presented in Section 7.4.6 (Algorithm 14), to which we refer as DA and DAW. Theorem 7.4.4 presents the complexities of all the betweenness update algorithms. In the following, we name $||A^{(i)}||$ the sum of affected nodes and their incident edges in the i-th sampled SSSP. We also name r the number of samples. In case we need to sample new additional paths after the update (in

Algorithm 13: BC initialization

```
Input  : Graph $G = (V, E)$, source node $s$, number of iterations $r$, batch $\beta$
Output : Approximated betweenness values $\forall v \in V$
foreach node $v \in V$ do
    $\tilde{c}_B\ (v) \leftarrow 0$; $vis(v) \leftarrow 0$;
end
$\tilde{VD} \leftarrow$ getApproxVertexDiameter($G$);
$r \leftarrow (c/\epsilon^2)(\lfloor \log_2(\tilde{VD} - 2) \rfloor + \ln(1/\delta))$;
for $i \leftarrow 1$ to $r$ do
    $(s_i, t_i) \leftarrow$ sampleUniformNodePair($V$);
    $\tilde{VD}_i \leftarrow$ initApprVD($G, s_i,$);
    $v \leftarrow t_i$;
    $p_{(i)} \leftarrow$ empty list;
    $P_{s_i}(v) \leftarrow \{z : \{z, v\} \in E \cap d_{s_i}(v) = d_{s_i}(z) + \omega(\{z, v\})\}$;
    while $P_{s_i}(v) \neq \{s_i\}$ do
        sample $z \in P_{s_i}(v)$ with probability $\sigma_{s_i}(z)/\sigma_{s_i}(v)$;
        $\tilde{c}_B(z) \leftarrow \tilde{c}_B(z) + 1/r$;
        add $z \rightarrow p_{(i)}$; $v \leftarrow z$;
        $P_{s_i}(v) \leftarrow \{z : \{z, v\} \in E \cap d_{s_i}(v) = d_{s_i}(z) + \omega(\{z, v\})\}$;
    end
end
$U \leftarrow V$;
$i \leftarrow r + 1$;
while $U \neq \emptyset$ do
    extract $s'$ from $U$;
    if $vis(s') = 0$ then
        $s'_i \leftarrow s'$;
        $\tilde{VD}_i \leftarrow$ initApprVD($G, s'_i$);
        $i \leftarrow i + 1$;
    end
end
$r' \leftarrow r - i - 1$;
return $\{(v, \tilde{c}_B(v)) : v \in V\}$
```

Algorithm 14: Dynamic update of betweenness approximation (DA)

```
Input   : Graph $G = (V, E)$, source node $s$, number of iterations $r$, batch $\beta$
Output  : New approximated betweenness values $\forall v \in V$
$U \leftarrow []$;
for $i \leftarrow 1$ to $r$ do
    $d_i^{old} \leftarrow d_{s_i}(t_i)$;
    $\sigma_i^{old} \leftarrow \sigma_{s_i}(t_i)$;
    // updateSSSPVD updates $vis$, inserts all $v : vis(v) = 0$ into $U$ and updates the $VD$
       approximation
    $\tilde{VD}_i \leftarrow$ updateSSSPVD($G, s_i, \beta$);
    // we replace the shortest path between $s_i$ and $t_i$
    foreach $w \in p_{(i)}$ do
        $\tilde{c}_B\ (w) \leftarrow \tilde{c}_B(w) - 1/r$;
    end
    $v \leftarrow t_i$;
    $p_{(i)} \leftarrow$ empty list;
    $P_{s_i}(v) \leftarrow \{z : \{z, v\} \in E \cap d_{s_i}(v) = d_{s_i}(z) + \omega(\{z, v\})\}$;
    while $P_{s_i}(v) \neq \{s_i\}$ do
        sample $z \in P_{s_i}(v)$ with probability $= \sigma_{s_i}(z)/\sigma_{s_i}(v)$;
        $\tilde{c}_B(z) \leftarrow \tilde{c}_B(z) + 1/r$;
        add $z$ to $p_{(i)}$;
        $v \leftarrow z$;
        $P_{s_i}(v) \leftarrow \{z : \{z, v\} \in E \cap d_{s_i}(v) = d_{s_i}(z) + \omega(\{z, v\})\}$;
    end
end
for $i \leftarrow r + 1$ to $r + r'$ do
    $\tilde{VD}_i \leftarrow$ updateApprVD($G, s_i, \beta$);
end
$i \leftarrow r + r' + 1$;
while $U \neq \emptyset$ do
    extract $s'$ from $U$;
    if $vis(s') = 0$ then
        $s'_i \leftarrow s'$;
        $\tilde{VD}_i \leftarrow$ initApprVD($G, s'_i$);
        $i \leftarrow i + 1$; $r' \leftarrow r' + 1$;
    end
end
// compute the maximum over all the $VD_i$ computed by updateApprVD
$\tilde{VD} \leftarrow \max_{i=1,\ldots,r+r'} \tilde{VD}_i$;
$r_{\text{new}} \leftarrow (c/\epsilon^2)(\lfloor \log_2(\tilde{VD} - 2) \rfloor + \ln(1/\delta))$;
if $r_{new} > r$ then
    sample new paths;
    foreach $v \in V$ do
        $\tilde{c}_B(v) \leftarrow \tilde{c}_B(v) \cdot r/r_{\text{new}}$
    end
    $r \leftarrow r_{\text{new}}$;
end
return $\{(v, \tilde{c}_B(v)) : v \in V\}$
```

DAD, DADW, DA and DAW), we refer to the difference between the value of r before and after the batch as Δr. In DA and DAW, we call r' the number of additional samples necessary for the *VD* approximation.

Theorem 7.4.4. *Given a graph $G = (V, E)$ with n nodes and m edges, the time required by the different algorithms to update the betweenness approximations after a batch β are the following:*

(i) *IA:* $O(r \cdot |\beta| + \sum_{i=1}^{r}(||A^{(i)}|| + d_{\max}^{(i)})$

(ii) *IAW:* $O(r \cdot |\beta| \log |\beta| + \sum_{i=1}^{r} ||A^{(i)}|| \log ||A^{(i)}||)$

(iii) *DAD:* $O(r \cdot |\beta| + \sum_{i=1}^{r}(||A^{(i)}|| + d_{\max}^{(i)}) + (\Delta r + 1)(n + m))$

(iv) *DADW:* $O((r \cdot |\beta| \log |\beta| + \sum_{i=1}^{r} ||A^{(i)}|| \log ||A^{(i)}|| + (\Delta r + 1)(n \log n + m))$

(v) *DA:* $O((r + r')|\beta| + \sum_{i=1}^{r+r'}(||A^{(i)}|| + d_{\max}^{(i)}) + \Delta r(n + m))$

(vi) *DAW:* $O((r + r')|\beta| \log |\beta| + \sum_{i=1}^{r+r'} ||A^{(i)}|| \log ||A^{(i)}|| + \Delta r(n \log n + m))$

Proof. We prove each case separately.

(i) IA updates each sampled path with updateSSSP-U. Therefore, the total complexity is the sum of the times required to update each of the r paths, i. e.$O(r \cdot |\beta| + \sum_{i=1}^{r}(||A^{(i)}|| + d_{\max}^{(i)})$.

(ii) Same as (i), with the only difference that we use updateSSSP-W for weighted graphs.

(iii) In DAD, we need to update the existing r samples, exactly as in IA. In addition to that, we might need to sample new Δr additional paths using a BFS, whose complexity is $O(n + m)$. Also, we need to recompute the upper bound on *VD*, whose complexity is also $O(n + m)$ (see Section 7.3.1). Therefore, in this case we have to add an additional $O((\Delta r + 1)(n + m))$ term to the complexity of IA.

(iv) Similarly to (iii), we need to sample Δr additional paths, but in weighted graphs the cost of a SSSP is $O(n \log n + m)$. Also the *VD* approximation described in Section 7.3.3 requires $O(n \log n + m)$ time.

(v) Let $\Delta r'$ be the difference between the values of r' before and after the batch. After processing β, we might need to sample new paths for the betweenness approximation ($\Delta r > 0$) and/or sample paths in new components that are not covered by any of the sampled paths ($\Delta r' > 0$). Then, the complexity for the betweenness approximation update is $O(r \cdot |\beta| + \sum_{i=1}^{r}(||A^{(i)}|| + d_{\max}^{(i)})) + O(\Delta r(n + m))$. The *VD* update requires $O(r' \cdot |\beta| + \sum_{i=1}^{r'}(||A^{(i)}|| + d_{\max}^{(i)}))$ to update the *VD* approximation in the already covered components and $\sum_{i=1}^{\Delta r}(|V_i| + |E_i|)$ for the new ones, where V_i and E_i are nodes and edges of the ith component, respectively. From this derives the total complexity.

(vi) Same as (v), using updateSSSP-W, approxVD-W.

□

Graph	Type	Nodes	Edges	Type
`ca-GrQc`	coauthorship	5 242	14 496	Unweighted, Undirected
`p2p-Gnutella09`	file sharing	8 114	26 013	Unweighted, Directed
`ca-HepTh`	coauthorship	9 877	25 998	Unweighted, Undirected
`PGPgiantcompo`	social / web of trust	10 680	24 316	Unweighted, Undirected
`as-22july06`	internet	22 963	48 436	Unweighted, Undirected

Table 13: Overview of small real-world networks used in the experiments.

Notice that, if $\tilde{VD}$ does not increase, $\Delta r = 0$ and the complexities of DA and DAD (DAW and DADW, respectively) are the same as the only-incremental algorithm IA (IAW, respectively). This case includes, for example, connected graphs subject to a batch of only edge insertions, or any batch that neither splits the graph into more components nor increases VD. Also, notice that in the worst case the complexity can be as bad as recomputing from scratch, or even slightly worse. Indeed, $||A^{(i)}||$ can be as large as m, for $i = 1, ..., r$, and $d^{(i)}_{\max}$ can be as large as n. Assuming $\beta = \Theta(m)$ as a worst-case batch size, the running times of IA and IAW are then $O(r \cdot (m+n))$ and $O(r \cdot (m \log m))$, respectively. Analogously, the complexities of DAD and DADW are $O((r + \Delta r) \cdot (n+m))$ and $O((r + \Delta r) \cdot (m \log m))$, where we recall that $(r + \Delta r)$ is the number of samples required after the batch. In the optimized versions DA and DAW, the worst-case running time is even larger: $O((r + \Delta r + n_c) \cdot (n+m))$ and $O((r + \Delta r + n_c) \cdot (m \log m))$, where n_c is the number of connected components of the graph. However, these worst-case running times are not observed in our experiments. Indeed, in the next section, we will show that our dynamic algorithms perform very well in practice. Also, notice that no dynamic SSSP (and so probably also no betweenness approximation) algorithm exists that is asymptotically faster than recomputation on all graphs.

7.5 EXPERIMENTS

IMPLEMENTATION AND SETTINGS. For an experimental comparison, we implemented our six approaches IA, IAW, DAD, DADW, DA, DAW, as well as the static approximation RK [107]. In the implementation of RK, we used the optimization proposed in [107], stopping all the SSSP searches once the target node has been found. Also, we computed the number of samples using our new bounds on VD presented in Section 7.3. We implemented all algorithms in C++, building on the open-source *NetworKit* framework [119]. We also used the NetworKit implementation of Brandes's algorithm BA as a reference in experiments regarding the accuracy. In all experiments we fix δ to 0.1 while the error bound ϵ varies. The machine we employ is used for its 256 GB RAM. Of the machine's 2 x 8 Intel(R) Xeon(R) E5-2680 cores running at 2.7 GHz we use only one; all computations are sequential to make the comparison to previous work more meaningful.

DATA SETS. We use both real-world and synthetic networks. For our experiments on the accuracy, we use relatively small networks, on which also BA can be executed. These networks are summarized in Table 13 and are publicly available from the collection compiled for the 10th DIMACS Challenge [9] (`http://www.cc.gatech.edu/dimacs10/downloads.shtml`) and from the SNAP collection (`http://snap.stanford.edu`). Due to

Graph	Type	Nodes	Edges	Type
`digg`	communication	30,398	85,155	Weighted
`slashdot`	communication	51,083	116,573	Weighted
`linux`	communication	63,399	159,996	Weighted
`facebook`	communication	46,952	183,412	Weighted
`enron`	communication	87,273	297,456	Weighted
`facebookFriends`	friendship	63,731	817,035	Unweighted
`ca-HepPh`	coauthorship	28,093	3,148,447	Unweighted
`wikipedia`	hyperlink	1,870,709	36,532,531	Unweighted

Table 14: Overview of real dynamic graphs used in the experiments, taken from `http://konect.uni-koblenz.de/`.

a shortage of actual dynamic networks in this size range, we simulate dynamics by removing a small fraction of random edges and adding them back in batches, as described in Section 3.2.2. We also use synthetic networks obtained with the Dorogovtsev-Mendes generator, a simple model for networks with power-law degree distribution [46].

To compare the running times of the scalable algorithms (RK and our dynamic algorithms), we use real dynamic networks, taken from The Koblenz Network Collection (KONECT) [76] and summarized in Table 14 (for more information about the properties of these networks, see Chapter 3.2). All the edges of the KONECT graphs are characterized by a time of arrival. In case of multiple edges between two nodes, we extract two versions of the graph: one unweighted, where we ignore additional edges, and one weighted, where we replace the set E_{st} of edges between two nodes with an edge of weight $1/|E_{st}|$ (more tightly coupled nodes receive a smaller distance). In our experiments, we let the batch size vary from 1 to 1024 and for each batch size, we average the running times over 10 runs. Since the networks do not include edge deletions, we implement additional simulated dynamics. In particular, we consider the following experiments. (i) *Real dynamics.* We remove the x edges with the highest timestamp from the network and we insert them back in batches, in the order of timestamps. (ii) *Random insertions and deletions.* We remove x edges from the graph, chosen uniformly at random. To create batches of both edge insertions and deletions, we add back the deleted edges with probability $1/2$ and delete other random edges with probability $1/2$. (iii) *Random weight changes.* In weighted networks, we choose x edges uniformly at random and we multiply their weight by a random value in the interval $(0, 2)$.

To study the scalability of the methods, we also use synthetic graphs obtained with a generator based on a unit-disk graph model in hyperbolic geometry [85], where edge insertions and deletions are obtained by moving the nodes in the hyperbolic plane. The networks produced by the model were shown to have many properties of real complex networks, like small diameter and power-law degree distribution (for details and references the interested reader is referred to von Looz et al. [85]). We generate seven networks, with $|E|$ ranging from about $2 \cdot 10^4$ to about $2 \cdot 10^7$ and $|V|$ approximately equal to $|E|/10$.

We also compare our new upper bound on *VD* for directed graphs presented in Section 7.3.1 with the one used in RK. For this, we use directed real-world graphs of different sizes taken from the SNAP collection.

7.5.1 *Accuracy.*

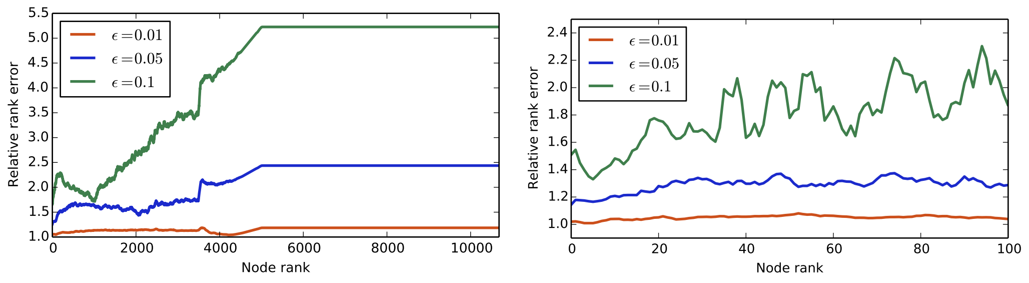


Figure 24: Relative rank error on `PGPgiantcompo` for nodes ordered by rank. Left: relative errors of all nodes. Right: relative errors of the 100 nodes with highest betweenness.

Graph	`ca-GrQc`	`ca-HepTh`	`PGPgiantcompo`	`as-22july06`	`p2p-Gnutella09`
max. error ($\epsilon = 0.1$)	1.70e-02	1.69e-02	3.10e-02	3.22e-02	1.56e-02
max. error ($\epsilon = 0.05$)	9.12e-03	7.62e-03	1.38e-02	1.60e-02	6.55e-03
max. error ($\epsilon = 0.01$)	1.67e-03	1.41e-03	2.99e-03	3.45e-03	1.23e-03
avg. error ($\epsilon = 0.1$)	4.55e-04	3.87e-04	4.56e-04	8.55e-05	5.92e-04
avg. error ($\epsilon = 0.05$)	2.42e-04	2.10e-04	2.54e-04	5.35e-05	3.15e-04
avg. error ($\epsilon = 0.01$)	4.63e-05	4.29e-05	5.10e-05	1.33e-05	6.55e-05

Table 15: Maximum and average absolute errors on real networks for different values of ϵ ($\delta = 0.1$). The values are averaged over 10 runs.

We consider the accuracy of the approximated centrality scores both in terms of absolute error and, more importantly, the preservation of the ranking order of nodes. Since we only replace the samples without changing their number, our dynamic algorithm has exactly the same accuracy as RK. The authors of [107] study the behavior of RK also experimentally, considering the average and maximum estimation error on a small set of real graphs. We study the experimental errors on additional graphs. For our tests we use the networks summarized in Table 13 and Dorogovtsev-Mendes graphs of several sizes. Our results confirm those of [107] in the sense that the measured absolute errors are *always* below the guaranteed maximum error ϵ and the measured average error is often orders of magnitude smaller than ϵ. Table 15 shows the measured errors for the real networks. We also study the *relative rank error* introduced by Geisberger et al. [56] (i. e. $\max\{\rho, 1/\rho\}$, denoting ρ the ratio between the estimated rank and the true rank), which we consider the most relevant measure of the quality of the approximations. Figure 24 shows the results for `PGPgiantcompo`, a similar trend can be observed on our other test instances as well. On the left, we see the errors for the whole set of nodes (ordered by exact rank) and, on the right, we focus on the top 100 nodes. The straight lines in the plot on the left correspond to nodes with betweenness 0, which are therefore undistinguishable. The plots show that for a small value of ϵ (0.01), the ranking is very well preserved over all the positions. With higher values of ϵ, the rank error of the nodes with low betweenness increases, as they are harder to approximate. However, the error remains small for the nodes with highest betweenness, the most important ones for many applications.

7.5.2 *New upper bound on VD for directed graphs.*

We compute the new upper bound on VD for directed graphs presented in Section 7.3.1 and compare it with the upper bound used in RK [107], i. e.the size of the largest weakly connected component. All the networks used in the experiment are real directed graphs. Since finding VD exactly would be expensive in most of the graphs we used, we also compute a lower bound on VD, by sampling nodes in the graph, computing their eccentricity (i. e., the maximum distance reachable from the node), and adding 1 to it. In Table 16, we report this lower bound ($VD^\star$), our new upper bound ($\tilde{VD}$) and the one used in RK ($\tilde{VD}_{\mathrm{RK}}$). The results show that $\tilde{VD}$ is always several orders of magnitude smaller than $\tilde{VD}_{\mathrm{RK}}$ and never more than a factor 4 from the lower bound $VD^\star$ (and therefore also from VD). This difference is mitigated by the logarithm in the number of samples required for the approximation (see Eq. (19)). However, Table 16 shows that $\tilde{VD}$ is almost always more than 2^{10} times smaller than $\tilde{VD}_{\mathrm{RK}}$, resulting in at least 10 times less samples required for the theoretical guarantee.

Graph	Nodes	Edges	$VD^\star$	$\tilde{VD}$	$\tilde{VD}_{\mathrm{RK}}$
`p2p-Gnutella24`	26 518	65 369	20	47	26 498
`soc-Epinions1`	75 879	508 837	13	41	75 877
`slashdot081106`	77 356	516 575	14	39	77 349
`twitter-comb`	81 306	1 768 149	9	34	81 306
`slashdot090216`	81 870	545 671	14	40	81 866
`amazon0302`	262 111	1 234 877	71	183	262 111
`email-EuAll`	265 214	420 045	9	23	224 832

Table 16: Lower bound on VD ($VD^\star$) and upper bounds (our new bound $\tilde{VD}$ and the one proposed in RK, $\tilde{VD}_{\mathrm{RK}}$) on real-world networks.

7.5.3 *Running times.*

In this section, we discuss the running times of the dynamic algorithms we propose and their speedups on RK. (Note that the term *speedup* is used to compare different algorithms, not sequential vs parallel execution.)

Figure 25 (left) reports the speedups of DA on RK in real graphs using real dynamics. Although some fluctuations can be noticed, the speedups tend to decrease as the batch size increases. We can attribute fluctuations to two main factors: First, different batches can affect areas of G of varying sizes, influencing also the time required to update the SSSPs. Second, changes in the VD approximation can require to sample new paths and therefore increase the running time of DA (and DAW). Nevertheless, DA is significantly faster than recomputation on all networks and for every tested batch size. Analogous results are reported in Figure 26 (left) for random dynamics. Table 17 summarizes the running times of DA and its speedups on RK with batches of size 1 and 1024 in unweighted graphs, under both real and random dynamics. Even on the larger graphs (`ca-HepPh` and `wikipedia`) and on large batches, DA requires at most a few seconds to recompute the betweenness scores, whereas RK requires about one hour for `wikipedia`. The results on weighted graphs are shown in Table 18. In both real dynamics and random updates, the

	Real				Random			
	Time [s]		Speedups		Time [s]		Speedups	
Graph	$\lvert\beta\rvert = 1$	$\lvert\beta\rvert = 1024$	$\lvert\beta\rvert = 1$	$\lvert\beta\rvert = 1024$	$\lvert\beta\rvert = 1$	$\lvert\beta\rvert = 1024$	$\lvert\beta\rvert = 1$	$\lvert\beta\rvert = 1024$
`digg`	0.078	1.028	76.11	5.42	0.008	0.832	94.00	4.76
`slashdot`	0.043	1.055	219.02	9.91	0.038	1.151	263.89	28.81
`linux`	0.049	1.412	108.28	3.59	0.051	2.144	72.73	1.33
`facebook`	0.023	1.416	527.04	9.86	0.015	1.520	745.86	8.21
`enron`	0.368	1.279	83.59	13.66	0.203	1.640	99.45	9.39
`facebookFriends`	0.447	1.946	94.23	18.70	0.448	2.184	95.91	18.24
`ca-HepPh`	0.038	0.186	2287.84	400.45	0.025	1.520	2188.70	28.81
`wikipedia`	1.078	6.735	3226.11	617.47	0.877	5.937	2833.57	703.18

Table 17: Times and speedups of DA on RK in unweighted real graphs under real dynamics and random updates, for batch sizes of 1 and 1024.

speedups vary between ≈ 50 and $\approx 6 \cdot 10^3$ for single-edge updates and between ≈ 5 and ≈ 75 for batches of size 1024.

On hyperbolic graphs (Figure 25, right), the speedups of DA on RK increase with the size of the graph. Table 19 contains the precise running times and speedups on batches of 1 and 1024 edges. The speedups vary between ≈ 100 and $\approx 3 \cdot 10^5$ for single-edge updates and between ≈ 3 and $\approx 5 \cdot 10^3$ for batches of 1024 edges.

Some graphs of Table 14 can also be interpreted as directed (i. e., `digg`, `slashdot`, `linux`, `facebook`, `enron`, `ca-HepPh`). We therefore test DAD on the directed version of the networks, using real dynamics. Also on directed networks, using DAD is always faster than recomputation with RK, by orders of magnitude on small batches. Figure 26 (right) shows the speedups of DAD on RK on the `facebook` graph.

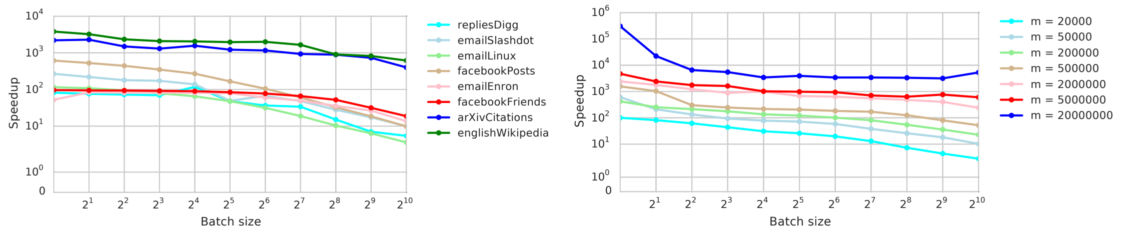


Figure 25: Speedups of DA on RK, with $\epsilon = 0.05$ and with batches of different sizes. Left: real unweighted networks using real dynamics. Right: hyperbolic unit-disk graphs of different sizes.

To summarize, our results show that our dynamic algorithms are faster than recomputation with RK in all the tested instances, even when large batches of 1024 edges are applied to the graph. With small batches, the algorithms are always orders of magnitude faster than RK, often with running times of fraction of seconds or seconds compared to minutes or hours. Such high speedups are made possible by the efficient update of the sampled shortest paths, which limits the recomputation to the nodes that are actually affected by the batch. Also, processing the edges in batches, we avoid to update multiple times nodes that are affected by several edges of the batch.

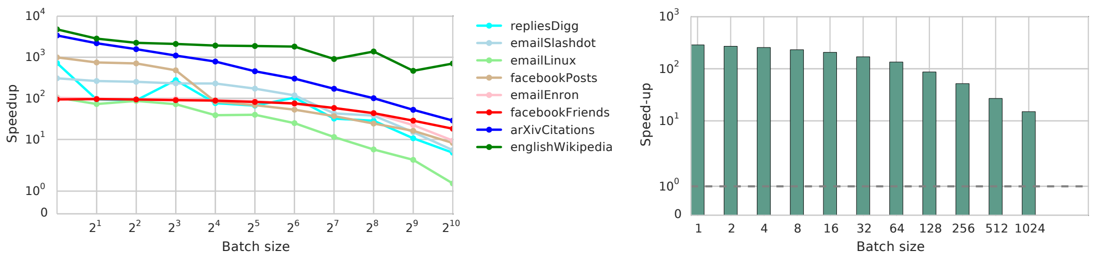


Figure 26: Left: speedups of DA on RK in real unweighted graphs under random updates. Right: Speedups of DAD on RK on the `facebook` directed graph using real dynamics.

	Real				Random			
	Time [s]		Speedups		Time [s]		Speedups	
Graph	$\lvert\beta\rvert = 1$	$\lvert\beta\rvert = 1024$	$\lvert\beta\rvert = 1$	$\lvert\beta\rvert = 1024$	$\lvert\beta\rvert = 1$	$\lvert\beta\rvert = 1024$	$\lvert\beta\rvert = 1$	$\lvert\beta\rvert = 1024$
`digg`	0.053	3.032	605.18	14.24	0.049	3.046	658.19	14.17
`slashdot`	0.790	5.387	50.81	16.12	0.716	5.866	56.00	14.81
`linux`	0.324	24.816	5780.49	75.40	0.344	24.857	5454.10	75.28
`facebook`	0.029	6.672	2863.83	11.42	0.029	6.534	2910.33	11.66
`enron`	0.050	9.926	3486.99	24.91	0.046	50.425	3762.09	4.90

Table 18: Times and speedups of DAW on RK in weighted real graphs under real dynamics and random updates, for batch sizes of 1 and 1024.

	Hyperbolic			
	Time [s]		Speedups	
Number of edges	$\lvert\beta\rvert = 1$	$\lvert\beta\rvert = 1024$	$\lvert\beta\rvert = 1$	$\lvert\beta\rvert = 1024$
$m = 20000$	0.005	0.195	99.83	2.79
$m = 50000$	0.002	0.152	611.17	10.21
$m = 200000$	0.015	0.288	422.81	22.64
$m = 500000$	0.012	0.339	1565.12	51.97
$m = 2000000$	0.049	0.498	2419.81	241.17
$m = 5000000$	0.083	0.660	4716.84	601.85
$m = 20000000$	0.006	0.401	304338.86	5296.78

Table 19: Times and speedups of DA on RK in hyperbolic unit-disk graphs, for batch sizes of 1 and 1024.

BIBLIOGRAPHIC NOTES

Results reguarding the incremental case in (strongly) connected graphs have ben published as “Approximating betweenness centrality in large evolving networks” (coauthored with Henning Meyerhenke and Christian Staudt) at the *Seventeenth Workshop on Algorithm Engineering and Experiments* (ALENEX 2015).

Results related to the decremental case in general undirected graphs have been published as “Fully-dynamic approximation of betweenness centrality” (coauthored with Henning Meyerhenke) at the *Twenty-third Annual European Symposium on Algorithms* (ESA 2015).

The methods have been unified and extended to directed graphs in “Approximating betweenness centrality in fully dynamic networks” (coauthored with Henning Meyerhenke), published in the journal *Internet Mathematics*.

CONCLUSIONS OF PART II

In this part we have presented several algorithms for updating betweenness centrality (or its approximation) in evolving networks. Betweenness is a shortest-path based centrality measure, and thus its computation, as well as its update, requires up-to-date information about shortest paths between pairs of nodes. Our experimental results in Chapter 3 already showed that most edge updates affect only a small fraction of node pairs in real-world complex networks. This phenomenon is reflected in the performance of the dynamic algorithms presented in this part, whose empirical running time is mostly several orders of magnitude faster than static recomputation. The much faster running time is achieved by storing information on the initial graph and pruning computations based on properties of the newly-created shortest paths. This, however, implies a higher memory footprint, which is in fact the primary limitation to the scalability of our dynamic algorithms. This limitation can be partially mitigated using approximation, as in the algorithm proposed in Chapter 7. Differently from the exact approaches presented in Chapters 5 and 6, and other existing dynamic approaches (which require $\Theta(n^2)$ memory), the algorithm in Chapter 7 requires $\Theta(n)$ memory for each sampled path, but the number of samples grows quadratically as the error bound ϵ is tightened. This leaves practitioners with a tradeoff between running time and memory consumption on the one hand and betweenness score error on the other hand. An interesting direction for future research is the development of algorithms with similarly good performance as the ones presented in this part, but smaller memory footprint.

Our results in Chapter 6 also highlight that updating betweenness centrality of a single node can be done much more efficiently than updating it for all nodes. Although this may sound intuitive, the same cannot be said for the static case, where no algorithm has been devised that computes betweenness of a single node faster than for all nodes, neither in theory (i.e., faster asymptotic running time), nor in practice (i.e. significantly better empirical performance). On the contrary, the incremental algorithm presented in Chapter 6 updates betweenness of a node in $O(n^2)$ time in the worst case, which – for dense graphs – is asymptotically faster than existing incremental algorithms for all nodes (requiring $O(nm)$ time in the worst case). Also in practice, our experiments show significantly better running times compared to the dynamic algorithm for all nodes presented in Chapter 5. Preliminary results show that the speedups are even higher when compared against other existing dynamic algorithms – between two and three orders of magnitude on average.

Our results from Chapter 7 also suggest that processing updates in batches can lead to significant speedups compared to processing them one by one. As an example, our dynamic approximation algorithm has an average speedup on recomputation of about 3200 for single-edge updates on the `wikipedia` graph, whereas the average speedup for batches of 1024 edges is above 600.

The algorithms presented in Chapter 5 and Chapter 7 have been made available to researchers and practitioners as part of the NetworKit tool suite [119]. The algorithm in Chapter 6 will be part of a future release.

Part III

EFFICIENT COMPUTATION OF NODES WITH HIGHEST CLOSENESS CENTRALITY

8 OVERVIEW OF ALGORITHMS FOR CLOSENESS CENTRALITY

8.1 INTRODUCTION

Closeness centrality is a very intuitive measure: given a node, it indicates the inverse average shortest-path distance to the other nodes of the network. The idea behind this definition is that a node is central if it is efficient in spreading information to the other nodes. The identification of the nodes with highest closeness finds application in a variety of research areas. Examples include facility location [71], marketing strategies [70] and identification of key infrastructure nodes as well as disease propagation control and crime prevention [13]. In fact, closeness is one of the oldest and one of the most widely-used centrality measures [12]: almost all books dealing with network analysis discuss it (for example, [95]), and almost all existing network analysis libraries implement algorithms to compute it.

Unfortunately, computing the closeness centrality of all nodes in a graph can be very expensive. The currently best algorithm solves an all-pairs shortest path (APSP) problem to compute the distance between each pair of nodes. For an unweighted graph $G = (V, E)$ with n nodes and m edges, this can be done in time $O(n^{2.373} \log n)$ using fast matrix multiplication [123] or in $O(nm)$ running a BFS from each node (or, on weighted graphs, in $O(nm + n^2 \log n)$ running Dijkstra's algorithm from each node). Since real-world networks are often sparse and since the first approach contains large hidden constants, BFS-based approaches are predominant in practice. Nevertheless, this running time becomes prohibitive already for networks with a few million nodes, restricting the exact computation of closeness to a small fraction of real applications. For this reason, several methods have been proposed to scale up the computation of closeness centrality, using different techniques. In the following, we discuss existing results on the computation of closeness centrality and related problems.

8.2 RELATED WORK

Closeness is a "traditional" definition of centrality, and consequently it was not "designed with scalability in mind", as stated in [65]. Also in [38], it is said that closeness centrality can "identify influential nodes", but it is "incapable to be applied in large-scale networks due to the computational complexity". The simplest solution considered was to define different measures that might be related to closeness centrality [65].

HARDNESS RESULTS. A different line of research has tried to develop more efficient algorithms, or lower bounds for the complexity of this problem. In particular, in [28] it is proved that finding the least closeness-central vertex is not subquadratic-time solvable, unless SETH is false. In the same line, it is proved in [3] that finding the most central vertex is not solvable in $O(m^{2-\epsilon})$, assuming the Hitting Set conjecture. This conjecture is very recent, and there are not strong evidences that it holds, apart from its similarity to the Orthogonal Vector conjecture. Conversely, the Orthogonal Vector conjecture is more established: it is implied both by the Hitting Set conjecture [3], and by SETH [122], a widely used assumption in the context of polynomial-time reductions [1–4, 27, 28, 63, 103, 109, 122, 124]. In [21], the authors prove that this result holds also if we assume SETH and the input graph is sparse and directed.

APPROXIMATION ALGORITHMS. In order to deal with the above hardness results, it is possible to design approximation algorithms: the simplest approach samples the distance between a node v and l other nodes w, and returns the average of all values $d(v, w)$ found [50]. The time complexity is $O(lm)$, to obtain an approximation $\tilde{c}_C(v)$ of the centrality of each node v such that $\mathbb{P}\left(\left|\frac{1}{\tilde{c}_C(v)} - \frac{1}{c_C(v)}\right| \geq \epsilon\,\mathsf{diam}\right) \leq 2e^{-\Omega\left(l\epsilon^2\right)}$, where diam is the diameter of the graph (we recall the diameter is the maximum distance between any two connected nodes). A more refined approximation algorithm is provided in [41], which combines the sampling approach with a 3-approximation algorithm: this algorithm has running time $O(lm)$, and it provides an estimate $\tilde{c}_C(v)$ of the centrality of each node v such that $\mathbb{P}\left(\left|\frac{1}{\tilde{c}_C(v)} - \frac{1}{c_C(v)}\right| \geq \frac{\epsilon}{c_C(v)}\right) \leq 2e^{-\Omega\left(l\epsilon^3\right)}$ (note that, differently from the previous algorithm, this algorithm provides a guarantee on the relative error). The recent result by Chechik et al. [35] allows to approximate closeness centrality with a coefficient of variation of ϵ using $O(\epsilon^{-2})$ single-source shortest path (SSSP) computations. Alternatively, one can make the probability that the maximum relative error exceeds ϵ polynomially small by using $O(\epsilon^{-2}\log n)$ SSSP computations.

However, these approximation algorithms have not been specifically designed for ranking nodes according to their closeness centrality, and turning them into a trustable top-k algorithm can be a challenging problem. Indeed, observe that, in many real-world cases, we work with so-called *small-world* networks, having a low diameter. Hence, in a typical graph, the average distance between v and a random node w is between 1 and 10. This implies that most of the n values $\frac{1}{c_C(v)}$ lie in this range, and that, in order to obtain a reliable ranking, we need the error to be close to $\epsilon = \frac{10}{n}$, which might be very small in the vast majority of real-word networks. As an example, performing $O(\epsilon^{-2})$ SSSPs as in [35] would then require $O(\frac{m}{\epsilon^2}) = O(mn^2)$ time in the unweighted case, which is impractical for large graphs.

Finally, an approximation algorithm was proposed in [96], where the sampling technique developed in [50] was used to actually compute the top k vertices: the result is not exact, but it is exact with high probability. The authors proved that the time complexity of their algorithm is $O(mn^{\frac{2}{3}}\log n)$, under the rather strong assumption that closeness centralities are uniformly distributed between 0 and the diameter diam (in the worst case, the time complexity of this algorithm is $O(mn)$).

HEURISTICS. Some works have tried to exploit properties of real-world networks in order to find more efficient algorithms. In [78], the authors develop a heuristic to compute the k most central vertices according to different measures. The basic idea is to identify central nodes according to a simple centrality measure (for instance, degree of nodes), and then to inspect a small set of central nodes according to this measure, hoping it contains the top k vertices according to the "complex" measure. The last approach [97], proposed by Olsen et al., tries to exploit the properties of real-world networks in order to develop exact algorithms with worst case complexity $O(mn)$, but performing much better in practice. As far as we know, this is the only exact algorithm that is able to efficiently compute the

k most central vertices in networks with up to 1 million nodes, before the work presented in Chapter 9.

DYNAMIC ALGORITHMS. Dynamic algorithms try to update some properties of the graph by limiting the computations to a subset of the nodes and edges. For updating the closeness of all nodes, a simple algorithm has been proposed by Kas et al. [66]. The authors use a dynamic algorithm by Ramalingam and Reps [104] for updating pairwise distances and either increase or decrease the closeness of nodes whose distance has changed.

For unweighted graphs, Sariyüce et al. [112] present optimizations that make the dynamic algorithm more efficient on complex networks. In particular, they show that the recomputation of closeness can be skipped for nodes s such that $|d(s,u) - d(s,v)| = 1$ (where u and v are the endpoints of the newly inserted or deleted edge). Also, they divide the graph into biconnected components and show that nodes outside the biconnected component of (u,v) can also be skipped. Finally, they notice that nodes with the same neighborhood have the same closeness, and therefore (re)computing it for only one of the nodes is sufficient. Differently from the algorithm by Kas et al. [66], the one by Sariyüce et al. [112] does not store pairwise distances, resulting in a memory requirement of $\Theta(n)$. Nevertheless, both algorithms require to compute exact closeness centrality at least once on the initial graph.

SOFTWARE LIBRARIES. Despite this huge amount of research, graph libraries still use the textbook algorithm: among them, Boost Graph Library [59], igraph [120] and NetworkX [115]. This is due to the fact that efficient available exact algorithms for top-k closeness centrality, like [97], are relatively recent and make use of several other non-trivial routines. We provide an implementation of the algorithm presented in Chapter 9 for NetworKit [119].

9 COMPUTING TOP-K CLOSENESS CENTRALITY

9.1 INTRODUCTION

In this chapter, we present a new algorithm for computing the k nodes with highest closeness in a real-world unweighted network. In fact, many research areas are interested in the most central nodes of the network, rather than in the closeness value of each single node. For example, somebody willing to open a store might be interested in knowing one or a few locations that are close, on average, to many potential customers and not in the closeness of each possible location. We provide a new exact algorithm that is much faster than computing closeness for all nodes in real-world networks, making it possible to compute the k most central vertices in networks with millions of nodes and hundreds of millions of edges. The new approach combines the BFS-based algorithm with a pruning technique: during the algorithm, we compute and update upper bounds on the closeness of all the nodes, and we exclude a node v from the computation as soon as its upper bound is "small enough", that is, we are sure that v does not belong to the top k nodes. We propose two different strategies to set the initial bounds, and two different strategies to update the bounds during the computation: this means that our algorithm comes in four different variants. The experimental results show that different variants perform well on different kinds of networks, and the best variant of our algorithm drastically outperforms both a probabilistic approach [96], and the best exact algorithm available until now [97]. We can now compute for the first time the 10 most central nodes in networks with millions of nodes and hundreds of millions of edges, and do so in very little time. Moreover, our approach is not only very efficient, but it is also very easy to code, making it a very good candidate to be implemented in existing graph libraries. We sketch the main ideas of the algorithm in Section 9.3, and we provide all details in Section 9.4-9.7. We experimentally evaluate the efficiency of the new algorithm in Section 9.8.

In case of disconnected graph, our results are described in terms of Lin's index (see Section 9.2), a well-known generalization of closeness centrality. However, they can be easily extended to any centrality measure in the form $c(v) = \sum_{w \neq v} f(d(v,w))$, where f is a decreasing function. The most popular among these measures is *harmonic centrality* [89], defined as $h(v) = \sum_{w \neq v} \frac{1}{d(v,w)}$. For the sake of completeness, in Section 9.8 we show that our algorithm performs well also for this measure.

