Karlsruhe Institute of Technology

Computing Top-k Closeness Centrality in Fully-dynamic Graphs

Master's Thesis

Patrick Bisenius
1640015

At the Department of Informatics
Institute of Theoretical Informatics

Reviewer: Prof. Dr. Henning Meyerhenke
Advisor: Elisabetta Bergamini

3rd May 2017

Abstract

Closeness is a widely-studied centrality measure in network analysis. Since it requires all pairwise distances, computing closeness centrality for all nodes is unfeasible for large real-world networks. However, for most applications, it is only necessary to find the k most central nodes and not all closeness values. Prior work has shown that computing the top-k nodes with highest closeness can be done much faster than computing closeness for all nodes in real-world networks.

However, for networks that evolve over time, no dynamic top-k closeness algorithm exists that improves on static recomputation. In this thesis, we present several techniques that allow us to efficiently recompute the k nodes with highest closeness after an edge insertion or an edge deletion. Our algorithms use information obtained during earlier computations to skip unnecessary work. However, they do not require asymptotically more memory than the static algorithms (i.e. linear in the number of nodes). We propose separate algorithms for complex networks and networks with large diameter, such as street networks. For edge insertions in undirected complex networks, our average speedup on recomputation is about 40 and it is up to 160 for some graphs. For street networks, the average speedup is 70 and it is up to 150. For some edges, our algorithm is up to three orders of magnitude faster than static recomputation.

We also study the problem of finding a group of k nodes with high closeness. We adapt a static greedy algorithm for the dynamic case. We use our dynamic algorithms for top-k closeness centrality and employ similar techniques to reduce the amount of computation. On average, our dynamic algorithm is between 8 and 15 times faster, depending on the group size.

Zusammenfassung

Nähezentralität ist eine weit verbreitete Metrik im Feld der Netzwerkanalyse. Da zur Berechnung die Distanzen aller Knotenpaare nötig wären, ist es unpraktikabel, die Nähezentralität aller Knoten in großen realen Netzwerken zu berechnen. Für einige Anwendungen reicht es allerdings aus, die k zentralsten Knoten zu bestimmen und nicht alle Nähezentralitäten. Frühere Arbeiten haben bereits gezeigt, dass die Berechnung der k zentralsten Knoten wesentlich schneller durchgeführt werden kann als die Berechnung der Nähezentralität für alle Knoten.

Allerdings existieren bisher keine Algorithmen für Netzwerke, die sich im Laufe der Zeit verändern, die besser als eine statische Neuberechnung sind. Im Rahmen dieser Arbeit stellen wir nun mehrere Techniken vor, die uns erlauben, die k zentralsten Knoten eines Netzwerks nach Einfügen oder Löschen einer Kante auf effiziente Weise neuzuberechnen. Unsere Algorithmen verwenden dabei Information aus früheren Berechnungen um unnötigen Rechenaufwand zu vermeiden. Dafür benötigen unsere Algorithmen nur linearen zusätzlichen Speicher, also asymptotisch gesehen nicht mehr als die statischen Algorithmen. Wir stellen verschiedene Algorithmen für komplexe Netzwerke und Netzwerke mit großen Durchmesser, wie zum Beispiel Straßennetzwerke vor. Unser dynamischer Algorithmus ist im Schnitt etwa 40 mal schneller als die statische Referenz für ungerichtete komplexe Netzwerke, in einzelnen Fällen bis zu 160 mal schneller. Für Straßennetzwerke wird die Berechnung im Schnitt um den Faktor 70 beschleunigt, für einzelne Graphen sogar um einen Faktor von 150. Für einige bestimmte Kanten ist unser dynamischer Algorithmus um bis zu drei Größenordnungen schneller als eine statische Neuberechnung.

Wir beschäftigen uns außerdem mit dem Problem der Gruppenzentralität. Dabei geht es darum, eine Gruppe mit k Knoten zu finden, die als Ganzes eine hohe Zentralität hat. Dabei passen wir einen statischen Greedy-Algorithmus für dynamische Graphen an. Wir verwenden dabei unter anderem unseren dynamischen Algorithmus für Top-k-Nähezentralität und adaptieren weitere ähnliche Techniken um den Rechenaufwand zu reduzieren. Im Durchschnitt ist unser dynamischer Algorithmus acht bis fünfzehn Mal schneller als die statische Referenz, je nachdem wie groß die gesuchte Gruppe ist.