In the last part of the chapter (Section 9.9, 9.10), we consider two case studies: the actor collaboration network ($\approx$ 2M vertices and $\approx$ 73M edges) and the Wikipedia citation network ($\approx$ 4M vertices and $\approx$ 102M edges). In the actor collaboration network, we analyze the evolution of the 10 most central vertices, considering snapshots taken every 5 years between 1940 and 2014. The computation was performed in little more than 45 minutes. In the Wikipedia case study, we consider both the standard citation network, that contains a directed edge (p, q) if p contains a link to q, and the reversed network, that contains a directed edge (p, q) if q contains a link to p. For most of these graphs, we are able to compute the 10 most central pages in a few minutes, making them available for further analyses.

Symbol	**Definition**
	Reachability set function
$R(v)$	Set of nodes reachable from v (by definition, $v \in R(v)$)
$r(v)$	$\|R(v)\|$
$\alpha(v)$	Lower bound on $r(v)$, that is, $\alpha(v) \leq r(v)$ (see Section 9.7.4)
$\omega(v)$	Upper bound on $r(v)$, that is, $r(v) \leq \omega(v)$ (see Section 9.7.4)
	Neighborhood functions
$N_d(v)$	Set of nodes at distance d from v: $\{w \in V : d(v,w) = d\}$
$N(v)$	Set of neighbors of v, that is $N_1(v)$
$n_d(v)$	Number of nodes at distance d from v, that is, $\|N_d(v)\|$
$\tilde{n}_d(v)$	Upper bound on $n_d(v)$ computed using the neighborhood-based lower bound (see Section 9.4)
$\tilde{n}_{d+1}(v)$	Upper bound on $n_{d+1}(v)$, defined as $\sum_{u \in N_d(v)} \mathtt{degree}(u) - 1$ if the graph is undirected, $\sum_{u \in N_d(v)} \mathtt{outdegree}(u)$ otherwise
$B_d(v)$	Set of nodes at distance *at most* d from v, that is, $\{w \in V : d(v,w) \leq d\}$
$b_d(v)$	Number of nodes at distance *at most* d from v, that is, $\|B_d(v)\|$
	Closeness functions
$c_C(v)$	Closeness of node v, that is, $\frac{(r(v)-1)^2}{(n-1)\sum_{w \in R(v)} d(v,w)}$
	Distance sum functions
$S(v)$	Total distance of node v, that is $\sum_{w \in R(v)} d(v,w)$
$S^{\mathrm{NB}}(v,r)$	Lower bound on $S(v)$ if $r(v) = r$, used in the `computeBoundsNB` function (see Prop. 9.4.1)
$S_d^{\mathrm{CUT}}(v,r)$	Lower bound on $S(v)$ if $r(v) = r$, used in the `updateBoundsBFSCut` function (see Lemma 9.5.1)
$S_s^{\mathrm{LB}}(v,r)$	Lower bound on $S(v)$ if $r(v) = r$, used in the `updateBoundsLB` function (see Eq. 23, 24)
	Farness functions
$f(v)$	Farness of node v, that is, $\frac{(n-1)S(v)}{(\|R(v)\|-1)^2}$
$L(v,r)$	Generic lower bound on $f(v)$, if $r(v) = r$
$L^{\mathrm{NB}}(v,r)$	Lower bound on $f(v)$, if $r(v) = r$, defined as $(n-1)\frac{S^{\mathrm{NB}}(v,r)}{(r-1)^2}$
$L_d^{\mathrm{CUT}}(v,r)$	Lower bound on $f(v)$, if $r(v) = r$, defined as $(n-1)\frac{S_d^{\mathrm{CUT}}(v,r)}{(r-1)^2}$
$L_s^{\mathrm{LB}}(v,r)$	Lower bound on $f(v)$, if $r(v) = r$, defined as $(n-1)\frac{S_s^{\mathrm{LB}}(v,r)}{(r-1)^2}$

Table 20: Notations used.

9.2 PRELIMINARIES

Our algorithmic results apply both to undirected and directed graphs. We will make clear in the respective context where results apply to only one of the two. All the notations and definitions used throughout this chapter are summarized in Table 20, but all notations are also defined in the text. Let us first define precisely the closeness centrality of a vertex v. In a connected graph, the farness of a node v in a graph $G = (V, E)$ is $f(v) = \frac{\sum_{w \in V} d(v,w)}{n-1}$, and the closeness centrality of v is $\frac{1}{f(v)}$. In the disconnected case, two are the most-common generalizations. One quite established generalization is Lin's index [83], defined as:

$$f(v) = \frac{\sum_{w \in R(v)} d(v,w)}{r(v)-1} \cdot \frac{n-1}{r(v)-1} \qquad c_C(v) = \frac{1}{f(v)} \tag{20}$$

where $R(v)$ is the set of vertices reachable from v, and $r(v) = |R(v)|$. If a vertex v has (out)degree 0, the previous fraction becomes $\frac{0}{0}$: in this case, the closeness of v is set to 0.

Another possibility is to consider a slightly different definition:

$$c_C(v) = \sum_{w \in V} f(d(v,w)),$$

for some decreasing function f. Usually, it is also assumed without loss of generality that $f(+\infty) = 0$, that is, we consider only reachable vertices: if this is not the case, it is enough to use a new function defined by $g(d) = f(d) - f(+\infty)$. One of the most common choices of f is $f(d) = \frac{1}{d}$: this way, we obtain *harmonic centrality* [89].

The work presented in this chapter will be described based on Lin's index, mostly because the best existing top-k closeness centrality algorithm uses this definition [97], and this allows for a simpler comparison. However, all our algorithms can be easily adapted to any centrality measure of the form $c(v) = \sum_{w \in V} f(d(v,w))$: indeed, in Section 9.8, we show that our algorithm performs very well also with harmonic centrality.

For simplicity, from now on, we will use the term *closeness* to indicate Lin's index and the term *farness* for the inverse of Lin's index, that is $\frac{1}{c_C(v)} := \frac{(n-1)\sum_{w \in R(v)} d(v,w)}{(r(v)-1)^2}$.

9.3 OVERVIEW OF THE ALGORITHM

In this section, we describe our new approach for computing the k nodes with maximum closeness (equivalently, the k nodes with minimum farness). If we have more than one node with the same score, we output all nodes having a centrality bigger than or equal to the centrality of the k-th node. The basic idea of our approach is to keep track of a lower bound on the farness of each node, and to skip the analysis of a vertex v if this lower bound implies that v is not among the top-k nodes.

More formally, let us assume that we know the farness of some vertices $v_1, \ldots, v_l$, and a lower bound $L(w)$ on the farness of any other vertex w. Furthermore, assume that there are k vertices among $v_1, \ldots, v_l$ verifying $f(v_i) < L(w)$ $\forall w \in V \setminus \{v_1, \ldots, v_l\}$, and hence $f(w) \geq L(w) > f(v_i)$ $\forall w \in V \setminus \{v_1, \ldots, v_l\}$. Then, we can safely skip the exact computation of $f(w)$ for all remaining nodes w, because the k vertices with smallest farness are among $v_1, \ldots, v_l$.

This idea is implemented in Algorithm 15: we use a list `Top` containing all "analysed" vertices $v_1, \ldots, v_l$ in increasing order of farness, and a priority queue `Q` containing all vertices "not analysed, yet", in increasing order of lower bound L (this way, the head of `Q` always has the smallest value of L among all vertices in `Q`). At the beginning, using the function `computeBounds`(), we compute a first bound $L(v)$ for each vertex v, and we fill the queue `Q` according to this bound. Then, at each step, we extract the first element v of `Q`: if $L(v)$ is larger than the k-th biggest farness computed until now (that is, the farness of the k-th vertex in variable `Top`), we can safely stop, because for each $x \in$ `Q`, $f(x) \geq L(x) \geq L(v) > f(\texttt{Top}[k])$, and x is not in the top k. Otherwise, we run the function `updateBounds`(v), which performs a BFS from v, returns the farness of v, and improves the bounds L of all other vertices. Finally, we insert v into `Top` in the right position, and we update `Q` if the lower bounds have changed.

Algorithm 15: Pseudocode of our algorithm for top-k closeness centralities.

```
   Input  : A graph $G = (V, E)$
   Output : Top $k$ nodes with highest closeness
1  global L, Q ← computeBounds($G$);
2  global Top ← [ ];
3  global Farn;
4  for $v \in V$ do Farn[$v$] = $+\infty$;
5  while Q is not empty do
6      $v$ ← Q.extractMin();
7      if |Top| ≥ $k$ and L[$v$] > Farn[Top[$k$]] then return Top;
8      Farn[$v$] ← updateBounds($v, L$); // This function might also modify L
9      add $v$ to Top, and sort Top according to Farn;
10     update Q according to the new bounds;
11 end
```

The crucial point of the algorithm is the definition of the lower bounds, that is, the definition of the functions `computeBounds` and `updateBounds`. We propose two alternative strategies for each of these these two functions: in both cases, one strategy is conservative, that is, it tries to perform as few operations as possible, while the other strategy is aggressive, that is, it needs many operations, but at the same time it improves many lower bounds.

Let us analyze the possible choices of the function `computeBounds`. The conservative strategy `computeBoundsDeg` needs time $O(n)$: it simply sets $L(v) = 0$ for each v, and it fills `Q` by inserting nodes in decreasing order of degree (the idea is that vertices with high degree have small farness, and they should be analysed as early as possible, so that the values in `Top` are correct as soon as possible). Note that the vertices can be sorted in time $O(n)$ using counting sort.

The aggressive strategy `computeBoundsNB` needs time $O(m\,\mathsf{diam})$, where diam is the diameter of the graph: it computes the neighborhood-based lower bound $L^{\mathrm{NB}}(v)$ for each vertex v (we will explain shortly afterwards how it works), it sets $L(v) = L^{\mathrm{NB}}(v)$, and it fills `Q` by adding vertices in decreasing order of L. The idea behind the neighborhood-based lower bound is to count the number of paths of length l starting from a given vertex v, which is also an upper bound U_l on the number of vertices at distance l from v. From U_l, it is possible to define a lower bound on $\sum_{x \in V} d(v, x)$ by "summing U_l times the distance l", until we have summed n distances: this bound yields the desired lower bound on the farness of v. The detailed explanation of this function is provided in Section 9.4.

For the function `updateBounds`(w), the conservative strategy `updateBoundsBFSCut`(w) does not improve L, and it cuts the BFS as soon as it is sure that the farness of w is larger than the k-th biggest farness found until now, that is, `Farn`[`Top`[k]]. If the BFS is cut, the function returns $+\infty$, otherwise, at the end of the BFS we have computed the farness of v, and we can return it. The running time of this procedure is $O(m)$ in the worst case, but it can be much better in practice. It remains to define how the procedure can be sure that the farness of v is at least x: to this purpose, during the BFS, we update a lower bound on the farness of v. The idea behind this bound is that, if we have already visited all nodes up to distance d, we can upper bound the closeness centrality of v by setting distance $d+1$ to a number of vertices equal to the number of edges “leaving” level d, and distance $d+2$ to all the remaining vertices. The details of this procedure are provided in Section 9.5.

The aggressive strategy `updateBoundsLB`(v) performs a complete BFS from v, and it bounds the farness of each node w using the level-based lower bound. The running time is $O(m)$ for the BFS, and $O(n)$ to compute the bounds. The idea behind the level-based lower bound is that $d(w,x) \geq |d(v,w) - d(v,x)|$, and consequently $\sum_{x \in V} d(w,x) \geq \sum_{x \in V} |d(v,w) - d(v,x)|$. The latter sum can be computed in time $O(n)$ for each w, because it depends only on the level d of w in the BFS tree, and because it is possible to compute in $O(1)$ the sum for a vertex at level $d+1$, if we know the sum for a vertex at level d. The details are provided in Section 9.6.

Finally, in order to transform these lower bounds on $\sum_{x \in V} d(v,x)$ into bounds on $f(v)$, we need to know the number of vertices reachable from a given vertex v. In Section 9.4, 9.5, 9.6, we assume that these values are known: this assumption is true in undirected graphs, where we can compute the number of reachable vertices in linear time at the beginning of the algorithm, and in strongly connected directed graphs, where the number of reachable vertices is n. The only remaining case is when the graph is directed and not strongly connected: in this case, we need some additional machinery, which are presented in Section 9.7.

9.4 NEIGHBORHOOD-BASED LOWER BOUND

In this section, we propose a lower bound $S^{\mathrm{NB}}(v, r(v))$ on the total sum of distances $S(v) = \sum_{w \in R(v)} d(v,w)$ of an undirected or strongly-connected graph. If we know the number $r(v)$ of vertices reachable from v, this bound translates into a lower bound on the farness of v, simply multiplying by $(n-1)/(r(v)-1)^2$. The basic idea is to find an upper bound $\tilde{n}_i(v)$ on the number of nodes $n_i(v)$ at distance i from v. Then, intuitively, if we assume that the number of nodes at distance i is greater than its actual value and “stop counting” when we have $r(v)$ nodes, we get something that is smaller than the actual total distance. This is because we are assuming that the distances of some nodes are smaller than their actual values. This argument is formalized in Proposition 9.4.1.

Proposition 9.4.1. *If $\tilde{n}_i(v)$ is an upper bound on $n_i(v)$, for $i = 0, \dots, \mathsf{diam}(G)$ and $\mathrm{ecc}(v) := \max_{w \in r(v)} d(v,w)$, then*

$$S^{NB}(v, r(v)) := \sum_{k=1}^{\mathrm{ecc}(v)} k \cdot \min\left\{\tilde{n}_k(v),\ \max\left\{r(v) - \sum_{i=0}^{k-1} \tilde{n}_i(v),\ 0\right\}\right\}$$

is a lower bound on $S(v)$.

Proof. First, we notice that $S(v) = \sum_{k=0}^{\mathrm{ecc}(v)} k \cdot n_k(v)$ and $r(v) = \sum_{k=0}^{\mathrm{ecc}(v)} n_k(v)$.

Let us assume that $\tilde{n}_0(v) < r(v)$. In fact, if $\tilde{n}_0(v) \geq r(v)$, the statement is trivially satisfied. Then, there must be a number $\mathrm{ecc}' > 0$ such that for $k < \mathrm{ecc}'$ the quantity $\min\left\{\tilde{n}_k(v),\ \max\left\{r(v) - \sum_{i=0}^{k-1} \tilde{n}_i(v),\ 0\right\}\right\}$ is equal to $\tilde{n}_k(v)$, for $k = \mathrm{ecc}'$, the quantity is

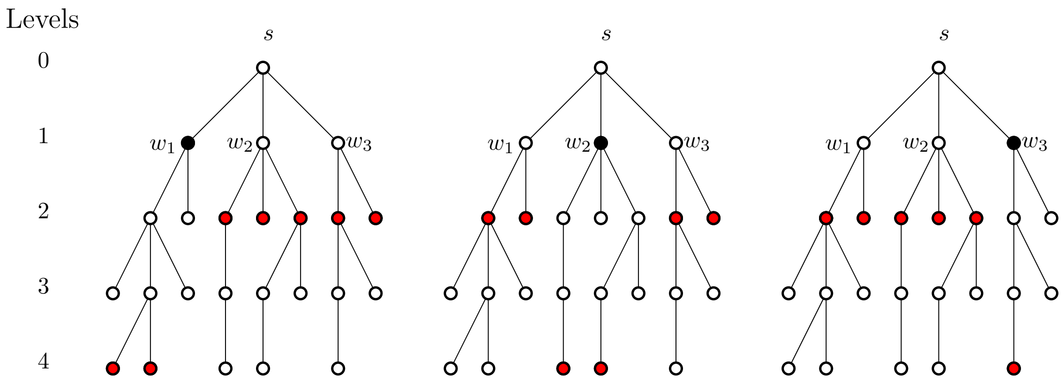


Figure 27: Relation between nodes at distance 4 for s and the neighbors of s. The red nodes represent the nodes at distance 3 for w_1 (left), for w_2 (center) and for w_3 (right).

equal to $\alpha := r(v) - \sum_{k=0}^{\text{ecc}'-1} \tilde{n}_k(v) > 0$ and, for $k > \text{ecc}'$, it is equal to 0. Therefore we can write $S^{\text{NB}}(v, r(v))$ as $\sum_{k=1}^{\text{ecc}'-1} k \cdot \tilde{n}_k(v) + \text{ecc}' \cdot \alpha$.

We show that $\text{ecc}' \leq \text{ecc}(v)$. In fact, we know that $\sum_{k=0}^{\text{ecc}'-1} \tilde{n}_k(v) < r(v) = \sum_{k=0}^{\text{ecc}(v)} n_k(v) \leq \sum_{k=0}^{\text{ecc}(v)} \tilde{n}_k(v)$. Therefore $\text{ecc}' - 1 < \text{ecc}(v)$, which implies $\text{ecc}' \leq \text{ecc}(v)$.

For each i, we can write $\tilde{n}_i(v) = n_i(v) + \epsilon_i$, $\epsilon_i \geq 0$. Therefore, we can write $\sum_{k=0}^{\text{ecc}'-1} \epsilon_i + \alpha = r(v) - \sum_{k=0}^{\text{ecc}'-1} n_k(v) = \sum_{k=\text{ecc}'}^{\text{ecc}(v)} n_k(v)$. Then, $S^{\text{NB}}(v, r(v)) = \sum_{k=0}^{\text{ecc}'-1} k \cdot n_k(v) + \sum_{k=0}^{\text{ecc}'-1} k \cdot \epsilon_i + \text{ecc}' \cdot \alpha \leq \sum_{k=0}^{\text{ecc}'-1} k \cdot n_k(v) + \text{ecc}'(\alpha + \sum_{k=0}^{\text{ecc}'-1} \epsilon_i) = \sum_{k=0}^{\text{ecc}'-1} k \cdot n_k(v) + \text{ecc}'(\sum_{k=\text{ecc}'}^{\text{ecc}(v)} n_k(v)) \leq \sum_{k=0}^{\text{ecc}(v)} k \cdot n_k(v) = S(v)$. □

In the following paragraphs, we propose upper bounds $\tilde{n}_i(v)$ for trees, undirected graphs and directed strongly-connected graphs. In case of trees, the bound $\tilde{n}_i(v)$ is actually equal to $n_i(v)$, which means that the algorithm can be used to compute closeness of all nodes in a tree exactly.

COMPUTING CLOSENESS ON TREES. Let us consider a node s for which we want to compute the total distance $S(s)$ (notice that in a tree $c_C(s) = (n-1)/S(s)$). The number of nodes at distance 1 in the BFS tree from s is clearly the degree of s. What about distance 2? Since there are no cycles, all the neighbors of the nodes in $N_1(s)$ are nodes at distance 2 from s, with the only exception of s itself. Therefore, naming $N_k(s)$ the set of nodes at distance k from s and $n_k(s)$ the number of these nodes, we can write $n_2(s) = \sum_{w \in N_1(s)} n_1(w) - \mathsf{degree}(s)$. In general, we can always relate the number of nodes at each distance k of s to the number of nodes at distance $k-1$ in the BFS trees of the neighbors of s. Let us now consider $n_k(s)$, for $k > 2$. Figure 27 shows an example where s has three neighbors w_1, w_2 and w_3. Suppose we want to compute $N_4(s)$ using information from w_1, w_2 and w_3. Clearly, $N_4(s) \subset N_3(w_1) \cup N_3(w_2) \cup N_3(w_3)$; however, there are also other nodes in the union that are not in $N_4(s)$. Furthermore, the nodes in $N_3(w_1)$ (red nodes in the leftmost tree) are of two types: nodes in $N_4(s)$ (the ones in the subtree of w_1) and nodes in $N_2(s)$ (the ones in the subtrees of w_2 and w_3). An analogous behavior can be observed for w_2 and w_3 (central and rightmost trees). If we simply sum all the nodes in $n_3(w_1)$, $n_3(w_2)$ and $n_3(w_3)$, we would be counting each node at level 2 twice, i. e.once for each node in $N_1(s)$ minus one. Hence, for each $k > 2$, we can write

$$n_k(s) = \sum_{w \in N_1(s)} n_{k-1}(w) - n_{k-2}(s) \cdot (\mathsf{degree}(s) - 1). \qquad (21)$$

Algorithm 16: Closeness centrality in trees

```
   Input  : A tree $T = (V, E)$
   Output : Closeness centralities $c_C(v)$ of each node $v \in V$
 1 $\mathsf{k} \leftarrow 2$;
 2 foreach $s \in V$ do
 3     $n_{k-1}(s) \leftarrow \mathsf{degree}(s)$;
 4     $S(s) \leftarrow \mathsf{degree}(s)$;
 5 end
 6 $\mathsf{nFinished} \leftarrow 0$;
 7 while $\mathsf{nFinished} < n$ do
 8     foreach $s \in V$ do
 9         if $\mathsf{k} = 2$ then
10             $n_k(s) \leftarrow \sum_{w \in N(s)} n_{k-1}(w) - \mathsf{degree}(s)$;
11         end
12         else
13             $n_k(s) \leftarrow \sum_{w \in N(s)} n_{k-1}(w) - n_{k-2}(s)(\mathsf{degree}(s) - 1)$;
14         end
15     end
16     foreach $s \in V$ do
17         $n_{k-2}(s) \leftarrow n_{k-1}(s)$;
18         $n_{k-1}(s) \leftarrow n_k(s)$;
19         if $n_{k-1}(s) > 0$ then
20             $S(s) \leftarrow S(s) + \mathsf{k} \cdot n_{k-1}(s)$;
21         end
22         else
23             $\mathsf{nFinished} \leftarrow \mathsf{nFinished} + 1$;
24         end
25     end
26     $\mathsf{k} \leftarrow \mathsf{k} + 1$;
27 end
28 foreach $s \in V$ do
29     $c_C(v) \leftarrow (n-1)/S(v)$;
30 end
31 return $c$
```

From this observation, we define a new method to compute the total distance of all nodes, described in Algorithm 16. Instead of computing the BFS tree of each node one by one, at each step we compute the number $n_k(v)$ of nodes at level k for *all* nodes v. First (Lines 2 - 4), we compute $n_1(v)$ for each node (and add that to $S(v)$). Then (Lines 7 - 26), we consider all the other levels k one by one. For each k, we use $n_{k-1}(w)$ of the neighbors w of v and $n_{k-2}(v)$ to compute $n_k(v)$ (Line 10 and 13). If, for some k, $n_k(v) = 0$, all the nodes have been added to $S(v)$. Therefore, we can stop the algorithm when $n_k(v) = 0 \ \ \forall v \in V$.

Proposition 9.4.2. *Algorithm 16 requires $O(\mathsf{diam} \cdot n)$ operations to compute the closeness centrality of all nodes in a tree T.*

Proof. The for loop in Lines 2 - 4 of Algorithm 16 clearly takes $O(n)$ time. For each level of the while loop of Lines 7 - 26, each node scans its neighbors in Line 10 or Line 13. In total, this leads to $O(n)$ operations per level since $m = O(n)$. Since the maximum number of levels that a node can have is equal to the diameter of the tree, the algorithm requires $O(\mathsf{diam} \cdot n)$ operations.

□

Note that closeness centrality on trees could even be computed in time $O(n)$ with a bottom-up followed by a top-down traversal (see [33] for more details). We choose to include Algorithm 16 here nonetheless since it paves the way for an algorithm computing a lower bound in general undirected graphs, described next.

LOWER BOUND FOR UNDIRECTED GRAPHS. For general undirected graphs, Eq. (21) is not true anymore – but a related upper bound $\tilde{n}_k(\cdot)$ on $n_k(\cdot)$ is still useful. Let $\tilde{n}_k(s)$ be defined recursively as in Eq. (21): in a tree, $\tilde{n}_k(s) = n_k(s)$, while in this case we prove that $\tilde{n}_k(s)$ is an upper bound on $N_k(s)$. Indeed, there could be nodes x for which there are multiple paths between s and x and that are therefore contained in the subtrees of more than one neighbor of s. This means that we would count x multiple times when considering $\tilde{n}_k(s)$, overestimating the number of nodes at distance k. However, we know for sure that at level k there cannot be *more nodes* than in Eq. (21). If, for each node v, we assume that the number $\tilde{n}_k(v)$ of nodes at distance k is that of Eq. (21), we can apply Proposition 9.4.1 and get a lower bound $S^{\text{NB}}(v, r(v))$ on the total sum for undirected graphs. The procedure is described in Algorithm 17. The computation of $S^{\text{NB}}(v, r(v))$ works basically like Algorithm 16, with the difference that here we keep track of the number of the nodes found in all the levels up to k ($\mathsf{nVisited}$) and stop the computation when $\mathsf{nVisited}$ becomes equal to $r(v)$ (if it becomes larger, in the last level we consider only $r(v) - \mathsf{nVisited}$ nodes, as in Proposition 9.4.1 (Lines 28 - 31).

Proposition 9.4.3. *For an undirected graph G, computing the lower bound $S^{NB}(v, r(v))$ described in Algorithm 17 takes $O(\mathsf{diam} \cdot m)$ time.*

Proof. Like in Algorithm 16, the number of operations performed by Algorithm 17 at each level of the while loop is $O(m)$. At each level i, all the nodes at distance i are accounted for (possibly multiple times) in Lines 12 and 15. Therefore, at each level, the variable $\mathsf{nVisited}$ is always greater than or equal to the the number of nodes v at distance $d(v) \leq i$. Since $d(v) \leq \mathsf{diam}$ for all nodes v, the maximum number of levels scanned in the while loop cannot be larger than diam, therefore the total complexity is $O(\mathsf{diam} \cdot m)$.

□

LOWER BOUND ON DIRECTED GRAPHS. In directed graphs, we can simply consider the out-neighbors, without subtracting the number of nodes discovered in the subtrees of the other neighbors in Eq. (21). The lower bound (which we still refer to as $S^{\text{NB}}(v, r(v))$) is obtained by replacing Eq. (21) with the following in Lines 12 and 15 of Algorithm 17:

$$\tilde{n}_k(s) = \sum_{w \in N(s)} \tilde{n}_{k-1}(w) \tag{22}$$

Algorithm 17: Neighborhood-based lower bound for undirected graphs

```
Input   : A graph $G = (V, E)$
Output  : Lower bounds $L^{\mathrm{NB}}(v, r(v))$ of each node $v \in V$
$k \leftarrow 2$;
foreach $s \in V$ do
    $n_{k-1}(s) \leftarrow \mathsf{degree}(s)$;
    $\tilde{S}^{(\mathrm{un})}(s) \leftarrow \mathsf{degree}(s)$;
    $\mathsf{nVisited}[s] \leftarrow \mathsf{degree}(s) + 1$;
    $\mathsf{finished}[s] \leftarrow false$;
end
$\mathsf{nFinished} \leftarrow 0$;
while $\mathsf{nFinished} < n$ do
    foreach $s \in V$ do
        if $k = 2$ then
            $n_k(s) \leftarrow \sum_{w \in N(s)} n_{k-1}(w) - \mathsf{degree}(s)$;
        end
        else
            $n_k(s) \leftarrow \sum_{w \in N(s)} n_{k-1}(w) - n_{k-2}(s)(\mathsf{degree}(s) - 1)$;
        end
    end
    foreach $s \in V$ do
        if $\mathsf{finished}[v]$ then
            continue;
        end
        $n_{k-2}(s) \leftarrow n_{k-1}(s)$;
        $n_{k-1}(s) \leftarrow n_k(s)$;
        $\mathsf{nVisited}[s] \leftarrow \mathsf{nVisited}[s] + n_{k-1}(s)$;
        if $\mathsf{nVisited}[s] < r(v)$ then
            $\tilde{S}^{(\mathrm{un})}(s) \leftarrow \tilde{S}^{(\mathrm{un})}(s) + k \cdot n_{k-1}(s)$;
        end
        else
            $\tilde{S}^{(\mathrm{un})}(s) \leftarrow \tilde{S}^{(\mathrm{un})}(s) + k(r(v) - (\mathsf{nVisited}[s] - n_{k-1}(s)))$;
            $\mathsf{nFinished} \leftarrow \mathsf{nFinished} + 1$;
            $\mathsf{finished}[s] \leftarrow true$;
        end
    end
    $k \leftarrow k + 1$;
end
foreach $v \in v$ do
    $L^{\mathrm{NB}}(v, r(v)) \leftarrow \frac{(n-1)\tilde{S}^{(\mathrm{un})}}{(r(v)-1)^2}$;
end
return $L^{NB}(\cdot, r(\cdot))$
```

9.5 THE `UPDATEBOUNDSBFSCUT` FUNCTION

The `updateBoundsBFSCut` function is based on a simple idea: if the k-th biggest farness found until now is x, and if we are performing a BFS from vertex v to compute its farness $f(v)$, we can stop as soon as we can guarantee that $f(v) \geq x$.

Informally, assume that we have already visited all nodes up to distance d: we can lower bound $S(v) = \sum_{w \in V} d(v, w)$ by setting distance $d+1$ to a number of vertices equal to the number of edges "leaving" level d, and distance $d+2$ to all the remaining reachable vertices. Then, this bound yields a lower bound on the farness of v. As soon as this lower bound is bigger than x, the `updateBoundsBFSCut` function may stop; if this condition never occurs, at the end of the BFS we have exactly computed the farness of x.

More formally, the following lemma defines a lower bound $S_d^{\mathrm{CUT}}(v, r(v))$ on $S(v)$, which is computable after we have performed a BFS from v up to level d, assuming we know the number $r(v)$ of vertices reachable from v (this assumption is lifted in Section 9.7).

Lemma 9.5.1. *Given a graph $G = (V, E)$, a vertex $v \in V$, and an integer $d \geq 0$, let $B_d(v)$ be the set of vertices at distance at most d from v, $b_d(v) = |B_d(v)|$, and let $\tilde{n}_{d+1}(v)$ be an upper bound on the number of vertices at distance $d+1$ from v (see Table 20). Then,*

$$S(v) \geq S_d^{CUT}(v, r(v)) := \sum_{w \in B_d(v)} d(v, w) - \tilde{n}_{d+1}(v) + (d+2)(r(v) - b_d(v)).$$

Proof. The sum of all the distances from v is lower bounded by setting the correct distance to all vertices at distance at most d from v, by setting distance $d+1$ to all vertices at distance $d+1$ (there are $n_{d+1}(v)$ such vertices), and by setting distance $d+2$ to all other vertices (there are $r(v) - b_{d+1}(v)$ such vertices, where $r(v)$ is the number of vertices reachable from v and $b_{d+1}(v)$ is the number of vertices at distance at most $d+1$). More formally, $f(v) \geq \sum_{w \in B_d(v)} d(v, w) + (d+1)n_{d+1}(v) + (d+2)(r(v) - b_{d+1}(v))$.

Since $b_{d+1}(v) = n_{d+1}(v) + b_d(v)$, we obtain that $f(v) \geq \sum_{w \in B_d(v)} d(v, w) - n_{d+1}(v) + (d+2)(r(v) - b_d(v))$. We conclude because, by assumption, $\tilde{n}_{d+1}(v)$ is an upper bound on $n_{d+1}(v)$.

□

Corollary 9.5.1. *For each vertex v and for each $d \geq 0$,*

$$f(v) \geq L_d^{CUT}(v, r(v)) := \frac{(n-1)S_d^{CUT}(v, r(v))}{(r(v)-1)^2}.$$

It remains to define the upper bound $\tilde{n}_{d+1}(v)$: in the directed case, this bound is simply the sum of the out-degrees of vertices at distance d from v. In the undirected case, since at least an edge from each vertex $v \in N_d(v)$ is directed towards $N_{d-1}(v)$, we may define $\tilde{n}_{d+1}(v) = \sum_{w \in N_d(v)} \deg(w) - 1$ (the only exception is $d = 0$: in this case, $\tilde{n}_1(v) = n_1(v) = \deg(v)$).

Remark 9.5.1. *When we are processing vertices at level d, if we process an edge (x, y) where y is already in the BFS tree, we can decrease $\tilde{n}_{d+1}(v)$ by one, obtaining a better bound.*

Assuming we know $r(v)$, all quantities necessary to compute $L_d^{\mathrm{CUT}}(v, r(v))$ are available as soon as all vertices in $B_d(v)$ are visited by a BFS. This function performs a BFS starting

Algorithm 18: The `updateBoundsBFSCut`(v) function in the case of directed graphs, if $r(v)$ is known for each v.

```
x ← Farn(Top[k]); // Farn and Top are global variables, as in Algorithm 15.
Create queue $Q$;
$Q$.enqueue($v$);
Mark $v$ as visited;
$d \leftarrow 0$; $S \leftarrow 0$; $\tilde{n} \leftarrow \text{outdeg}(v)$; $nd \leftarrow 1$;
while $Q$ is not empty do
    $u \leftarrow Q$.dequeue();
    if $d(v,u) > d$ then
        $d \leftarrow d+1$;
        $L_d^{\text{CUT}}(v,r(v)) \leftarrow \frac{(n-1)(S-\tilde{n}+(d+2)(r(v)-nd))}{(r(v)-1)^2}$;
        if $L_d^{CUT}(v,r(v)) \geq x$ then return $+\infty$;
        $\tilde{n} \leftarrow 0$
    end
    for $w$ in adjacency list of $u$ do
        if $w$ is not visited then
            $S \leftarrow S + d(v,w)$;
            $\tilde{n} \leftarrow \tilde{n} + \text{outdegree}(w)$;
            $nd \leftarrow nd + 1$;
            $Q$.enqueue($w$);
            Mark $w$ as visited
        end
        else
            // we use Remark 9.5.1
            $L_d^{\text{CUT}}(v,r(v)) \leftarrow L_d^{\text{CUT}}(v,r(v)) + \frac{(n-1)}{(r(v)-1)^2}$;
            if $L_d^{CUT}(v,r(v)) \geq x$ then return $x$;
        end
    end
end
return $\frac{S(n-1)}{(r(v)-1)^2}$;
```

from v, continuously updating the upper bound $L_d^{\text{CUT}}(v,r(v)) \leq f(v)$ (the update is done whenever all nodes in $N_d(v)$ have been reached, or Remark 9.5.1 can be used). As soon as $L_d^{\text{CUT}}(v,r(v)) \geq x$, we know that $f(v) \geq L_d^{\text{CUT}}(v,r(v)) \geq x$, and we return $+\infty$.

Algorithm 18 is the pseudocode of the function `updateBoundsBFSCut` when implemented for directed graphs, assuming we know the number $r(v)$ of vertices reachable from each v (for example, if the graph is strongly connected). This code can be easily adapted to all the other cases.

9.6 THE UPDATEBOUNDLB FUNCTION

Differently from `updateBoundsBFSCut` function, `updateBoundsLB` computes a complete BFS traversal, but uses information acquired during the traversal to update the bounds on the other nodes. Let us first consider an undirected graph G and let s be the source node from which we are computing the BFS. We can see the distances $d(s,v)$ between s and all the nodes v reachable from s as *levels*: node v is at level i if and only if the distance between s and v is i, and we write $v \in N_i(s)$ (or simply $v \in N_i$ if s is clear from the context). Let i and j be two levels, $i \leq j$. Then, the distance between any two nodes v at level i and w at level j must be at least $j-i$. Indeed, if $d(v,w)$ was smaller than $j-i$,

w would be at level $i + d(v, w) < j$, which contradicts our assumption. It follows directly that $\sum_{w \in V} |d(s, w) - d(s, v)|$ is a lower bound on $S(v)$, for all $v \in R(s)$:

Lemma 9.6.1. *$\sum_{w \in R(s)} |d(s, w) - d(s, v)| \leq S(v) \quad \forall v \in R(s)$.*

To improve the approximation, we notice that the number of nodes at distance 1 from v is exactly the degree of v. Therefore, all the other nodes w such that $|d(s, v) - d(s, w)| \leq 1$ must be at least at distance 2 (with the only exception of v itself, whose distance is of course 0). This way we can define the following lower bound on $S(v)$:

$$2(\#\{w \in R(s) : |d(s, w) - d(s, v)| \leq 1\} - \mathsf{degree}(v) - 1) + \\ + \mathsf{degree}(v) + \sum_{\substack{w \in R(s) \\ |d(s,w)-d(s,v)|>1}} |d(s, w) - d(s, v)|,$$

that is:

$$2 \cdot \sum_{|j-d(s,v)| \leq 1} n_j + \sum_{|j-d(s,v)|>1} n_j \cdot |j - d(s, v)| - \mathsf{degree}(v) - 2, \tag{23}$$

where $n_j = |N_j|$.

Multiplying the bound of Eq. (23) by $\frac{(n-1)}{(r(v)-1)^2}$, we obtain a lower bound on the farness $f(v)$ of node v, named $L_s^{\mathrm{LB}}(v, r(v))$. A straightforward way to compute $L_s^{\mathrm{LB}}(v, r(v))$ would be to first run the BFS from s and then, for each node v, to consider the level difference between v and all the other nodes. This would require $O(n^2)$ operations, which is clearly too expensive. However, we can notice two things: First, the bounds of two nodes at the same level differ only by their degree. Therefore, for each level i, we can compute $2 \cdot \sum_{|j-i| \leq 1} n_j + \sum_{|j-i|>1} n_j \cdot |j - i| - 2$ only once and then subtract $\mathsf{degree}(v)$ for each node at level i. We call the quantity $2 \cdot \sum_{|j-i| \leq 1} n_j + \sum_{|j-i|>1} n_j \cdot |j - i| - 2$ the level-bound $\mathsf{L}(i)$ of level i. Second, we can prove that $\mathsf{L}(i)$ can actually be written as a function of $\mathsf{L}(i - 1)$.

Lemma 9.6.2. *Let $\mathsf{L}(i) := 2 \cdot \sum_{|j-i| \leq 1} n_j + \sum_{|j-i|>1} n_j \cdot |j - i| - 2$. Also, let $n_j = 0$ for $j \leq 0$ and $j > \mathsf{maxD}$, where $\mathsf{maxD} = \max_{v \in R(s)} d(s, v)$. Then $\mathsf{L}(i) - \mathsf{L}(i - 1) = \sum_{j<i-2} n_j - \sum_{j>i+1} n_j$, $\forall i \in \{1, ..., \mathsf{maxD}\}$.*

Proof. Since $n_j = 0$ for $j \leq 0$ and $j > \mathsf{maxD}$, we can write $\mathsf{L}(i)$ as $2 \cdot (n_{i-1} + n_i + n_{i+1}) + \sum_{|j-i|>1} n_j \cdot |j - i| - 2$, $\forall i \in \{1, ..., \mathsf{maxD}\}$. The difference between $\mathsf{L}(i)$ and $\mathsf{L}(i - 1)$ is: $2 \cdot (n_{i-1} + n_i + n_{i+1}) + \sum_{|j-i|>1} |j - i| \cdot n_j - 2 \cdot (n_{i-2} + n_{i-1} + n_i) + \sum_{|j-i+1|>1} |j - i + 1| \cdot n_j = 2 \cdot (n_{i+1} - n_{i-2}) + 2 \cdot n_{i-2} - 2 \cdot n_{i+1} + \sum_{j<i-2 \cup j>i+1} (|j - i| - |j - i + 1|) \cdot n_j = \sum_{j<i-2} n_j - \sum_{j>i+1} n_j$.

□

Algorithm 19 describes the computation of $L_s^{\mathrm{LB}}(v, r(v))$. First, we compute all the distances between s and the nodes in $R(s)$ with a BFS, storing the number of nodes in each level and the number of nodes in levels $j \leq i$ and $j > i$ respectively (Lines 1 - 9). Then we compute the level bound $\mathsf{L}(1)$ of level 1 according to its definition (Line 10) and those of the other level according to Lemma 9.6.2 (Line 12). The lower bound $L_s^{\mathrm{LB}}(v, r(v))$ is then computed for each node v by subtracting its degree to $\mathsf{L}(d(s, v))$ and normalizing (Line 16). The complexity of Lines 1 - 9 is that of running a BFS, i. e.$O(n + m)$. Line 12 is

Algorithm 19: The `updateBoundsLB` function for undirected graphs

```
Input  : A graph $G = (V, E)$, a source node $s$
Output : Lower bounds $L_s^{\mathrm{LB}}(v, r(v))$ of each node $v \in R(s)$
$d \leftarrow$ BFSfrom($s$);
maxD $\leftarrow \max_{v \in V} d(s, v)$;
sum$N_{\leq 0} \leftarrow 0$; sum$N_{\leq -1} \leftarrow 0$; sum$N_{>\mathsf{maxD}+1} \leftarrow 0$;
for $i = 1, 2, \ldots,$ maxD do
    $N_i \leftarrow \{w \in V : d(s, w) = i\}$;
    $n_i \leftarrow \#N_i$;
    sum$N_{\leq i} \leftarrow$ sum$N_{\leq i-1} + n_i$;
    sum$N_{>i} \leftarrow |V| -$ sum$N_{\leq i}$;
end
L(1) $\leftarrow n_1 + n_2 +$ sum$N_{>2} - 2$;
for $i = 2, \ldots,$ maxD do
    L($i$) $\leftarrow$ L($i-1$) + sum$N_{\leq i-3}$ − sum$N_{>i+1}$;
end
for $i = 1, \ldots,$ maxD do
    foreach $v \in N_i$ do
        $L_s^{\mathrm{LB}}(v, r(v)) \leftarrow (\mathsf{L}(i) - \mathsf{degree}(v)) \cdot \frac{(n-1)}{(r(v)-1)^2}$;
    end
end
return $L_s^{LB}(v, r(v)) \quad \forall v \in V$
```

repeated once for each level (which cannot be more than n) and Line 16 is repeated once for each node in $R(s)$. Therefore, the following proposition holds.

Proposition 9.6.1. *Computing the lower bound $L_s^{LB}(v, r(v))$ takes $O(n + m)$ time.*

For directed strongly-connected graphs, the result does not hold for nodes w whose level is smaller than $l(v)$, since there might be a directed edge or a shortcut from v to w. Yet, for nodes w such that $d(s, w) > d(s, v)$, it is still true that $d(v, w) \geq d(s, w) - d(s, v)$. For the remaining nodes (apart from the outgoing neighbors of v), we can only say that the distance must be at least 2. The upper bound $L_s^{\mathrm{LB}}(v, r(v))$ for directed graphs can therefore be defined as:

$$2 \cdot \#\{w \in R(s) : d(s, w) - d(s, v) \leq 1\} + \sum_{\substack{w \in R(s) \\ d(s,w) - d(s,v) > 1}} (d(s, w) - d(s, v)) - \mathsf{degree}(v) - 2. \tag{24}$$

The computation of $L_s^{\mathrm{LB}}(v, r(v))$ for directed strongly-connected graphs is analogous to the one described in Algorithm 19.

9.7 THE DIRECTED DISCONNECTED CASE

In the directed disconnected case, even if the time complexity of computing strongly connected components is linear in the input size, the time complexity of computing the number of reachable vertices is much bigger (assuming SETH, it cannot be $O(m^{2-\epsilon})$ [27]). For this reason, when computing our upper bounds, we cannot rely on the exact value of $r(v)$: for now, let us assume that we know a lower bound $\alpha(v) \leq r(v)$ and an upper bound $\omega(v) \geq r(v)$. The definition of these bounds is postponed to Section 9.7.4.

Furthermore, let us assume that we have a lower bound $L(v, r(v))$ on the farness of v, depending on the number $r(v)$ of vertices reachable from v: in order to obtain a bound not depending on $r(v)$, the simplest approach is $f(v) \geq L(v, r(v)) \geq \min_{\alpha(v) \leq r \leq \omega(v)} L(v, r)$. However, during the algorithm, computing the minimum among all these values might be quite expensive, if $\omega(v) - \alpha(v)$ is big. In order to solve this issue, we find a small set $X \subseteq [\alpha(v), \omega(v)]$ such that $\min_{\alpha(v) \leq r \leq \omega(v)} L(v, r) = \min_{r \in X} L(v, r)$.

More specifically, we find a condition that is verified by "many" values of r, and that implies $L(v, r) \geq \min(L(v, r-1), L(v, r+1))$: this way, we may define X as the set of values of r that either do not verify this condition, or that are extremal points of the interval $[\alpha(v), \omega(v)]$ (indeed, all other values cannot be minima of $L(v, r)$). Since all our bounds are of the form $L(v, r) = \frac{(n-1)S(v,r)}{(r-1)^2}$, where $S(v, r)$ is a lower bound on $\sum_{w \in R(v)} d(v, w)$, we state our condition in terms of the function $S(v, r)$. For instance, in the case of the `updateBoundsBFSCut` function, $S_d^{\mathrm{CUT}}(v, r) = \sum_{w \in B_d(v)} d(v, w) - \tilde{n}_{d+1}(v) + (d+2)(r - b_d(v))$, as in Lemma 9.5.1.

Lemma 9.7.1. *Let v be a vertex, and let $S(v, r)$ be a positive function such that $S(v, r(v))) \leq \sum_{w \in R(v)} d(v, w)$ (where $r(v)$ is the number of vertices reachable from v). Assume that $S(v, r+1) - S(v, r) \leq S(v, r) - S(v, r-1)$. Then, if $L(v, r) := \frac{(n-1)S(v,r)}{(r-1)^2}$ is the corresponding bound on the farness of v,* $\min(L(v, r+1), L(v, r-1)) \leq L(v, r)$.

Proof. Let us define $d = S(v, r+1) - S(v, r)$. Then, $L(v, r+1) \leq L(v, r)$ if and only if $\frac{(n-1)S(v,r+1)}{r^2} \leq \frac{(n-1)S(v,r)}{(r-1)^2}$ if and only if $\frac{S(v,r)+d}{r^2} \leq \frac{S(v,r)}{(r-1)^2}$ if and only if $(r-1)^2(S(v, r) + d) \leq r^2 S(v, r)$ if and only if $S(v, r)(r^2 - (r-1)^2) \geq (r-1)^2 d$ if and only if $S(v, r)(2r - 1) \geq (r-1)^2 d$.

Similarly, if $d' = S(v, r) - S(v, r-1)$, $L(v, r-1) \leq L(v, r)$ if and only if $\frac{(n-1)S(v,r-1)}{(r-2)^2} \leq \frac{(n-1)S(v,r)}{(r-1)^2}$ if and only if $\frac{S(v,r)-d'}{(r-2)^2} \leq \frac{S(v,r)}{(r-1)^2}$ if and only if $(r-1)^2(S(v, r) - d') \leq (r-2)^2 S(v, r)$ if and only if $S(v, r)((r-1)^2 - (r-2)^2) \leq (r-1)^2 d'$ if and only if $S(v, r)(2r - 3) \leq (r-1)^2 d'$ if and only if $S(v, r)(2r - 1) \leq (r-1)^2 d' + 2S(v, r)$.

We conclude that, assuming $d \leq d'$, $(r-1)^2 d \leq (r-1)^2 d' \leq (r-1)^2 d + 2S(v, r)$, and one of the two previous conditions is always satisfied. □

9.7.1 *The Neighborhood-Based Lower Bound*

In the neighborhood-based lower bound, we computed upper bounds $\tilde{n}_k(v)$ on $n_k(v)$, and we defined the lower bound $S^{\mathrm{NB}}(v, r(v)) \leq \sum_{w \in R(v)} d(v, w)$, by

$$S^{\mathrm{NB}}(v, r(v)) := \sum_{k=1}^{\mathsf{diam}(G)} k \cdot \min\left\{\tilde{n}_k(v),\ \max\left\{r(v) - \sum_{i=0}^{k-1} \tilde{n}_i(v),\ 0\right\}\right\}.$$

The corresponding bound on $f(v)$ is $L^{\mathrm{NB}}(v, r(v)) := \frac{(n-1)S^{\mathrm{NB}}(v,r(v))}{(r(v)-1)^2}$: let us apply Lemma 9.7.1 with $S(v, r) = S^{\mathrm{NB}}(v, r)$ and $L(v, r) = L^{\mathrm{NB}}(v, r)$. We obtain that the local minima of $L^{\mathrm{NB}}(v, r(v))$ are obtained on values r such that $S^{\mathrm{NB}}(v, r+1) - S^{\mathrm{NB}}(v, r) > S^{\mathrm{NB}}(v, r) - S^{\mathrm{NB}}(v, r-1)$, that is, when $r = \sum_{i=0}^{l} \tilde{n}_i(v)$ for some l. Hence, our final bound $L^{\mathrm{NB}}(v)$ becomes:

$$\min\left(L^{\mathrm{NB}}(v, \alpha(v)), L^{\mathrm{NB}}(v, \omega(v)), \min\left\{L^{\mathrm{NB}}(v, r) : \alpha(v) < r < \omega(v), r = \sum_{i=0}^{l} \tilde{n}_i(v)\right\}\right). \quad (25)$$

This bound can be computed with no overhead, by modifying Lines 25 - 31 in Algorithm 17. Indeed, when $r(v)$ is known, we have two cases: either `nVisited[s]` $< r(v)$, and we continue, or `nVisited[s]` $\geq r(v)$, and $S^{\mathrm{NB}}(v, r(v))$ is computed. In the disconnected case, we need to distinguish three cases:

- if `nVisited[v]` $< \alpha(v)$, we simply continue the computation;
- if $\alpha(v) \leq$ `nVisited[v]` $< \omega(v)$, we compute $L^{\mathrm{NB}}(v, \texttt{nVisited[v]})$, and we update the minimum in Eq. 25 (if this is the first occurrence of this situation, we also have to compute $L^{\mathrm{NB}}(v, \alpha(v))$);
- if `nVisited[v]` $\geq \omega(v)$, we compute $L^{\mathrm{NB}}(v, \omega(v))$, and we update the minimum in Eq. 25.

Since this procedure needs time $O(1)$, it has no impact on the running time of the computation of the neighborhood-based lower bound.

9.7.2 *The* `updateBoundsBFSCut` *Function*

Let us apply Lemma 9.7.1 to the bound used in the `updateBoundsBFSCut` function. In this case, by Lemma 9.5.1, $S_d^{\mathrm{CUT}}(v, r) = \sum_{w \in B_d(v)} d(v, w) - \tilde{n}_{d+1}(v) + (d+2)(r - b_d(v))$, and $S_d^{\mathrm{CUT}}(v, r+1) - S_d^{\mathrm{CUT}}(v, r) = d + 2$, which does not depend on r. Hence, the condition in Lemma 9.7.1 is always verified, and the only values we have to analyze are $\alpha(v)$ and $\omega(v)$. Hence, the lower bound becomes $f(v) \geq L_d^{\mathrm{CUT}}(v, r(v)) \geq \min_{\alpha(v) \leq r \leq \omega(v)} L_d^{\mathrm{CUT}}(v, r) = \min(L_d^{\mathrm{CUT}}(v, \alpha(v)), L_d^{\mathrm{CUT}}(v, \omega(v)))$ (which does not depend on $r(v)$).

This means that, in order to adapt the `updateBoundsBFSCut` function (Algorithm 18), it is enough to replace Lines 10, 24 in order to compute both $L_d^{\mathrm{CUT}}(v, \alpha(v))$ and $L_d^{\mathrm{CUT}}(v, \omega(v))$), and to replace Lines 11, 25 in order to stop if $\min(L_d^{\mathrm{CUT}}(v, \alpha(v)), L_d^{\mathrm{CUT}}(v, \omega(v))) \geq x$.

9.7.3 *The* `updateBoundsLB` *Function*

In this case, we do not apply Lemma 9.7.1 to obtain simpler bounds. Indeed, the function `updateBoundsLB` improves the bounds of vertices that are quite close to the source of the BFS, and hence are likely to be in the same component as this vertex. Consequently, if we perform a BFS from a vertex s, we can simply compute $L_s^{\mathrm{LB}}(v, r(v))$ for all vertices in the same strongly connected component as s, and for these vertices we know the value $r(v) = r(s)$. The computation of better bounds for other vertices is left as an open problem.

9.7.4 *Computing* $\alpha(v)$ *and* $\omega(v)$

It now remains to compute $\alpha(v)$ and $\omega(v)$. This can be done during the preprocessing phase of our algorithm, in linear time. To this purpose, let us precisely define the node-weighted directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of strongly connected components (in short, SCCs) corresponding to a directed graph $G = (V, E)$. In this graph, $\mathcal{V}$ is the set of SCCs of G, and, for any two SCCs $C, D \in \mathcal{V}$, $(C, D) \in \mathcal{E}$ if and only if there is an arc in E from a node in C to the a node in D. For each SCC $C \in \mathcal{V}$, the weight $w(C)$ of C is equal to

Algorithm 20: Estimating the number of reachable vertices in directed disconnected graphs.

```
Input  : A graph $G = (V, E)$
Output : Lower and upper bounds $\alpha(v), \omega(v)$ on the number of vertices reachable from $v$
1  $(\mathcal{V}, \mathcal{E}, w) \leftarrow$ computeSCCGraph($G$);
2  $\tilde{C} \leftarrow$ the biggest SCC;
3  $\alpha_{SCC}(\tilde{C}), \omega_{SCC}(\tilde{C}) \leftarrow$ the number of vertices reachable from $\tilde{C}$;
4  for $X \in \mathcal{V}$ in reverse topological order do
5      if $X == \tilde{C}$ then continue;
6      $\alpha_{SCC}(X), \omega_{SCC}(X), \omega'_{SCC}(X) \leftarrow 0$ for $Y$ neighbor of $X$ in $\mathcal{G}$ do
7          $\alpha_{SCC}(X) \leftarrow \max(\alpha_{SCC}(X), \alpha_{SCC}(Y))$;
8          $\omega_{SCC}(X) \leftarrow \omega_{SCC}(X) + \omega_{SCC}(Y)$;
9          if $W$ not reachable from $\tilde{C}$ then $\omega'_{SCC}(X) \leftarrow \omega'_{SCC}(X) + \omega_{SCC}(Y)$;
10     end
11     if $X$ reaches $\tilde{C}$ then $\omega_{SCC}(X) \leftarrow \omega'_{SCC}(X) + \omega_{SCC}(\tilde{C})$;
12     $\alpha_{SCC}(X) \leftarrow \alpha_{SCC}(X) + w(X)$;
13     $\omega_{SCC}(X) \leftarrow \omega_{SCC}(X) + w(X)$;
14 end
15 for $v \in V$ do
16     $\alpha(v) = \alpha_{SCC}$(the component of $v$);
17     $\omega(v) = \omega_{SCC}$(the component of $v$);
18 end
19 return $\alpha, \omega$
```

$|C|$, that is, the number of nodes in the SCC C. Note that the graph $\mathcal{G}$ is computable in linear time.

For each node $v \in C$, $r(v) = \sum_{D \in R(C)} w(D)$, where $R(C)$ denotes the set of SCCs that are reachable from C in $\mathcal{G}$. This means that we simply need to compute a lower (respectively, upper) bound $\alpha_{SCC}(C)$ (respectively, $\omega_{SCC}(C)$) on $\sum_{D \in \mathcal{R}(C)} w(D)$, for every SCC C. To this aim, we first compute a topological sort $\{C_1, \dots, C_l\}$ of $\mathcal{V}$ (that is, if $(C_i, C_j) \in \mathcal{E}$, then $i < j$). Successively, we use a dynamic programming approach, and, by starting from C_l, we process the SCCs in reverse topological order, and we set:

$$\alpha_{SCC}(C) = w(C) + \max_{(C,D) \in \mathcal{E}} \alpha_{SCC}(D) \qquad \omega_{SCC}(C) = w(C) + \sum_{(C,D) \in \mathcal{E}} \omega_{SCC}(D).$$

Note that processing the SCCs in reverse topological ordering ensures that the values $\alpha(D)$ and $\omega(D)$ on the right hand side of these equalities are available when we process the SCC C. Clearly, the complexity of computing $\alpha(C)$ and $\omega(C)$, for each SCC C, is linear in the size of $\mathcal{G}$, which in turn is smaller than G.

Observe that the bounds obtained through this simple approach can be improved by using some “tricks”. First of all, when the biggest SCC $\tilde{C}$ is processed, we do not use the dynamic programming approach and we exactly compute $\sum_{D \in \mathcal{R}(\tilde{C})} w(D)$ by performing a BFS starting from any node in $\tilde{C}$. This way, not only $\alpha(\tilde{C})$ and $\omega(\tilde{C})$ are exact, but also $\alpha_{SCC}(C)$ and $\omega_{SCC}(C)$ are improved for each SCC C from which it is possible to reach $\tilde{C}$. Finally, in order to compute the upper bounds for the SCCs that are able to reach $\tilde{C}$, we can run the dynamic programming algorithm on the graph obtained from $\mathcal{G}$ by removing all components reachable from $\tilde{C}$, and we can then add $\sum_{D \in \mathcal{R}(\tilde{C})} w(D)$.

The pseudocode is available in Algorithm 20.

9.8 EXPERIMENTAL RESULTS

In this section, we test the four variants of our algorithm on several real-world networks, in order to evaluate their performances. All the networks used in our experiments come from the datasets SNAP (http://snap.stanford.edu/), LASAGNE (http://lasagne-unifi.sourceforge.net/), KONECT (http://konect.uni-koblenz.de/networks/), and IMDB (http://www.imdb.com). The platform for our tests is a shared-memory server with 256 GB RAM and 2x8 Intel(R) Xeon(R) E5-2680 cores (32 threads due to hyperthreading) at 2.7 GHz. The algorithms are implemented in C++, building on the open-source *NetworKit* framework [119].

9.8.1 *Comparison with the State of the Art*

In order to compare the performance of our algorithm with state-of-the-art approaches, we select 19 directed complex networks, 17 undirected complex networks, 6 directed street networks, and 6 undirected street networks (the undirected versions of the previous ones). The number of nodes of most of these networks ranges between 5 000 and 100 000. We test four different variants of our algorithm, that provide different implementations of the functions `computeBounds` and `updateBounds` (for more information, we refer to Section 9.3):

- DegCut uses the conservative strategies `computeBoundsDeg` and `updateBoundsBFSCut`;
- DegBound uses the conservative strategy `computeBoundsDeg` and the aggressive strategy `updateBoundsLB`;
- NBCut uses the aggressive strategy `computeBoundsNB` and the conservative strategy `updateBoundsBFSCut`;
- NBBound uses the aggressive strategies `computeBoundsNB` and `updateBoundsLB`.

We compare these algorithms with our implementations of the best existing algorithms for top-k closeness centrality (note that the source code of our competitors is not available). The first one [97] is based on a pruning technique and on Δ-BFS, a method to reuse information collected during a BFS from a node to speed up the computation for one of its in-neighbors; we denote this algorithm as Olh. The second one, Ocl, provides top-k closeness centralities with high probability [96]. It performs some BFSes from a random sample of nodes to estimate the closeness centrality of all the other nodes, then it computes the exact centrality of all the nodes whose estimate is big enough. Note that this algorithm requires the input graph to be (strongly) connected: for this reason, differently from the other algorithms, we have run this algorithm on the largest (strongly) connected component of the input graph. Furthermore, this algorithm offers different tradeoffs between the time needed by the sampling phase and the second phase: in our tests, we try all possible tradeoffs, and we choose the best alternative in each input graph (hence, our results are upper bounds on the real performance of the Ocl algorithm).

In order to perform a fair comparison, we consider the *edge traversal ratio*, which is defined as $\frac{mn}{m_{\text{vis}}}$ in directed graphs, $\frac{2mn}{m_{\text{vis}}}$ in undirected graphs, where m_{vis} is the number of arcs visited during the algorithm, and mn (resp., $2mn$) is an estimate of the number of arcs visited by the textbook algorithm in directed (resp., undirected) graphs (this estimate is correct whenever the graph is connected). Note that the edge traversal ratio does not

		DIRECTED		UNDIRECTED		BOTH	
k	ALGORITHM	GMEAN	GSTDDEV	GMEAN	GSTDDEV	GMEAN	GSTDDEV
1	OLH	21.24	5.68	11.11	2.91	15.64	4.46
	OCL	1.71	1.54	2.71	1.50	2.12	1.61
	DEGCUT	104.20	6.36	171.77	6.17	131.94	6.38
	DEGBOUND	3.61	3.50	5.83	8.09	4.53	5.57
	NBCUT	**123.46**	7.94	**257.81**	8.54	**174.79**	8.49
	NBBOUND	17.95	10.73	56.16	9.39	30.76	10.81
10	OLH	21.06	5.65	11.11	2.90	15.57	4.44
	OCL	1.31	1.31	1.47	1.11	1.38	1.24
	DEGCUT	56.47	5.10	60.25	4.88	58.22	5.00
	DEGBOUND	2.87	3.45	2.04	1.45	2.44	2.59
	NBCUT	**58.81**	5.65	**62.93**	5.01	**60.72**	5.34
	NBBOUND	9.28	6.29	10.95	3.76	10.03	5.05
100	OLH	20.94	5.63	11.11	2.90	15.52	4.43
	OCL	1.30	1.31	1.46	1.11	1.37	1.24
	DEGCUT	22.88	4.70	15.13	3.74	18.82	4.30
	DEGBOUND	2.56	3.44	1.67	1.36	2.09	2.57
	NBCUT	**23.93**	4.83	**15.98**	3.89	**19.78**	4.44
	NBBOUND	4.87	4.01	4.18	2.46	4.53	3.28

Table 21: Complex networks: geometric mean and standard deviation of the edge traversal ratios of the algorithm in [97] (OLH), the algorithm in [96] (OCL), and the four variants of the new algorithm (DEGCUT, DEGBOUND, NBCUT, NBBOUND).

depend on the implementation, nor on the machine used for the algorithm, and it does not consider parts of the code that need subquadratic time in the worst case. These parts are negligible in our algorithm, because their worst case running time is $O(n \log n)$ or $O(m\,\mathsf{diam})$ where diam is the diameter of the graph, but they can be significant when considering the competitors. For instance, in the particular case of OLH, we have just counted the arcs visited in BFS and Δ-BFS, ignoring all the operations done in the pruning phases (see [97]).

We consider the geometric mean of the edge traversal ratios over all graphs in the dataset. In our opinion, this quantity is more informative than the arithmetic mean, which is highly influenced by the maximum value: for instance, in a dataset of 20 networks, if all edge traversal ratios are 1 apart from one, which is 10 000, the arithmetic mean is more than 500, which makes little sense, while the geometric mean is about 1.58. Our choice is further confirmed by the geometric standard deviation, which is always quite small.

The results are summarised in Table 21 for complex networks and Table 22 for street networks. For the edge traversal ratios of each graph, we refer to Table 27, Table 28 and Table 29.

On complex networks, the best algorithm is NBCUT: when $k = 1$, the edge traversal ratios are always bigger than 100 and up to 258. When $k = 10$ they are close to 60, and when $k = 100$ they are close to 20. Another good option is DEGCUT, which achieves results slightly worse than NBCUT, but it has almost no overhead at the beginning (while NBCUT needs a preprocessing phase with cost $O(m\,\mathsf{diam})$). Furthermore, DEGCUT is very easy to implement, becoming a very good candidate for state-of-the-art graph libraries. The edge traversal ratios of the competitors are smaller: OLH has edge traversal ratios between 10 and 20, and OCL provides almost no improvement with respect to the textbook algorithm.

		Directed		Undirected		Both	
k	Algorithm	GMean	GStdDev	GMean	GStdDev	GMean	GStdDev
1	Olh	4.11	1.83	4.36	2.18	4.23	2.01
	Ocl	3.39	1.28	3.23	1.28	3.31	1.28
	DegCut	4.14	2.07	4.06	2.06	4.10	2.07
	DegBound	187.10	1.65	272.22	1.67	225.69	1.72
	NBCut	4.12	2.07	4.00	2.07	4.06	2.07
	NBBound	**250.66**	1.71	**382.47**	1.63	**309.63**	1.74
10	Olh	4.04	1.83	4.28	2.18	4.16	2.01
	Ocl	2.93	1.24	2.81	1.24	2.87	1.24
	DegCut	4.09	2.07	4.01	2.06	4.05	2.07
	DegBound	172.06	1.65	245.96	1.68	205.72	1.72
	NBCut	4.08	2.07	3.96	2.07	4.02	2.07
	NBBound	**225.26**	1.71	**336.47**	1.68	**275.31**	1.76
100	Olh	4.03	1.82	4.27	2.18	4.15	2.01
	Ocl	2.90	1.24	2.79	1.24	2.85	1.24
	DegCut	3.91	2.07	3.84	2.07	3.87	2.07
	DegBound	123.91	1.56	164.65	1.67	142.84	1.65
	NBCut	3.92	2.08	3.80	2.09	3.86	2.08
	NBBound	**149.02**	1.59	**201.42**	1.69	**173.25**	1.67

Table 22: Street networks: geometric mean and standard deviation of the edge traversal ratios of the algorithm in [97] (Olh), the algorithm in [96] (Ocl), and the four variants of the new algorithm (DegCut, DegBound, NBCut, NBBound).

On street networks, the best option is NBBound: for $k = 1$, the average improvement is about 250 in the directed case and about 382 in the undirected case, and it always remains bigger than 150, even for $k = 100$. It is worth noting that also the performance of DegBound are quite good, being at least 70% of NBBound. Even in this case, the DegBound algorithm offers some advantages: it is very easy to be implemented, and there is no overhead in the first part of the computation. All the competitors perform relatively poorly on street networks, since their improvement is always smaller than 5.

Overall, we conclude that the preprocessing function `computeBoundsNB` always leads to better results (in terms of visited edges) than `computeBoundsDeg`, but the difference is quite small: hence, in some cases, `computeBoundsDeg` could be even preferred, because of its simplicity. Conversely, the performance of `updateBoundsBFSCut` is very different from the performance of `updateBoundsLB`: the former works much better on complex networks, while the latter works much better on street networks. Currently, these two approaches exclude each other: an open problem left by this work is the design of a "combination" of the two, that works both in complex networks and in street networks. Finally, the experiments show that the best variant of our algorithm outperforms all competitors in all frameworks considered: both in complex and in street networks, both in directed and undirected graphs.

HARMONIC CENTRALITY. As mentioned in the introduction, all our methods can be easily generalized to any centrality measure in the form $c(v) = \sum_{w \neq v} f(d(v, w))$, where f is a decreasing function such that $f(+\infty) = 0$. We also implemented a version of DegCut, DegBound, NBCut and NBBound for *harmonic centrality*, which is defined as $h(v) = \sum_{w \neq v} \frac{1}{d(v,w)}$. Also for harmonic centrality, we compute the edge traversal ratios on the textbook algorithm.

Input	k	DIRECTED GMEAN	DIRECTED GSTDDEV	UNDIRECTED GMEAN	UNDIRECTED GSTDDEV	BOTH GMEAN	BOTH GSTDDEV
Street	1	742.42	2.60	1681.93	2.88	1117.46	2.97
	10	724.72	2.67	1673.41	2.92	1101.25	3.03
	100	686.32	2.76	1566.72	3.04	1036.95	3.13
Complex	1	247.65	11.92	551.51	10.68	339.70	11.78
	10	117.45	9.72	115.30	4.87	116.59	7.62
	100	59.96	8.13	49.01	2.93	55.37	5.86

Table 23: Big networks: geometric mean and standard deviation of the edge traversal ratios of the best variant of the new algorithm (NBCut in complex networks, NBBound in street networks).

For the complex networks used in our experiments, finding the k nodes with highest harmonic centrality is *always faster* than finding the k nodes with highest closeness, for all four methods and k values in $\{1, 10, 100\}$. For example, for NBCut and $k = 1$, the geometric mean (over both directed and undirected networks) of the edge traversal ratios is 486.07, whereas for closeness it is 174.79 (as reported in Table 21).

For street networks, the version of harmonic centrality is faster than the version for closeness for DegCut and NBCut, but it is slower for DegBound and NBBound. In particular, the average (geometric mean) edge traversal ratio of NBBound for harmonic centrality is 103.58 for $k = 1$, 93.49 for $k = 10$ and 62.22 for $k = 100$, which is about a factor 3 smaller than the edge traversal ratio of NBBound for closeness (see Table 22). Nevertheless, this is significantly faster than the textbook algorithm.

9.8.2 *Real-World Large Networks*

In this section, we run our algorithm on bigger inputs, by considering a dataset containing 23 directed networks, 15 undirected networks, and 5 street networks, with up to $\approx$ 4M nodes and $\approx$ 117M edges. On this dataset, we run the fastest variant of our algorithm (DegBound in complex networks, NBBound in street networks), using 64 threads (however, the server used has only 16 cores and runs 32 threads with hyperthreading; we account for memory latency in graph computations by oversubscribing slightly).

Once again, we consider the *edge traversal ratio*, which is defined as $\frac{mn}{m_{\text{vis}}}$ in directed graphs, $\frac{2mn}{m_{\text{vis}}}$ in undirected graphs. It is worth observing that we are able to compute for the first time the k most central nodes of networks with millions of nodes and hundreds of millions of arcs, with $k = 1$, $k = 10$, and $k = 100$. The detailed results are shown in Table 30, where for each network we report the running time and the edge traversal ratio. A summary of these results is available in Table 23, which contains the geometric means of the edge traversal ratios, with the corresponding standard deviations.

For $k = 1$, the geometric mean of the edge traversal ratios is always above 200 in complex networks, and above 700 in street networks. In undirected graphs, the edge traversal ratios are even bigger: close to 500 in complex networks and close to 1 600 in street networks. For bigger values of k, the performance does not decrease significantly: on complex networks, the edge traversal ratios are bigger than or very close to 50, even for $k = 100$. In street networks, the performance loss is even smaller, always below 10% for $k = 100$.

Regarding the robustness of the algorithm, we outline that the algorithm always achieves performance improvements bigger than $\sqrt{n}$ in street networks, and that in complex net-

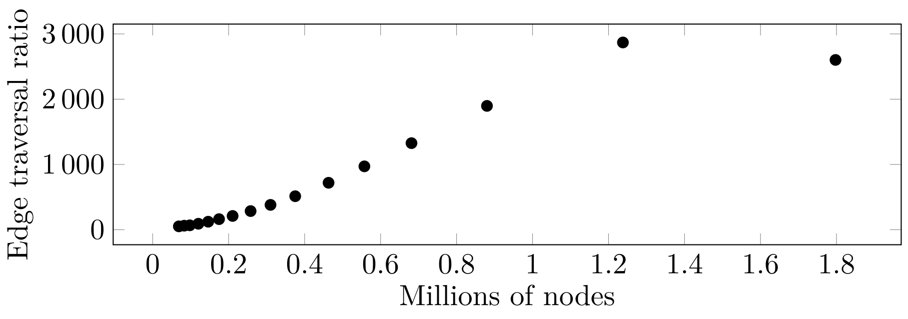


Figure 28: Growth of performance ratio with respect to the number of nodes ($k = 1$).

works, with $k = 1$, 64% of the networks have edge traversal ratios above 100, and 33% of the networks above 1 000. In some cases, the edge traversal ratio is even bigger: in the `com-Orkut` network, our algorithm for $k = 1$ is almost 35 000 times faster than the textbook algorithm.

In our experiments, we also report the running time of our algorithm. Even for $k = 100$, a few minutes are sufficient to conclude the computation on most networks, and, in all but two cases, the total time is smaller than 3 hours. For $k = 1$, the computation always terminates in at most 1 hour and a half, apart from two street networks where it needs less than 2 hours and a half. Overall, the total time needed to compute the most central vertex in all the networks is smaller than 1 day. In contrast to this, if we extrapolate the results in Tables 21 and 22, it seems plausible that the fastest competitor OLH would require a month or so.

9.9 IMDB CASE STUDY

In this section, we apply the new algorithm NBBound to analyze the IMDB graph, where nodes are actors, and two actors are connected if they played together in a movie (TV-series are ignored). The data collected comes from the website `http://www.imdb.com`. In line with `http://oracleofbacon.org`, we decide to exclude some genres from our database: awards-shows, documentaries, game-shows, news, realities and talk-shows. We analyse snapshots of the actor graph, taken every 5 years from 1940 to 2010, and 2014. The results are reported in Table 24 and Table 25.

RUNNING TIMES. Thanks to this experiment, we can evaluate the performance of our algorithm on increasing snapshots of the same graph. This way, we can have an informal idea on the asymptotic behavior of its complexity. In Figure 28, we have plotted the edge traversal ratio with respect to the number of nodes: if the edge traversal ratio is I, the running time is $O(\frac{mn}{I})$. Hence, assuming that $I = cn$ for some constant c (which is approximately verified in the actor graph, as shown by Figure 28), the running time is linear in the input size. The total time needed to perform the computation on all snapshots is little more than 30 minutes for $k = 1$, and little more than 45 minutes for $k = 10$.

RESULTS. In 2014, the most central actor is Michael Madsen, whose career spans 25 years and more than 170 films. Among his most famous appearances, he played as *Jimmy Lennox* in *Thelma & Louise* (Ridley Scott, 1991), as *Glen Greenwood* in *Free Willy* (Simon Wincer, 1993), as *Bob* in *Sin City* (Frank Miller, Robert Rodriguez, Quentin Tarantino),

	1940	**1945**	**1950**	**1955**
1	Semels, Harry (I)	Corrado, Gino	Flowers, Bess	Flowers, Bess
2	Corrado, Gino	Steers, Larry	Steers, Larry	Harris, Sam (II)
3	Steers, Larry	Flowers, Bess	Corrado, Gino	Steers, Larry
4	Bracey, Sidney	Semels, Harry (I)	Harris, Sam (II)	Corrado, Gino
5	Lucas, Wilfred	White, Leo (I)	Semels, Harry (I)	Miller, Harold (I)
6	White, Leo (I)	Mortimer, Edmund	Davis, George (I)	Farnum, Franklyn
7	Martell, Alphonse	Boteler, Wade	Magrill, George	Magrill, George
8	Conti, Albert (I)	Phelps, Lee (I)	Phelps, Lee (I)	Conaty, James
9	Flowers, Bess	Ring, Cyril	Ring, Cyril	Davis, George (I)
10	Sedan, Rolfe	Bracey, Sidney	Moorhouse, Bert	Cording, Harry

	1960	**1965**	**1970**	**1975**
1	Flowers, Bess	Flowers, Bess	Flowers, Bess	Flowers, Bess
2	Harris, Sam (II)	Harris, Sam (II)	Harris, Sam (II)	Harris, Sam (II)
3	Farnum, Franklyn	Farnum, Franklyn	Tamiroff, Akim	Tamiroff, Akim
4	Miller, Harold (I)	Miller, Harold (I)	Farnum, Franklyn	Welles, Orson
5	Chefe, Jack	Holmes, Stuart	Miller, Harold (I)	Sayre, Jeffrey
6	Holmes, Stuart	Sayre, Jeffrey	Sayre, Jeffrey	Miller, Harold (I)
7	Steers, Larry	Chefe, Jack	Quinn, Anthony (I)	Farnum, Franklyn
8	Parìs, Manuel	Parìs, Manuel	O'Brien, William H.	Kemp, Kenner G.
9	O'Brien, William H.	O'Brien, William H.	Holmes, Stuart	Quinn, Anthony (I)
10	Sayre, Jeffrey	Stevens, Bert (I)	Stevens, Bert (I)	O'Brien, William H.

	1980	**1985**	**1990**	**1995**
1	Flowers, Bess	Welles, Orson	Welles, Orson	Lee, Christopher (I)
2	Harris, Sam (II)	Flowers, Bess	Carradine, John	Welles, Orson
3	Welles, Orson	Harris, Sam (II)	Flowers, Bess	Quinn, Anthony (I)
4	Sayre, Jeffrey	Quinn, Anthony (I)	Lee, Christopher (I)	Pleasence, Donald
5	Quinn, Anthony (I)	Sayre, Jeffrey	Harris, Sam (II)	Hitler, Adolf
6	Tamiroff, Akim	Carradine, John	Quinn, Anthony (I)	Carradine, John
7	Miller, Harold (I)	Kemp, Kenner G.	Pleasence, Donald	Flowers, Bess
8	Kemp, Kenner G.	Miller, Harold (I)	Sayre, Jeffrey	Mitchum, Robert
9	Farnum, Franklyn	Niven, David (I)	Tovey, Arthur	Harris, Sam (II)
10	Niven, David (I)	Tamiroff, Akim	Hitler, Adolf	Sayre, Jeffrey

	2000	**2005**	**2010**	**2014**
1	Lee, Christopher (I)	Hitler, Adolf	Hitler, Adolf	Madsen, Michael (I)
2	Hitler, Adolf	Lee, Christopher (I)	Lee, Christopher (I)	Trejo, Danny
3	Pleasence, Donald	Steiger, Rod	Hopper, Dennis	Hitler, Adolf
4	Welles, Orson	Sutherland, Donald (I)	Keitel, Harvey (I)	Roberts, Eric (I)
5	Quinn, Anthony (I)	Pleasence, Donald	Carradine, David	De Niro, Robert
6	Steiger, Rod	Hopper, Dennis	Sutherland, Donald (I)	Dafoe, Willem
7	Carradine, John	Keitel, Harvey (I)	Dafoe, Willem	Jackson, Samuel L.
8	Sutherland, Donald (I)	von Sydow, Max (I)	Caine, Michael (I)	Keitel, Harvey (I)
9	Mitchum, Robert	Caine, Michael (I)	Sheen, Martin	Carradine, David
10	Connery, Sean	Sheen, Martin	Kier, Udo	Lee, Christopher (I)

Table 24: Detailed ranking of the IMDB actor graph.

and as *Deadly Viper Budd* in *Kill Bill* (Quentin Tarantino, 2003-2004). The second is Danny Trejo, whose most famous movies are *Heat* (Michael Mann, 1995), where he played as *Trejo*, *Machete* (Ethan Maniquis, Robert Rodriguez, 2010) and *Machete Kills* (Robert Rodriguez, 2013), where he played as *Machete*. The third "actor" is not really an actor: he

YEAR	**1940**	**1945**	**1950**	**1955**
NODES	69 011	83 068	97 824	120 430
EDGES	3 417 144	5 160 584	6 793 184	8 674 159
IMPR ($k = 1$)	51.74	61.46	67.50	91.46
IMPR ($k = 10$)	32.95	40.73	44.72	61.52
YEAR	**1960**	**1965**	**1970**	**1975**
NODES	146 253	174 826	210 527	257 896
EDGES	11 197 509	12 649 114	14 209 908	16 080 065
IMPR ($k = 1$)	122.63	162.06	211.05	285.57
IMPR ($k = 10$)	80.50	111.51	159.32	221.07
YEAR	**1980**	**1985**	**1990**	**1995**
NODES	310 278	375 322	463 078	557 373
EDGES	18 252 462	20 970 510	24 573 288	28 542 684
IMPR ($k = 1$)	380.52	513.40	719.21	971.11
IMPR ($k = 10$)	296.24	416.27	546.77	694.72
YEAR	**2000**	**2005**	**2010**	**2014**
NODES	681 358	880 032	1 237 879	1 797 446
EDGES	33 564 142	41 079 259	53 625 608	72 880 156
IMPR ($k = 1$)	1326.53	1897.31	2869.14	2601.52
IMPR ($k = 10$)	838.53	991.89	976.63	1390.32

Table 25: Detailed edge traversal ratios on the IMDB actor graph.

is the German dictator Adolf Hitler: he was also the most central actor in 2005 and 2010, and he was in the top 10 since 1990. This a consequence of his appearances in several archive footages, that were re-used in several movies (he counts 775 credits, even if most of them are in documentaries or TV shows, which were eliminated). Among the movies where Adolf Hitler is credited, we find *Zelig* (Woody Allen, 1983), and *The Imitation Game* (Morten Tyldum, 2014). Among the other most central actors, we find many people who played a lot of movies, and most of them are quite important actors. However, this ranking does not discriminate between important roles and marginal roles: for instance, the actress Bess Flowers is not widely known, because she rarely played significant roles, but she appeared in over 700 movies in her 41 years career, and for this reason she was the most central for 30 years, between 1950 and 1980. Finally, it is worth noting that we never find Kevin Bacon in the top 10, even if he became famous for the “Six Degrees of Kevin Bacon” game (`http://oracleofbacon.org`). In this game the player receives an actor x and has to find a path of length at most 6 from x to Kevin Bacon in the actor graph. Kevin Bacon was chosen as the goal because he played in several movies, and he was thought to be one of the most central actors: this work shows that, actually, he is quite far from the top. Indeed, his closeness centrality is 0.336, while the most central actor has centrality 0.354, the 10th actor has centrality 0.350, and the 100th actor has centrality 0.341.

9.10 WIKIPEDIA CASE STUDY

In this section, we apply the new algorithm NBBOUND to analyze the Wikipedia graph, where nodes are pages, and there is a directed edge from page p to page q if p contains a link to q. The data collected comes from DBPedia 3.7 (`http://wiki.dbpedia.org/`). We analyse both the standard graph and the reverse graph, which contains an edge from page p to page q if q contains a link to p. The 10 most central pages are available in Table 26.

Position	Standard Graph	Reversed Graph
1ST	1989	United States
2ND	1967	World War II
3RD	1979	United Kingdom
4TH	1990	France
5TH	1970	Germany
6TH	1991	English language
7TH	1971	Association football
8TH	1976	China
9TH	1945	World War I
10TH	1965	Latin

Table 26: Top 10 pages in Wikipedia directed graph, both in the standard graph and in the reversed graph.

RUNNING TIMES. In the standard graph, the edge traversal ratio is 1 784 for $k = 1$, 1 509 for $k = 10$, and 870 for $k = 100$. The total running time is about 39 minutes for $k = 1$, 45 minutes for $k = 10$, and less than 1 hour and 20 minutes for $k = 100$. In the reversed graph, the algorithm performs even better: the edge traversal ratio is 87 918 for $k = 1$, 71 923 for $k = 10$, and 21 989 for $k = 100$. The total running times are less than 3 minutes for both $k = 1$ and $k = 10$, and less than 10 minutes for $k = 100$.

RESULTS. If we consider the standard graph, the results are quite unexpected: indeed, all the most central pages are years (the first is *1989*). However, this is less surprising if we consider that these pages contain a lot of links to events that happened in that year: for instance, the out-degree of *1989* is 1 560, and the links contain pages from very different topics: historical events, like the fall of Berlin wall, days of the year, different countries where particular events happened, and so on. A similar argument also works for other years: indeed, the second page is *1967* (with out-degree 1 438), and the third is *1979* (with out-degree 1 452). Furthermore, all the 10 most central pages have out-degree at least 1 269. Overall, we conclude that the central page in the Wikipedia standard graph are not the "intuitively important" pages, but they are the pages that have a biggest number of links to pages with different topics, and this maximum is achieved by pages related to years.

Conversely, if we consider the reversed graph, the most central page is *United States*, confirming a common conjecture. Indeed, in `http://wikirank.di.unimi.it/`, it is shown that the United States are the center according to harmonic centrality, and many other measures (however, in that work, the ranking is only approximated). A further evidence for this conjecture comes from the Six Degree of Wikipedia game (`http://thewikigame.com/6-degrees-of-wikipedia`), where a player is asked to go from one page to the other following the smallest possible number of link: a hard variant of this game forces the player not to pass the *United States* page, which is considered to be central. In this work, we show that this conjecture is true. The second page is *World War II*, and the third is *United Kingdom*, in line with the results obtained by other centrality measures (see `http://wikirank.di.unimi.it/`), especially for the first two pages.