Acknowledgements

First of all, I would like to thank Professor Meyerhenke for giving me the opportunity to work on this very interesting topic.

My special thanks goes to my advisor Elisabetta Bergamini. She helped me find the topic and assisted me in every way I needed. We had many productive discussions that produced the ideas and solutions presented in this thesis.

I would like to thank all of my friends who were very accepting when I did not have the time or mood to attend social events.

At last, I would like to thank my parents, my brother, and my sister for supporting me all this time.

Contents

List of Figures

List of Tables

1. Introduction

The concept of a *network* is used as a tool to model interactions and connections in several fields of science. “The scientific study of networks [...] is an interdisciplinary field that combines ideas from mathematics, physics, biology, computer science, the social sciences and many other areas”, Mark Newman writes in his book *Networks. An introduction* [20].

One fundamental concept in network analysis is *Centrality.* Centrality measures are used to ascertain the importance of nodes within a graph. Practical applications include finding the most influential people in a social network [15], identifying key infrastructure nodes in computer networks, analyzing the effects of human land use to organism movement [13], or finding so-called super-spreaders in disease transmission networks [10].

One of the most widely-known centrality measures is *PageRank*, proposed by the founders of Google, Sergey Brin and Larry Page [21]. The idea behind PageRank is that a webpage is important if it is linked to by other important webpages. Instead of simply relying on the absolute number of links to a webpage (a metric that was often used in citation networks), the quality of the links is taken into account. PageRank is similar to Eigenvector centrality. At the time, PageRank was a considerable improvement over other search engines.

Another popular centrality measure is called Betweenness Centrality. A node is important if it is part of many shortest paths between other node pairs. For instance, this can be useful to find the routers in a computer network that are most integral to its stability.

Closeness centrality

Closeness centrality is based on the intuition, first presented by Alexander Bavelas in 1950 [2], that a node is important if its distance to other nodes in the graph is small. There are different definitions of closeness centrality that are applicable in different contexts. For strongly connected graphs, one can simply compute the sum of the distances from one node to all the other nodes and compute the inverse value. However, this approach leads to problems on disconnected graphs. It is not initially clear how to treat nodes which are not connected by a path in the graph. Simply assuming an infinite (or arbitrarily large) distance would completely distort the resulting closeness values of affected nodes, since closeness centrality involves computing the sum of distances to other nodes. Arbitrarily large or infinite terms automatically lead to a sum total that is arbitrarily large and therefore loses any informative value. One possible solution for this problem is to only take into account the distances of reachable nodes, then scaling the result with the number of reachable nodes. Another approach is called *harmonic closeness centrality* which has been shown to have nice properties on arbitrary graphs [6]. Instead of summing all distances and then computing the inverse, the harmonic closeness centrality is obtained by computing the sum of the inverse distances between nodes. Disconnected node pairs do not contribute to the total sum of inverse distances.

Computing the closeness centrality of a node in an unweighted graph requires a complete breadth-first search (BFS), and a complete run of Dijkstra’s algorithm [11] with weighted graphs. It requires solving the *all-pairs-shortest-path* problem to compute the closeness centrality of each node in the graph. The computational effort for this is often impractical, especially on large real-world networks. Unfortunately, this effort cannot be avoided if the application requires the exact closeness ranking of all nodes.

For some applications, however, it is enough to compute a list of the k most central nodes. This problem is called Top-k closeness centrality. Limiting the problem to the k most central nodes can decrease the required computational effort significantly in real-world networks.

The idea of existing approaches [4, 7] is to compute upper bounds on the closeness of each node. If we find k nodes whose closeness is higher than the upper bounds on the remaining nodes, we know these nodes have to be the top-k (and we do not need to compute the exact closeness of the other nodes). Notice that, although these approaches were shown to work very well in practice, they are not asymptotically faster than computing closeness for all nodes. However, Bergamini et al. [4] have shown that the most central node in a graph cannot be computed in $\mathcal{O}(|E|^{2-\epsilon})$ in directed graphs in the worst-case, under reasonable complexity assumptions.

Dynamic Top-k closeness centrality

In some cases, it is enough to compute closeness centralities only once because the underlying graph is static. Now consider a social network which constantly adds new users, which is effectively a node insertion in the underlying graph, and existing users befriend other users, which is an edge insertion. Terminating a friendship in the social network corresponds to an edge removal.