Overall, we conclude that most of the central pages in the reversed graph are nations, and that the results capture our intuitive notion of “important” pages in Wikipedia. Thanks to this new algorithm, we can compute these pages in a bit more than 1 hour for the original graph, and less than 10 minutes for the reversed one.

BIBLIOGRAPHIC NOTES

The method `updateBoundsBFSCut`, as well as the computation of $\alpha(v)$ and $\omega(v)$ in Section 9.7.4 were presented as a technical report in [29], authored by Michele Borassi, Pierluigi Crescenzi and Andrea Marino.

The methods `updateBoundsLB` and `computeBoundsNB` were presented as a technical report in [15] (coauthored with Henning Meyerhenke) and subsequently published as “Computing top-k Closeness Centrality Faster in Unweighted Graphs” (coauthored with Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke) at the *Eighteenth Workshop on Algorithm Engineering and Experiments* (ALENEX 2016).

The unified framework presented in this chapter, as well as the extension to disconnected graphs, has been presented as a technical report in [21], and is currently in revision for international journal publication. In [21], we also prove that the problem of selecting the k most central vertices is not solvable in time $\mathcal{O}(|E|^{2-\epsilon})$ on directed graphs, for any constant $\epsilon > 0$, if we assume the Strong Exponential Time Hypothesis (SETH).

Network	Olh	Ocl	DegCut	DegBound	NBCut	NBBound
Directed Street						
faroe-islands	4.080	3.742	4.125	338.011	4.086	437.986
liechtenstein	2.318	2.075	2.114	130.575	2.115	137.087
isle-of-man	2.623	3.740	2.781	224.566	2.769	314.856
malta	5.332	4.351	4.147	73.836	4.141	110.665
belize	2.691	3.969	2.606	253.866	2.595	444.849
azores	13.559	3.038	19.183	230.939	19.164	266.488
Undirected Street						
faroe-islands	4.126	3.276	4.118	361.593	3.918	444.243
liechtenstein	2.318	2.027	2.107	171.252	2.122	183.240
isle-of-man	2.613	3.661	2.767	266.734	2.676	370.194
malta	4.770	4.164	3.977	122.729	3.958	232.622
belize	2.565	3.945	2.510	340.270	2.481	613.778
azores	22.406	2.824	18.654	589.985	18.810	727.528
Directed Complex						
polblogs	3.201	1.131	31.776	1.852	31.974	5.165
out.opsahl-openflights	13.739	1.431	73.190	2.660	73.888	18.255
ca-GrQc	9.863	1.792	36.673	3.630	38.544	6.307
out.subelj_jung-j_jung-j	125.219	1.203	79.559	1.024	79.882	1.897
p2p-Gnutella08	5.696	1.121	66.011	4.583	81.731	6.849
out.subelj_jdk_jdk	116.601	1.167	74.300	1.023	74.527	1.740
wiki-Vote	9.817	2.760	261.242	1.479	749.428	395.278
p2p-Gnutella09	5.534	1.135	41.214	4.650	43.236	6.101
ca-HepTh	7.772	2.121	40.068	3.349	42.988	5.217
freeassoc	33.616	1.099	12.638	2.237	12.700	2.199
ca-HepPh	7.682	2.836	10.497	3.331	10.516	4.387
out.lasagne-spanishbook	13.065	2.553	1871.296	7.598	6786.506	3160.750
out.cfinder-google	16.725	1.782	38.321	2.665	25.856	3.020
ca-CondMat	7.382	3.526	409.772	5.448	517.836	29.282
out.subelj_cora_cora	14.118	1.700	14.098	1.345	14.226	2.299
out.ego-twitter	2824.713	1.000	1870.601	28.995	3269.183	278.214
out.ego-gplus	722.024	1.020	3481.943	236.280	3381.029	875.111
as-caida20071105	20.974	3.211	2615.115	1.737	2837.853	802.273
cit-HepTh	4.294	3.045	16.259	1.514	16.398	3.290
Undirected Complex						
HC-BIOGRID	5.528	1.581	15.954	3.821	14.908	3.925
facebook_combined	10.456	3.726	56.284	18.786	56.517	98.512
Mus_musculus	18.246	1.743	70.301	3.253	104.008	7.935
Caenorhabditis_elegans	11.446	2.258	86.577	2.140	110.677	9.171
ca-GrQc	6.567	1.904	38.279	3.551	41.046	6.824
as20000102	19.185	2.402	1550.351	3.213	1925.916	498.000
advogato	8.520	2.018	315.024	18.181	323.163	142.654
p2p-Gnutella09	3.744	2.336	90.252	1.708	100.427	13.846
hprd_pp	6.543	2.397	392.853	2.091	407.261	63.953
ca-HepTh	7.655	2.075	42.267	3.308	46.326	5.593
Drosophila_melanogaster	5.573	2.346	69.457	1.822	75.456	6.904
oregon1_010526	20.474	3.723	1603.739	2.703	1798.822	399.071
oregon2_010526	17.330	4.748	1138.475	2.646	1227.105	520.955
Homo_sapiens	6.689	2.700	1475.113	1.898	1696.909	130.381
GoogleNw	15.591	8.389	107.902	15763.000	15763.000	15763.000
dip20090126_MAX	2.883	3.826	5.833	6.590	5.708	7.392
com-amazon.all.cmty	415.286	2.499	5471.982	3.297	8224.693	373.294

Table 27: Detailed comparison of the edge traversal ratios, with $k = 1$.

Network	Olh	Ocl	DegCut	DegBound	NBCut	NBBound
Directed Street						
faroe-islands	3.713	2.884	4.037	290.626	4.025	361.593
liechtenstein	2.318	2.002	2.104	111.959	2.106	116.713
isle-of-man	2.623	2.933	2.711	209.904	2.720	288.123
malta	5.325	3.861	4.094	70.037	4.086	101.546
belize	2.690	3.638	2.592	244.275	2.580	416.210
azores	13.436	2.644	19.043	222.073	19.045	254.206
Undirected Street						
faroe-islands	3.702	2.594	4.046	320.588	3.848	388.713
liechtenstein	2.316	1.965	2.097	142.047	2.114	150.608
isle-of-man	2.612	2.889	2.695	241.431	2.636	323.185
malta	4.768	3.615	3.920	115.574	3.910	208.192
belize	2.564	3.634	2.496	323.257	2.469	563.820
azores	22.392	2.559	18.541	539.032	18.712	653.372
Directed Complex						
polblogs	3.199	1.039	13.518	1.496	13.544	2.928
out.opsahl-openflights	13.739	1.130	32.297	1.984	32.405	6.867
ca-GrQc	9.863	1.356	25.238	3.096	25.786	4.565
out.subelj_jung-j_jung-j	124.575	1.000	79.284	1.024	79.657	1.884
p2p-Gnutella08	5.684	1.064	12.670	3.241	12.763	3.599
out.subelj_jdk_jdk	116.228	1.000	74.106	1.023	74.363	1.730
wiki-Vote	9.812	1.205	166.941	1.453	174.775	25.411
p2p-Gnutella09	5.532	1.084	16.293	3.624	16.265	4.213
ca-HepTh	7.772	1.586	31.314	3.013	32.604	4.356
freeassoc	33.414	1.034	10.612	2.210	10.704	2.178
ca-HepPh	7.682	2.077	10.322	3.042	10.340	4.010
out.lasagne-spanishbook	13.063	1.483	303.044	1.067	351.262	94.351
out.cfinder-google	16.725	1.413	36.364	2.665	24.765	3.017
ca-CondMat	7.382	2.318	91.209	3.507	93.548	7.027
out.subelj_cora_cora	13.699	1.287	12.763	1.334	12.909	2.072
out.ego-twitter	2689.884	1.000	1817.032	28.157	2872.213	218.411
out.ego-gplus	722.024	1.000	951.983	201.949	1085.361	482.204
as-caida20071105	20.974	1.615	997.996	1.371	1266.443	448.729
cit-HepTh	4.030	2.179	11.361	1.486	11.423	2.832
Undirected Complex						
HC-BIOGRID	5.528	1.240	10.714	3.102	10.036	3.058
facebook_combined	10.456	1.292	9.103	2.236	9.371	2.694
Mus_musculus	18.246	1.316	18.630	2.279	20.720	3.288
Caenorhabditis_elegans	11.445	1.405	58.729	1.904	68.905	7.605
ca-GrQc	6.567	1.340	26.050	3.052	26.769	5.011
as20000102	19.185	1.529	196.538	1.314	209.674	52.210
advogato	8.520	1.405	131.173	2.043	132.207	11.155
p2p-Gnutella09	3.744	1.632	79.093	1.623	87.357	12.941
hprd_pp	6.543	1.436	47.945	1.837	47.866	8.620
ca-HepTh	7.655	1.546	32.612	2.961	34.407	4.677
Drosophila_melanogaster	5.573	1.672	50.840	1.646	54.637	5.743
oregon1_010526	20.474	1.451	418.099	1.282	429.161	109.549
oregon2_010526	17.330	1.560	364.277	1.302	371.929	71.186
Homo_sapiens	6.689	1.599	81.496	1.620	82.250	15.228
GoogleNw	15.591	1.320	23.486	1.252	23.053	2.420
dip20090126_MAX	2.881	1.836	4.055	4.556	4.065	4.498
com-amazon.all.cmty	414.765	1.618	3407.016	3.279	3952.370	199.386

Table 28: Detailed comparison of the edge traversal ratios, with $k = 10$.

Network	Olh	Ocl	DegCut	DegBound	NBCut	NBBound
Directed Street						
faroe-islands	3.713	2.823	3.694	150.956	3.691	168.092
liechtenstein	2.318	1.998	2.078	84.184	2.086	86.028
isle-of-man	2.620	2.902	2.551	139.139	2.567	167.808
malta	5.282	3.850	3.933	56.921	3.942	76.372
belize	2.688	3.617	2.526	184.718	2.516	268.634
azores	13.334	2.628	18.380	194.724	18.605	220.013
Undirected Street						
faroe-islands	3.702	2.548	3.693	159.472	3.523	171.807
liechtenstein	2.311	1.959	2.072	96.782	2.095	99.768
isle-of-man	2.607	2.847	2.533	153.859	2.468	183.982
malta	4.758	3.605	3.745	89.929	3.730	137.538
belize	2.562	3.629	2.428	226.582	2.406	323.257
azores	22.345	2.548	18.092	411.760	18.384	476.253
Directed Complex						
polblogs	3.198	1.037	3.951	1.245	3.961	1.731
out.opsahl-openflights	13.739	1.124	5.524	1.456	5.553	1.740
ca-GrQc	9.863	1.339	11.147	2.353	10.407	2.926
out.subelj_jung-j_jung-j	123.393	1.000	78.473	1.021	78.798	1.787
p2p-Gnutella08	5.684	1.063	6.611	2.935	7.750	3.278
out.subelj_jdk_jdk	114.210	1.000	73.522	1.020	73.755	1.669
wiki-Vote	9.812	1.186	61.375	1.236	60.475	9.436
p2p-Gnutella09	5.531	1.083	6.370	3.109	7.650	3.508
ca-HepTh	7.772	1.570	16.135	2.477	16.747	3.135
freeassoc	33.266	1.032	6.314	2.154	6.428	2.138
ca-HepPh	7.682	2.032	9.605	2.549	9.619	3.340
out.lasagne-spanishbook	13.063	1.467	56.689	1.043	80.069	33.271
out.cfinder-google	16.725	1.392	13.521	2.655	12.298	2.722
ca-CondMat	7.382	2.288	16.884	2.602	16.950	2.824
out.subelj_cora_cora	13.231	1.280	11.171	1.315	11.350	1.870
out.ego-twitter	2621.659	1.000	1574.836	26.893	1908.731	110.236
out.ego-gplus	722.024	1.000	522.333	181.754	522.576	236.280
as-caida20071105	20.974	1.606	17.971	1.216	18.694	5.479
cit-HepTh	3.969	2.143	8.867	1.466	9.068	2.662
Undirected Complex						
HC-BIOGRID	5.528	1.236	4.452	2.154	4.345	1.999
facebook_combined	10.456	1.292	3.083	1.470	3.074	1.472
Mus_musculus	18.245	1.305	7.940	1.944	9.518	2.631
Caenorhabditis_elegans	11.445	1.391	11.643	1.463	12.296	3.766
ca-GrQc	6.567	1.331	11.311	2.346	10.389	3.105
as20000102	19.185	1.512	7.318	1.174	7.956	3.593
advogato	8.520	1.398	32.629	1.706	33.166	7.784
p2p-Gnutella09	3.744	1.625	11.378	1.374	11.867	3.695
hprd_pp	6.543	1.422	21.053	1.547	22.191	3.468
ca-HepTh	7.655	1.539	16.406	2.454	17.030	3.301
Drosophila_melanogaster	5.573	1.655	29.115	1.487	30.979	4.614
oregon1_010526	20.474	1.443	13.300	1.163	14.611	6.569
oregon2_010526	17.330	1.530	18.203	1.173	21.758	7.258
Homo_sapiens	6.689	1.577	19.350	1.445	20.182	3.080
GoogleNw	15.591	1.320	16.224	1.172	16.506	2.010
dip20090126_MAX	2.880	1.815	2.789	2.602	2.784	2.546
com-amazon.all.cmty	414.765	1.605	1368.675	3.236	1654.150	97.735

Table 29: Detailed comparison of the edge traversal ratios, with $k = 100$.

			$k = 1$		$k = 10$		$k = 100$	
Input	Nodes	Edges	Impr.	Time	Impr.	Time	Impr.	Time
Directed Street								
egypt	1054242	2123036	144.91	0:03:55	132.86	0:04:25	116.74	0:04:48
new_zealand	2759124	5562944	447.55	0:02:34	443.95	0:02:35	427.31	0:02:38
india	16230072	33355834	1370.32	0:43:42	1369.05	0:44:17	1326.31	0:45:05
california	16905319	34303746	1273.66	0:54:56	1258.12	0:56:00	1225.73	0:56:02
north_am	35236615	70979433	1992.68	2:25:58	1967.87	2:29:25	1877.78	2:37:14
Undirected Street								
egypt	1054242	1159808	344.86	0:01:54	340.30	0:01:54	291.71	0:02:11
new_zealand	2759124	2822257	811.75	0:02:47	786.52	0:03:02	734.20	0:03:02
india	16230072	17004400	2455.38	0:44:21	2484.70	0:44:38	2422.40	0:44:21
california	16905319	17600566	2648.08	0:39:15	2620.17	0:42:04	2504.86	0:44:19
north_am	35236615	36611653	7394.88	1:13:37	7530.80	1:15:01	7263.78	1:10:28
Directed Complex								
cit-HepTh	27769	352768	16.34	0:00:01	11.41	0:00:01	9.06	0:00:02
cit-HepPh	34546	421534	23.68	0:00:01	19.88	0:00:01	14.41	0:00:02
p2p-Gnut31	62586	147892	194.19	0:00:01	44.24	0:00:01	19.34	0:00:04
soc-Eps1	75879	508837	243.14	0:00:01	43.75	0:00:01	33.60	0:00:05
soc-Slash0811	77360	828161	1007.70	0:00:00	187.46	0:00:00	21.09	0:00:18
twitter_comb	81306	2684592	1024.32	0:00:01	692.96	0:00:01	145.68	0:00:05
Slash090221	82140	549202	177.82	0:00:02	162.30	0:00:02	108.53	0:00:03
gplus_comb	107614	24476570	1500.35	0:00:04	235.17	0:00:04	62.54	0:02:19
soc-sign-eps	131828	840799	225.91	0:00:03	161.58	0:00:03	39.26	0:00:16
email-EuAll	265009	418956	4724.80	0:00:00	3699.48	0:00:00	1320.22	0:00:01
web-Stanford	281903	2312497	13.59	0:04:00	8.70	0:04:00	7.47	0:07:15
web-NotreD	325729	1469679	1690.08	0:00:02	132.83	0:00:02	66.88	0:00:49
amazon0601	403394	3387388	10.81	0:14:54	8.87	0:14:54	6.84	0:22:04
web-BerkStan	685230	7600595	3.95	1:36:21	3.67	1:36:21	3.47	1:49:12
web-Google	875713	5105039	228.61	0:01:51	96.63	0:01:51	38.69	0:10:29
youtube-links	1138494	4942297	662.78	0:01:33	200.68	0:01:33	125.72	0:07:02
in-2004	1382870	16539643	43.68	0:41:45	29.89	0:41:45	16.68	1:48:42
trec-wt10g	1601787	8063026	33.86	0:36:01	20.39	0:36:01	16.73	1:10:54
soc-pokec	1632803	22301964	21956.64	0:00:17	2580.43	0:06:14	1106.90	0:12:35
zhishi-hudong	1984484	14682258	30.37	1:25:38	27.71	1:25:38	24.95	1:53:27
zhishi-baidu	2141300	17632190	44.05	1:17:52	38.61	1:17:52	23.17	3:08:05
wiki-Talk	2394385	5021410	34863.42	0:00:08	28905.76	0:00:08	9887.18	0:00:18
cit-Patents	3774768	16518947	9454.04	0:02:07	8756.77	0:02:07	8340.18	0:02:13
Undirected Complex								
ca-HepPh	12008	118489	10.37	0:00:00	10.20	0:00:00	9.57	0:00:01
CA-AstroPh	18772	198050	62.47	0:00:00	28.87	0:00:01	14.54	0:00:01
CA-CondMat	23133	93439	247.35	0:00:00	84.48	0:00:00	17.06	0:00:01
email-Enron	36692	183831	365.92	0:00:00	269.80	0:00:00	41.95	0:00:01
loc-brightkite	58228	214078	308.03	0:00:00	93.85	0:00:01	53.49	0:00:02
flickrEdges	105938	2316948	39.61	0:00:23	17.89	0:00:55	15.39	0:01:16
gowalla	196591	950327	2412.26	0:00:01	33.40	0:01:18	28.13	0:01:33
com-dblp	317080	1049866	500.83	0:00:10	300.61	0:00:17	99.64	0:00:52
com-amazon	334863	925872	37.76	0:02:21	31.33	0:02:43	18.68	0:04:34
com-lj.all	477998	530872	849.57	0:00:07	430.72	0:00:13	135.14	0:00:45
com-youtube	1134890	2987624	2025.32	0:00:32	167.45	0:06:44	110.39	0:09:16
soc-pokec	1632803	30622564	46725.71	0:00:18	8664.33	0:02:16	581.52	0:18:12
as-skitter	1696415	11095298	185.91	0:19:06	164.24	0:21:53	132.38	0:27:06
com-orkut	3072441	117185083	23736.30	0:02:32	255.17	2:54:58	69.23	15:02:06
youtube-u-g	3223585	9375374	11473.14	0:01:07	91.17	2:07:23	66.23	2:54:12

Table 30: Detailed comparison of the edge traversal ratios on big networks. For street networks, the results refer to NBBound, whereas for complex networks they refer to NBCut.

10

COMPUTING TOP-K CLOSENESS CENTRALITY IN FULLY-DYNAMIC GRAPHS

10.1 INTRODUCTION

As already mentioned in Chapter 1 and Chapter 3, many real-world networks evolve over time at a quick pace. For applications that require to keep track of the top-k nodes with highest closeness in a dynamic network, even rerunning the fast algorithms proposed in Chapter 9 after each edge modification might be too expensive. In this chapter we present dynamic algorithms for top-k closeness centrality that handle both edge insertions and edge deletions. Our new algorithms are based on the static algorithms presented in Chapter 9 and reuse information obtained during the initial run to skip the recomputation of closeness centralities for nodes that are unaffected by the edge modification. In contrast to other dynamic algorithms for closeness centrality, it is not required to compute the exact closeness centralities of all nodes in the initial graph, making it possible to target networks with tens of millions of edges. Moreover, we specifically design our algorithms to use only a linear amount of additional memory, since a quadratic memory footprint (typical of most existing dynamic algorithms for problems based on shortest paths, see also Chapter 8) would be impractical for large instances.

In experiments we obtain significant speedups compared to static recomputation. For example, for $k = 10$, our average speedup (geometric mean over the tested instances) is about 76 for insertions in undirected complex networks. For deletions in directed street networks, we reach an average speedup of 743. Also, our experiments show some interesting results: deletions are mostly faster than insertions and speedups increase with k for complex networks, whereas they decrease as k increases in street networks.

We recall from Section 9.2 that traditional closeness does not apply to disconnected graphs. For comparison with previous work, in Chapter 9, the proposed algorithms for top-k closeness have been generalized to disconnected graphs using Lin's index. Since this allows for a simpler description (while still applying to disconnected graphs), in this chapter we present all algorithms based on *harmonic closeness*. In addition to extending to disconnected graphs in a very natural way, harmonic closeness has been shown in [25] to satisfy all axioms presented in the same paper (i.e., size, density and score monotonicity). However, our algorithms can be easily adapted to Lin's index as well.

Before moving on to the description of the dynamic algorithms, we first introduce some notation in Section 10.2.1 and briefly describe the variant of the static top-k closeness algorithms based on harmonic centrality in Section 10.2.2 (see Chapter 9 for exhaustive description based on Lin's index).

10.2 PRELIMINARIES

10.2.1 *Notation and Problem Definition*

Let G be an unweighted graph (either directed or undirected) with n nodes and m edges. We use $d(u, v)$ to denote the shortest-path distance between two nodes u and v. We recall from Chapter 9 that the set of nodes at distance i from u is denoted by $N_i(u) := \{v : d(u, v) = i\}$ and its cardinality by $n_i(u)$. The reachable nodes $R(u) := \{v : d(u, v) < +\infty\}$ are the nodes with finite distance from u (we denote their cardinality by $r(u)$). We consider a variant of closeness called *harmonic closeness centrality* [25], defined as follows (see also Chapter 2):

$$c(u) := \sum_{v \in V,\ v \neq u} \frac{1}{d(u, v)} .$$

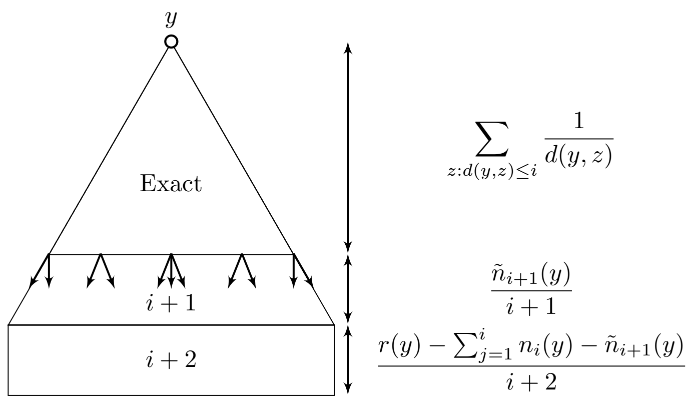
Throughout this chapter, we will use the term "closeness centrality" to indicate the harmonic variant. When talking about dynamic graphs, we refer to G as the graph before the edge update, and to G' as the modified graph. Similarly, d is the distance on G and d' the distance on G'.

10.2.2 *Static Top-k Closeness*

The top-k closeness algorithm proposed in Chapter 9 (Algorithm 15) tries to reduce the running time of the textbook algorithm (i.e., running a BFS from each node) by exploiting upper bounds on the closeness values. If k nodes are found, whose exact closeness is higher than the upper bounds of all other nodes, than the algorithm can stop. In particular, we recall that we proposed two approaches for each of `computeBounds` (Line 1 of Algorithm 15) and `updateBounds` (Line 8). For `computeBounds`, our experiments show that the `computeBoundsNB` approach in Section 9.4 (which computes bounds based on the number of walks starting from each node) always leads to better results than `computeBoundsDeg` (which simply orders nodes by degree). As for `updateBounds`, we proposed two strategies. The conservative strategy `updateBoundsBFSCut` does not improve existing bounds, but tries to interrupt the BFS traversal as soon as possible. On the contrary, the aggressive strategy `updateBoundsLB` runs a complete BFS, but tries to improve the bounds on the other nodes. Our experiments in Section 9.8 show that the strategy `updateBoundsBFSCut` leads to better results on complex networks and `updateBoundsLB` works better for street networks. As a result, NBCut (combining `computeBoundsNB` with `updateBoundsBFSCut`) is the best-performing approach for complex networks, whereas NBBound (combining `computeBoundsNB` with `updateBoundsLB`) works best for street networks.

Since both algorithms were proposed using Lin's index, in the following we briefly recall them using harmonic closeness.

NBCUT ALGORITHM FOR COMPLEX NETWORKS. Assume we are performing a BFS from a node y, and we have visited all nodes up to distance i. We know that all remaining nodes have at least distance $i + 1$, otherwise they would have been visited already. Also, we know that at most $\tilde{n}_{i+1}(y) := \sum_{w \in N_i(y)} \texttt{degree}(w)$ can be at distance $i + 1$, since all nodes at distance $i + 1$ must have at least one neighbor at distance i from y.



y
Exact
$i+1$
$i+2$
$\sum_{z:d(y,z)\leq i} \frac{1}{d(y,z)}$
$\frac{\tilde{n}_{i+1}(y)}{i+1}$
$\frac{r(y) - \sum_{j=1}^{i} n_i(y) - \tilde{n}_{i+1}(y)}{i+2}$

Figure 29: Upper bound on $c(y)$ computed by the NBCUT algorithm. For nodes up to distance $d_{cut}(y)$ we know the exact distance. Then, $\tilde{n}_{d_{cut}(y)+1}$ nodes are assumed to be at distance $d_{cut}(y)+1$ and the remaining at distance $d_{cut}(y)+2$.

Thus, all remaining nodes have to be at distance at least $i+2$. Based on this observation, we can define the following upper bound on $c(y)$:

$$\tilde{c}(y) := \sum_{z:d(y,z)\leq i} \frac{1}{d(y,z)} + \frac{\tilde{n}_{i+1}(y)}{i+1} + \frac{r(y) - \sum_{j=1}^{i} n_j(y) - \tilde{n}_{i+1}(y)}{i+2}, \tag{26}$$

where the first term in the sum accounts for the nodes for which we computed the exact distance, the second term assumes that exactly $\tilde{n}_{i+1}(y)$ nodes are at distance $i+1$, and the third term assumes that all remaining nodes are at distance $d+2$ (see Figure 29). Notice that the more nodes are visited during the BFS, the tighter the bound is (if all nodes are visited, $\tilde{c}(y) = c(y)$). Thus, the NBCUT algorithm works as follows: assume that we already computed the *exact* closeness for at least k nodes, and let x_k be the k-th highest closeness value found so far. While running a BFS from node y, `updateBoundsBFSCut` computes the bound of Eq. (26). If at some point $\tilde{c}(y) < x_k$, we can interrupt the search from y, since y cannot be one of the top-k nodes. We call the distance i at which we interrupt the BFS the *cutoff distance*, and we refer to it as $d_{cut}(y)$. If the exact closeness $c(y)$ is actually larger than x_k, `updateBoundsBFSCut` runs a complete BFS and stores y as one of the top-k nodes.

Notice that Eq. (26) requires the number $r(y)$ of nodes reachable from y. In undirected graphs, this is the size of the connected component of y, which can be easily computed in a preprocessing step. In directed graphs, we proposed in Chapter 9 (Section 9.7.4) an upper bound based on a topological sorting of the strongly-connected-components DAG (SCC DAG).

For simplicity, from now on we will write `BFSCut` to indicate `updateBoundsBFSCut`. Here we assume that, in addition to the computed bound $\tilde{c}(y)$, `BFSCut` returns also the cutoff distance $d_{cut}(y)$ and $\mathsf{isExact}(y)$ (which is true if and only if $\tilde{c}(y)$ is the exact closeness, i.e. if a whole BFS has been performed) for each node y.

NBBOUND ALGORITHM FOR STREET NETWORKS. NBBOUND initially computes an upper bound $\tilde{c}$ on the closeness of each node and enters the nodes into a priority queue Q, sorted by their value of $\tilde{c}$. Then, the node v with highest $\tilde{c}$ is extracted and its actual closeness $c(v)$ is computed using the `updateBoundsLB` function. In addition to computing the exact closeness of v, this function also modifies the upper bounds $\tilde{c}$ of the other nodes. In particular, let w and y be any two nodes visited during the BFS from v. Then, we can prove (see Chapter 9) that $d(w, y) \geq |d(v, w) - d(v, y)|$. By assuming that $d(w, y) = |d(v, w) - d(v, y)|$, $\forall w, y \in V$, we get an upper bound on the closeness of all nodes (see Chapter 9 for more details). If, for some node w, this upper bound is tighter than the current $\tilde{c}$, then $\tilde{c}$ and the priority of w in Q are updated. The algorithm terminates when k nodes are found such that their exact closeness is higher than the upper bound on the closeness of the other nodes.

10.3 DYNAMIC TOP-k CLOSENESS CENTRALITY

Let us assume an edge (u, v) has been inserted into or deleted from the graph. Our goal is to update the list of the top-k nodes and their closeness values faster than computing it from scratch with NBCut or NBBound. In the following, we first show how we update the number of reachable nodes (Section 10.3.1) and compute the set of affected nodes (Section 10.3.2), which is a preprocessing necessary for both edge insertions and deletions. Then, we consider edge insertions and deletions separately in Section 10.3.3 and Section 10.3.4.

10.3.1 *Updating the Number of Reachable Nodes*

The upper bound in Eq. (26) requires the number of reachable nodes $r(u)$ (or an upper bound on $r(u)$). For undirected graphs, this is simply the number of nodes in the connected component of u. Instead of recomputing connected components from scratch after each update, we use a simple dynamic algorithm similar to the one presented in [49]. Initially, we build a spanning forest of G. When an edge (u, v) is inserted, we check whether u and v belong to the same component and if not, we merge the components and (u, v) becomes part of the forest. If we delete an edge (u, v) that is part of the forest, we then run a pruned BFS from u and interrupt it as soon as we hit v (and, if we do not hit v, we split the components). If, in turn, we delete an edge (u, v) that is not part of the spanning forest, we know there has to be another path between u and v and we are done.

For directed graphs, we proposed in Chapter 9 (Section 9.7.4) an upper bound based on a topological sorting of the SCC DAG. Since preliminary experiments showed that recomputing this after each update was a bottleneck for the dynamic algorithm, we replace this bound with the number of nodes in the weakly-connected component. This does not affect the correctness of the algorithm and can be updated much more efficiently using basically the same algorithm we use for updating the connected components in undirected graphs (we simply need to ignore the direction of the edges).

10.3.2 *Finding Affected Nodes*

When an edge (u, v) is inserted or deleted from G, some nodes might increase or decrease their closeness centrality. We call such nodes *affected*. More precisely, the set A of affected nodes is defined as $A := \{y \in V : \exists w \in V \text{ such that } d'(y, w) \neq d(y, w)\}$. It is easy to see that, if $d'(y, w) \neq d(y, w)$, then either $d'(y, u) \neq d(y, u)$ or $d'(y, v) \neq d(y, v)$. Indeed, if the distances from y to both u and v stay the same, it means that the insertion or deletion of (u, v) does not move any node up or down the BFS DAG rooted in y. Thus, the set of affected nodes can be easily identified by running two BFSs from u (one on G and one on G') and two BFSs from v (one on G and one on G'). If the graph is directed, the BFSs have to be run on G and G' transposed, since we are interested in the nodes that change their distance *to* u or v. Once we know all the distances to u and v in G and G', we can simply compare them and mark all nodes y such that $d'(y, u) \neq d(y, u)$ or $d'(y, v) \neq d(y, v)$ as affected.

We know that the closeness of all unaffected nodes v does not change. Therefore, the previously computed upper bound $\tilde{c}$ and, if it was computed, the previous exact closeness value $c(v)$ are still valid.

Algorithm 21: Recomputation of the top-k nodes after an edge insertion (based on NBCut).

```
   Data: $G = (V, E), (u, v) \notin E$
   Result: Top-$k$ nodes with the highest closeness in $G' := (V, E \cup \{u, v\})$
1  Compute $r'(y) \quad \forall y \in V$;
2  Compute the set $A$ of affected nodes, $d(\cdot, u)$, $d'(\cdot, u)$;
3  $x_k \leftarrow$ TopK.getMin();
4  forall the $w \in A$ do
5      if $w \in$ TopK then
6          TopK.remove($w$);
7      end
8  end
9  foreach $y \in A$ do
10     if $d_{cut}(y) < d(y, u) \wedge$ not isExact($y$) then
           /* $y$ is a far-away node */
11         $\tilde{c}'(y) \leftarrow \tilde{c}(y) - \frac{r(y)}{d_{cut}(y)+2} + \frac{r'(y)}{d_{cut}(y)+2}$;
12     else if $d_{cut}(y) == d(y, u) \wedge$ not isExact($y$) then
           /* $y$ is a boundary node */
13         $\tilde{c}'(y) \leftarrow \tilde{c}(y) - \frac{r(y)-r'(y)+1}{d_{cut}(y)+2} + \frac{1}{d_{cut}(y)+1}$;
14     else
           /* we compute the distance-based bounds */
15         $\tilde{c}'(y) \leftarrow \tilde{c}(y) + \sum_{i=1}^{\text{diam}} \frac{1}{i+d(y,u)} \left(n'_i(u) - n_i(u)\right)$;
16         $d_{cut}(y) \leftarrow$ diam;
17     if $\tilde{c}'(y) \geq x_k$ then
           /* we have to run a new BFSCut */
18         $(\tilde{c}'(y), \text{isExact}(y), d_{cut}(y)) \leftarrow$ BFScut($y, x_k$);
19         if isExact($y$) $\wedge\ \tilde{c}'(y) > x_k$ then
20             TopK.insert($\tilde{c}'(y)$, $y$);
21             if TopK.size() $> k$ then
22                 TopK.removeMin();
23             end
24             if TopK.size() $= k$ then
25                 $x_k \leftarrow$ TopK.getMin();
26             end
27         end
28     end
29 end
30 Set $\tilde{c}$, $r$, $n_i$ to $\tilde{c}'$, $r'$, $n'_i$;
```

10.3.3 *Update after an edge insertion*

We first focus on updating the NBCUT top-k closeness algorithm. Before the insertion, we assume that, for each node y, we know $\tilde{c}(y)$, $d_{cut}(y)$ and $\mathsf{isExact}(y)$ computed by the function `BFSCut`. Also, we assume the nodes are sorted by their $\tilde{c}$ value in a priority queue Q. After an insertion, affected nodes increase their closeness. One first simple strategy would be to run `BFSCut` from each affected node, using as x_k the current node with the k-th highest closeness. This would already save some time compared to the static algorithm, since no work is performed for unaffected nodes. In the following, we propose some further improvements. Algorithm 21 shows the pseudocode of the dynamic algorithm. Initially, Line 1 and Line 2 update the number of reachable nodes and compute the set A of affected nodes, as described in Section 10.3.1 and Section 10.3.2, respectively. Then (Lines 3-8), the affected nodes that are among the top k are removed from TopK. For each affected node y, Lines 10-16 try to efficiently update $\tilde{c}$, in order to avoid a new `BFSCut` (as we will see in the following). If this is not possible, Lines 17-28 run `BFSCut` and update TopK as in the static algorithm.

SKIPPING FAR-AWAY NODES. Let y be an affected node such that $\mathsf{isExact}(y)$ is false, i.e., a node for which `BFSCut` has been interrupted at some cutoff level $d_{cut}(y)$. W.l.o.g., let us assume that $d(y,u) < d(y,v)$ if G is undirected. We recall that we know $d(\cdot,u)$ from the identification of the affected nodes described in Section 10.3.2.

If $d(y,u) > d_{cut}(y)$ and $r'(y) = r(y)$, the upper bound $\tilde{c}(y)$ is still valid. Figure 30 (left) shows this case. The reason for this is that u has not been visited by `BFSCut` and therefore the existence of edge (u,v) does not affect the bound in Eq. (26). If $d(y,u) > d_{cut}(y)$ but $r'(y) \neq r(y)$ (i.e., the insertion has increased the number of nodes reachable from y), we can simply replace $r(y)$ with $r'(y)$ in Eq. (26), obtaining $\tilde{c}'(y) = \tilde{c}(y) - \frac{r(y)}{d_{cut}(y)+2} + \frac{r'(y)}{d_{cut}(y)+2}$. If $\tilde{c}(y) < x_k$, y can therefore be skipped and no `BFSCut` on G' has to be run from it. We call such nodes *far-away nodes* (Line 11 of Algorithm 21).

SKIPPING BOUNDARY NODES. If $d(y,u)$ is equal to $d_{cut}(y)$ (Figure 30, right), then the bound in Eq. (26) is affected, since the degree of u changes (we recall that $\tilde{n}_{i+1}(y) := \sum_{w \in N_i(y)} \mathsf{degree}(w)$). In particular, $\tilde{n}'_{d_{cut}(y)+1}(y)$ after the insertion is equal to $\tilde{n}_{d_{cut}(y)+1}(y) + 1$, since the degree of u has increased by one. Thus, we can easily compute the new bound from the old one without running a new `BFSCut` from y as follows (we use $\tilde{n}_i$ instead of $\tilde{n}_i(y)$ and d_{cut} instead of $d_{cut}(y)$ for simplicity):

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

Boundary nodes are handled in Line 13 of Algorithm 21.

DISTANCE-BASED BOUNDS. The improvements described in the previous two paragraphs do not apply to affected nodes y for which $d(y,u) < d_{cut}(y)$. Let z be any node such that $d'(y,z) < d(y,z)$ (if y is affected, there has to exist such a node). Since all new shortest paths have to go through (u,v) (and thus through u), we can write $d'(y,z)$ as $d'(y,u) + d'(u,z) = d(y,u) + d'(u,z)$, since the distance from y to u cannot change as a consequence of the insertion of (u, v). As for $d(y, z)$, there are two options: either u was part of a shortest path from y to z also before the insertion – and thus $d(y, z) = d(y, u) + d(u, z)$, or there was a shorter path from y to z not going through u – and therefore $d(y, z) < d(y, u) + d(u, z)$. In both cases, we can say that $d(y, z) \leq d(y, u) + d(u, z)$. Putting this together, we get $\frac{1}{d'(y,z)} - \frac{1}{d(y,z)} \leq \frac{1}{d(y,u)+d'(u,z)} - \frac{1}{d(y,u)+d(u,z)}$. Thus:

$$\begin{aligned} c'(y) - c(y) &= \sum_{z \in V} \left(\frac{1}{d'(y,z)} - \frac{1}{d(y,z)} \right) \\ &\leq \sum_{z \in V} \left(\frac{1}{d(y,u) + d'(u,z)} - \frac{1}{d(y,u) + d(u,z)} \right) \\ &\leq \sum_{i=1}^{\mathsf{diam}} \frac{1}{i + d(y,u)} \left(n'_i(u) - n_i(u) \right) \end{aligned} \tag{27}$$

where diam is the diameter of G. Notice that Eq. (27) implies that, if $\tilde{c}(y)$ is an upper bound on $c(y)$, then $\tilde{c}'(y) := \tilde{c}(y) + \sum_{i=1}^{\mathsf{diam}} \frac{1}{i+d(y,u)} (n'_i(u) - n_i(u))$ is an upper bound on $c'(y)$. The values $(n'_i(u) - n_i(u))$ can be easily computed with one BFS from u in G and G' (this can also be combined with the BFSs we run to identify the affected node, see Section 10.3.2). Then, for each affected node that is neither a boundary node nor a far-away node, we compute a new upper bound as in Eq. (27). If this is still smaller than x_k, no `BFSCut` has to be performed from the node. Notice that the computation of the new bound requires $\Theta(\mathsf{diam})$ operations. Since the diameter in complex networks is very small (often assumed to be constant), this is much faster than running a `BFSCut`, which can take up to $\Theta(n + m)$ time. The distance-based bounds are computed in Line 15 of Algorithm 21. In Line 16, we set $d_{cut}(y)$ to diam to indicate that the current bound is not a result of a `BFSCut` and should not be used in the future to skip far-away or boundary nodes.

UPDATING NBBOUND. So far we described how to update top-k closeness assuming that NBCut has been run on the initial graph. We recall that NBBound only runs complete BFSs, until we find k nodes whose closeness is higher than the upper bounds on the remaining nodes. Thus, there is no cutoff threshold that we can use to skip far-away or boundary nodes. However, we can still make some considerations. First, also in this case $c(y)$ and $\tilde{c}(y)$ of unaffected nodes are still valid and do not need to be changed. Also, the distance-based bounds described in the previous paragraph can be applied to NBBound as well. If there are nodes y whose new bound is higher than the k-th highest closeness value, we run a BFS from y. We stop when there is no affected node left whose $\tilde{c}$ is higher than $\mathsf{TopK}[k]$, similarly to the static algorithm. Algorithm 22 shows the pseudocode.

10.3.4 *Update After an Edge Deletion*

In some sense, edge deletions are easier to handle than edge insertions. Indeed, since closeness can only decrease as a consequence of a deletion, nothing needs to be done in case none of the top-k nodes is affected (whereas for insertions there could be nodes that increase their closeness and "overtake" the previous top k). Also, notice that for affected nodes the previous upper bounds are still valid (although they might be less tight). If for

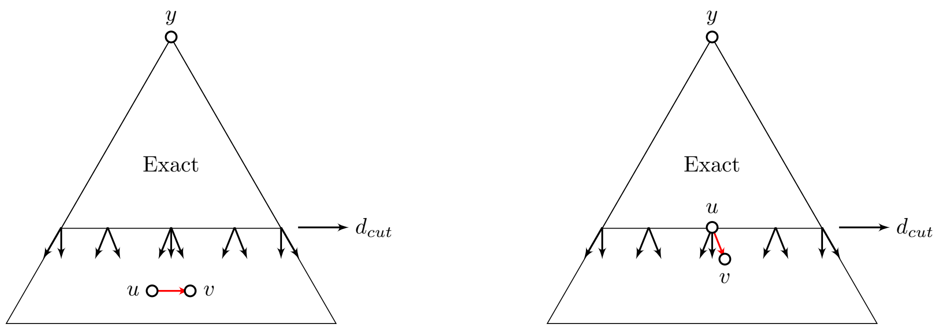


Figure 30: Left: u and v are far-away nodes for y. Right: u is a boundary node.

some affected node y we know the exact closeness $c(y)$ before the insertion, this becomes now an upper bound on the closeness of y in the new graph (thus, we set isExact(y) to true).

Once this is done, the algorithm goes through the nodes in Q ordered by $\tilde{c}$. If we find some node y for which isExact(y) = false, then we compute its new $c'(y)$ and update its priority in Q. We stop when isExact is true for the first k nodes in Q. Notice that this approach works for both NBCut and NBBound. Algorithm 23 shows the pseudocode for deletions, based in NBCut. The one based on NBBound is basically the same, with the difference that in Line 14 we do not run a `BFSCut`, but a BFSbound.

Algorithm 22: Recomputation of the top-k nodes after an edge insertion (based on NBBound).

```
Data: $G = (V, E), (u, v) \notin E$
Result: Top-$k$ nodes with the highest closeness in $G' := (V, E \cup \{u, v\})$
1  Compute the set $A$ of affected nodes, $d(\cdot, u)$, $d'(\cdot, u)$;
2  forall the $w \in A$ do
3      if $w \in$ TopK then
4          TopK.remove($w$);
5      end
6  end
7  Q ← $\emptyset$;
8  foreach $y \in A$ do
       /* we update the bounds on the affected nodes */
9      $\tilde{c}'(y) \leftarrow \tilde{c}(y) + \sum_{i=1}^{\text{diam}} \frac{1}{i+d(y,u)} \left(n'_i(u) - n_i(u)\right)$;
10     insert $y$ into Q with priority $\tilde{c}'(y)$;
11 end
12 while Q is not empty do
13     $y \leftarrow$ Q.extractMax();
14     if $|\text{TopK}| \geq k$ and $\tilde{c}(y) >$ TopK$[k]$ then return TopK;
15     $c(y) \leftarrow$ updateBoundsLB($v$); // This function might also modify $\tilde{c}$
16     add $y$ to TopK, and sort TopK according to $c$;
17     update Q according to the new bounds;
18 end
```

10.3.5 *Running Times and Memory Requirements*

The update of the number of reachable nodes described in Section 10.3.1 takes $O(n + m)$ in the worst case – the time to run a BFS from scratch. Computing the set of affected

Algorithm 23: Recomputation of the top-k nodes after an edge deletion (based on NBCut).

```
Data: $G = (V, E), (u, v) \notin E$
Result: Top-$k$ nodes with the highest closeness in $G' := (V, E \cup \{u, v\})$
Assume : $Q$ = priority queue with nodes sorted by $\tilde{c}$;
1  Compute $r'(y) \quad \forall y \in V$;
2  Compute the set $A$ of affected nodes, $d(\cdot, u)$, $d'(\cdot, u)$;
3  forall the $w \in A$ do
4      if $w \in$ TopK then
5          TopK.remove($w$);
6      end
7      isExact($w$) ← false;
8  end
9  $x_k \leftarrow 0$;
10 while Q is not empty do
11     $y \leftarrow$ Q.extractMax();
12     if $|$TopK$| \geq k$ and $\tilde{c}(y) >$ TopK$[k]$ then return TopK;
13     if not isExact($y$) then
           /* We run a new BFSCut                                    */
14         $(\tilde{c}'(y), \text{isExact}(y), d_{cut}(y)) \leftarrow$ BFScut($y, x_k$);
15     end
16     if isExact($y$) $\wedge$ $\tilde{c}'(y) > x_k$ then
17         TopK.insert($\tilde{c}'(y)$, $y$);
18         if TopK.size() $> k$ then
19             TopK.removeMin();
20         end
21         if TopK.size() $= k$ then
22             $x_k \leftarrow$ TopK.getMin();
23         end
24     end
25 end
26 Set $\tilde{c}$, $r$, $n_i$ to $\tilde{c}'$, $r'$, $n_i'$;
```

nodes takes $\Theta(n + m)$ time, since we need to run a BFS from u in G and G' (in the graph is undirected, also from v). Then, the algorithms described in Section 10.3.3 and Section 10.3.4 have to run, in the worst case, a BFSCut (or BFSbound, if we are considering the algorithm based on NBBound) for each affected node. Since the worst-case running time of both BFSCut and BFSbound is $O(n + m)$ (see Chapter 9), in the worst case the running time of the dynamic algorithms is $O(|A|(n + m))$, where $|A|$ is the number of affected nodes. However, we will see in Section 10.4 that the number of calls to BFSCut is usually a small fraction of the total number of affected nodes (which is, in turn, often only a small fraction of the total number of nodes).

Concerning memory, our algorithms need to store the bound $\tilde{c}(y)$, isExact(y), the number $r(y)$ of reachable nodes and the cutoff distance $d_{cut}(y)$, for each $v \in V$ (as well as the list TopK with the k nodes with maximum closeness). This requires only $\Theta(n)$ memory, which is asymptotically the same as the static top-k algorithms.

10.4 EXPERIMENTS

10.4.1 *Experimental Setup*

DATA SETS. We test our algorithms on numerous directed and undirected real-world complex networks (e. g., social networks or web graphs) and street networks. All of them can be retrieved from the public repositories SNAP (`snap.stanford.edu`), LASAGNE (`lasagne-unifi.sourceforge.net`), KONECT (`konect.uni-koblenz.de/networks`) and GEOFABRIC (`download.geofabrik.de`); the graphs are listed in Tables 31, 32 and 33. Since the street networks in Table 33 are all directed, we also consider an undirected version of them, by ignoring the direction of edges (multiple edges between nodes are ignored). For each tested graph, we either add or remove 100 edges one at a time and run the dynamic algorithm after each update. Due to time constraints, we only run the static algorithm once every 10 updates (this does not affect results considerably, since the running time of the static algorithm is always approximately the same). For edge insertions we remove 100 random edges from the original graph before running the algorithms, and then add them one-by-one (a motivation for this technique can be found in Section 3.2.2), whereas for deletions we just delete 100 random edges.

Graph	Nodes	Edges
polblogs	1224	19025
p2p-Gnutella08	6301	20777
wiki-Vote	7115	103689
p2p-Gnutella09	8114	26013
p2p-Gnutella06	8717	31525
p2p-Gnutella04	10876	39994
freeassoc	14213	72176
as-caida20071105	26475	106762
p2p-Gnutella31	62586	147892
soc-Epinions1	75879	508837
web-Stanford	281903	2312497
web-NotreDame	325729	1469679
wiki-Talk	2394385	5021410
cit-Patents	3774768	16518948

Table 31: Overview of directed complex networks used in the experiments.

Graph	Nodes	Edges
HC-BIOGRID	4039	10321
Mus_musculus	4610	5747
Caenorhabditis_elegans	4723	9842
ca-GrQc	5241	14484
advogato	7418	48037
hprd_pp	9465	37039
ca-HepTh	9877	25998
Drosophila_melanogaster	10625	40781
oregon1_010526	11174	23409
oregon2_010526	11461	32730
Homo_sapiens	13690	61130
GoogleNw	15763	148585
dip20090126_MAX	19928	41202
ca-HepPh	12008	118521
CA-AstroPh	18772	198110
CA-CondMat	23133	93497
email-Enron	36692	183831
loc-brightkite	58228	214078
gowalla	196591	950327
com-dblp	317080	1049866
com-amazon	334863	925872
com-youtube	1134890	2987624

Table 32: Overview of undirected complex networks used in the experiments.

IMPLEMENTATION AND SETTINGS. Since NBCUT outperforms NBBOUND on complex networks, we only use NBCUT and the new dynamic algorithm based on it (Algorithms 21 and 23) for our experiments on complex networks. Similarly, we only use

Graph	Nodes	Edges
maldives	22591	27088
faroe-islands	31097	31974
liechtenstein	54972	56616
isle-of-man	61082	63793
equatorial-guinea	80262	82245
malta	91188	101437
belize	96977	103198
azores	237174	243185

Table 33: Overview of street networks used in the experiments.

NBBound and the dynamic algorithm based on it for our experiments on street networks. We recall that our dynamic algorithm for directed graphs uses the number of nodes in the weakly connected components instead of the bound originally proposed in Chapter 9. However, for the static case, we use the original bound, in order to make a fair comparison. We recall that all algorithms are *exact*, i.e., they find the k nodes with highest closeness and their exact scores, so they only differ in their running time and not in the results they find.

The machine we used for our experiments is a shared-memory server equipped with 256 GB RAM and 2 x 8 Intel(R) Xeon(R) E5-2680 cores at 2.7 GHz of which we use only one since we executed our algorithms sequentially (i. e., using a single thread). The code has been written in C++ and uses the open-source *NetworKit* framework [119]. We plan to publish our code in future releases of the package.

10.4.2 *Speedups on Recomputation*

DYNAMIC COMPLEX NETWORKS. First, we study the effect of the optimizations proposed in Section 10.3.3. Table 35 in Section 10.4.2 contains the average number of affected nodes over the tested edge insertions and the percentage of nodes skipped due to each of the optimizations in undirected graphs (results for $k = 10$). The average number of affected nodes is never higher than 34% of the total number of nodes. Also, in all graphs, skipping far-away nodes allows us to ignore the vast majority of affected nodes after an update. Combined with the cheap updates for boundary nodes and applying the distance-based bounds, we need to run a new `BFSCut` for less than 1% of the affected nodes on most graphs. Notice that the column `BFScuts` contains the percentage of *affected nodes* for which we run a new `BFSCut`. The percentage of the total number of nodes is therefore much smaller. Table 34 shows the results for the directed case. Compared to undirected graphs, insertions typically affect smaller portions of the graph (average values at most $\approx$ 7%). Among them, usually a smaller percentage (compared to undirected graphs) are far-away nodes, probably because mostly nodes that are very close to the inserted edge are affected. However, several nodes are skipped because of the distance-based bounds, resulting in a very small number of BFSs. The highest numbers are for `p2p-Gnutella08`, for which the `BFScuts` are $\approx$ 31% of the affected nodes and $\approx$ 1.5% of the total.

The speedups (ratio between the running times) of our dynamic algorithm on the static one for complex networks are summarized in the lower part of Figure 31. Table 36 and Table 37 report the detailed values for insertions in directed and undirected graphs, respectively. For undirected networks, the geometric mean of the speedups (over the 100 edge insertions) are always at least in the double-digit range (with the only exception of `Mus_musculus`, where the average speedup for $k = 1$ is 8.4). Also, the speedups grow for bigger values of k, reaching an average speedup (over all tested undirected networks) of 123 for $k = 100$. Although for directed graphs the average speedups vary a lot (from ≈ 10 for `as-caida20071105` to ≈ 3368 for `web-Stanford`, for $k = 1$), the results are mostly even better than for the undirected case: the average speedups over all tested networks are 62 for $k = 1$, 93 for $k = 10$ and 174 for $k = 100$. This can be explained by the smaller number of affected nodes in directed networks (see Tables 35 and 34).

Tables 38 and 39 show the results for edge deletions (on directed and undirected graphs, respectively). Interestingly, deletions are mostly faster than insertions for directed graphs, whereas they are usually slower in the undirected case. For most shortest-path based problems insertions are easier than deletions: for example, pairwise distances can be updated in time $O(n^2)$ after an edge insertion, but not after an edge deletion [44]. In our case, we know deletions can only *decrease* centrality. Thus, all previous upper bounds on the centralities are still valid and, if none of the top-k nodes is affected, nothing needs to be updated. In insertions, on the contrary, any affected node could increase its centrality and become one of the top-k. If the number of affected nodes is small (as it is usually the case for directed graphs, see Table 34), it is quite unlikely that a top-k node is among the affected ones. This happens much more often in undirected graphs, where a larger number of nodes is often affected. As for insertions, the speedups increase with k: for directed graphs, the geometric mean of the speedups is 74 for $k = 1$, 160 for $k = 10$ and 314 for $k = 100$, whereas for undirected graphs it is 12.5 for $k = 1$, 25.8 for $k = 10$ and 50.2 for $k = 100$. All detailed running times for both insertions and deletions can be found in Tables 40, 41, 42 and 43.

DYNAMIC STREET NETWORKS. Tables 44, 45, 46 and 47 show the speedups for street networks, for both edge insertions and deletions, on directed and undirected graphs. Figure 31 summarizes all results for both complex and street networks (the results for street networks are in the upper part of the figure). As for complex networks, speedups in street networks are considerably higher in the directed case. However, differently from complex networks, speedups generally decrease as k increases. In this respect, notice that the running times of the static algorithm (Tables 48, 49, 50 and 51) do not change considerably for different values of k (at least, compared to most complex networks). On the other hand, if k is larger, it is also more likely that some of the affected nodes is either among the top-k or overtakes one of the top-k, slowing down the dynamic algorithm. Nevertheless, even for $k = 100$, the dynamic algorithm is on average ≈ 49 times faster than recomputation for insertions in undirected street networks and ≈ 242 times in directed street networks. The results for deletions are even better: ≈ 67 in the undirected case and ≈ 519 in the directed one. The results are significantly better for $k = 1$, reaching an average speedup of ≈ 848 for edge deletions in directed graphs, and ≈ 187 in undirected graphs.

BIBLIOGRAPHIC NOTES

The results presented in this chapter are joint work with Patrick Bisenius, Eugenio Angriman and Henning Meyerhenke and have been accepted for publication at the *Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX 2018)*. Preliminary re-

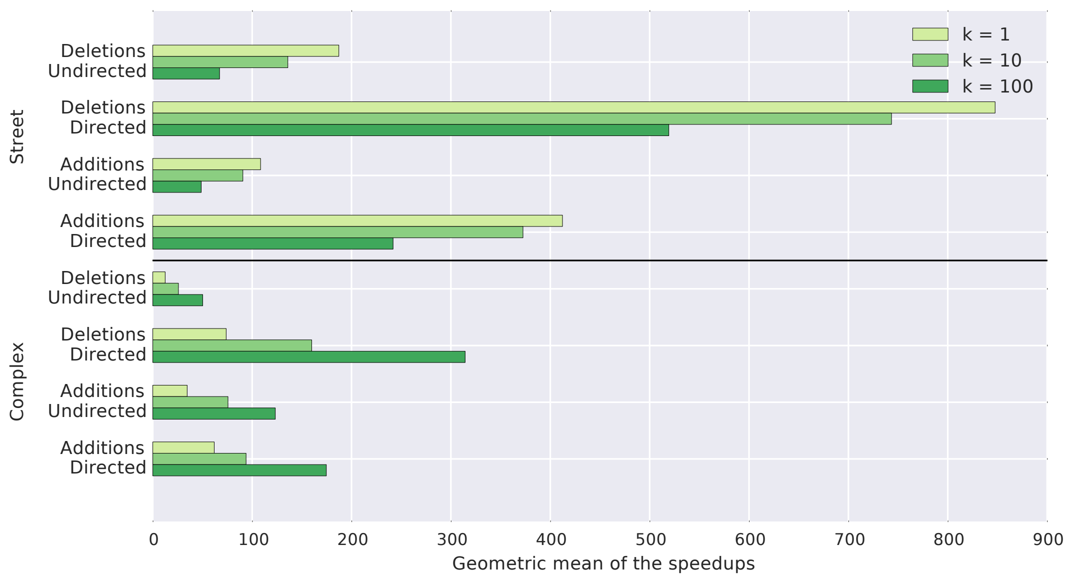


Figure 31: Geometric mean of the average speedups over all tested networks, for different values of k. The upper part of the plot shows the results for street networks, whereas the lower part shows the results for complex networks. Detailed numbers can be found in Table 36, 37, 38 and 39 for complex networks, and Tables 44, 45, 46 and 47 for street networks.

sults are part of Patrick Bisenius's Master Thesis, entitled "Computing Top-k Closeness Centrality in Fully-dynamic Graphs".

Graph	affected	% affected	far-away	boundary	dist. bound	BFScuts
polblogs	39	3.166%	41.084%	13.265%	36.310%	9.342%
p2p-Gnutella08	311	4.942%	42.941%	19.577%	6.102%	31.380%
wiki-Vote	31	0.440%	0.287%	0.255%	79.355%	20.102%
p2p-Gnutella09	426	5.245%	50.895%	17.709%	12.175%	19.220%
p2p-Gnutella06	589	6.752%	60.853%	20.203%	7.343%	11.601%
p2p-Gnutella04	773	7.108%	67.576%	17.189%	6.639%	8.597%
freeassoc	323	2.273%	0.000%	0.000%	95.220%	4.780%
as-caida20071105	2076	7.843%	87.803%	11.912%	0.247%	0.038%
p2p-Gnutella31	2895	4.625%	71.353%	13.087%	10.360%	5.200%
soc-Epinions1	659	0.868%	13.611%	38.786%	20.705%	26.898%
wiki-Talk	3411	0.142%	96.579%	3.306%	0.106%	0.009%
web-Stanford	10968	3.891%	66.812%	12.232%	14.690%	6.265%
web-NotreDame	3896	1.196%	94.599%	2.200%	3.112%	0.090%
cit-Patents	161	0.004%	0.000%	0.000%	99.901%	0.099%

Table 34: Impact of optimizations in directed networks for $k = 10$, averaged over 100 insertions. The column "affected" contains the average number of affected nodes and "% affected" its percentage of the total number of nodes. The next three columns report the percentage of affected nodes that can be skipped for each optimization and the last column the percentage of affected nodes for which a `BFSCut` has been run.

Graph	affected	% affected	far-away	boundary	dist. bound	BFScuts
HC-BIOGRID	598	14.812%	61.555%	8.876%	27.527%	2.043%
Mus_musculus	1556	33.762%	62.127%	7.547%	29.682%	0.644%
Caenorhabditis_elegans	508	10.758%	70.768%	6.794%	21.452%	0.986%
ca-GrQc	690	13.170%	81.017%	7.090%	10.792%	1.101%
advogato	281	3.784%	58.696%	18.037%	21.496%	1.771%
hprd_pp	731	7.728%	92.468%	5.731%	1.471%	0.329%
ca-HepTh	1362	13.795%	89.968%	5.780%	2.641%	1.611%
Drosophila_melanogaster	836	7.872%	85.411%	10.178%	2.483%	1.927%
oregon1_010526	1932	17.288%	76.625%	22.486%	0.805%	0.084%
oregon2_010526	1470	12.824%	76.226%	22.781%	0.774%	0.220%
Homo_sapiens	692	5.054%	93.008%	5.686%	1.003%	0.304%
GoogleNw	838	5.316%	39.976%	59.204%	0.717%	0.103%
dip20090126_MAX	3082	15.467%	44.315%	11.963%	35.153%	8.569%
ca-HepPh	745	6.204%	77.089%	3.473%	18.230%	1.208%
CA-AstroPh	795	4.233%	93.756%	4.596%	1.506%	0.142%
CA-CondMat	3026	13.081%	93.658%	5.090%	1.050%	0.202%
email-Enron	2097	5.716%	93.122%	6.144%	0.665%	0.069%
loc-brightkite	5973	10.259%	72.360%	7.362%	19.820%	0.459%
gowalla	16581	8.434%	74.731%	1.456%	23.786%	0.026%
com-dblp	67781	21.377%	92.218%	2.708%	5.044%	0.029%
com-amazon	71059	21.220%	88.512%	3.888%	7.057%	0.543%
com-youtube	101963	8.984%	86.257%	12.424%	0.908%	0.411%

Table 35: Impact of optimizations in undirected complex networks for $k = 10$, averaged over 100 insertions. The column "affected" contains the average number of affected nodes and "% affected" its percentage of the total number of nodes. The next three columns report the percentage of affected nodes that can be skipped for each optimization and the last column the percentage of affected nodes for which a `BFSCut` has been run.

Graph	k = 1			k = 10			k = 100		
	gmean	min	max	gmean	min	max	gmean	min	max
polblogs	41.6	11.3	67.5	78.5	1.9	149.2	146.8	3.5	252.4
p2p-Gnutella08	18.9	1.7	34.1	32.6	0.7	268.6	53.7	1.2	503.1
wiki-Vote	52.6	2.0	85.1	53.1	1.2	156.0	191.4	7.2	298.2
p2p-Gnutella09	37.8	5.8	75.5	24.7	1.0	189.3	45.7	1.3	707.7
p2p-Gnutella06	21.4	9.4	34.7	26.0	1.4	210.2	45.8	1.8	689.9
p2p-Gnutella04	23.9	9.0	35.3	30.6	2.2	277.6	34.4	2.4	804.3
freeassoc	433.8	91.2	685.8	347.4	1.6	766.1	403.1	11.8	831.6
as-caida20071105	9.6	2.7	12.7	14.1	2.1	19.5	99.3	3.9	221.2
p2p-Gnutella31	76.1	16.4	194.2	48.0	3.1	491.8	21.2	1.1	1084.7
soc-Epinions1	11.1	0.1	73.8	35.1	0.1	386.7	84.7	0.3	1284.7
web-Stanford	3367.6	0.4	21500.8	3364.7	0.6	28577.8	5270.0	0.9	41562.7
web-NotreDame	50.7	12.5	72.3	569.7	6.0	901.6	1363.2	15.2	2560.3
wiki-Talk	14.5	11.1	15.7	43.6	6.0	51.2	174.3	11.1	235.0
cit-Patents	1699.1	1127.4	1897.9	2237.3	823.2	2636.3	2513.0	365.6	3393.6
(geometric) mean	61.9	6.8	118.2	93.9	2.3	362.7	174.7	3.8	922.6

Table 36: Speedups on recomputation over 100 edge insertions in directed complex networks, for $k \in \{1, 10, 100\}$. The column “gmean” contains the geometric mean of the achieved speedups, “min” and “max” the minimum and the maximum speedup.

Graph	k = 1			k = 10			k = 100		
	gmean	min	max	gmean	min	max	gmean	min	max
HC-BIOGRID	37.8	6.1	65.3	42.0	6.2	97.0	79.0	4.9	332.4
Mus_musculus	8.4	3.6	39.6	14.0	3.7	101.8	20.5	2.6	358.4
Caenorhabditis_elegans	12.1	3.7	17.5	18.9	3.7	35.9	35.7	2.9	162.6
ca-GrQc	25.9	8.5	84.3	39.7	7.2	159.7	53.4	3.4	420.9
advogato	37.5	11.8	42.6	62.1	2.6	108.2	107.4	3.2	268.3
hprd_pp	18.1	4.4	26.5	31.8	3.6	55.2	51.8	3.6	323.1
ca-HepTh	30.8	3.6	205.9	36.8	4.7	385.0	54.7	4.4	957.5
Drosophila_melanogaster	19.3	5.3	27.0	59.6	9.5	156.1	60.2	3.5	272.6
oregon1_010526	12.2	2.9	19.4	15.1	2.4	29.5	50.8	2.6	225.4
oregon2_010526	16.7	3.8	24.1	22.4	2.9	44.8	80.8	3.4	258.8
ca-HepPh	365.4	15.6	704.1	397.6	3.7	3352.4	487.0	7.1	972.2
Homo_sapiens	20.6	5.5	26.6	34.8	6.2	64.2	59.1	4.8	424.8
GoogleNw	31.3	8.8	36.9	798.4	78.8	1045.3	886.6	11.4	1452.1
CA-AstroPh	133.1	6.8	288.1	280.1	28.5	414.5	418.4	11.3	6602.3
dip20090126_MAX	53.1	6.5	645.1	54.1	6.2	1096.5	57.1	9.3	1878.0
CA-CondMat	35.8	9.3	183.8	50.1	5.0	417.8	94.7	3.3	1458.5
email-Enron	56.0	2.0	267.2	105.9	11.0	442.8	222.6	5.2	446.8
loc-brightkite	23.2	4.3	33.1	133.2	2.9	1200.6	132.9	5.0	2028.6
gowalla	15.9	3.8	22.1	142.1	8.4	257.3	936.0	6.7	3463.2
com-dblp	140.1	17.6	282.7	130.4	15.1	357.3	108.7	4.7	766.4
com-amazon	445.1	19.6	1389.8	304.6	8.8	1853.6	518.3	9.0	5538.1
com-youtube	11.6	2.8	17.0	505.9	4.1	2122.3	479.2	2.0	3669.0
(geometric) mean	34.6	5.9	77.3	75.7	6.3	265.9	123.3	4.6	790.7

Table 37: Speedups on recomputation over 100 edge insertions in undirected complex networks, for $k \in \{1, 10, 100\}$. The column “gmean” contains the geometric mean of the achieved speedups, “min” and “max” the minimum and the maximum speedup.

Graph	k = 1			k = 10			k = 100		
	gmean	min	max	gmean	min	max	gmean	min	max
polblogs	43.0	29.7	47.7	94.7	78.4	102.5	106.6	61.2	151.3
p2p-Gnutella08	16.6	2.0	31.6	59.2	0.8	229.6	114.2	2.5	523.9
wiki-Vote	32.1	28.6	33.9	61.2	59.1	63.3	98.2	32.4	116.7
p2p-Gnutella09	36.6	8.4	75.2	36.1	1.1	182.4	35.7	4.3	155.7
p2p-Gnutella06	45.0	13.4	53.4	33.4	1.6	148.9	145.0	4.8	614.4
p2p-Gnutella04	23.5	8.9	27.4	63.7	5.2	236.2	89.7	7.7	794.0
freeassoc	373.5	305.6	1412.8	397.8	331.3	1398.6	332.1	72.2	1683.0
as-caida20071105	8.6	8.1	8.8	14.0	12.1	14.5	154.2	124.5	161.9
p2p-Gnutella31	68.8	10.3	220.0	132.5	2.9	652.2	190.4	7.5	1585.5
soc-Epinions1	41.1	37.0	43.4	138.6	108.7	165.0	629.3	266.8	754.6
web-Stanford	3796.1	3711.1	3887.1	6019.9	4281.5	55778.7	9471.6	6989.9	104810.2
web-NotreDame	58.7	49.6	69.6	859.7	781.9	896.5	2353.9	2166.3	2486.6
wiki-Talk	21.6	20.1	22.8	81.1	75.8	84.8	348.4	335.9	366.2
cit-Patents	3717.3	2923.6	4394.9	6357.6	3543.7	7096.1	3472.4	180.5	7523.4
(geometric) mean	73.9	38.8	104.3	159.9	40.9	360.5	314.3	62.2	879.0

Table 38: Speedups on recomputation over 100 edge deletions in directed complex networks, for $k \in \{1, 10, 100\}$. The column "gmean" contains the geometric mean of the achieved speedups, "min" and "max" the minimum and the maximum speedup.

Graph	k = 1			k = 10			k = 100		
	gmean	min	max	gmean	min	max	gmean	min	max
HC-BIOGRID	8.8	1.8	15.2	11.3	1.3	24.6	24.2	2.4	86.8
Mus_musculus	6.0	2.9	104.8	9.6	2.8	350.2	13.9	2.2	679.9
Caenorhabditis_elegans	4.2	1.6	6.3	6.1	1.5	13.2	12.9	1.8	58.8
ca-GrQc	17.1	1.7	339.4	22.6	2.2	576.8	39.8	2.4	1679.5
advogato	11.1	2.4	407.8	20.7	1.1	1207.4	35.9	1.4	3046.0
hprd_pp	5.6	1.6	6.9	8.2	1.4	14.4	22.9	1.3	86.3
ca-HepTh	12.7	1.3	478.3	17.2	1.2	937.3	27.3	2.0	2436.8
Drosophila_melanogaster	5.6	1.6	6.8	21.3	1.4	39.2	20.3	1.7	69.7
oregon1_010526	3.9	2.0	5.1	4.5	1.7	7.9	14.9	1.6	63.1
oregon2_010526	4.6	2.0	5.9	6.1	1.5	11.3	20.2	2.4	66.0
ca-HepPh	116.8	1.0	8902.1	109.4	1.0	9336.2	129.7	1.0	12759.1
Homo_sapiens	5.5	1.9	6.5	9.1	1.2	15.8	31.5	1.5	106.9
GoogleNw	9.1	2.3	13.0	149.5	1.0	354.1	164.7	1.1	507.9
CA-AstroPh	65.6	1.1	4020.2	88.4	1.1	5886.6	207.8	1.1	18076.6
dip20090126_MAX	89.4	1.7	182.4	81.2	1.4	264.5	102.9	1.8	487.4
CA-CondMat	13.5	2.2	470.7	21.4	1.3	996.5	47.2	1.8	3807.3
email-Enron	17.1	1.3	780.5	30.1	7.6	1204.6	63.5	4.8	3607.3
loc-brightkite	6.0	1.6	230.0	40.2	1.1	2566.3	45.6	1.4	4319.3
gowalla	4.6	2.2	5.4	36.2	1.2	60.0	336.8	1.3	836.0
com-dblp	37.3	2.7	81.6	39.3	2.3	101.9	50.3	1.9	219.3
com-amazon	155.9	8.1	461.2	99.4	3.0	539.6	211.3	9.5	1595.0
com-youtube	3.4	1.9	4.4	117.4	1.0	585.6	156.9	1.1	1029.9
(geometric) mean	12.5	1.9	70.0	25.8	1.5	213.6	50.2	1.8	662.9

Table 39: Speedups on recomputation over 100 edge deletions in undirected complex networks, for $k \in \{1, 10, 100\}$. The column "gmean" contains the geometric mean of the achieved speedups, "min" and "max" the minimum and the maximum speedup.

Graph	k = 1		k = 10		k = 100	
	static [s]	dynamic [s]	static[s]	dynamic [s]	static [s]	dynamic [s]
polblogs	0.0121	0.0003	0.0298	0.0004	0.0586	0.0004
p2p-Gnutella08	0.0148	0.0008	0.1026	0.0031	0.2636	0.0049
wiki-Vote	0.0536	0.0010	0.0969	0.0018	0.1857	0.0010
p2p-Gnutella09	0.0446	0.0012	0.1072	0.0043	0.3640	0.0080
p2p-Gnutella06	0.0242	0.0011	0.1293	0.0050	0.4541	0.0099
p2p-Gnutella04	0.0323	0.0013	0.2238	0.0073	0.8322	0.0242
freeassoc	0.2658	0.0006	0.2977	0.0009	0.3493	0.0009
as-caida20071105	0.0416	0.0043	0.0702	0.0050	0.7898	0.0079
p2p-Gnutella31	0.8200	0.0108	2.4506	0.0511	6.0270	0.2841
soc-Epinions1	0.8373	0.0752	2.9520	0.0840	12.7048	0.1500
web-Stanford	207.7361	0.0617	288.4620	0.0857	406.5858	0.0772
web-NotreDame	1.1105	0.0219	13.9541	0.0245	40.1074	0.0294
wiki-Talk	2.9550	0.2035	9.6525	0.2212	43.4118	0.2491
cit-Patents	351.6200	0.2069	371.7316	0.1662	428.1452	0.1704

Table 40: Update times for 100 random edge insertions with $k \in \{1, 10, 100\}$ in directed complex networks. The columns "static" and "dynamic" contain the average time for the static and dynamic algorithm, respectively.

Graph	k = 1		k = 10		k = 100	
	static [s]	dynamic [s]	static [s]	dynamic [s]	static [s]	dynamic [s]
HC-BIOGRID	0.0317	0.0008	0.0527	0.0013	0.1791	0.0023
Mus_musculus	0.0083	0.0010	0.0260	0.0019	0.0976	0.0048
Caenorhabditis_elegans	0.0106	0.0009	0.0217	0.0011	0.0980	0.0027
ca-GrQc	0.0270	0.0010	0.0469	0.0012	0.1324	0.0025
advogato	0.0491	0.0013	0.1289	0.0021	0.3229	0.0030
hprd_pp	0.0339	0.0019	0.0708	0.0022	0.4217	0.0081
ca-HepTh	0.0705	0.0023	0.1328	0.0036	0.3514	0.0064
Drosophila_melanogaster	0.0385	0.0020	0.2217	0.0037	0.3956	0.0066
oregon1_010526	0.0173	0.0014	0.0263	0.0017	0.2057	0.0040
oregon2_010526	0.0233	0.0014	0.0441	0.0020	0.2568	0.0032
ca-HepPh	1.4801	0.0041	1.5431	0.0039	2.0807	0.0043
Homo_sapiens	0.0506	0.0025	0.1228	0.0035	0.8286	0.0140
GoogleNw	0.0810	0.0026	2.3284	0.0029	3.3147	0.0037
CA-AstroPh	1.0297	0.0077	1.4850	0.0053	4.5794	0.0109
dip20090126_MAX	1.6690	0.0314	2.4519	0.0453	4.5685	0.0800
CA-CondMat	0.1445	0.0040	0.3113	0.0062	1.1810	0.0125
email-Enron	0.3866	0.0069	0.6290	0.0059	1.9659	0.0088
loc-brightkite	0.2249	0.0097	2.3622	0.0177	4.3341	0.0326
gowalla	0.8203	0.0515	9.0723	0.0638	127.8335	0.1366
com-dblp	16.8464	0.1202	20.9283	0.1605	44.7540	0.4117
com-amazon	102.1518	0.2295	118.0336	0.3875	345.3928	0.6663
com-youtube	3.5802	0.3074	427.8860	0.8458	759.7169	1.5854

Table 41: Update times for 100 random edge insertions with $k \in \{1, 10, 100\}$ in undirected complex networks. The columns "static" and "dynamic" contain the average time for the static and dynamic algorithm, respectively.

Graph	k = 1		k = 10		k = 100	
	static [s]	dynamic [s]	static [s]	dynamic [s]	static [s]	dynamic [s]
polblogs	0.0188	0.0004	0.0401	0.0004	0.0580	0.0005
p2p-Gnutella08	0.0163	0.0010	0.1189	0.0020	0.2669	0.0023
wiki-Vote	0.0527	0.0016	0.0995	0.0016	0.1859	0.0019
p2p-Gnutella09	0.0448	0.0012	0.1087	0.0030	0.3623	0.0101
p2p-Gnutella06	0.0444	0.0010	0.1245	0.0037	0.5147	0.0035
p2p-Gnutella04	0.0307	0.0013	0.2646	0.0042	0.8984	0.0100
freeassoc	0.2904	0.0008	0.3084	0.0008	0.3455	0.0010
as-caida20071105	0.0422	0.0049	0.0713	0.0051	0.7937	0.0051
p2p-Gnutella31	0.8146	0.0118	2.5231	0.0190	6.1104	0.0321
soc-Epinions1	0.8558	0.0208	2.9626	0.0214	12.7745	0.0203
web-Stanford	207.5102	0.0547	276.8251	0.0460	406.2662	0.0429
web-NotreDame	1.1191	0.0191	13.9595	0.0162	40.1689	0.0171
wiki-Talk	2.8892	0.1339	9.8011	0.1208	43.7422	0.1256
cit-Patents	337.1497	0.0907	375.2206	0.0590	396.8926	0.1143

Table 42: Update times for 100 random edge deletions with $k \in \{1, 10, 100\}$ in directed complex networks. The columns "static" and "dynamic" contain the average time for the static and dynamic algorithm, respectively.

Graph	k = 1		k = 10		k = 100	
	static [s]	dynamic [s]	static [s]	dynamic [s]	static [s]	dynamic [s]
HC-BIOGRID	0.0317	0.0036	0.0496	0.0044	0.1720	0.0071
Mus_musculus	0.0086	0.0014	0.0261	0.0027	0.0989	0.0071
Caenorhabditis_elegans	0.0101	0.0024	0.0209	0.0035	0.0950	0.0073
ca-GrQc	0.0260	0.0015	0.0443	0.0020	0.1297	0.0033
advogato	0.0492	0.0044	0.1272	0.0062	0.3217	0.0090
hprd_pp	0.0339	0.0061	0.0692	0.0084	0.4139	0.0180
ca-HepTh	0.0656	0.0051	0.1278	0.0074	0.3346	0.0122
Drosophila_melanogaster	0.0410	0.0074	0.2310	0.0109	0.4046	0.0199
oregon1_010526	0.0163	0.0042	0.0250	0.0056	0.2009	0.0135
oregon2_010526	0.0235	0.0051	0.0445	0.0073	0.2601	0.0129
ca-HepPh	1.4666	0.0126	1.5403	0.0141	2.1020	0.0162
Homo_sapiens	0.0497	0.0091	0.1217	0.0133	0.8283	0.0263
GoogleNw	0.0875	0.0096	2.3516	0.0157	3.3499	0.0203
CA-AstroPh	1.0323	0.0157	1.4989	0.0170	4.6072	0.0222
dip20090126_MAX	1.6675	0.0187	2.4406	0.0301	4.5202	0.0439
CA-CondMat	0.1444	0.0107	0.3036	0.0142	1.1564	0.0245
email-Enron	0.3726	0.0217	0.6040	0.0201	1.9093	0.0300
loc-brightkite	0.2105	0.0352	2.3587	0.0586	4.3993	0.0964
gowalla	0.8376	0.1825	9.4024	0.2595	131.2249	0.3896
com-dblp	16.0662	0.4307	20.3018	0.5169	43.8383	0.8710
com-amazon	98.1340	0.6294	116.4843	1.1724	349.3862	1.6538
com-youtube	3.6342	1.0569	427.4117	3.6402	751.0728	4.7879

Table 43: Update times for 100 random edge deletions with $k \in \{1, 10, 100\}$ in undirected complex networks. The columns "static" and "dynamic" contain the average time for the static and dynamic algorithm, respectively.

Graph	k = 1			k = 10			k = 100		
	gmean	min	max	gmean	min	max	gmean	min	max
maldives	485.9	9.3	734.5	482.7	3.7	883.5	584.8	10.9	980.4
faroe-islands	431.5	2.8	13250.3	276.5	1.7	11531.9	148.1	3.2	15783.5
liechtenstein	162.8	6.7	12125.3	159.5	2.2	15470.6	93.2	1.3	27578.6
isle-of-man	193.9	9.4	6855.9	117.3	1.2	7589.3	70.2	3.2	971.1
equatorial-guinea	455.2	8.2	6440.1	453.2	37.1	16065.3	405.8	23.7	21588.5
malta	438.7	20.7	6884.2	519.3	11.0	14307.0	283.3	4.9	13639.7
belize	534.3	70.7	14497.9	502.0	19.2	25484.5	153.7	7.3	28820.4
azores	1181.4	86.6	4154.8	1257.0	63.2	6980.3	1167.9	33.3	5929.5
(geometric) mean	412.3	14.2	6191.9	372.5	7.3	9144.8	241.8	6.7	8220.6

Table 44: Speedups on recomputation over 100 edge insertions in directed street networks, for $k \in \{1, 10, 100\}$. The column “gmean” contains the geometric mean of the achieved speedups, “min” and “max” the minimum and the maximum speedup.

Graph	k = 1			k = 10			k = 100		
	gmean	min	max	gmean	min	max	gmean	min	max
maldives	285.3	26.0	481.6	311.2	17.1	564.5	308.0	7.6	663.7
faroe-islands	110.2	1.1	4115.4	61.6	1.7	4473.0	36.8	1.2	6374.2
liechtenstein	69.1	7.0	5274.2	37.9	4.6	8756.1	11.2	2.4	13198.3
isle-of-man	49.9	2.4	4410.2	36.2	4.0	4264.1	14.3	2.3	5294.5
equatorial-guinea	86.8	16.8	4434.3	71.6	8.2	6111.2	55.1	4.1	7957.4
malta	107.1	3.9	2902.7	130.7	2.9	4338.3	62.3	2.9	8218.3
belize	86.2	5.3	5135.8	68.1	20.9	9255.8	23.6	4.0	10407.4
azores	220.9	9.1	1318.3	272.3	8.5	2563.5	218.0	8.7	1631.4
(geometric) mean	108.5	5.9	2821.6	90.7	6.2	3950.4	48.8	3.5	4892.4

Table 45: Speedups on recomputation over 100 edge insertions in undirected street networks, for $k \in \{1, 10, 100\}$. The column “gmean” contains the geometric mean of the achieved speedups, “min” and “max” the minimum and the maximum speedup.