Each modification of an undirected graph affects at least the closeness centralities of the directly affected nodes, that is, the nodes incident to a newly inserted or removed edge. In directed graphs, only the source node of the directed edge is directly affected. It is also possible that there are new shortest paths between pairs of nodes that use the newly inserted edge. Analogously, removing an edge might increase the distance between node pairs because every shortest path between them contained the removed edge. A simple strategy to get the new closeness centralities of each node is to re-run the static algorithm on the modified graph, ignoring any information collected by previous runs of the algorithm.

Group closeness

The concept of closeness centralities for single nodes can be extended to groups of nodes. The distance between a node v and a group S is defined as the smallest distance between v and any node of the group. The problem to find a group of size k such that the total distance of all nodes in the graph to the group is minimal is called the *maximum closeness centrality group identification* (MCGI) problem by Chen et al. in [9]. Since the problem is shown to be NP-hard, no efficient exact algorithm exists at this point. However, there are approximative greedy algorithms [9, 27] for the problem. Bergamini et al. [5] improve on the work in [9] by reducing the memory requirements and total number of operations.

Types of networks

In this thesis, we will work with two types of networks and will adapt our algorithms to exploit their individual structure.

Complex networks are usually highly irregular graphs that exhibit the *small-world phenomenon* [25]. These networks have very few nodes that have high degree (*hubs*) and a vast majority of nodes with comparably low degree. The degree distribution follows a *power law*, i.e. the amount of nodes with degree k is proportional to $k^{-\gamma}$ where γ is a positive constant. Since many nodes have a connection to at least one hub, these complex networks often have a small diameter ("small world"). Social networks like Twitter or Facebook have underlying social graphs with this structure.

Street networks are based on real-world infrastructure and model the roads of a certain region. They have a large diameter compared to complex networks and hubs are not as distinguishable from other nodes.

Contributions

This thesis contributes a dynamic algorithm for Top-k closeness that handles both edge insertions and edge removals. It is based on the static algorithm first proposed by Borassi et al for complex networks in [7], and the additional optimizations for street networks proposed by Bergamini et al. in [4]. Our algorithm re-uses information obtained by an initial run of the static algorithm and tries to skip the re-computation of closeness centralities for nodes that are unaffected by modifications of the graph. In some cases, even the upper bounds for the closeness centralities of affected nodes can be updated with little computational effort. In contrast to other dynamic algorithms for closeness centrality, it is not required to compute the exact closeness centralities of all nodes in the graph. Our dynamic algorithms require only a linear amount of additional memory.

We also contribute an algorithm to update the group of nodes with the highest group closeness after an edge insertion. It is based on the work by Bergamini et al. who improved the algorithm by Chen et al. Our basic idea is to verify whether the choices of the greedy algorithm are still valid on the modified graph with as little computational effort as possible. Once the greedy algorithm would choose a different node than on the previous graph, the dynamic algorithm discards all information from the previous run and falls back to the static algorithm.

All our dynamic algorithms are based on existing implementations in NetworKit which is an open-source toolkit for high-performance network analysis [23]. In experiments, we obtain significant speedups compared to static recomputation. For instance, our dynamic algorithm for top-k closeness centrality is about 40 times faster on average on undirected complex networks. For directed street networks, we reach speedups of up to 140 on average. Our dynamic group closeness algorithm is between 8 and 15 times faster on average, depending on the group size.

Outline

In Chapter 2, we will provide basic definitions. Chapter 3 contains a description of the static algorithms for Top-k closeness centrality in complex networks and street networks. We will adapt these algorithms for dynamic graphs in Chapter 4. In Chapter 5, we will make use of our dynamic algorithm for Top-k closeness centrality and describe a dynamic algorithm for the group closeness problem. Chapter 6 describes our implementation in NetworKit and provides some additional implementation details. We evaluate our dynamic algorithms and compare them to static recomputation in Chapter 7. We conclude with Chapter 8 and provide some ideas for future work.

2. Preliminaries

In this thesis, $G = (V, E)$ denotes a simple, unweighted graph with a set of nodes V and a set of edges E between the nodes. An undirected edge is a subset of V with exactly two distinct elements. A directed edge is an element of $V \times V$. For simplicity, we will use the notation (u, v) for an edge between two nodes u and v interchangeably for directed and undirected edges. The distance between two nodes u and v is denoted by $d(u, v)$. There are sometimes multiple graph instances G and G' when dealing with dynamic graphs. In these cases, we will use $d_G(u, v)$ to refer to distances in G.

The set of neighbor nodes for v is denoted by $N(v) := \{u : \exists(v, u) \in E\}$ in a directed graph and by $N(v) := \{u : \exists\{v, u\} \in E\}$ in an undirected graph. It is sometimes useful to consider the incoming edges of a node in directed graphs. $N^{\leftarrow}(v) := \{u : \exists(u, v) \in E\}$ denotes the set of incoming neighbors of a node v.

We now want to define the concept of closeness centrality. The most simple definition, while only meaningful for connected graphs, is the following:

Definition 2.1. *Let $G = (V, E)$ be a connected, unweighted graph. The closeness centrality of a node $v \in V$ is defined as*

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

In the disconnected case, there are node pairs without a path between them. Using Definition 2.1 and assuming $d(u, v) = \infty$ for such node pairs, the sum over all the distances in the denominator would blow up and make the resulting closeness centralities useless. To solve this problem, we first define $R(v) := \{u \in V : u \text{ is reachable from } v \text{ in } G\}$ and $r(v) := |R(v)|$. This leads to a generalized version of Definition 2.1:

$$c(v) = \frac{r(v) - 1}{\sum_{u \in R(v)} d(v, u)}. \tag{2.1}$$

However, this definition does not differentiate between central nodes in small components and central nodes in large components of a graph. Intuitively, a node v with $r(v) = 2000$ and a total distance of 4000 to all reachable nodes is more central than a node w with $r(w) = 20$ and a total distance of 40. In order to give preference to nodes in large components, we scale Equation 2.1 by $\frac{r(v)}{|V|-1}$.

Definition 2.2. *Let $G = (V, E)$ be an unweighted graph. The closeness centrality of a node $v \in V$ is defined as*

$$c(v) = \frac{r(v) - 1}{\sum_{u \in R(v)} d(v, u)} \cdot \frac{r(v)}{|V| - 1}.$$

Another approach to handle disconnected graphs is called *harmonic closeness centrality.*

Definition 2.3. *Let $G = (V, E)$ be an unweighted graph. The harmonic closeness centrality of a node $v \in V$ is defined as*

$$h(v) = \sum_{u \in V} \frac{1}{d(v, u)} \text{ [6].}$$

If there is no path between u and v, the inverse distance is set to 0. Unless stated otherwise, *closeness centrality* refers to *harmonic closeness centrality* in this thesis.

3. Static Top-k closeness

Computing the exact closeness centrality of each node in a graph requires solving the *all-pair-shortest-path* (APSP) problem. A simple approach is to compute a breadth-first search (or Dijkstra's algorithm) from each node. This results in a time complexity of $\mathcal{O}(|V| \cdot (|V| + |E|))$ for unweighted graphs and $\mathcal{O}(|V| \cdot (|V| \log |V| + |E|))$ for weighted graphs. The Floyd-Warshall algorithm has a time complexity of $\mathcal{O}(n^3)$ [14]; Johnsons's algorithm for weighted graphs without negative cycles has a time complexity of $\mathcal{O}(|V| \cdot (|V| \log |V| + |E|))$ [16]. There are also algorithms that are based on Fast Matrix Multiplication [26, 28] which solve the problem in $\mathcal{O}(n^{2.3727})$. In practice, methods using graph traversal such as breadth-first search or Dijkstra's algorithm are preferred because real-world graphs are often sparse. Fast Matrix Multiplication algorithms also require a larger amount of memory.

There have been some ideas to reduce the effort to compute closeness centralities in large networks.

3.1 Approximation of closeness centrality

Eppstein and Wang propose a fast approximation algorithm in [12] for large networks exhibiting the *small world phenomenon*. It provides an $(1 + \epsilon)$-approximation in near-linear time. The algorithm works as follows:

1. Let k be the number of iterations to obtain the desired error bound
2. In iteration i, pick vertex v_i uniformly at random from G and solve the SSSP problem with v_i as the source
3. Let

$$\hat{c}_u = \frac{1}{\sum_{i=1}^{k} \frac{n \cdot d \cdot (v_i, u)}{k \cdot (n-1)}}$$

be the centrality estimator for vertex u.