Graph	k = 1			k = 10			k = 100		
	gmean	min	max	gmean	min	max	gmean	min	max
maldives	740.9	29.9	1227.1	795.1	25.8	1338.0	857.4	10.3	1655.0
faroe-islands	809.5	0.8	12583.8	697.2	0.7	13509.7	518.8	0.6	18936.6
liechtenstein	609.7	39.4	11082.1	382.0	27.4	24102.8	214.1	9.2	35979.0
isle-of-man	332.6	10.3	758.7	237.9	7.6	803.2	116.0	5.1	988.9
equatorial-guinea	2018.3	165.4	14889.7	2333.4	78.0	28155.3	2098.5	25.1	35160.2
malta	688.6	169.2	8015.5	613.3	111.1	14959.3	392.5	32.7	17787.7
belize	769.2	11.8	14131.3	556.2	7.3	34348.3	292.8	4.2	46801.4
azores	2049.9	288.8	8574.0	2326.7	150.5	11610.8	1986.6	39.3	13917.1
(geometric) mean	847.7	31.1	6084.0	743.4	20.7	9357.4	519.3	9.0	12082.7

Table 46: Speedups on recomputation over 100 edge deletions in directed street networks, for $k \in \{1, 10, 100\}$. The column “gmean” contains the geometric mean of the achieved speedups, “min” and “max” the minimum and the maximum speedup.

Graph	k = 1			k = 10			k = 100		
	gmean	min	max	gmean	min	max	gmean	min	max
maldives	328.9	24.5	539.9	389.2	23.1	658.6	404.1	8.9	806.0
faroe-islands	123.6	0.2	1795.7	93.7	0.2	4421.8	55.5	0.3	6637.0
liechtenstein	170.5	16.3	4666.8	57.9	15.7	6926.5	13.8	6.0	11481.9
isle-of-man	94.2	2.4	4243.1	53.2	3.2	4610.9	11.7	2.4	5920.1
equatorial-guinea	399.3	26.6	2919.2	406.2	28.4	4518.0	347.9	8.8	5519.6
malta	182.1	64.0	1306.5	144.3	54.9	1521.2	63.0	18.6	2144.7
belize	95.0	0.1	4535.5	54.0	0.1	6283.8	21.4	0.1	7672.6
azores	334.8	36.4	1314.7	326.8	28.8	1995.2	244.1	11.1	2319.9
(geometric) mean	187.2	5.2	2138.0	135.9	5.2	3076.0	67.2	2.9	4078.9

Table 47: Speedups on recomputation over 100 edge deletions in undirected street networks, for $k \in \{1, 10, 100\}$. The column "gmean" contains the geometric mean of the achieved speedups, "min" and "max" the minimum and the maximum speedup.

Graph	k = 1		k = 10		k = 100	
	static [s]	dynamic [s]	static [s]	dynamic [s]	static [s]	dynamic [s]
maldives	0.1959	0.0004	0.2038	0.0004	0.2897	0.0005
faroe-islands	4.0750	0.0094	4.2421	0.0153	4.0176	0.0271
liechtenstein	10.2787	0.0631	12.5527	0.0787	18.3544	0.1969
isle-of-man	6.4567	0.0333	7.0189	0.0598	8.3399	0.1188
equatorial-guinea	15.7533	0.0346	15.8417	0.0350	19.8871	0.0490
malta	16.3053	0.0372	17.8396	0.0344	20.9484	0.0739
belize	32.5222	0.0609	34.8647	0.0695	47.5453	0.3092
azores	46.1730	0.0391	52.2209	0.0415	54.7890	0.0469

Table 48: Update times for 100 random edge insertions with $k \in \{1, 10, 100\}$ in directed street networks. The columns "static" and "dynamic" contain the average time for the static and dynamic algorithm, respectively.

Graph	k = 1		k = 10		k = 100	
	static [s]	dynamic [s]	static [s]	dynamic [s]	static [s]	dynamic [s]
maldives	0.1691	0.0006	0.2071	0.0007	0.2508	0.0008
faroe-islands	2.7337	0.0248	1.8672	0.0303	2.8906	0.0786
liechtenstein	9.5222	0.1377	11.2397	0.2963	21.6126	1.9293
isle-of-man	5.4850	0.1100	5.5143	0.1524	7.0543	0.4931
equatorial-guinea	7.5368	0.0868	9.2345	0.1290	10.2358	0.1858
malta	9.0850	0.0849	10.0963	0.0772	13.6970	0.2198
belize	21.6908	0.2517	23.7571	0.3488	38.5866	1.6374
azores	17.4890	0.0792	19.3370	0.0710	25.3668	0.1163

Table 49: Update times for 100 random edge insertions with $k \in \{1, 10, 100\}$ in undirected street networks. The columns "static" and "dynamic" contain the average time for the static and dynamic algorithm, respectively.

Graph	k = 1		k = 10		k = 100	
	static [s]	dynamic [s]	static [s]	dynamic [s]	static [s]	dynamic [s]
maldives	0.2083	0.0003	0.2424	0.0003	0.2888	0.0003
faroe-islands	3.4412	0.0043	3.6236	0.0052	4.8986	0.0094
liechtenstein	11.2583	0.0185	12.0275	0.0315	17.9281	0.0838
isle-of-man	5.8541	0.0176	6.2707	0.0264	7.6060	0.0656
equatorial-guinea	19.8018	0.0098	19.9973	0.0086	23.3713	0.0111
malta	15.1507	0.0220	14.7942	0.0241	18.3038	0.0466
belize	29.0321	0.0377	31.2502	0.0562	42.3123	0.1445
azores	43.8422	0.0214	45.3105	0.0195	51.5531	0.0260

Table 50: Update times for 100 random edge deletions with $k \in \{1, 10, 100\}$ in directed street networks. The columns “static” and “dynamic” contain the average time for the static and dynamic algorithm, respectively.

Graph	k = 1		k = 10		k = 100	
	static [s]	dynamic [s]	static [s]	dynamic [s]	static [s]	dynamic [s]
maldives	0.1689	0.0005	0.2073	0.0005	0.2535	0.0006
faroe-islands	1.8495	0.0150	2.0199	0.0216	3.0493	0.0549
liechtenstein	9.2484	0.0542	10.2024	0.1761	17.7007	1.2800
isle-of-man	4.7809	0.0508	5.1547	0.0968	6.6677	0.5684
equatorial-guinea	9.2114	0.0231	10.0027	0.0246	12.8164	0.0368
malta	8.6663	0.0476	9.1300	0.0633	12.4407	0.1975
belize	16.8559	0.1775	18.9041	0.3498	30.4396	1.4236
azores	16.5104	0.0493	17.9961	0.0551	23.8098	0.0975

Table 51: Update times for 100 random edge deletions with $k \in \{1, 10, 100\}$ in undirected street networks. The columns “static” and “dynamic” contain the average time for the static and dynamic algorithm, respectively.

11 SCALING UP GROUP CLOSENESS MAXIMIZATION

11.1 INTRODUCTION

In Chapter 9 and Chapter 10 we presented algorithms for computing the k nodes that individually have highest closeness. In their seminal work, Borgatti and Everett [52] extended the concept of centrality to *groups* of nodes. For a node v and a group S of other nodes, the distance between v and S is defined as the minimum distance between v and the elements of S. Then, a group of nodes has high closeness when its average distance to the other nodes is small. Finding central groups of nodes is an important task for many applications. For example, in social networks, retailers might want to select a group of nodes as promoters of their product, in order to maximize the spread among users [69]. In this context, picking the k most central nodes might lead to a large overlap in the set of influenced nodes, whereas there might be k nodes that are not among the most central when considered individually, but that influence different areas of the graph.

Closely related to finding the group with highest closeness is p-median, a fundamental facility location problem in operations research [60]. One can see Group Closeness Maximization (GCM) as a special case of p-median: the standard GCM formulation applies only to graphs without vertex weights, whereas p-median also applies to geometric inputs and weighted objects (to name only few of the possible generalizations [47]). For p-median, several (meta)heuristics and approximation algorithms have been proposed over the years (see [106] for an annotated bibliograohy). Yet, these methods are mostly applicable to relatively small networks only. In [105], the authors compare state-of-the-art methods on a street network of Sweden ($\approx$ 190K nodes) and show that existing methods either fail due to their memory requirements (>32 GB) or take more than 14 hours to find an approximation. Other recent methods have been shown to scale to inputs with up to 90000 points/nodes [6, 64].

Specifically for GCM, an $(1 - 1/e)$-approximation algorithm has been proposed recently by Chen et al. [37]. Unfortunately, the algorithm does not scale easily to graphs with more than about 10^4 vertices, since it requires to compute pairwise distances. Thus, Chen et al. proposed in the same paper also a more scalable heuristic without guarantees on the solution quality.

In this chapter, we present techniques that can reduce considerably the memory and the number of operations required by the greedy algorithm presented in [37] (in both directed and undirected networks), without losing its theoretical guarantee on the quality of the approximation. First, instead of computing and storing all pairwise distances, we use the algorithm presented in Chapter 9 to find the node with maximum closeness. Then, we reduce the subsequent computations using pruned SSSPs (Section 11.4.1) and exploiting the submodularity of the objective function (Section 11.4.2). We also propose an approach based on bit vectors (Section 11.4.3) which is faster than pruned SSSPs but requires more memory. In our experiments in Section 11.6, we compare our algorithm (Greedy++) with the greedy approach presented in [37] and show that Greedy++ is orders of magnitude faster. Also, we compare Greedy++ with the heuristic proposed in [37] and show that

Greedy++ is often faster (or has a comparable running time) and that it always finds a better solution in all our experiments. We also provide an Integer Linear Programming (ILP) formulation of the GCM problem in Section 11.5 and compare the quality of our solution with the optimum. Our results show that the solution found by Greedy++ is actually much better than the theoretical guarantee and the empirical approximation ratio is never lower than 0.97. Finally, we study the overlap between the group with maximum closeness and the k nodes with highest closeness and highest degree in real-world networks, showing that in most cases this is relatively small (between 30% and 60% of the group size). This confirms the intuition that a central group of nodes is not necessarily composed of nodes that are individually central.

11.2 PRELIMINARIES

We model a network as a graph $G = (V, E)$ with $|V| =: n$ nodes and $|E| =: m$ edges. Unless stated explicitly, we assume the graph to be connected (or, if directed, strongly connected) and unweighted. Let $d(u, v)$ represent the shortest-path distance between node u and node v. We define the distance between $u \in V$ and a set $S \subseteq V$ of nodes as $d(u, S) := \min_{s \in S} d(u, s)$. Then, the closeness centrality of a set S is:

$$c(S) := \frac{n - |S|}{\sum_{v \notin S} d(S, v)} \ .$$

The Group Closeness Maximization (GCM) problem is defined as finding a set $S^\star \subseteq V$ of a given size k, with maximum group closeness: $S^\star = \arg\max_{S \subseteq V}\{c(S) : |S| = k\}$.

11.3 RELATED WORK

GCM has been very recently considered in [37], where the authors show that finding the group with maximum closeness is an NP-hard problem. Also, they propose a greedy algorithm and prove that the solution found by the algorithm is at most a factor $(1 - 1/e)$ away from the optimum. Since the greedy algorithm is still expensive (its complexity is $\Theta(kn^2)$ plus the cost of an APSP, for a group of size k), the authors propose an alternative heuristic based on sampling. In particular, they first propose a baseline heuristic (BSA), which basically samples a set of nodes and then selects iteratively the node that minimizes the distance of the current solution to the samples. Then, they show that the running time of BSA can be improved by dividing the set of samples in partitions (and they call this second heuristic Order-based Sampling Algorithm, OSA). However, the two heuristics do not have the theoretical guarantee of the greedy algorithm, so we do not know how well they approximate the optimum. Since the algorithm proposed in this chapter builds on the greedy algorithm of [37], we describe it in more detail in Section 11.3.1.

In [126], an algorithm for computing and maximizing group closeness on disk-resident graphs has been proposed. The basic idea is to estimate the closeness of a group using the nodes at distance at most H from the group (where H can be any integer value greater than 0). Although they show that their approach can scale quite well for small values of H, there is no guarantee on how close their estimation is to the real centrality of the group. The problem of finding a central group of nodes has also been considered for betweenness centrality, for which sampling-based approximation algorithms have been proposed [87, 125].

11.3.1 *Greedy approximation algorithm*

Chen et al. [37] proposed a greedy approximation algorithm (Greedy) for group closeness. We recall that the objective is to find a set $S^\star$ such that $S^\star = \arg\max_{S \subseteq V}\{c(S) : |S| = k\}$. Greedy runs k iterations, after which it returns a set S. Within each iteration, Greedy adds to the set S the node u with the largest marginal gain $c(S \cup \{u\}) - c(S)$. Since the objective function c is monotone and submodular (as proven in [37]), Greedy provides a $(1 - 1/e)$-approximation for the GCM problem, i. e.$c(S) \geq (1 - 1/e)c(S^\star)$. Algorithm 24 shows the pseudocode of Greedy. In Line 2 the pairwise distances are computed and stored in the $n \times n$ matrices d and M. In each iteration, d always contains the pairwise distances, whereas M contains, for each node pair (u, w), the distance $d(S \cup \{u\}, w)$. Initially $d = M$, since $S = \emptyset$. Then, every time a node s is added to S, M is updated in Line 10. *Score* contains $c(S \cup \{u\})$ for each node u, which is computed in Line 13 by summing over $M(u, w)$, $\forall w \in V$.

Since it needs to store two $n \times n$ matrices, the memory requirement of Greedy is $\Theta(n^2)$. The running time is $\Theta(n(m + n \log n))$ for the initial APSP computation (when running a SSSP from each node in a weighted graph) and then $\Theta(kn^2)$ for the remaining part.

Algorithm 24: Greedy algorithm for GCM [37].

```
    Input   : A graph $G = (V, E)$, a number $k$
    Output : A set $S$ of nodes of size $k$ such that $c(S) \geq (1 - 1/e)c(S^\star)$
 1  $d \leftarrow$ APSP($G$);
 2  $M \leftarrow$ APSP($G$);
 3  $Score \leftarrow \{c(u) | u \in V\}$;
 4  $s \leftarrow \arg\max_{u \in V \setminus S} Score[u]$;
 5  $S \leftarrow \{s\}$;
 6  while $|S| < k$ do
 7      foreach $u \in V \setminus S$ do
 8          foreach $w \in V$ do
 9              if $d[u, w] > d[s, w]$ then
10                  $M[u, w] \leftarrow d[s, w]$;
11              end
12          end
            /* $Score[u]$ is set to $c(S \cup \{u\})$ */
13          $Score[u] \leftarrow (n - |S| - 1) / \sum_{w \in V \setminus S} M[u, w]$;
14          $s \leftarrow \arg\max_{w \in V \setminus S} Score[w]$;
15          $S \leftarrow S \cup \{s\}$;
16      end
17  end
18  return $S$;
```

11.4 A SCALABLE GREEDY ALGORITHM

First of all, we notice that we can reduce the memory requirement of Greedy by not storing the matrices d and S. In fact, to find the first element s_0 of S (i. e.the node with maximum closeness) we can simply use the algorithm NBCut presented in Chapter 9.

Algorithm 25: Memory-efficient greedy algorithm.

```
Input  : A graph $G = (V, E)$, a number $k$
Output : A set $S$ of nodes of size $k$ such that $c(S) \geq (1 - 1/e)c(S^\star)$
1  $s_0 \leftarrow$ NBCut$(G, 1)$;
2  $S \leftarrow \{s_0\}$;
3  SSSP$(s_0)$;
4  $d_S[u] \leftarrow d(s_0, u) \quad \forall u \in V$;
5  while $|S| < k$ do
6      foreach $u \in V \setminus S$ do
7          SSSP$(u)$;
           /* $Score[u]$ is set to $c(S \cup \{u\})$ */
8          $t \leftarrow \sum_{w \in V \setminus S} \min\{d(u, w),\ d_S[w]\}$;
9          $Score[u] \leftarrow (n - |S| - 1)/t$;
10     end
11     $s \leftarrow \arg\max_{w \in V \setminus S} Score[w]$;
12     $S \leftarrow S \cup \{s\}$;
13     SSSP$(s)$;
14     foreach $u \in V$ do
15         $d_S[u] \leftarrow \min\{d_S[u],\ d(s, u)\}$;
16     end
17 end
18 return $S$;
```

Then, we can use a vector d_S containing, for each node v, the distance between S and v (i.e. $d_S[v] := d(S, v)$). Since initially S is composed of only one element s_0, d_S simply contains the distances between s_0 and the other nodes, which can be computed with a SSSP rooted in s_0. Then, for each node $u \in V \setminus S$, Lines 8-10 can be replaced with a SSSP rooted in u where we sum, over each node w visited in the SSSP, the minimum between $d_S(w)$ and $d(u, w)$. This sum is exactly the same as $\sum_{w \in V \setminus S} M[u, w]$ and can therefore be used in Line 13 to update $Score[u]$. The memory-efficient version of Greedy is described in Algorithm 25. In the pseudocode we report explicitly every time we need to run a SSSP. In Line 3 and Line 13, the SSSP is needed to compute d_S, whereas in Line 7 we need it to compute $Score[u]$.

Since we have to re-run a SSSP for each node u and for each element of S other than s_0, the running time complexity of the while loop of Algorithm 25 is $O(kn(m + n \log n))$ (for weighted graphs). The worst-case complexity of finding s_0 with NBCut is the same as that of an APSP (i. e.$n(m + n \log n)$), although in practice it is basically linear in the size of the graph (see Chapter 9). For unweighted graphs, the complexity of Algorithm 25 is $O(knm)$, since we can use BFS instead of Dijkstra to compute the SSSPs. Although the memory requirement is now only $\Theta(n)$ (in addition to the memory required to store the graph), the time complexity is too high to target large networks. For this reason, in the following we propose improvements that, as we will see in Section 11.6, increase the scalability of Greedy considerably.

11.4.1 *Pruned SSSP*

In Line 7 of Algorithm 25, we need to run a SSSP rooted in u to recompute $Score[u]$. However, the only nodes w for which we need to compute $d(u, w)$ are those for which $d(u, w) < d_S[w]$, i. e.the ones that are closer to u than to S. Indeed, for all the other nodes, the distance from u does not contribute to the sum in Line 8 and therefore to

$Score[u]$. Thus, if we know that $d(u, w)$ is larger than or equal to $d_S[w]$, we do not need to visit w in the SSSP. It is not hard to see that, if $d(u, w) \geq d_S[w]$, then the same holds for *all the nodes* in the SSSP subtree rooted in w. In fact, let t be a node in the SSSP subtree of w, i. e.$d(u, t) = d(u, w) + d(w, t)$. There is a path between a node in S and t going through w of length $d_S[w] + d(w, t)$. Therefore $d_S[t] \leq d_S[w] + d(w, t) \leq d(u, t)$. Figure 32 illustrates this concept. This allows us to *prune* the SSSP when we find a node whose distance from u is not smaller than its distance from S. When we visit a new node w, we compare $d(u, w)$ with $d_S[w]$. If the first is not strictly smaller than the second, we do not enqueue its neighbors into the SSSP (priority) queue. Notice that, since only nodes u for which $d_S[u] \leq d(s, u)$ are pruned, the value of $d_S[u]$ in Line 15 is not affected, for any $u \in V$. This means that the solution returned by the improved algorithm is exactly the same as the solution returned by Algorithm 25.

11.4.2 *Submodularity improvement*

A function f is submodular whenever $f(S \cup \{u\}) - f(S) \geq f(T \cup \{u\}) - f(T)$, for $S \subseteq T$. It is not hard to see that the closeness c of a set is a submodular function [37]. We can use this property to reduce the number of evaluations of $Score$ (and SSSP computations) in Lines 7-9. Let us name S_i the set S computed by Algorithm 25 in the i-th iteration of the while loop (S_i is the set composed of i elements). Since $S_i \subseteq S_{i+1}$, because of submodularity $c(S_i \cup \{u\}) - c(S_i) \geq c(S_{i+1} \cup \{u\}) - c(S_{i+1})$. The difference $c(S_i \cup \{u\}) - c(S_i)$ is then the *marginal gain* $\Delta(u, S_i)$ ($\Delta_i(u)$, in short) of u with respect to S_i.

In other words, we can say that at each iteration of the while loop in Algorithm 25, the marginal gain of each node can only decrease. Now, let us assume that there is a node s whose marginal gain $\Delta_i(s)$ with respect to S_i is larger than the marginal gain $\Delta_{i-1}(u)$ of a node u in the previous iteration. This means that the marginal gain of u at iteration i cannot be larger than $\Delta_i(s)$ (since $\Delta_i(u) \leq \Delta_{i-1}(u) \leq \Delta_i(s)$). This allows us to skip the computation of the score of u in Lines 7 - 9. All we need to do is keep track of the marginal gain of each node in the previous iteration and compare it with the maximum marginal gain found in the current iteration. Notice that this improvement is compatible with the pruned SSSP improvement proposed in the previous section. For the nodes that cannot be skipped because of what was described in this section, we compute their score with a pruned SSSP. We name our version of Algorithm 25 using pruned SSSPs and the submodularity improvement Greedy++. As explained in Section 11.4.1, using pruned SSSPs does not affect the solution found by the algorithm. The improvement described in this section might return a different solution only in case there are nodes with the same marginal gain. Indeed, if there are two nodes u and v with the same marginal gain $\Delta^\star$ and

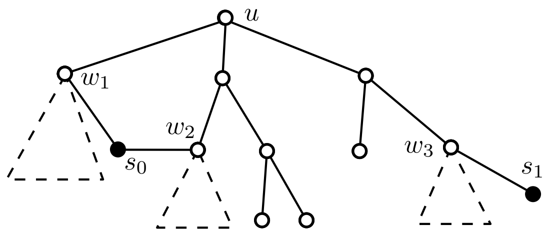


Figure 32: Pruned SSSP. If a node w is such that $d_S[w] \leq d(u, w)$, the same holds for the whole SSSP subtree rooted in w. In the figure, black nodes represent elements of S.

such that $\Delta^\star \geq \Delta(w)\ \forall w \in V$, whether we choose u or v depends on which comes first in the ordering of the nodes. Nevertheless, this does not influence the guarantee on the quality of the approximation.

Consequently, the following theorem holds.

Theorem 11.4.1. *Let $S \subseteq V$, $|S| = k$, be the solution returned by* Greedy++*. Then, it holds that $c(S) \geq (1 - 1/e)c(S^\star)$, where $S^\star = \arg\max_{S \subseteq V}\{c(S) : |S| = k\}$.*

11.4.3 *Bit-parallel group closeness*

To further speed up Greedy++, we propose an optimization for unweighted graphs exploiting bit-level parallelism. Bit-parallel methods try to exploit the fact that computers can perform bitwise operations on a word at once. Let $B_i(u)$ be a bit vector with the j-th bit set to 1 if $d(u,j) \leq i$ and set to 0 otherwise. It is easy to see that $B_i(u) = \bigoplus_{v \in N(u)} B_{i-1}(v)$, for $i \geq 1$, where $\oplus$ represents a bitwise-OR operation and $N(u)$ are the neighbors of u. Then, if we indicate the number of ones in a bit vector B as $|B|$, the closeness $c(u)$ of u can be expressed as $(n-1)/\sum_{i=1}^{\mathsf{diam}} i(|B_i(u)| - |B_{i-1}(u)|)$, where diam is the diameter of G. A simple algorithm for computing the closeness of all nodes could therefore work as follows: Initialize $B_0(u)$ as a bit vector with a 1 in position u and 0 everywhere else, for each $u \in V$. Then, for $i = 1, \ldots, \mathsf{diam}$, compute $B_i(u)$ as $\bigoplus_{v \in N(u)} B_{i-1}(v)$. Although the complexity of this algorithm $(O(\mathsf{diam} \cdot nm))$ is higher than that of running a BFS from each node $(O(nm))$, diam is usually very small in complex networks, and bitwise operations are very fast (see for example [113]).

We can use bitwise operations also to compute group closeness. Similarly to $B_i(u)$, we can define $B_i(S)$ of a set S as a bit vector where the j-th bit set to 1 if $d(S,j) \leq i$. Then, using $\otimes$ to indicate a bitwise-AND, and $\neg$ for a bitwise-NOT, we can prove the following.

Theorem 11.4.2. *The node $u^\star$ with the highest marginal gain with respect to set S is*

$$u^\star = \arg\max_{u \in V \setminus S} \sum_{i=0}^{\mathsf{maxD}(u)} |B_i(u) \otimes \neg B_i(S)|$$

where $\mathsf{maxD}(u) := \max\{i \geq 0 : |B_i(u) \otimes \neg B_i(S)| > 0\}$.

Proof. We recall that the marginal gain $\Delta(u,S)$ of node u with respect to set S is $c(S \cup \{u\}) - c(S)$. Clearly, $\Delta(u,S) > \Delta(v,S) \iff \sum_{w \in V}(d(S,w) - d(S \cup \{u\}, w)) > \sum_{w \in V}(d(S,w) - d(S \cup \{v\}, w))$, for any two nodes u and v. Now, naming $V(u)$ the set of nodes w such that $d(S,w) > d(u,w)$, we can write $\sum_{w \in V}(d(S,w) - d(S \cup \{u\}, w))$ as $\sum_{w \in V(u)}(d(S,w) - d(u,w))$. Thus, the node $u^\star$ with maximum marginal gain is

$$\arg\max_{u \in V \setminus S} \sum_{w \in V(u)} (d(S,w) - d(u,w)) \ .$$

For $i \geq 0$, $|B_i(u) \otimes \neg B_i(S)|$ is the number of nodes w such that $d(u,w) \leq i$ (as they are in $B_i(u)$) and $d(S,w) > i$ (as they are not in $B_i(S)$). Let w be any node in $V(u)$. For each i such that $d(u,w) \leq i < d(S,w)$, the bit corresponding to w in $B_i(u) \otimes \neg B_i(S)$ is set to 1. This means that, for each $w \in V(u)$, $\sum_{i=0}^{\mathsf{maxD}(u)} |B_i(u) \otimes \neg B_i(S)|$ adds one to the sum a number of times equal to $d(S,w) - d(u,w)$. This means that $\sum_{i=0}^{\mathsf{maxD}(u)} |B_i(u) \otimes \neg B_i(S)| = \sum_{w \in V(u)}(d(S,w) - d(u,w))$, which proves the theorem. □

Theorem 11.4.2 gives us a simple algorithm for finding the node with maximum marginal gain: First, we compute $B_i(S)$, for $i \leq \mathsf{diam}$. Then, for each distance i starting from 1 and for each node u, we compute $B_i(u)$ as $\bigoplus_{v \in N(u)} B_{i-1}(v)$. Notice that, if $|B_i(u) \otimes \neg B_i(S)| = 0$ for some value of i, this will also be true for any $j > i$, so the search from u can be interrupted at distance i. This is in some sense equivalent to the pruned SSSP described in Section 11.4.1, but using bit vectors. Also, notice that the algorithm can be combined with the submodularity improvement in Section 11.4.2. Although using bit vectors can speed up the algorithm (up to a factor 4, in our experiments in Section 11.6.5), a major limitation of this approach is its memory requirement: for each node, we need to store a bit vector of length n, leading to a total of $\Theta(n^2)$ memory. This yields a tradeoff between memory and speed.

11.5 ILP FORMULATION OF GROUP CLOSENESS

To evaluate the quality of the solution found by Greedy++, we want to know how far it is from the optimum. Computing the closeness centrality of all possible subsets of size k would clearly be prohibitive even for tiny networks. Hence, we formulate GCM as an ILP problem. This will be used in the experiments in Section 11.6.1.

For each node $v_j \in V$, we define a binary variable y_j, which is 1 if node v_j is part of the group with maximum closeness $S^\star$, and is equal to 0 otherwise. We say a node v_i is *assigned* to a node $v_j \in S^\star$ if $d(v_i, S^\star) = d(v_i, v_j)$. If there are multiple nodes $v_j \in S^\star$ that satisfy this property, v_i can be arbitrarily assigned to one of them. Thus, we also define a variable x_{ij} that, for each node pair (v_i, v_j) is equal to 1 if $v_j \in S^\star$ and v_i is assigned to v_j, and 0 otherwise. We can rewrite our problem in the following form:

$$\max \frac{n-k}{\sum_{i=1}^{n} \sum_{j=1}^{n} d(v_i, v_j) x_{ij}} \tag{28}$$

s.t.: $(i)\ \sum_{j=1}^{n} x_{ij} = 1,\ \forall i \in \{1, ..., n\}$; $(ii)\ \sum_{j=1}^{n} y_j = k$; $(iii)\ x_{ij} \leq y_j,\ \forall i, j \in \{1, ..., n\}$.

Condition (i) indicates that each node in $v_i \in V$ is assigned to exactly one node in $v_j \in S^\star$, (ii) indicates that $|S^\star| = k$ and (iii) indicates that nodes v_i are only assigned to nodes v_j that are in $S^\star$, i. e.nodes for which $y_j = 1$. Since the numerator in Eq. (28) is constant, we can rewrite Eq. (28) as:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} d(v_i, v_j) x_{ij}, \tag{29}$$

which gives us an ILP formulation.

11.6 EXPERIMENTS

In the following, we present experimental results concerning several aspects of our new algorithm Greedy++. Apart from Section 11.6.5 (where we compare the two versions), we always refer to the version using pruned SSSPs described in Section 11.4.1 and not to the one using bit vectors described in Section 11.4.3. In Section 11.6.1 we study the accuracy of Greedy++ in comparison with the optimum. In Section 11.6.2, we show the speedup of Greedy++ on the greedy algorithm proposed in [37] (which we call Greedy). Then, in

Section 11.6.3, we compare Greedy++ with OSA, the heuristic based on sampling proposed in [37] (we did not implement the other heuristic BSA, since the authors of [37] show that OSA always finds a solution with a similar accuracy as BSA in a smaller running time). In Section 11.6.4, we study the running time of Greedy++ on additional larger networks, both for a sequential and a parallel implementation (the other algorithms are either too slow or would require too much memory for these networks). Finally, in Section 11.6.6, we study the correlation between the group with maximum closeness and the top-k nodes with highest closeness in real-world networks.

All algorithms are implemented in C++, building on the open-source network analysis tool NetworKit [119]. All experiments were done on a machine equipped with 256 GB RAM and a 2.7 GHz Intel Xeon CPU E5-2680 having 2 sockets with 8 cores each. The machine runs 64 bit SUSE Linux and we compiled our code with g++-4.8.1 and OpenMP 3.1. For comparability with previous work, unless stated explicitly, running times refer to a sequential implementation.

The graphs used in the experiments are taken from the SNAP [81], KONECT [76] and LASAGNE(`piluc.dsi.unifi.it/lasagne`) data sets. The `easyjet` graph in Table 52 was taken from [42]. All graphs are connected, undirected and unweighted.

11.6.1 *Accuracy*

The quality comparison between the quality of the solution found by Greedy++ and the optimum is performed on several small real-world networks; the optimum is computed using the ILP formulation described in Section 11.5. The ILP model is implemented using the Java optimization modeling library and interface ILOG Concert Technology. The problems are solved with ILOG CPLEX 12.6 (`www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`). The results for $k = 10$ are reported in Table 52. Among all networks, the empirical approximation ratio (ratio between the objective function of the optimum and that of the solution found by Greedy++) is always higher than 0.97. This is much higher than the theoretical guarantee of $(1 - 1/e) \approx 0.63$. Similar results can be observed for $k = 2$ and $k = 20$, reported in Table 53 and Table 54. For $k = 10$, the geometric mean of the approximation ratios is 0.994, for $k = 2$ it is 0.998 and for $k = 20$ it is 0.995. Notice that Greedy++ never takes more than one second on the tested networks, whereas finding the optimum with CPLEX takes hours for the larger instances of Table 52.

11.6.2 *Speedup on Greedy*

Recall that the solution found by the two algorithms Greedy++ and Greedy is the same, thus we only compare running times between the two. Due to the time and space complexity of Greedy, we compare the two approaches on two relatively small networks (`ca-HepTh`: 8638 nodes and 24806 edges and `oregon_1_010526`: 11174 nodes and 23409 edges). Figure 33 shows the running times of the two algorithms for different values of group size k between 10 and 1000. For both graphs, Greedy++ outperforms Greedy by orders of magnitude. For all tested group sizes, Greedy++ finds the solution in less than one second, whereas for $k = 1000$ Greedy requires 25 minutes on the `ca-HepTh` graph and 34 minutes on the `oregon_1_010526` graph. The speedups of Greedy++ on Greedy ranges between 93

Graph	Nodes	Edges	Category	Optimum	Greedy++	Approx. ratio
`karate`	35	78	friendship	25	25	1.0
`contiguous-usa`	49	107	transport.	40	41	0.976
`easyjet`	136	755	transport.	126	126	1.0
`jazz`	198	2742	collaboration	191	192	0.995
`coli1-1Inter`	328	456	metabolic	475	482	0.985
`pro-pro`	1458	1993	metabolic	4213	4217	0.999
`hamster-friend`	1788	12476	social	2871	2871	1.0
`dnc-temporal`	1833	4366	communicat.	2398	2407	0.996
`caenorhab-eleg`	4428	9659	metabolic	10003	10075	0.993

Table 52: Comparison with optimum on small real-world networks, for $k = 10$. The fifth and the sixth columns show the objective function of Eq. (29) for the optimum and Greedy++, respectively.

Graph	Nodes	Edges	Category	Optimum	Greedy++	Approx. ratio
`karate`	35	78	friendship	37	37	1.0
`contiguous-usa`	49	107	transport.	99	99	1.0
`easyjet`	136	755	transport.	143	143	1.0
`jazz`	198	2742	collaboration	259	261	0.992
`coli1-1Inter`	328	456	metabolic	780	780	1.0
`pro-pro`	1458	1993	metabolic	5573	5573	1.0
`hamster-friend`	1788	12476	social	3596	3596	1.0
`dnc-temporal`	1833	4366	communicat.	3236	3236	1.0
`caenorhab-eleg`	4428	9659	metabolic	12535	12631	0.992

Table 53: Comparison with optimum on small real-world networks, for $k = 2$. The fifth and the sixth columns show the objective function of Eq. (29) for the optimum and Greedy++, respectively.

Graph	Nodes	Edges	Category	Optimum	Greedy++	Approx. ratio
`karate`	35	78	friendship	15	15	1.0
`contiguous-usa`	49	107	transport.	29	29	1.0
`easyjet`	136	755	transport.	116	116	1.0
`jazz`	198	2742	collaboration	178	178	1.0
`coli1-1Inter`	328	456	metabolic	367	373	0.984
`pro-pro`	1458	1993	metabolic	3488	3518	0.991
`hamster-friend`	1788	12476	social	2556	2573	0.993
`dnc-temporal`	1833	4366	communicat.	2066	2082	0.992

Table 54: Comparison with optimum on small real-world networks, for $k = 20$. The fifth and the sixth columns show the objective function of Eq. (29) for the optimum and Greedy++, respectively. The results for `caenorhab-eleg` are not included, because the CPLEX solver did not find the optimum within 13 hours.

($k = 10$) and 1765 ($k = 1000$) for `ca-HepTh` and between 581 ($k = 10$) and 6125 ($k = 1000$) for `oregon_1_010526`.

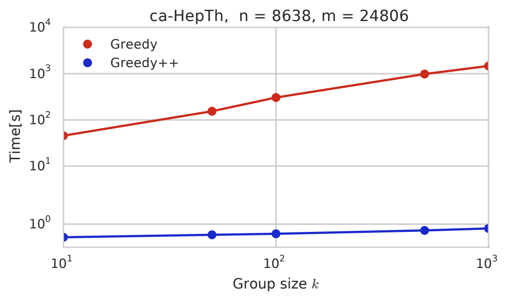


Figure 33: Running times of Greedy and Greedy++ for different group sizes (log-log scale). left: running times for `ca-HepTh`; right: running times for `oregon_1_010526`.

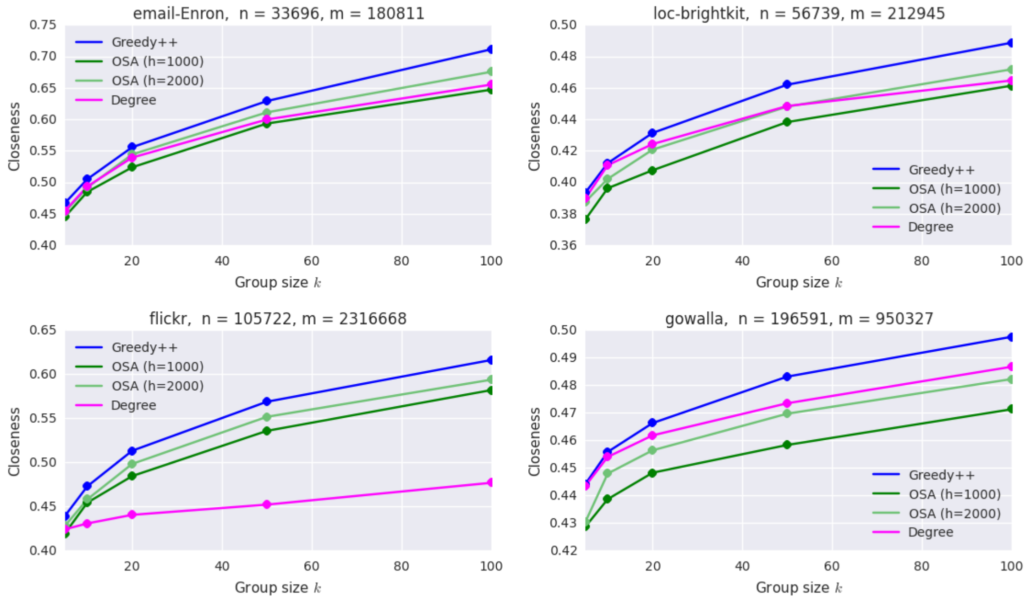


Figure 34: Closeness centrality of the solution found by the methods for different group sizes and different graphs. The plot shows the results of Greedy++, OSA with sample sizes of 1000 and 2000, and the group consisting of the k nodes with highest degree.

11.6.3 *Comparison with OSA*

Since OSA is a sampling-based algorithm, the number h of samples influences its performance, both in terms of accuracy and running time. In [37], the authors suggest $h = 1000$ samples as a good tradeoff for group sizes up to 50. Since we are also testing the algorithms on groups with up to 100 nodes, we run OSA both with $h = 1000$ and with a larger sample size of $h = 2000$. We test OSA and Greedy++ on all the networks of Table 55 with $m < 10^7$ (11 networks). We did not run experiments on larger networks because of the high memory requirements of OSA. Since OSA is a sampling-based approach, we repeat each experiment 10 times and report the average running time and accuracy.

Figure 34 shows the group closeness of the solutions found by OSA and Greedy++ on four of the tested graphs (`email-Enron`, `loc-brightkit`, `flickr`, and `gowalla`), for group sizes ranging between 5 and 100. As a baseline, we also report the closeness of the group composed of the k nodes with maximum degree (Degree).

In addition to having a theoretical guarantee (whereas OSA has none), the results show that Greedy++ always finds the best solution, for all graphs and group sizes. Interestingly, for all the four graphs but `flickr`, the set of nodes with maximum degree has a higher

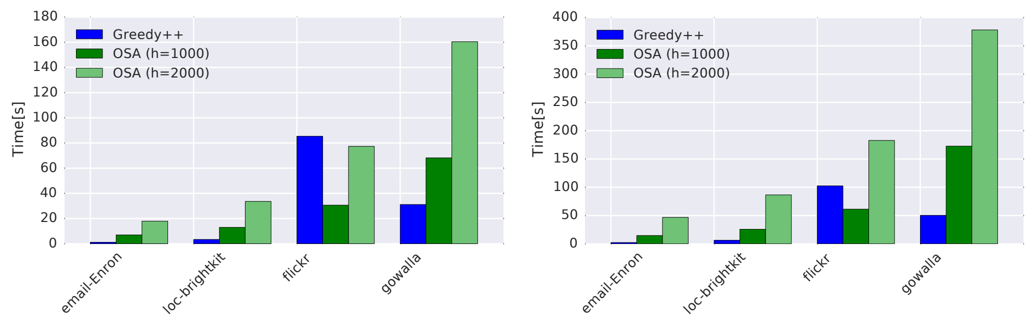


Figure 35: Running times of Greedy++ and OSA with sample sizes of 1000 and 2000 for $k = 20$ (left) and $k = 100$ (right).

closeness than the solution found by OSA with $h = 1000$ samples. For the `gowalla` graph, Degree finds a better solution than OSA even with $h = 2000$ samples.

Figure 35 shows the running times of Greedy++ and OSA on the four graphs, for group size $k = 20$ (left) and $k = 100$ (right). On all graphs but `flickr`, Greedy++ is significantly faster than OSA (both with $h = 1000$ and $h = 2000$ samples). On the `flickr` graph, for group size $k = 20$, Greedy++ takes 85 seconds, whereas OSA with $h = 2000$ takes 77 seconds. However, when the group size increases ($k = 100$), Greedy++ becomes faster (102 seconds versus 182 seconds required by OSA with $h = 2000$).

Also, notice that the memory requirement of Greedy++ is significantly lower than that of OSA. In fact, Greedy++ only needs $\Theta(n)$ memory for its data structures, whereas OSA requires $\Theta(hn)$ to store the distances between the sampled nodes and the other nodes. This means that, using OSA with the number of samples suggested in [37], it needs about one thousand times more memory than Greedy++, which might be problematic for large graphs.

On average (geometric mean) over the 11 tested networks, Greedy++ is faster than OSA with $h = 1000$ by a factor of 1.1 and than OSA with $h = 2000$ by a factor of 1.7. Although our average running times are not very different from those of OSA with $h = 1000$, our accuracy is better on all tested networks (the same is true also for OSA with $h = 2000$). Also, on 7 out of the 11 tested networks, OSA with $h = 1000$ returns a result with a worse accuracy than choosing the k nodes with maximum degree, suggesting that OSA should be run using a larger number of samples. With $h = 2000$, the solution of OSA is worse than picking the k nodes with maximum degree on 4 out of 11 networks (the solution returned by Greedy++ is better on all tested networks).