Eppstein and Wang show that the expected value of $\frac{1}{\hat{c}_u}$ is equal to $\frac{1}{c_u}$. Using Hoeffding's bound, they also show that for $k = \mathcal{O}(\frac{\log n}{\epsilon^2})$ the additive error is at most $\Delta\epsilon$ with high probability, where Δ denotes the diameter of the graph. The total runtime of the algorithm is $\mathcal{O}\left(\frac{\log n}{\epsilon^2}(n \log n + m)\right)$ for weighted graphs. For unweighted graphs it is $\mathcal{O}\left(\frac{\log n}{\epsilon^2} \cdot (n + m)\right)$.

3.2 Top-k closeness centrality

For some real-world applications, it is unnecessary to know the exact closeness centrality and the exact ranking of unimportant nodes. In these cases, it might be enough to compute a list of the k nodes with the highest exact closeness centrality. For all the other nodes it is enough to provide an upper bound that is lower than the exact closeness centrality of the k-th most central node.

Borassi et al. propose an efficient algorithm for the problem in [7] which works especially well on complex networks. Angriman presents an adaptation of the algorithms to work with harmonic closeness centrality [1]. Bergamini et al. present some modifications for street networks (i.e. graphs with large diameter) in [4]. Since the dynamic algorithm presented in this thesis is based on these algorithms, we will provide detailed descriptions and analysis. The following explanation is based on the version of the algorithm for harmonic closeness centrality to avoid redundancy and since it is easily applicable to disconnected graphs. However, the adaptation of these algorithms to other definitions of closeness centrality is fairly straightforward.

3.2.1 Upper bounds for the closeness centrality of a node

The algorithm by Borassi et al. is briefly outlined in Algorithm 3. The main idea is to run a BFS from each node in the graph, but abort the search once it is clear that the node does not belong to the k nodes with highest closeness. During the BFS from a specific node v, the algorithm keeps track of an upper bound $\widetilde{h}(v)$ for the harmonic closeness of that node.

For that purpose, the algorithm keeps track of the current level d of the search, that is the current distance to the source node v of the search. When the BFS reaches a new level, the upper bound $\widetilde{h}(v)$ is recomputed. If the new upper bound is smaller than the known exact harmonic closeness centrality of the k-th node in the list, the search is aborted.

Let Γ_d denote the set of nodes on level d from v, γ_d the number of nodes on level d, i.e. $\gamma_d = |\Gamma_d|$. For each level, the algorithm computes an upper bound $\widetilde{\gamma}_{d+1} = \sum_{u \in \Gamma_d} deg(u)$ for the number of nodes on level $d+1$. Let $r(v)$ denote the number of nodes reachable from v. Let n_d denote the number of nodes up to level d. Let $h_d(v)$ denote the harmonic closeness based on all nodes up to level d from v, i.e. $h_d(v) = \sum_{d(v,u) \leq d} \frac{1}{d(v,u)}$.

During the BFS, the algorithms sums up the inverse distances of visited nodes. After visiting all nodes on level d, the resulting sum is $h_d(v)$. For the upper bound, we start with

$$h(v) \leq h'(v) = h_d(v) + \frac{\gamma_{d+1}}{d+1} + \frac{r(v) - n_{d+1}}{d+2}. \tag{3.1}$$

Basically, all nodes on level $d+1$ contribute $\frac{1}{d+1}$ to the harmonic closeness centrality. All remaining unvisited nodes have at least distance $d+2$.

Since $n_{d+1} = n_d + \gamma_{d+1}$, we can write

$$h'(v) = h_d(v) + \frac{\gamma_{d+1}}{d+1} + \frac{r(v) - n_d - \gamma_{d+1}}{d+2} \tag{3.2}$$

We can now replace γ_{d+1} with its upper bound $\widetilde{\gamma}_{d+1}$, which is computed after visiting all nodes on level d.

$$h'(v) \leq h_d(v) + \frac{\widetilde{\gamma}_{d+1}}{d+1} + \frac{r(v) - n_d - \widetilde{\gamma}_{d+1}}{d+2} \tag{3.3}$$

$$= h_d(v) + \frac{(d+2) \cdot \widetilde{\gamma}_{d+1} + (d+1) \cdot (r(v) - n_d - \widetilde{\gamma}_{d+1})}{(d+1) \cdot (d+2)} \tag{3.4}$$

$$= h_d(v) + \frac{\widetilde{\gamma}_{d+1} + (d+1) \cdot (r(v) - n_d)}{(d+1) \cdot (d+2)} \tag{3.5}$$

$$\widetilde{h}(v) = h_d(v) + \frac{\widetilde{\gamma}_{d+1}}{(d+1) \cdot (d+2)} + \frac{r(v) - n_d}{d+2} \tag{3.6}$$

3.2.2 Computing the number of reachable nodes

The number of reachable nodes $r(v)$ in Equation 3.6 can be computed in a preprocessing step. For undirected graphs, the number of reachable nodes from v is equal to the size of the connected component containing v. Algorithm 1 computes the number of reachable nodes for each node in $\mathcal{O}(n + m)$ with a single BFS.

Directed graphs

For directed graphs, $r(v)$ cannot be computed as easily if the graph is not strongly-connected. Instead, an upper bound for $r(v)$ is computed. The strongly-connected components of a graph G can be computed in linear time with Tarjan's algorithm [24]. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote the graph of the strongly-connected components of G, with

- $\mathcal{V}$ as the set of strongly-connected components in G
- $C \subset V$ and $D \subset V$ denoting strongly-connected components
- $(C, D) \in \mathcal{E} \iff \exists (u, v) \in E,$ such that $u \in C \wedge v \in D$
- the weight $w(C) = |C|$ for each strongly-connected component C.

Tarjan's algorithm already provides a topologically sorted list of the strongly-connected components of G. Components without any outgoing edges, so-called *sinks*, are placed at the end of the list. The list is then processed in reverse order and an upper bound $\omega(C)$ for the number of reachable nodes from that component can be computed with

$$\omega(C) = w(C) + \sum_{(C,D) \in \mathcal{E}} \omega(D).$$

Since the components are traversed in reverse topological order, the upper bound of each neighbor component is already known. Please note that there are no circles in this graph of strongly-connected components. Otherwise, all components comprising the circle would belong to one larger strongly-connected components.

3.2.3 Pruned breadth-first search

In order to compute the closeness centrality of a single node in an unweighted graph, a complete breadth-first search starting from that node is necessary. The algorithm only

Algorithm 1: Computing connected components in an undirected graph

```
Data: $G = (V, E)$
Result: $r(v)$ for each node $v$
$P(v) \leftarrow$ none for each node $v$
$i \leftarrow 0$
forall $v \in V$ do
    if $P(v) = none$ then
        Run BFS from $v$ and set $P(u) = i$ for each visited node $u$, keep track of the
          component sizes
        $i \leftarrow i + 1$
    end
end
forall $v \in V$ do
    $r(v) \leftarrow componentSizes[P(v)]$
end
```

requires the exact closeness centrality of the k most central nodes. Therefore, the breadth-first searches can be aborted early for many nodes, once the upper bound obtained with Equation 3.6 is smaller than the exact closeness centrality of the k-th most central node among the already processed nodes. Algorithm 2 outlines the steps necessary to perform such a pruned BFS. The algorithm is structured like a standard BFS, but also keeps track of the closeness centrality of the source node. The algorithm sums the inverse distances of all visited nodes in h (Line 22). Once the BFS reaches a new level (Line 7), $\widetilde{h}$ is updated (Line 10). If the new upper bound is smaller than the known exact closeness x_k of the k-th node, the algorithm returns with the last computed upper bound and a flag that indicates that the value is not exact. The algorithm also keeps track of an upper bound $\widetilde{\gamma}$ for the number of nodes on level $d+1$ by summing the out-degrees of all nodes on level d (Line 24 and Line 26). If the search encounters a node w that has already been marked as visited, that node must be on a level $l < d$. It is then possible to compute a new, lower upper bound $\widetilde{h}$ in some cases (Line 28).

3.2.4 Algorithm outline

Algorithm 3 is used to compute the list of the k nodes with the highest closeness in G. The preprocessing in Line 1 is used to compute the number of reachable nodes for each node in the graph in linear time (see Section 3.2.2). The algorithm then iterates over all nodes in decreasing order of degree (Line 5) and starts a pruned BFS from each node. If the pruned BFS returns an exact closeness value, and if the value is larger than the current k-th largest value, it can be added to the list with the most central nodes. x_k, representing the k-th largest value, can then also be updated.