11.6.4 *Running time evaluation*

To test the scalability of Greedy++, we now run it on all networks from Table 55 (for the comparison with OSA, only the first 11 networks could be used). The networks belong to different domains, including friendship, collaboration, communication and internet topology graphs. To further speed up the running time of Greedy++, we also implement a parallel version of it. The first element of $|S|$ is computed using the parallel top-k closeness implementation of NBCut (see Chapter 9). Then, in each iteration of Greedy++, Line 6 of Algorithm 25 is executed in parallel, i.e. each thread runs a pruned SSSP from the nodes assigned to it. Table 55 reports the running times of Greedy++ for $k = 10$, for both the sequential and the parallel implementation (using 16 threads). On all networks with less than 10^5 nodes, our parallel implementation takes less than 1 second. On all remaining graphs, it always takes less than 1 hour, apart from the `com-orkut` graph ($>$ 3M nodes and $>$ 100M edges), where it takes a bit more than one and a half hours. The parallel speedup varies significantly among the tested networks, ranging from 5.4 (`com-youtube`) to 13.8 (`flickr`). These values should be appreciated in the context of complex networks, for which it is often difficult to obtain even higher speedups (see for example [91] and [118]). Low speedup values are in our case also due to the fact that, in some networks, the work done by the pruned SSSPs is extremely imbalanced (some nodes can be pruned early, whereas others need almost a full SSSP). Load balancing mechanisms beyond what OpenMP offers are outside the scope of this work, as they would require very fine-grained and inexpensive context switches between threads. Also, as expected, the parallel speedup decreases as k increases. Indeed, whereas the geometric mean of the speedups is 9.1 for $k = 10$, it is 8.7 for $k = 20$ and 5.6 for $k = 100$. This results from the fact that, for higher k values, more and more pruned SSSPs can be skipped because of submodularity. Since less work is done in each iteration, the overhead due to parallelism and imbalance becomes more significant. The fact that less and less work is done in each iteration as k increases is also confirmed by the fact that the running times do not increase linearly as k increases. For $k = 20$, the running times are only about 10% higher (on average) that they are for $k = 10$ and, for $k = 100$, they are about 50% higher than for $k = 10$ (running times for $k = 20$ and $k = 100$ can be found in Table 56.

Graph	Nodes	Edges	Time seq. [s]	Time par. [s]	Speedup
`ca-HepPh`	11204	117649	7.70	0.58	13.4
`email-Enron`	33696	180811	1.94	0.20	9.9
`CA-AstroPh`	17903	197031	3.78	0.32	12.0
`loc-brightkite`	56739	212945	5.74	0.55	10.5
`com-lj`	303526	427701	127.35	17.00	7.5
`com-amazon`	334863	925872	808.70	88.37	9.2
`gowalla`	196591	950327	60.14	8.74	6.9
`com-dblp`	317080	1049866	232.51	30.99	7.5
`flickr`	105722	2316668	314.11	22.76	13.8
`com-youtube`	1134890	2987624	1323.31	245.50	5.4
`youtube-u-growth`	3216075	9369874	22298.52	2196.42	10.2
`as-skitter`	1694616	11094209	12014.09	1611.09	7.5
`soc-pokec-relationships`	1632803	22301964	11912.29	1104.82	10.8
`com-orkut`	3072441	117185083	60252.10	5792.81	10.4

Table 55: Networks used in the experiments and performance of Greedy++ for $k = 10$. The fourth and fifth columns report the sequential and parallel running times with 16 threads, respectively. The last column reports the speedup of the parallel implementation on the sequential one.

11.6.5 *Greedy++ using bit vectors*

We now test the performance of the version of Greedy++ using bit vectors described in Section 11.4.3. Our implementation of bit vectors is based on the C++ `std::bitset` and

Graph	Nodes	Edges	Time $k = 20$ [s]	Time $k = 100$ [s]
`ca-HepPh`	11204	117649	0.61	0.7
`email-Enron`	33696	180811	0.26	0.6
`CA-AstroPh`	17903	197031	0.34	0.5
`loc-brightkite`	56739	212945	0.67	1.2
`com-lj`	303526	427701	18.16	24.2
`com-amazon`	334863	925872	94.56	116.2
`gowalla`	196591	950327	9.09	11.2
`com-dblp`	317080	1049866	34.26	49.5
`flickr`	105722	2316668	23.04	24.6
`com-youtube`	1134890	2987624	263.17	473.9
`youtube-u-growth`	3216075	9369874	2412.60	2901.9
`as-skitter`	1694616	11094209	1620.43	2024.6
`soc-pokec-relationships`	1632803	22301964	1179.33	1288.1
`com-orkut`	3072441	117185083	6233.67	8387.0

Table 56: Performance of Greedy++ for $k = 20$ and $k = 100$, using 16 threads.

we test both versions sequentially. Table 57 shows the ratio between the running times of Greedy++ using pruned SSSPs and Greedy++ using bit vectors, for $k = 10$, $k = 100$ and $k = 1000$ (we call the version using bit vectors bitGreedy++). The ratio is never smaller than 0.9 and bitGreedy++ is up to a factor 4 faster than Greedy++. The geometric means of the ratios are 1.1 for $k = 10$, 1.6 for $k = 100$ and 2.8 for $k = 1000$. On the other hand, the memory required by bitGreedy++ is usually much higher. On `com-amazon`, Greedy++ requires only about 312 MB, whereas bitGreedy++ needs 226 GB. For this reason, we were not able to test bitGreedy++ on the 5 largest networks of Table 55. To summarize, bitGreedy++ is mostly faster than Greedy++, and the improvement is more apparent for larger values of k. Thus, using bitGreedy++ is recommended if enough memory is available and k is relatively large (e.g. $k \geq 100$).

	Speedup of bitGreedy++ on Greedy++				
Graph	$k = 10$	$k = 100$	$k = 1000$	Mem. Greedy++	Mem. bitGreedy++
`ca-HepPh`	0.95	1.59	3.90	$\approx$ 136 MB	$\approx$ 210 MB
`email-Enron`	1.16	1.74	4.09	$\approx$ 151 MB	$\approx$ 603 MB
`CA-AstroPh`	0.90	1.48	3.08	$\approx$ 164 MB	$\approx$ 367 MB
`loc-brightkite`	0.97	1.39	3.00	$\approx$ 273 MB	$\approx$ 1 GB
`com-lj`	0.96	1.24	2.10	$\approx$ 318 MB	$\approx$ 78 GB
`com-amazon`	0.97	1.46	2.12	$\approx$ 312 MB	$\approx$ 226 GB
`gowalla-edges`	1.35	1.52	2.45	$\approx$ 279 MB	$\approx$ 17 GB
`com-dblp`	1.13	1.70	2.57	$\approx$ 310 MB	$\approx$ 94 GB
`flickrEdges`	1.60	2.04	2.73	$\approx$ 339 MB	$\approx$ 5 GB

Table 57: Performance of the new algorithm for group closeness using pruned SSSPs (Greedy++) and using bit vectors (bitGreedy++). The first three columns represent the speedup of bitGreedy++ on Greedy++ (i.e. the ratio between their running times). The last two columns report the memory requirements.

11.6.6 *Group closeness versus top-k closeness*

A natural question is how many elements of the group of nodes with highest closeness have high closeness or high degree individually. We investigate this on the networks of Table 55. In particular, for a given group size k, we compute the overlap (i. e.the size of the intersection) between the group returned by Greedy++ and the set of the top-k nodes with highest closeness (computed using the algorithm described in Chapter 9) and highest degree. The percentage overlap is then the overlap divided by k and multiplied by 100. Figure 36 shows the results. The plot on the bottom right corner shows the average over all networks of Table 55, whereas the other three plots refer to the `com-youtube` graph, to `soc-pokec-relationships` and to `com-orkut`, respectively. As it appears from the plots, the overlap changes significantly among the graphs. For the `com-youtube` graph, the percentage overlap decreases as the group size increases, and the overlap with Degree is always larger than the one with Top-k. Partially similar are the results for `soc-pokec-relationships`, although there is more fluctuation in the overlap of Degree and the initial overlap of Top-k is higher than it is for `com-youtube` ($\approx$ 80% vs. $\approx$ 60%). On the other hand, the results for `com-orkut` are quite different: The overlap with Degree increases with the group size, and is lower than the one with Top-k. On average, the overlap with both Degree and Top-k tends to decrease as the group size increases (as expected), with Degree having a higher overlap than Top-k (except for $k = 5$). Also, on average the overlap ranges between 30% and 60%. This clearly indicates that there is a dependence between the group with maximum closeness and the degrees of nodes and their centralities. However, the strength of this dependence varies significantly among the tested networks and suggests that picking the k nodes with highest closeness or highest degree is not always a good heuristic for finding the group with maximum closeness.

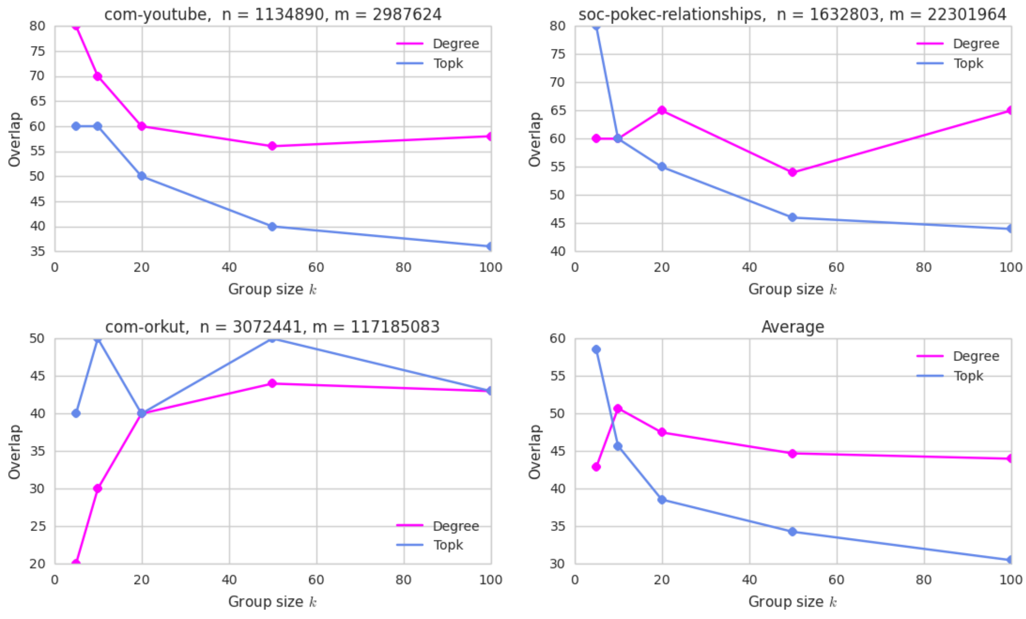


Figure 36: Percentage overlap between the group found by Greedy++ and the k nodes with highest closeness (Top-k) and between the group found by Greedy++ and the k nodes with highest degree (Degree).

BIBLIOGRAPHIC NOTES

The results presented in this chapter are joint work with Tanya Gonser and Henning Meyerhenke and have been accepted for publication at the *Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX 2018).*

CONCLUSIONS OF PART III

Closeness centrality indicates the inverse average shortest-path distance between one node and the other nodes of the network. Although computing it for all nodes requires all pair-wise distances, the closeness centrality of a single node can be obtained in linear time (in the number of edges) by running a Breadth-First Search (or Dijkstra's algorithm in weighted graphs). This is in contrast with betweenness centrality, for which the computation of a single nodes requires all shortest paths between existing pairs of nodes. In this part, we exploited this property of closeness centrality to quickly identify (and update) the most-central nodes or groups of nodes of a network.

In Chapter 9, we proposed upper bounds on the closeness values that efficiently allow us to skip the computation of closeness for the majority of nodes, allowing us to find the most-central nodes in a fraction of the time needed by the textbook algorithm. As an example, using our new approach we are able to find the top-10 nodes with highest closeness in the whole street network of north America (with 36 millions edges) in about one hour, where exhaustive computation would take years. Compared with the state of the art, our approach is always faster than the currently best algorithms for closeness centrality, both exact and approximate.

In Chapter 10, we presented fully-dynamic algorithms for top-k closeness centrality. Using properties of the modified graph, we are able to significantly reduce the number of operations required to update the most central nodes. As a result, we achieve high speedups on static recomputation, in line with those obtained by other dynamic algorithms for related problems (e.g. [58, 66] and the dynamic algorithms presented in Part II), confirming the fact that efforts in developing dynamic algorithms are well spent. Differently from most existing algorithms updating shortest-paths-based centralities, the techniques we propose use a linear (in the number of nodes) amount of additional memory. Although storing more information (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 nodes.

Future work includes extension to *batch updates*, where several edge updates occur at the same time, and for which the results presented in Chapter 7 are particularly encouraging. Also, finding methods for keeping track of the nodes with highest betweenness centrality (as well as other centrality measures) would be a very interesting research direction. Whereas computing betweenness of a single node cannot be done (much) more efficiently than computing it for all nodes, our results in Chapter 6 show that this is not true in the incremental case. Thus, it might be possible to devise dynamic algorithms that keep track of the k nodes with highest betweenness faster than updating betweenness of all nodes.

Finally, Chapter 11 presents a method for finding the group with maximum closeness in large complex networks. Our algorithm is the first that scales to networks with tens or hundreds of millions of edges *and* delivers a guaranteed approximation ratio of $(1 - 1/e)$ at the same time. Pruning the SSSP searches and exploiting the submodularity of the objective function allows us to reduce the amount of work done by the greedy algorithm proposed in [37] by orders of magnitude.

The algorithm presented in Chapter 9 has been made available to researchers and practitioners as part of the NetworKit tool suite [119]. The algorithms in Chapter 10 and Chapter 11 will be part of a future release.

Part IV

ESTIMATION OF ELECTRICAL CLOSENESS

12 ESTIMATION OF ELECTRICAL CLOSENESS

12.1 INTRODUCTION

In this chapter, we present two approximation algorithms for electrical closeness, a variant of closeness centrality based on *resistance distance* [33]. Differently from shortest-path distance, resistance distance takes all paths of the network into account, weighted by their length. Both our approximation algorithms are based on solving Laplacian linear systems. One algorithm uses sampling, whereas the other one builds on the Johnson-Lindenstrauss transform. According to our experimental study, our approximations are extremely accurate and are reasonably fast in scenarios where the centrality of a single node or a subset of nodes has to be computed. An exact computation of electrical closeness requires to invert the Laplacian of the graph, which takes $\Omega(n^2)$ time, being the inverse of the Laplacian in general a dense matrix. This would be prohibitive for networks millions of nodes and edges, even when the closeness of only a single node or of a subset of nodes is required. With this approach, in fact, computing the closeness of one node is just as expensive as computing it for all nodes. On the contrary, our approach can estimate the closeness of a single node very quickly: For example, it requires less than 2 minutes on a network with 50 millions edges.

Thanks to our approach, we can study for the first time the properties of electrical closeness in large networks with tens of millions of nodes. We compare electrical closeness with traditional shortest-path closeness and show that the former succeeds in differentiating nodes significantly better than the latter, and is also more resilient to noise. In addition, we study the correlation between centrality measures and degrees in real-world networks, in relation to a recent theoretical result for random geometric graphs [86]. Our experiments show that there is a strong correlation between degrees and electrical closeness in complex networks, whereas there is basically no correlation in street networks.

12.2 PRELIMINARIES: GRAPHS AS ELECTRICAL NETWORKS.

Throughout this chapter we consider connected undirected graphs $G = (V, E, w)$ having $n = |V|$ nodes and $m = |E|$ edges (disconnected graphs can be handled by treating each connected component separately). We recall that the Laplacian matrix $L = L(G)$ of G is defined as $L := D - A$, where $A = A(G)$ is the (weighted) adjacency matrix of G and $D = D(G)$ the diagonal matrix storing the (weighted) node degrees: $D_{ii} = \sum_{j=1}^{n} \omega_{ij}$.

One can regard a graph as an *electrical network* where each edge $\{u, v\}$ corresponds to a resistor with conductance ω_{uv} (the edge weight) or resistance $1/\omega_{uv}$. We can interpret the conductance as the ease with which an electrical current can flow through the edge. We can associate a *supply* $b : V \to \mathbb{R}$ with the electrical network, representing the nodes where current enters and leaves the network. A positive supply $b(v)$ means that current is entering the network from node v and a negative supply means that current is leaving the network. In the following, we will always assume that $\sum_{v \in V} b(v) = 0$ and that $b(s) = +1$ and $b(t) = -1$ for two nodes s and t, and that $b(w) = 0 \; \forall w \neq s, t$. In the following, we will refer to such a supply as vector $b_{st} \in \mathbb{R}^{n\times 1}$. We could interpret this as s and t being the two poles of a battery: this generates a current $e_{st} : \vec{E} \to \mathbb{R}$ flowing through the network. To each node v we can associate a *potential* $p_{st}(v)$ such that the vector $p_{st} \in \mathbb{R}^{n\times 1}$ satisfies the following linear system:

$$Lp_{st} = b_{st} \tag{30}$$

Then, the current flowing through edge (u, v) is defined as $(p_{st}(u) - p_{st}(v))/\omega_{st}$. Notice that, since G is connected, the rank of the Laplacian is $n - 1$ and there are infinitely many vectors p_{st} satisfying Eq. (30), each of them differing from the other by an additive constant. However, the current is well defined, since it depends on the difference between two potentials.

The difference $\rho(s, t)$ of potential between s and t is called *resistance distance* and, as the name indicates, it can be interpreted as an alternative distance measure between s and t. Notice that $\rho(s, t)$ is equal to the commute-time distance between s and t divided by the volume of G (i.e. the sum of the weights of the edges in G). The commute time between nodes u and v is defined as $H(u, v) + H(v, u)$, where the hitting time $H(x, y)$ is the expected time step in which a random walk in the graph starting in x reaches y for the first time. Thus, the commute time can be seen as the expected time a random walk needs for going from u to v and back again. Since the commute time is based on random walks, it depends on *all* the paths between two nodes (weighted by their length), and so does resistance distance.

12.3 RELATED WORK

12.3.1 *Solving Laplacian linear systems.*

We focus our description on iterative solvers due to their better time complexity on sparse graphs compared to direct solvers. Most iterative solvers reduce the norm of the residual $r = \|b - Ax\|$ iteratively by altering the current preliminary solution vector x in every iteration. One usually stops when the (relative) residual is below a certain tolerance τ, which yields a vector x' that is a good enough approximation to the actual solution x. While there are recent advances in theory to solve special linear systems including Laplacians in nearly-linear time [68, 117], those algorithms are not competitive in practice yet [26, 62]. In fact, the Conjugate Gradient (CG) algorithm outperforms the nearly-linear time algorithms in practice even though its asymptotic running time is typically higher.

A popular class of iterative algorithms to solve linear equations quickly in practice is called Algebraic Multigrid (AMG) [110]. The basic idea is to solve the actual linear system by iteratively solving coarser (i.e. smaller) yet similar systems and projecting the solutions of those back to the original system. AMG algorithms can be distinguished by the class of matrices they can handle and the way they construct the coarser systems. Two fast algorithms that are specifically designed for solving Laplacian systems are CMG by Koutis et al. [75] and LAMG by Livne and Brandt [84].

We decided to use LAMG as linear solver due to its particular design for complex networks. To this end, LAMG alternates between two stages called *elimination* and *aggregation* to construct the coarser systems. The former eliminates low degree nodes in the corresponding graph, the latter partitions nodes into *aggregates* based on a special affinity measure [84]. Both stages reduce the number of nodes and thus define a coarsening mechanism. Based on an extensive evaluation, Livne and Brandt state that running times of LAMG and CMG are comparable but LAMG tends to be more robust in the sense that CMG has large outliers on a small set of systems [84].

12.3.2 *Laplacian linear systems for network analysis.*

The connection between the graph Laplacian and electrical networks (see Section 12.2) has allowed for the solution of several graph algorithmic problems in terms of Laplacian systems. One of them is a centrality measure called *spanning edge centrality*, which indicates whether an edge is vital for the connectedness of a network [90]. The notion of importance for connectedness is also helpful for graph sparsification. A sparsification algorithm takes a dense graph and wants to find a sparser representation (= with fewer edges) with the same vertex set and similar properties [116], e. g. approximately the same cut sizes or eigenvalues. Since processes described by Laplacian linear systems can distinguish sparse from dense graph regions, edges in dense areas are, intuitively speaking, redundant and can be "sparsified" without doing much harm to the cut sizes when the weights of retained edges are properly scaled. This idiosyncrasy allows the use of processes described by Laplacian linear systems also for graph partitioning [92], approximate maximum network flow [40], and graph drawing [55, 93]. Moreover, the connection to electrical flow makes the use in dynamic load balancing of divisible tokens by diffusion [45] possible.

The interpretation of a graph as an electrical network has also led to the definition of two centrality measures based on current flow, electrical closeness and electrical betweenness [33]. Compared to traditional closeness and betweenness centrality, these two measures take *all* paths between two nodes into account and not only *shortest* paths.

12.4 ELECTRICAL CLOSENESS CENTRALITY

Closeness centrality measures the efficiency of a node in spreading information to the other nodes of the network. Formally, let $d(u, v)$ be the shortest-path distance between u and v (i.e. the length of the shortest path(s) between u and v). Then, the closeness centrality of node v is defined as the inverse of the expected shortest-path distance between v and and a random node w:

$$c_{\mathrm{C}}(v) := \frac{n-1}{\sum_{w \neq v} d(v, w)}. \tag{31}$$

The smaller the average distance between v and the other nodes, the higher is the closeness of v. To better understand the meaning of closeness, let us consider the two graphs in Figure 37. Since closeness takes only shortest-path distances into account, the closeness of node x_1 in the graph on the left and the score of node x_2 in the graph on the right will be exactly the same. However, there is only one path connecting x_1 to each of the other nodes. This means that if just a single edge is removed from the graph, x_1 will become disconnected from part of the other nodes. For example, let us assume the edges represent streets and x_1 is the location of an ambulance. If a congestion occurs, the ambulance in x_1 will not be able to reach part of the nodes (or it will take a very long time to reach them).

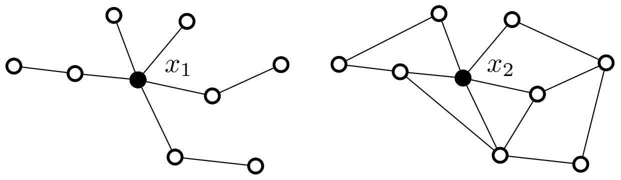


Figure 37: Shortest-path closeness centrality cannot distinguish between node x_1 and node x_2.

On the other hand, if the ambulance was in x_2, a congestion would limit only partially (or not at all) the ability of the ambulance to reach the nodes of the network.

The example above illustrates that traditional closeness is unable to model scenarios where the distance between two nodes does not only depend on the length of the shortest path between them, but also on the number of shortest or relatively short paths between the nodes. For this reason, a variant of closeness named *electrical closeness centrality* has been introduced [33]. If we see the graph as an electrical network, we can use *resistance distance* $\rho(u, v) := p_{uv}(u) - p_{uv}(v)$ as an alternative to shortest-path distance and define a slightly-modified version of closeness centrality [33]:

$$c_{\mathrm{EC}}(v) := \frac{n-1}{\sum\limits_{w \neq v} \rho(v, w)} = \frac{n-1}{\sum\limits_{w \neq v} p_{vw}(v) - p_{vw}(w)}. \tag{32}$$

By convention, we define $\rho(v, v) := 0 \; \forall v \in V$. To compute $c_{\mathrm{EC}}(v)$ of a node v, we can solve $n-1$ linear systems. Alternatively, we could invert the Laplacian matrix L of G (after omitting the row and column corresponding to a node, in order to get a regular matrix $\tilde{L}$), using the property that $p_{vw}(v) - p_{vw}(w) = \tilde{L}^{-1}_{vv} - 2\tilde{L}^{-1}_{vw} + \tilde{L}^{-1}_{ww}$ [33].

12.5 APPROXIMATING ELECTRICAL CLOSENESS

As outlined in [90], fast Laplacian linear solvers with a theoretical time complexity guarantee run in $O(m \log n \log(1/\tau))$ time, where τ is the tolerance of the solver. Multigrid methods such as CMG and LAMG are much faster in practice and have an *empirical* running time of $O(m \log(1/\tau))$.

Computing electrical closeness for only one node to the desired tolerance would already require the solution of $n-1$ linear systems, yielding $O(n^2 \log(1/\tau))$ time in practice assuming a sparse graph. This is infeasible for large networks with millions of nodes and edges. For this reason, we propose two approximation techniques for computing electrical closeness for a subset of the nodes in large graphs, and we compare them in our experimental evaluation. The first one is based on a simple sampling approach, which recalls the one used for classical closeness [50]. The second one uses the Johnson-Lindenstrauss transform (JLT), which allows to project the system into a lower-dimensional space by using $O(\log n)$ random vectors. The two approximations are different in nature and we were able to prove a theoretical guarantee on the quality of the approximation only for the second one. However, our experiments in Section 12.6 show that both approaches work very well in practice.

12.5.1 *Sampling-based approximation.*

The idea is to sample uniformly at random a set $S \subseteq V$ of nodes $S = \{s_1, ..., s_k\}$, which we call *pivots*. To approximate the electrical closeness of a node v, we compute the resistance distance $\rho(s, v)$ between all nodes $s \in S$ and v. Then, the closeness of v can be approximated as

$$\tilde{c}_{\mathrm{EC}}(v) := \frac{k}{n} \cdot \frac{n-1}{\sum_{i=1}^{k} \rho(v, s_i)}.$$

Proposition 12.5.1. *$\tilde{c}_{EC}(v)$ is un unbiased estimator for $c_{EC}(v)$ (i.e. $E[\tilde{c}_{EC}(v)] = c_{EC}(v)$).*

Proof. We show that $Y := \frac{n}{k} \sum_{s_i \in S} \rho(v, s_i)$ is an unbiased estimator for $s(v) = \sum_{w \in V} \rho(v, w)$, then the theorem follows directly from the properties of expected value. In the following, we denote the set of k-combinations of V with V_k.

$$\begin{aligned} E(Y) &= \sum_{S=\{s_1,...,s_k\} \in V_k} \frac{1}{\binom{n}{k}} \frac{n}{k} \sum_{s_i \in S} \rho(v, s_i) = \\ &= \frac{1}{\binom{n}{k}} \frac{n}{k} \sum_{w \in V} \binom{n-1}{k-1} \rho(v, w) \\ &= \sum_{w \in V} \rho(v, w). \qquad \square \end{aligned}$$

□

With k pivots, the empirical complexity of our approach is $O(km \log(1/\tau))$ with a multigrid solver. Our experiments in Section 12.6.1 show that a very small k (e.g., $k = 10$) is already enough to get a very good approximation.

12.5.2 *Projection-based approximation.*

Spielman and Srivastava [116] show how to compute an approximation of resistance distance based on the JLT. Let B be the $m \times n$ incidence matrix where each row corresponds to an edge of G and each node corresponds to a node such that, for edge $e = \{u, v\}$, $B(e, u) = +1$, $B(e, v) = -1$ and $B(e, w) = 0\ \forall w \neq u, v$ (since G is undirected, the direction of edge e can be chosen arbitrarily). Then, they show that the resistance distance between node u and node v can be re-written as $\rho(u, v) = ||W^{1/2} B L^{\dagger} (e_u - e_v)||_2^2$, where W is the diagonal $m \times m$ matrix such that $W(e, e) = \omega(e)$, $L^{\dagger}$ is the Moore-Penrose pseudoinverse [57] of L and e_u is the $n \times 1$ vector such that $e(u) = 1$ and equal to 0 everywhere else. The resistance distances can therefore be seen as pairwise distances between vectors in $\{W^{1/2} B L^{\dagger} e_u\}_{u \in V}$, which allows to apply the JLT: If we project the vectors into a lower-dimensional space spanned by $k = O(\log n)$ random vectors, the pairwise distances are approximately preserved. In other words, we can consider the pairwise distances between vectors in $\{Q W^{1/2} B L^{\dagger} e_u\}_{u \in V}$, where Q is a random projection matrix of size $k \times m$ with elements in $\{0, +\frac{1}{\sqrt{k}}, -\frac{1}{\sqrt{k}}\}$.

Since we do not want to compute $Q W^{1/2} B L^{\dagger}$ directly (it would require to compute the pseudoinverse $L^{\dagger}$ of L), we approximate it by solving k linear systems: for $i = 1, ..., k$, the

i-th row z_i^T of $QW^{1/2}BL^\dagger$ can be computed by solving the system $Lz_i = \{QW^{1/2}B\}_{\cdot,i}$, see Algorithm 26 (which we reuse from [116]). Note that the multiplication in Line 2 requires only $O(2m \log n)$ operations, since B is sparse (with $2m$ non-zero entries) and W is diagonal. When choosing k in Algorithm 26 equal to $O(\log n/\epsilon^2)$ for any $\epsilon > 0$, it was

Algorithm 26: Resistance distance approximation [116]

```
   Input  : $G = (V, E)$
   Output : Approx. $\tilde{\rho}(u, v)$ $\forall(u, v) \in V \times V$
1  Construct random matrix $Q$;
2  Compute $Y = QW^{1/2}B$;
3  $Z \leftarrow$ empty $k \times n$ matrix;
4  for $i = 1, ..., k$ do
5      solve the system $Lz_i = Y_{\cdot,i}$;
6      $Z_{i,\cdot} \leftarrow z_i^T$;
7  end
8  foreach $(u, v) \in V \times V$ do
9      $\tilde{\rho}(u, v) \leftarrow ||Z_{\cdot,u} - Z_{\cdot,v}||_2^2$;
10 end
11 return $\tilde{\rho}$
```

shown [116] that, with probability $\geq 1 - 1/n$,

$$(1-\epsilon)\rho(v, w) \leq \tilde{\rho}(v, w) \leq (1+\epsilon)\rho(v, w)$$

for all $(v, w) \in V \times V$. An approximation of electrical closeness for node v can therefore be computed as $\tilde{c}_{\mathrm{EC}}(v) := (n-1)/\sum_{w \neq v} \tilde{\rho}(v, w)$. This requires $O(2m \log n)$ for Line 2 and $O(km \log(1/\tau))$ empirical time for Lines 4 - 7. Then we need to compute $\tilde{\rho}(v, w)$ for all $w \neq w$, which requires $O(nk)$ operations. Assuming $n = O(m)$, the total (empirical) running time is $O(m(k \log(1/\tau) + log(n)))$.

If $(1-\epsilon)\rho(v, w) \leq \tilde{\rho}(v, w) \leq (1+\epsilon)\rho(v, w)$ for each $w \neq v$, then also $(1-\epsilon)c_{\mathrm{EC}}(v) \leq \tilde{c}_{\mathrm{EC}}(v) \leq (1+\epsilon)c_{\mathrm{EC}}(v)$. This is provably true only with probability $(1-1/n)^{n-1}$. However, our experimental results show that the approximation works well in practice: on all tested instances, $\tilde{c}_{\mathrm{EC}}$ is *always* within a $(1+\epsilon)$-factor from c_{EC} (see Section 12.6.1).

12.6 EXPERIMENTAL EVALUATION

In this section we evaluate the performance of the two approximation algorithms described in Section 12.5. First, we want to give some more details on the implementation, the benchmarking setup and the graph instances we used, before elaborating on the results of our evaluation.

IMPLEMENTATION. We implemented both approximation algorithms in NetworKit [119], the open-source tool for fast exploratory analysis of massive networks. For solving Laplacian systems, we rely on the NetworKit implementation of the Laplacian solver LAMG by Livne and Brandt [84]. When solving linear systems, in all our experiments we set the relative residual error τ to 10^{-5}.

Graph	Nodes	Edges	Description	Reference
PGP	10680	24316	PGP trust network	[9]
advogato	5272	42816	Advocato trust network	[77]
Drosophila_melanogaster	10424	40660	Interactome	[77]
Caenorhabditis_elegans	4428	9659	Metabolic network	[77]
CA-HepTh	8638	24806	Collaboration Network	[81]
HC-BIOGRID	4039	10321	Genetic interaction	[77]
hprd_pp	9219	36900	Human proteine interaction	[77]
Mus_musculus	3745	5170	Interactome	[77]
GoogleNw	15763	148585	Hyperlinks between web pages	[77]
Homo_sapiens	13478	61006	Metabolic network	[77]
oregon2_010526	11461	32730	AS peering network	[81]
as-caida20071105	26475	53381	CAIDA AS relationships	[81]

Table 58: Properties of smaller benchmark instances used.

BENCHMARKING SETUP. All experiments were done on a machine equipped with 256 GB RAM and a 2.7 GHz Intel Xeon CPU E5-2680 having 2 sockets with 8 cores each and hyperthreading enabled. The machine runs 64 bit SUSE Linux and we compiled our code with g++-4.8.1 and OpenMP 3.1.

INSTANCES. Tables 58 and 59 show the set of instances we use for our experiments. While Table 58 includes rather small complex networks with up to about 150 000 edges, Table 59 includes larger networks with up to 56 million edges. If a network has more than one connected component, we used the largest connected component (LCC). We ignore self-loops and the direction of edges in case a graph is directed. All the graphs are unweighted.

12.6.1 *Approximation algorithms.*

In this section we compare the two approximation algorithms described in Section 12.5.1 and Section 12.5.2, respectively. We refer to the first one as SAMPLING and to the second one as PROJECTION. SAMPLING depends on the number $|S|$ of samples, whereas PROJECTION depends on the dimension k of the $k \times n$ random projection matrix. For simplicity, we call *exact* the approach computing c_{EC} as in equation 32, solving $n - 1$ linear systems to the desired tolerance τ.

For our experiments, we select 100 nodes for each of the networks shown in Table 58. For each of these nodes, we compute electrical closeness exactly (to the desired tolerance τ) and the two approximations with different parameters. In particular, we set the number $|S|$ of samples of SAMPLING to 10, 20, 50, 100, 200, 500, and 1000. When running PROJECTION, we fix k to $\lceil \log n / \epsilon^2 \rceil$ and set ϵ equal to 0.5, 0.2, 0.1, and 0.05. To measure the accuracy of the algorithms, we use the well-known Spearmann rank correlation coefficient, which measures how close the ranking of nodes determined by the approximation algorithm is close to that of the exact algorithm. We recall that the closer the Spearmann coefficient is to 1, the more correlated are the two rankings, with 0 meaning no correlation and 1 meaning the two ranks are identical.

Graph	Nodes	Edges	Description	Reference
cit-Patents	3764117	16511740	Citation Network	[81]
com-Amazon	334863	925872	Amazon Product Network	[81]
com-DBLP	317080	1049866	Collaboration Network	[81]
com-Youtube	1134890	2987624	Youtube Social Network	[81]
hollywood-2009	1069126	56306653	Collaboration Network	[43]
com-LiveJournal	3997962	34681189	LiveJournal Social Network	[81]
Slashdot0902	82168	504230	Slashdot Zoo Social Network	[81]
soc-Epinions1	75877	405739	Epinions Social Network	[81]
roadNet-TX	1351137	1879201	Road Network of Texas	[81]
luxembourg.osm	114599	119666	Road Network of Luxembourg	[9]
belgium.osm	1441295	1549970	Road Network of Belgium	[9]
netherlands.osm	2216688	2441238	Road Network of the Netherlands	[9]
italy.osm	6686493	7013978	Road Network of Italy	[9]
great_britain.osm	7733822	8156517	Road Network of Great Britain	[9]
europe.osm	50912018	54054660	Road Network of Europe	[9]

Table 59: Properties of large benchmark instances used.

Figure 38 reports the accuracy (Spearmann coefficient) and the running times in seconds for each approximation algorithm and for each parameter. We do not report explicitly to which parameter each point in the plot corresponds to, but this can be easily deduced from the running times: a smaller sample size corresponds to a smaller running time for Sampling and a larger ϵ corresponds to a smaller running time for Projection. Figure 38 reports, for each approximation algorithm and for each parameter, the average over all networks of Table 58 of time and Spearmann coefficient.

The results are quite self-explanatory: the Sampling approach clearly outperforms Projection and its accuracy is extremely high already with only 10 samples. We also compute for each algorithm and parameter the number of rank inversions, i.e. the number of node pairs $\{u, v\}$ for which the approximated closeness of u is smaller than the approximated closeness of v, but the exact closeness of u is larger than or equal to the exact one of v (or vice versa). With ten pivots, the average number of rank inversions of Sampling is 12.5; it is always below 10 for higher number of samples. This means that, out of $\binom{100}{2} = 4950$ pairs, less than 10 are inverted, corresponding to 0.2%.

In addition to accuracy in terms of ranks, we also evaluate the maximum relative error. We define the relative error for a node v as $e(v) = \max\{r(v), 1/r(v)\}$, where $r(v)$ is the ratio between the exact electrical closeness of v and its approximation. The maximum relative error is then defined as $\max_{v \in V} e(v)$. Figure 39 reports the results. It is interesting to notice that, with respect to this measure, the two algorithms behave quite similarly. Also, notice that the maximum relative error for Projection is always smaller than ϵ (we recall the values of ϵ used are 0.05, 0.1, 0.2 and 0.5), although we can only prove that this is true with probability at least $(1 - 1/n)^{n-1}$.

To summarize, our results show that both algorithms lead to very good accuracy in terms of maximum relative error, whereas the sampling approach better preserves the ranking of nodes, even when the number of samples is very small. For this reason, in our experiments on large graphs, we make use of the sampling approach. On average (over the

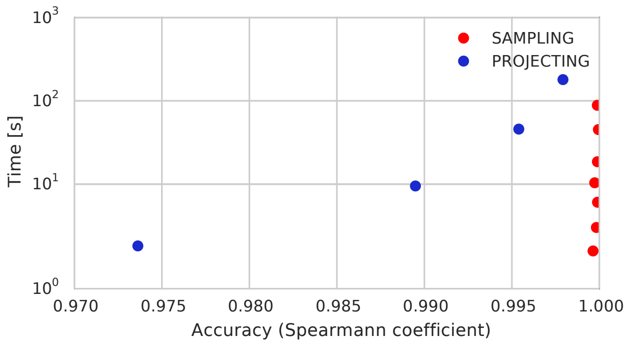


Figure 38: Time vs. Spearmann coefficient for the two approximation algorithms, using different parameters. The points represent the average among the networks of Table 58.

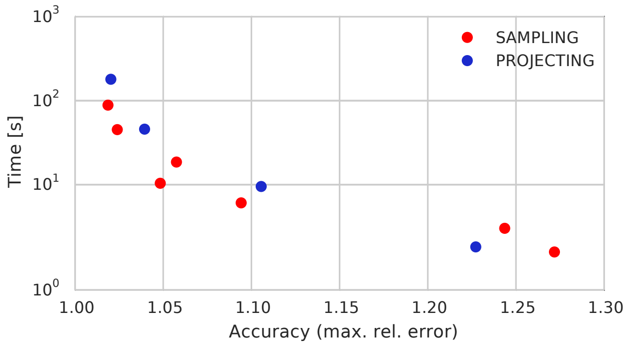


Figure 39: Time vs. maximum relative error for the two approximation algorithms, using different parameters. The points represent the average among the networks of Table 58.

instances of Table 58), computing c_{EC} on 100 nodes takes more than 20 minutes, whereas using SAMPLING with 20 pivots takes only 2.87 seconds. Table 60 in the Appedix shows the detailed running times.

12.6.2 *Comparison with shortest-path closeness.*

As explained in Section 12.4, our intuition is that electrical closeness should represent the efficiency of a node reaching the other nodes of the network better than shortest-path closeness. To verify this assumption, we first compare the two measures in terms of their capability to discriminate between different nodes. In this experiments, we use the networks of Table 58 and compute (exactly) electrical and shortest-path closeness on 100 randomly chosen nodes. Figure 40 shows the relative standard deviation for shortest-path and electrical closeness. The relative standard deviation is defined as the standard deviation divided by the average. It is always significantly higher for electrical closeness than it is for shortest-path closeness, meaning that there is much more variation in the scores computed by the former.

Also, similarly to what has been done in [90] for spanning edge centrality and edge betweenness centrality, we measure the resilience to noise, in this case for electrical closeness and shortest-path closeness. The idea is to add edges to the graph and see how well the

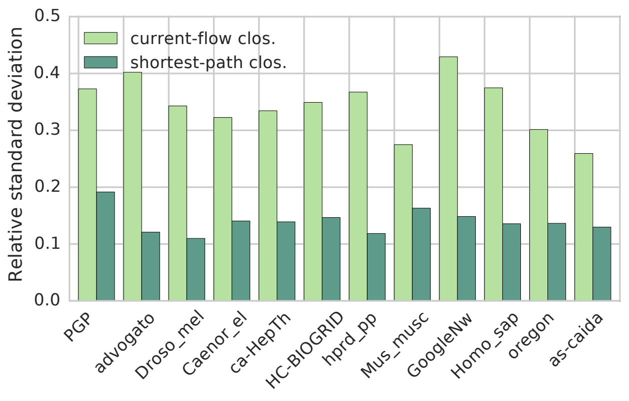


Figure 40: Relative standard deviation for shortest-path and electrical closeness.

initial rankings are preserved. Our intuition is that, if we add some edge that creates a shortcut between a node v and some other nodes, the shortest-path closeness of v will be more affected than its electrical closeness, since the former takes only shortest paths into account. This is confirmed by our experiments, summarized in Figure 41. For each network in Table 58, we insert a percentage of the total number of edges varying from 1% to 10%. To have a high number of shortcuts involving the sampled nodes, we always add edges between one of the sampled nodes and other nodes of the graph. Figure 41 shows, for each percentage of inserted edges, the average among all tested networks of the Spearmann correlation coefficient between the initial ranking and the ranking after the insertions. Figure 41 shows that electrical closeness is more resilient to edge insertions and the difference between the resilience of the two measures increases the more the graph changes.

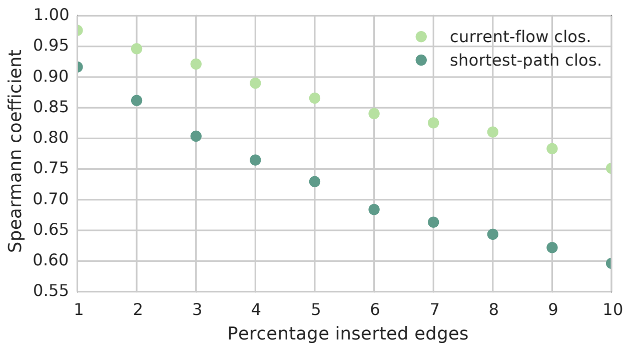


Figure 41: Resilience to noise for different percentages of inserted edges. The points represent the average among the networks of Table 58.

12.6.3 *Correlation with degree.*

In certain random geometric graph models, such as ϵ-graphs, kNN graphs, and Gaussian similarity graphs, it was recently shown [86] that the resistance distance between two nodes u and v converges to $1/\deg(u) + 1/\deg(v)$ when the number of nodes goes to infinity.

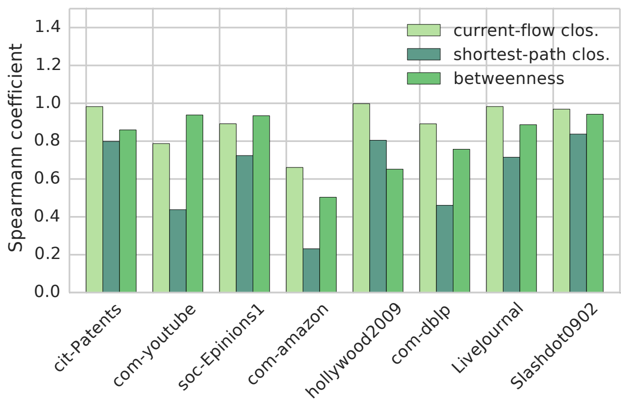


Figure 42: Correlation with c_A for complex networks.

This result also has implications on electrical closeness: when the number of nodes goes to infinity in such graphs, $c_{EC}(v)$ goes to $c_A(v) := (n-1)/\sum_{w \neq v}(1/\deg(v) + 1/\deg(w))$. However, the structure of real-world networks (e.g. complex or street networks) is significantly different from random graphs and it is not clear how close electrical closeness is to this asymptotic value in reality. In this section we therefore study the correlation between electrical closeness, as well as shortest-path closeness and betweenness, with c_A. In our experiments, we consider large networks with up to 56 millions nodes and edges and we use the sampling approach to approximate electrical closeness (with 20 pivots). Table 61 shows the running times of our approximation when computing the closeness of a single node. We approximate betweenness using the approach presented in [56], which is already implemented in NetworKit. Since the results are very different for street and complex networks, we study them separately.

Figure 42 shows the Spearmann coefficient computed between the three centrality measures and c_A on the complex networks of Table 59. The results show that there is in fact a strong positive correlation. This is weaker for shortest-path closeness, with an average Spearmann coefficient of 0.63 and stronger for betweenness and electrical closeness, with an average of 0.81 and 0.89, respectively. We obtain similar results on the smaller instances of Table 58, where the averages are 0.63, 0.85 and 0.89 for shortest-path closeness, betweenness and electrical closeness, respectively. The results are very different for street networks (Figure 43). Here the correlation with the degrees is generally very low and sometimes even negative, with an average of -0.02 for closeness, 0.35 for betweenness and 0.07 for electrical closeness.

While c_A and c_{EC} are unrelated on street networks, our results show that there is actually a strong correlation between them in complex networks. This behavior is likely due to the different type of degree distributions in the two network classes. While complex networks usually feature a skewed degree distribution with many small, but also some high-degree nodes, the degrees in street networks are closely concentrated. Consequently, in some applications where a very good accuracy is not needed, c_A might be used as an approximation of c_{EC} in complex networks. The same thing can be said for betweenness, which is only slightly less correlated with c_A than c_{EC}. This is very convenient, since c_A can be computed in $O(m)$ time. However, our results in Section 12.6.1 show that our sampling-based approach can compute an extremely accurate approximation in time

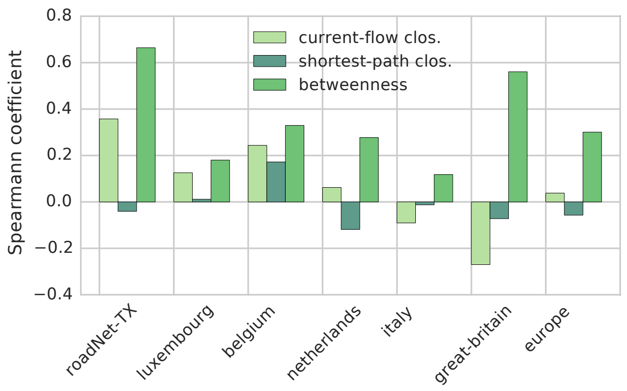


Figure 43: Correlation with c_A for street networks.

Graph	Time exact [s]	Time SAMP. 20 [s]	Spearmann coeff.	Rank Inver.
PGP	558.97	1.68	0.99990	0.14%
advogato	383.42	2.39	0.99986	0.16%
Drosophila_melanogaster	1077.78	3.50	0.99986	0.12%
Caenorhabditis_elegans	68.50	0.64	0.99975	0.28%
CA-HepTh	800.65	2.87	0.99989	0.14%
HC-BIOGRID	186.47	1.92	0.99975	0.28%
hprd_pp	988.58	4.01	0.99990	0.10%
Mus_musculus	33.66	0.33	0.99958	0.44%
GoogleNw	4612.19	8.16	0.99987	0.14%
Homo_sapiens	1913.06	4.90	0.99999	0.02%
oregon2_010526	640.97	1.49	0.99988	0.14%
as-caida20071105	3354.62	2.62	0.99990	0.12%

Table 60: Comparison between exact (= within desired tolerance τ) and SAMPLING approach, with 20 pivots. The first two columns report the running times of the two approaches, when computing electrical closeness on 100 nodes. The third column reports the Spearmann rank correlation coefficient between the approaches and the fourth the percentage of rank inversions.

$O(km \log(1/\epsilon))$ even when the number k of samples is very small. For this reason, we believe the sampling approach is probably the best option for most applications.

12.7 CONCLUSIONS

In this chapter, we provided two algorithms SAMPLING and PROJECTION for electrical closeness, both based on solving Laplacian linear systems. Thanks to them and to the fast NetworKit implementation of the LAMG solver, we have computed electrical closeness centrality and provided the first published results on its behavior on large real-world networks. Our algorithms lead to very accurate results and using them we are now able to compute an estimation of electrical closeness of a subset of nodes on networks with tens of millions of nodes and edges within a few seconds or minutes.

Graph	Time approximation [s]
cit-Patents	125.99
com-youtube	5.06
soc-Epinions1	0.28
com-amazon	2.56
hollywood2009	107.52
com-dblp	1.90
LiveJournal	287.27
Slashdot0902	0.41
roadNet-TX	6.28
luxembourg	0.13
belgium	2.39
netherlands	5.33
italy	10.91
great-britain	12.72
europe	103.34

Table 61: Running time of SAMPLING with 20 pivots when computing c_{EC} of a single node.

In our experiments electrical closeness alleviates two known problems of shortest-path closeness and can thus be seen as a viable alternative in many scenarios. We have also shown empirically that there is a strong correlation between degrees and both electrical closeness and betweenness centrality in complex networks, whereas the degree and electrical closeness are basically unrelated in street networks.

Our approach based on SAMPLING or PROJECTION is very fast in scenarios where one only needs to compute the electrical closeness for a subset of nodes. However, it might become too expensive if closeness has to be computed for all nodes. In these scenarios an interesting aspect of future work is whether our approximation would still be the best approach or whether inverting the Laplacian of the matrix would be faster in this case.

BIBLIOGRAPHIC NOTES

The results presented in this chapter have been published as “Estimating current-flow closeness centrality with a multigrid laplacian solver” (coauthored with Michael Wegner, Dimitar Lukarski, and Henning Meyerhenke) at the *Seventh SIAM Workshop on Combinatorial Scientific Computing* (CSC 2016).

Part V

CONCLUSION

CONCLUSION

Finding the most central nodes is a major task in network analysis. In this work, we presented several algorithms for computing centrality efficiently, both on static networks and on networks that evolve over time.

In Chapters 5, 6, 7 and 10, we presented dynamic algorithms that update betweenness centrality and the k nodes with highest closeness after a change occurs to the graph. As suggested from our experiments in Chapter 3, most edge updates only affect (i.e., increase or decrease the distance between) a small fraction of node pairs in complex networks. Thus, a simple idea common to all our dynamic algorithms is to first identify the affected pairs and limit the SSSP searches on the new graph to them. Another common general idea is that of identifying properties of the SSSP subtree (or sub-DAG) of a given node and prune the search based on these properties.

To this aim, storing information on the graph before the update has turned out to be extremely beneficial, since it allows to skip unnecessary work. On the other hand, the choice of which information should be stored has to be made carefully, based on the specific problem and targeted applications. Indeed, a large memory footprint can be a major limitation for the scalability of a dynamic algorithm. For example, storing all pairwise distances requires a quadratic amount of additional memory, meaning that networks with millions of edges would be out of reach on most workstations, regardless of how fast the dynamic algorithm is.

This leaves the algorithm developer with a tradeoff between running time and memory consumption. In contexts where the running time of the static algorithm on the initial graph is already a bottleneck for scalability, it might be reasonable to allocate additional memory to speedup the running time of the updates. This was the case for the exact betweenness algorithms in Chapter 5 and Chapter 6, where networks with millions of edges would be out of reach in any case, due to the quadratic running time of the static algorithm. On the contrary, the static top-k closeness algorithms proposed in Chapter 9 can target networks with hundreds of millions of edges, meaning that a dynamic algorithm with quadratic memory footprint in this context would be a major limitation for scalability. For this reason the dynamic top-k closeness algorithms presented in Chapter 10 were designed to use only a linear (in the number of nodes) amount of additional memory. Although storing additional information would probably lead to even higher speedups, the algorithms are orders of magnitude faster than recomputation on many instances.

The approaches in Chapter 9 and Chapter 11 target static networks, but scale up existing algorithms by pruning unnecessary work (without changing the returned results). In both cases, computing fast and accurate upper bounds on quantities of interest (i.e., closeness in the first case, and marginal gain of a node with respect to a set in the second case) is crucial. Also, we point out once again that the top-k closeness algorithm proposed in Chapter 9 builds on the fact that the closeness centrality of a single node can be computed quickly (in linear time in the size of the graph) by running a SSSP from the node. This is not the case for other centrality measures, such as betweenness, for which at the moment no algorithm for a single node exists that is significantly faster than computing all pairwise distances. Although achieving a better worst-case complexity for this problem seems unlikely, it might be possible to devise algorithms that perform better on real-world instances. Such a result would not only be interesting per se (there might be applications which require the betweenness of a single node, see for example the MBI problem described in Chapter 6), but also because it might allow to develop fast algorithms for computing the k nodes with highest betweenness. As for electrical closeness, the approximation algorithms proposed in Chapter 12 make it possible to compute an approximation for a single node on networks with tens of millions of nodes and edges in a matter of seconds or at most minutes. Although this is an approximation, our experimental results show that the returned ranking is very close to the exact one. Thus, the methods in Chapter 12 could be used as a building block to devise an algorithm for finding (an approximation of) the k nodes with highest electrical closeness. The techniques used for top-k closeness in Chapter 9 would not directly apply to this context, as they all rely on properties of shortest paths. However, it might be possible to devise upper bounds on electrical closeness based on properties of resistance distance.

BIBLIOGRAPHY

[1] Amir Abboud, Fabrizio Grandoni, and Virginia V. Williams. “Subcubic equivalences between graph centrality problems, APSP and diameter.” In: *Proceedings of the 26th ACM/SIAM Symposium on Discrete Algorithms (SODA)*. 2015, pp. 1681–1697.

[2] Amir Abboud and Virginia V. Williams. “Popular conjectures imply strong lower bounds for dynamic problems.” In: *Proceedings of the 55th Annual Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 434–443.

[3] Amir Abboud, Virginia V. Williams, and Joshua Wang. “Approximation and Fixed Parameter Subquadratic Algorithms for Radius and Diameter.” In: *Proceedings of the 27th ACM/SIAM Symposium on Discrete Algorithms (SODA)*. 2016, pp. 377–391.

[4] Amir Abboud, Virginia V. Williams, and Oren Weimann. “Consequences of Faster Alignment of Sequences.” In: *Proceedings of the 41st International Colloquium on Automata, Languages and Programming (ICALP)*. 2014, pp. 39–51.

[5] Guilherme Ferraz de Arruda, André Luiz Barbieri, Pablo Martín Rodriguez, Yamir Moreno, Luciano da Fontoura Costa, and Francisco Aparecido Rodrigues. “The role of centrality for the identification of influential spreaders in complex networks.” In: *CoRR* abs/1404.4528 (2014).

[6] Pasquale Avella, Maurizio Boccia, Saverio Salerno, and Igor Vasilyev. “An aggregation heuristic for large scale p-median problem.” In: *Computers & OR* 39.7 (2012), pp. 1625–1632.

[7] Lars Backstrom and Jon M. Kleinberg. “Romantic partnerships and the dispersion of social ties: a network analysis of relationship status on facebook.” In: *Proceedings of the 17th ACM conference on Computer Supported Cooperative Work (CSCW)*. 2014, pp. 831–841.

[8] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. “Approximating Betweenness Centrality.” In: *Proceedings of the Workshop on Algorithms and Models for the Web Graph (WAW)*. 2007, pp. 124–137.

[9] David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. “Benchmarking for Graph Clustering and Partitioning.” In: *Encyclopedia of Social Network Analysis and Mining*. Springer, 2014, pp. 73–82.

[10] Reinhard Bauer and Dorothea Wagner. “Batch Dynamic Single-Source Shortest-Path Algorithms: An Experimental Study.” In: *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA)*. 2009, pp. 51–62.

[11] Alex Bavelas. “A mathematical model of Group Structure.” In: *Human Organizations* 7 (1948), pp. 16–30.

[12] Alex Bavelas. “Communication patterns in task-oriented groups.” In: *Journal of the Acoustical Society of America* 22 (1950), pp. 725–730.

[13] David C. Bell, John S. Atkinson, and Jerry W. Carlson. “Centrality measures for disease transmission networks.” In: *Social Networks* 21.1 (1999), pp. 1–21.

[14] Elisabetta Bergamini, Tanya Gonser, and Henning Meyerhenke. “Scaling up Group Closeness Maximization.” In: *Accepted at the 20th Workshop on Algorithm Engineering and Experiments (ALENEX).* To appear. 2018.

[15] Elisabetta Bergamini and Henning Meyerhenke. “Computing Top-k Closeness Centrality Faster in Unweighted Graphs.” In: *Karlsruhe Reports in Informatics* (2015).

[16] Elisabetta Bergamini and Henning Meyerhenke. “Fully-Dynamic Approximation of Betweenness Centrality.” In: *Proceedings of the 23rd Annual European Symposium on Algorithms, (ESA).* 2015, pp. 155–166.

[17] Elisabetta Bergamini and Henning Meyerhenke. “Approximating Betweenness Centrality in Fully Dynamic Networks.” In: *Internet Mathematics* 12.5 (2016), pp. 281–314.

[18] Elisabetta Bergamini, Henning Meyerhenke, and Christian Staudt. “Approximating Betweenness Centrality in Large Evolving Networks.” In: *Proceedings of the 17th Workshop on Algorithm Engineering and Experiments, (ALENEX).* 2015, pp. 133–146.

[19] Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke. “Computing Top-k Closeness Centrality Faster in Unweighted Graphs.” In: *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments, (ALENEX).* 2016, pp. 68–80.

[20] Elisabetta Bergamini, Michael Wegner, Dimitar Lukarski, and Henning Meyerhenke. “Estimating Current-Flow Closeness Centrality with a Multigrid Laplacian Solver.” In: *Proceedings of the 7th SIAM Workshop on Combinatorial Scientific Computing, (CSC).* 2016, pp. 1–12.

[21] Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke. “Computing top-k Closeness Centrality Faster in Unweighted Graphs.” In: *CoRR* abs/1704.01077 (2017).

[22] Elisabetta Bergamini, Henning Meyerhenke, Mark Ortmann, and Arie Slobbe. “Faster Betweenness Centrality Updates in Evolving Networks.” In: *Proceedings of the 16th International Symposium on Experimental Algorithms, (SEA).* 2017, 23:1–23:16.

[23] Elisabetta Bergamini, Pierluigi Crescenzi, Gianlorenzo D’Angelo, Henning Meyerhenke, Lorenzo Severini, and Yllka Velaj. “Improving the betweenness centrality of a node by adding links.” In: *CoRR* abs/1702.05284 (2017).

[24] Patrick Bisenius, Elisabetta Bergamini, Eugenio Angriman, and Henning Meyerhenke. “Computing Top-k Closeness Centrality in Fully-dynamic Graphs.” In: *Accepted at the 20th Workshop on Algorithm Engineering and Experiments (ALENEX).* To appear. 2018.

[25] Paolo Boldi and Sebastiano Vigna. “Axioms for Centrality.” In: *Internet Mathematics* 10.3-4 (2014), pp. 222–262.

[26] Erik G. Boman, Kevin Deweese, and John R. Gilbert. “Evaluating the Dual Randomized Kaczmarz Laplacian Linear Solver.” In: *Informatica* 40 (2016), pp. 95–107.

[27] Michele Borassi. “A Note on the Complexity of Computing the Number of Reachable Vertices in a Digraph.” In: *CoRR* abs/1602.02129 (2016). arXiv: 1602.02129.

[28] Michele Borassi, Pierluigi Crescenzi, and Michel Habib. “Into the square - On the complexity of some quadratic-time solvable problems.” In: *Proceedings of the 16th Italian Conference on Theoretical Computer Science (ICTCS)*. 2015, pp. 1–17.

[29] Michele Borassi, Pierluigi Crescenzi, and Andrea Marino. *Fast and Simple Computation of Top-k Closeness Centralities*. http://arxiv.org/abs/1507.01490. 2015.

[30] Michele Borassi and Emanuele Natale. “KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation.” In: *Proceedings of the 24th Annual European Symposium on Algorithms, (ESA)*. 2016, 20:1–20:18.

[31] Ulrik Brandes. “A faster algorithm for betweenness centrality.” In: *Journal of Mathematical Sociology* 25 (2001), pp. 163–177.

[32] Ulrik Brandes and Thomas Erlebach. *Network Analysis: Methodological Foundations (Lecture Notes in Computer Science)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[33] Ulrik Brandes and Daniel Fleischer. “Centrality Measures Based on Current Flow.” In: *Proceedings of the 22nd Annual Symposium on Theoretical Aspects of Computer Science (STACS)*. 2005, pp. 533–544.

[34] Ulrik Brandes and Christian Pich. “Centrality Estimation in Large Networks.” In: *International Journal on Bifurcation and Chaos* 17.7 (2007), pp. 2303–2318.

[35] Shiri Chechik, Edith Cohen, and Haim Kaplan. “Average Distance Queries through Weighted Samples in Graphs and Metric Spaces: High Scalability with Tight Statistical Guarantees.” In: *Proceedings of the 2015 International Workshop on Approximation, Randomization, and Combinatorial Optimization (APPROX) and International Workshop on Algorithms and Techniques (RANDOM)*. Vol. 40. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 659–679.

[36] Mostafa Haghir Chehreghani. “An Efficient Algorithm for Approximate Betweenness Centrality Computation.” In: *The Computer Journal* 57.9 (2014), pp. 1371–1382.

[37] Chen Chen, Wei Wang, and Xiaoyang Wang. “Efficient Maximum Closeness Centrality Group Identification.” In: *Proceedings of the 27th Australasian Database Conference (ADC)*. 2016, pp. 43–55.

[38] Duanbing Chen, Linyuan Lu, Ming-Sheng Shang, Yi-Cheng Zhang, and Tao Zhou. “Identifying influential nodes in complex networks.” In: *Physica A: Statistical Mechanics and its Applications* 391.4 (2012), pp. 1777–1787.

[39] Nicholas A. Christakis and James H. Fowler. “The spread of obesity in a large social network over 32 years.” In: *The New England Journal of Medicine* 357.4 (2007), pp. 370–379.

[40] Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. “Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs.” In: *Proceedings of the 43rd ACM Symposium on Theory of Computing (STOC)*. 2011, pp. 273–282.

[41] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F. Werneck. “Computing classic closeness centrality, at scale.” In: *Proceedings of the 2nd ACM conference on Online social networks (COSN).* 2014, pp. 37–50.

[42] Pierluigi Crescenzi, Gianlorenzo D’Angelo, Lorenzo Severini, and Yllka Velaj. “Greedily Improving Our Own Centrality in A Network.” In: *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA).* 2015, pp. 43–55.

[43] Timothy A. Davis and Yifan Hu. “The University of Florida Sparse Matrix Collection.” In: *ACM Transactions on Mathematical Software* 38.1 (2011), 1:1–1:25.

[44] Camil Demetrescu and Giuseppe F. Italiano. “A new approach to dynamic all pairs shortest paths.” In: *Journal of the ACM* 51.6 (2004), pp. 968–992.

[45] R. Diekmann, A. Frommer, and B. Monien. “Efficient schemes for nearest neighbor load balancing.” In: *Parallel Computing* 25.7 (1999), pp. 789–812.

[46] Sergei N Dorogovtsev and José FF Mendes. *Evolution of networks: From biological nets to the Internet and WWW.* Oxford University Press, 2003.

[47] Zvi Drezner. *Facility Location.* Springer, 1995.

[48] Jean-Guillaume Dumas and Victor Pan. “Fast Matrix Multiplication and Symbolic Computation.” In: *CoRR* abs/1612.05766 (2016).

[49] David Ediger, Jason Riedy, David A. Bader, and Henning Meyerhenke. “Computational Graph Analytics for Massive Streaming Data.” In: *Large Scale Network-Centric Distributed Systems.* John Wiley & Sons, Inc., 2013, pp. 619–648.

[50] David Eppstein and Joseph Wang. “Fast Approximation of Centrality.” In: *Journal of Graph Algorithms and Applications* (2004), pp. 39–45.

[51] Dóra Erdös, Vatche Ishakian, Azer Bestavros, and Evimaria Terzi. “A Divide-and-Conquer Algorithm for Betweenness Centrality.” In: *Proceedings of the 2015 SIAM International Conference on Data Mining (SDM).* 2015, pp. 433–441.

[52] Martin G. Everett and Stephen P. Borgatti. “The centrality of groups and classes.” In: *The Journal of Mathematical Sociology* 23.3 (1999), pp. 181–201.

[53] Linton C. Freeman. “Centrality in social networks: Conceptual clarification.” In: *Social Networks* 1.3 (1979), pp. 215–239.

[54] François Le Gall. “Powers of tensors and fast matrix multiplication.” In: *Proceedings of the 2014 International Symposium on Symbolic and Algebraic Computation (ISSAC).* 2014, pp. 296–303.

[55] Emden R. Gansner, Yifan Hu, and Stephen C. North. “A Maxent-Stress Model for Graph Layout.” In: *IEEE Transactions on Visualization and Computer Graphics* 19.6 (2013), pp. 927–940.

[56] Robert Geisberger, Peter Sanders, and Dominik Schultes. “Better Approximation of Betweenness Centrality.” In: *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX).* 2008, pp. 90–100.

[57] Gene H. Golub and Charles F. Van Loan. *Matrix computations.* Johns Hopkins University Press, 1996.

[58] Oded Green, Robert McColl, and David A. Bader. “A Fast Algorithm for Streaming Betweenness Centrality.” In: *Proceedings of the 2012 International Conference on Privacy, Security, Risk and Trust (PASSAT) and 2012 International Conference on Social Computing (SocialCom)*. 2012, pp. 11–20.

[59] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. “Exploring network structure, dynamics, and function using NetworkX.” In: *Proceedings of the 7th Python in Science Conference (SCIPY)*. 2008, pp. 11–15.

[60] S. Louis Hakimi. “Optimum distribution of switching centers in a communication network and some related graph theoretic problems.” In: *Operations Research* 13.3 (1965), pp. 462–475.

[61] Takanori Hayashi, Takuya Akiba, and Yuichi Yoshida. “Fully Dynamic Betweenness Centrality Maintenance on Massive Networks.” In: *Proceedings of the 41st International Conference on Very Large Data Bases (PVLDB)*. Vol. 9. 2. 2015, pp. 48–59.

[62] Daniel Hoske, Dimitar Lukarski, Henning Meyerhenke, and Michael Wegner. “Is Nearly-linear the Same in Theory and Practice? A Case Study with a Combinatorial Laplacian Solver.” In: *Proceedings of 14th International Symposium on Experimental Algorithms (SEA)*. Vol. 9125. Springer. 2015, p. 205.

[63] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. “Which Problems Have Strongly Exponential Complexity?” In: *Journal of Computer and System Sciences* 63.4 (Dec. 2001), pp. 512–530.

[64] Chandra Ade Irawan and Saïd Salhi. “Solving large p-median problems by a multistage hybrid approach using demand points aggregation and variable neighbourhood search.” In: *Journal of Global Optimization* 63.3 (2015), pp. 537–554.

[65] U Kang, Spiros Papadimitriou, Jimeng Sun, and Tong Hanghang. “Centralities in large networks: Algorithms and observations.” In: *Proceedings of the SIAM International Conference on Data Mining (SDM)*. 2011, pp. 119–130.

[66] Miray Kas, Kathleen M. Carley, and L. Richard Carley. “Incremental closeness centrality for dynamically changing social networks.” In: *Proceedings of the 5th International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. 2013, pp. 1250–1258.

[67] Miray Kas, Matthew Wachs, Kathleen M. Carley, and L. Richard Carley. “Incremental algorithm for updating betweenness centrality in dynamically growing networks.” In: *Proceedings of the 5th International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. 2013, pp. 33–40.

[68] Jonathan A. Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. “A Simple, Combinatorial Algorithm for Solving SDD Systems in Nearly-linear Time.” In: *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*. Palo Alto, California, USA, 2013, pp. 911–920.

[69] David Kempe, Jon M. Kleinberg, and Éva Tardos. “Maximizing the spread of influence through a social network.” In: *Proceedings of the 9thACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2003, pp. 137–146.

[70] Christine Kiss and Martin Bichler. "Identification of influencers – Measuring influence in customer networks." In: *Decision Support Systems* 46.1 (2008), pp. 233 –253.

[71] Dirk Koschützki, Katharina Anna Lehmann, Leon Peeters, Stefan Richter, Dagmar Tenfelde-Podehl, and Oliver Zlotowski. "Centrality Indices." English. In: *Network Analysis*. Vol. 3418. LNCS. Springer Berlin Heidelberg, 2005, pp. 16–61.

[72] Nicolas Kourtellis, Gianmarco De Francisci Morales, and Francesco Bonchi. "Scalable Online Betweenness Centrality in Evolving Graphs." In: *IEEE Transactions on Knowledge and Data Engineering* PP.99 (2015), pp. 1–1.

[73] Ioannis Koutis, Gary L. Miller, and Richard Peng. "Approaching Optimality for Solving SDD Linear Systems." In: *Proceedings of the 51th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2010, pp. 235–244.

[74] Ioannis Koutis, Gary L. Miller, and Richard Peng. "A Nearly-m log n Time Solver for SDD Linear Systems." In: *Proceedings of the 52nd IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. 2011, pp. 590–598.

[75] Ioannis Koutis, Gary L. Miller, and David Tolliver. "Combinatorial Preconditioners and Multilevel Solvers for Problems in Computer Vision and Image Processing." In: *Computer Vision and Image Understanding* 115.12 (2011), pp. 1638–1646.

[76] Jérôme Kunegis. "KONECT: the Koblenz network collection." In: *Proceedings of the 22nd International World Wide Web Conference (WWW)*. 2013, pp. 1343–1350.

[77] *LASAGNE Network Dataset*. http://lasagne-unifi.sourceforge.net.

[78] Erwan Le Merrer, Nicolas Le Scouarnec, and Gilles Trédan. "Heuristical Top-k: Fast Estimation of Centralities in Complex Networks." In: *Information Processing Letters* 114 (2014), pp. 432–436.

[79] Min-Joong Lee, Sunghee Choi, and Chin-Wan Chung. "Efficient algorithms for updating betweenness centrality in fully dynamic graphs." In: *Information Sciences* 326 (2016), pp. 278–296.

[80] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. "Graph evolution: Densification and shrinking diameters." In: *TKDD* 1.1 (2007), p. 2.

[81] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. 2014.

[82] Chih-Chung Lin and Ruei-Chuan Chang. "On the Dynamic Shortest Path Problem." In: *Journal of Information Processing* 13.4 (1991), pp. 470–476.

[83] Nan Lin. *Foundations of social research*. McGraw-Hill, 1976.

[84] Oren E. Livne and Achi Brandt. "Lean algebraic multigrid (LAMG): Fast Graph Laplacian Linear Solver." In: *SIAM Journal on Scientific Computing* 34.4 (2012), B499–B522.

[85] Moritz von Looz, Henning Meyerhenke, and Roman Prutkin. "Generating Random Hyperbolic Graphs in Subquadratic Time." In: *Proceedings of the 26th International Symposium on Algorithms and Computation (ISAAC)*. LNCS. Springer, 2015, pp. 467–478.

[86] Ulrike von Luxburg, Agnes Radl, and Matthias Hein. “Hitting and commute times in large random neighborhood graphs.” In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1751–1798.

[87] Ahmad Mahmoody, Charalampos E. Tsourakakis, and Eli Upfal. “Scalable Betweenness Centrality Maximization via Sampling.” In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD).* 2016, pp. 1765–1773.

[88] Paolo Malighetti, Gianmaria Martini, Stefano Paleari, and Renato Redondi. *The Impacts of Airport Centrality in the EU Network and Inter-Airport Competition on Airport Efficiency.* Tech. rep. MPRA-7673. 2009.

[89] Massimo Marchiori and Vito Latora. “Harmony in the small-world.” In: *Physica A: Statistical Mechanics and its Applications* 285.3-4 (2000), pp. 539–546.

[90] Charalampos Mavroforakis, Richard Garcia-Lebron, Ioannis Koutis, and Evimaria Terzi. “Spanning Edge Centrality: Large-scale Computation and Applications.” In: *Proceedings of the 24th International Conference on World Wide Web (WWW).* 2015, pp. 732–742.

[91] Robert McColl, Oded Green, and David A. Bader. “A new parallel algorithm for connected components in dynamic graphs.” In: *Proceedings of the 20th Annual International Conference on High Performance Computing (HiPC).* 2013, pp. 246–255.

[92] Henning Meyerhenke, Burkhard Monien, and Stefan Schamberger. “Graph partitioning and disturbed diffusion.” In: *Parallel Computing* 35.10-11 (2009), pp. 544–569.

[93] Henning Meyerhenke, Martin Nöllenburg, and Christian Schulz. “Drawing Large Graphs by Multilevel Maxent-Stress Optimization.” In: *Proceedings of the 23rd International Symposium on Graph Drawing and Network Visualization (GD).* 2015, pp. 30–43.

[94] Meghana Nasre, Matteo Pontecorvi, and Vijaya Ramachandran. “Betweenness Centrality - Incremental and Faster.” In: *Proceedings of the 39th International Symposium on Mathematical Foundations of Computer Science(MFCS).* 2014, pp. 577–588.

[95] Mark E. J. Newman. *Networks: An Introduction.* OUP Oxford, 2010.

[96] Kazuya Okamoto, Wei Chen, and Xy Li. “Ranking of closeness centrality for large-scale social networks.” In: *Frontiers in Algorithmics* 5059 (2008), pp. 186–195.

[97] Paul W. Olsen, Alan G. Labouseur, and Jeong-Hyon Hwang. “Efficient top-k closeness centrality search.” In: *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE).* 2014, pp. 196–207.

[98] John F. Padgett and Christofer K. Ansell. *Robust Action and the Rise of the Medici, 1400-1434.* University of Chicago, 1993.

[99] Romualdo Pastor-Satorras and Alessandro Vespignani. “Epidemic Spreading in Scale-Free Networks.” In: *Physical Review Letters* 86.14 (2001), pp. 3200–3203.

[100] Forrest R. Pitts. “A graph theoretic approach to historical geography.” In: *The Professional Geographer* 17 (1965), pp. 15–20.

[101] Matteo Pontecorvi and Vijaya Ramachandran. “Fully Dynamic Betweenness Centrality.” In: *Proceedings of the 26th International Symposium on Algorithms and Computation (ISAAC)*. 2015, pp. 331–342.

[102] Rami Puzis, Polina Zilberman, Yuval Elovici, Shlomi Dolev, and Ulrik Brandes. “Heuristics for Speeding Up Betweenness Centrality Computation.” In: *Proceedings of the 2012 International Conference on Privacy, Security, Risk and Trust (PASSAT) and 2012 International Conference on Social Computing (SocialCom)*. 2012, pp. 302–311.

[103] Mihai Pătraşcu and Ryan Williams. “On the possibility of faster SAT algorithms.” In: *Proceedings of the 21st ACM/SIAM Symposium on Discrete Algorithms (SODA)*. 2010.

[104] G. Ramalingam and Thomas W. Reps. “On the Computational Complexity of Dynamic Graph Problems.” In: *Theoretical Computer Science* 158.1&2 (1996), pp. 233–277.

[105] Pascal Rebreyend, Laurent Lemarchand, and Reinhardt Euler. “A Computational Comparison of Different Algorithms for Very Large p -median Problems.” In: *Proceedings of the 15th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP)*. 2015, pp. 13–24.

[106] J. Reese. “Solution methods for the p-median problem: An annotated bibliography.” In: *Networks* 48.3 (2006), pp. 125–142.

[107] Matteo Riondato and Evgenios M. Kornaropoulos. “Fast approximation of betweenness centrality through sampling.” In: *Data Mining and Knowledge Discovery* 30.2 (2016), pp. 438–475.

[108] Matteo Riondato and Eli Upfal. “ABRA: Approximating Betweenness Centrality in Static and Dynamic Graphs with Rademacher Averages.” In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2016, pp. 1145–1154.

[109] Liam Roditty and Virginia V. Williams. “Fast approximation algorithms for the diameter and radius of sparse graphs.” In: *Proceedings of the 45th annual ACM Symposium on Theory of Computing (STOC)*. 2013, pp. 515–524.

[110] Jurgen Ruge and Klaus Stüben. “Algebraic multigrid.” In: *Multigrid methods* 3 (1987), pp. 73–130.

[111] Peter Sanders. “Algorithm Engineering - An Attempt at a Definition.” In: *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*. 2009, pp. 321–340.

[112] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. “Incremental algorithms for closeness centrality.” In: *Proceedings of the 1st IEEE International Conference on Big Data*. 2013, pp. 487–492.

[113] Ahmet Erdem Sariyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. “Hardware/Software Vectorization for Closeness Centrality on Multi-/Many-Core Architectures.” In: *Proceedings of the IEEE International Parallel & Distributed Processing Symposium Workshops*. 2014, pp. 1386–1395.

[114] Avi Shoshan and Uri Zwick. “All Pairs Shortest Paths in Undirected Graphs with Integer Weights.” In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science, (FOCS).* 1999, pp. 605–615.

[115] Jeremy G. Siek, Lie Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual.* Pearson Education, 2001.

[116] Daniel A. Spielman and Nikhil Srivastava. “Graph Sparsification by Effective Resistances.” In: *SIAM Journal on Computing* 40.6 (2011), pp. 1913–1926.

[117] Daniel A. Spielman and Shang-Hua Teng. “Nearly-linear Time Algorithms for Graph Partitioning, Graph Sparsification, and Solving Linear Systems.” In: *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC).* 2004, pp. 81–90.

[118] Christian L. Staudt and Henning Meyerhenke. “Engineering Parallel Algorithms for Community Detection in Massive Networks.” In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 27.1 (2016), pp. 171–184.

[119] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. “NetworKit: A Tool Suite for High-Performance Network Analysis.” In: *Network Science* 4.4 (2016), pp. 508–530.

[120] William Stein and David Joyner. “Sage: System for algebra and geometry experimentation.” In: *SIGSAM Bulletin* 39.2 (2005), pp. 61–64.

[121] Andreia S. Teixeira, Francisco C. Santos, and Alexandre P. Francisco. “Spanning Edge Betweenness in Practice.” In: *Proceedings of the 7th Workshop on Complex Networks (CompleNet).* 2016, pp. 3–10.

[122] Ryan Williams. “A new algorithm for optimal 2-constraint satisfaction and its implications.” In: *Theoretical Computer Science* 348.2-3 (2005), pp. 357–365.

[123] Virginia V. Williams. “Multiplying matrices faster than Coppersmith-Winograd.” In: *Proceedings of the 44th Symposium on Theory of Computing, (STOC).* 2012, pp. 887–898.

[124] Virginia V. Williams and Ryan Williams. “Subcubic Equivalences between Path, Matrix and Triangle Problems.” In: *Proceedings of the 51st IEEE Annual Symposium on Foundations of Computer Science (FOCS).* 2010, pp. 645–654.

[125] Yuichi Yoshida. “Almost linear-time algorithms for adaptive betweenness centrality using hypergraph sketches.” In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD).* 2014, pp. 1416–1425.

[126] Junzhou Zhao, John C. S. Lui, Don Towsley, and Xiaohong Guan. “Measuring and maximizing group closeness centrality over disk-resident graphs.” In: *Proceedings of the 23rd International World Wide Web Conference (WWW).* 2014, pp. 689–694.

[127] Uri Zwick. “All Pairs Shortest Paths in Weighted Directed Graphs ¾ Exact and Almost Exact Algorithms.” In: *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS).* 1998, pp. 310–319.

[128] Jon delEtoile and Hojjat Adeli. “Graph Theory and Brain Connectivity in Alzheimer’s Disease.” In: *The Neuroscientist* (2017).

APPENDICES

PUBLICATIONS

Some of the research leading to this thesis has appeared previously in the following publications.

Journal Articles

- Elisabetta Bergamini, Henning Meyerhenke: **Approximating betweenness centrality in fully dynamic networks**. – *Internet Mathematics*, 2016

Conference Papers

- Elisabetta Bergamini, Tanya Gonser, Henning Meyerhenke: **Scaling up Group Closeness Maximization**. – Accepted at *SIAM Workshop on Algorithm Engineering and Experiments (ALENEX 2018)*, January 2018, New Orleans, USA
- Patrick Bisenius, Elisabetta Bergamini, Eugenio Angriman, Henning Meyerhenke: **Computing Top-k Closeness Centrality in Fully-dynamic Graphs**. – Accepted at *SIAM Workshop on Algorithm Engineering and Experiments (ALENEX 2018)*, January 2018, New Orleans, USA
- Elisabetta Bergamini, Henning Meyerhenke, Mark Ortmann, Arie Slobbe: **Faster betweenness centrality updates in evolving networks**. – *International Symposium on Experimental Algorithms (SEA 2017)*, June 2017, London, UK
- Elisabetta Bergamini, Henning Meyerhenke, Christian Staudt: **Estimating current-flow closeness centrality with a multigrid laplacian solver**. – *SIAM Workshop on Combinatorial Scientific Computing (CSC 2016)*, October 2016, Albuquerque, USA
- Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, Henning Meyerhenke: **Computing top-k Closeness Centrality Faster in Unweighted Graphs**. – *SIAM Workshop on Algorithm Engineering and Experiments (ALENEX 2016)*, January 2016, Arlington, USA
- Elisabetta Bergamini, Henning Meyerhenke: **Fully-dynamic approximation of betweenness centrality**. – *European Symposium on Algorithms (ESA 2015)*, September 2015, Patras, Greece
- Elisabetta Bergamini, Henning Meyerhenke, Christian Staudt: **Approximating Betweenness Centrality in Large Evolving Networks**. – *SIAM Workshop on Algorithm Engineering and Experiments (ALENEX 2015)*, January 2015, San Diego, USA

Journal Articles in Revision Process

- Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, Henning Meyerhenke: **Computing top-k Closeness Centrality Faster in Unweighted Graphs**. – preprint available at `https://arxiv.org/abs/1704.01077`
- Elisabetta Bergamini, Pierluigi Crescenzi, Gianlorenzo D'Angelo, Henning Meyerhenke, Lorenzo Severini, Yllka Velaj: **Improving the betweenness centrality of a node by adding links**. – preprint available at `https://arxiv.org/abs/1702.05284`