3.2.5 Networks with large diameter

The previously described algorithm works well for complex (social) networks with small diameter. Bergamini et al. present a different approach to solve the problem faster on networks with large diameter, for instance street networks [4]. The basic idea is to always run a complete breadth-first search from a source node and then use the number of nodes on each level to compute upper bounds for the closeness centrality of each other node in the graph. The nodes are kept in a list sorted in decreasing order by the corresponding upper bound for their closeness centrality. We provide a detailed explanation of this strategy in the following paragraphs.

Algorithm 2: Pruned BFS

```
Data: $G = (V, E), v, x_k$
Result: A tuple $(h, isExact)$ with $isExact$ =false if $h$ is only an upper bound for the
        exact harmonic closeness centrality.
Create queue $Q$
$Q$.enqueue($v$)
Mark $v$ as visited
$d \leftarrow 0; h \leftarrow 0; \widetilde{\gamma} \leftarrow 0; nd \leftarrow 0$
while $!Q.isEmpty$ do
    $u \leftarrow$ Q.dequeue()
    if $d(v, u) > d$ then
        $d \leftarrow d + 1$
        $r \leftarrow r(u)$
        $\widetilde{h} \leftarrow h + \frac{\widetilde{\gamma}}{(d+1)\cdot(d+2)} + \frac{r - n_d}{d+2}$
        if $\widetilde{h} \leq x_k$ then
            return $(\widetilde{h}$, false)
        end
    end
    forall $w \in N(u)$ do
        if $w$ is not marked as visited then
            Mark $w$ as visited
            $Q$.enqueue($w$)
            $n_d \leftarrow n_d + 1$
            $pred[w] \leftarrow u$
            $d(v, w) \leftarrow d(v, u) + 1$
            $h \leftarrow h + \frac{1}{d(v,w)}$
            if $G$ is directed then
                $\widetilde{\gamma} \leftarrow \widetilde{\gamma} + outdegree(w) - 1$
            else
                $\widetilde{\gamma} \leftarrow \widetilde{\gamma} + outdegree(w)$
            end
        else if $d(v, w) > 1 \wedge pred[u] \neq w$ then
            $\widetilde{h} \leftarrow \widetilde{h} - \frac{1}{d+1} + \frac{1}{d+2}$
            if $\widetilde{h} \leq x_k$ then
                return $(\widetilde{h}$, false)
            end
    end
    return $(h$, true)
end
```

Algorithm 3: Static computation of the k nodes with the highest closeness

```
Data: $G = (V, E)$
Result: A list with the $k$ nodes with the highest closeness centrality
Compute $r(v) \forall v \in V$
Top ← empty priority queue $h(v) \leftarrow 0$ for each node $v$
$isExact(v) \leftarrow$ false for each node $v$
$x_k \leftarrow 0$
forall $v \in V$ in decreasing order of degree do
    $(h, isExact) \leftarrow$ BFSCut$(v, x_k)$
    $h(v) \leftarrow h$
    $isExact(v) \leftarrow isExact$
    if $isExact \wedge h > x_k$ then
        Top.insert(h, v)
        if Top.size() $> k$ then
            Top.removeMin()
        end
        if Top.size() $= k$ then
            $x_k \leftarrow$ Top.getMin()
        end
    end
end
```

Level-based upper bounds

It is possible to compute upper bounds for the closeness centrality of all nodes in a graph with a single BFS. Let G denote an undirected graph and s the source node of the BFS. A node v is on level i ($l(v) = i$) if $d(s, v) = i$ and we write $v \in \Gamma_i(s) \iff d(s, v) = i$ (see Section 3.2.1). The distance between two arbitrary nodes $v \in \Gamma_i(s)$ and $w \in \Gamma_j(s)$ for $i \leq j$ is at least $j - i$. If $d(v, w)$ was smaller than $j - i$, w would have been discovered earlier and its level would be $i + d(v, w) < j$. This is a contradiction to the assumption that w is on level j. Using this property, it is possible to obtain an upper bound for the harmonic closeness of each node $v \in V$:

$$h(v) \leq \widetilde{h}(v) = \sum_{w \in R(s)} \left| \frac{1}{d(s, w)} - \frac{1}{d(s, v)} \right|.$$

This bound can be improved further. The degree of a node v is equal to the number of nodes w at distance 1. All the other nodes with $|d(s, w) - d(s, v)| \leq 1$ must have at least distance 2. This leads to

$$\begin{aligned}\widetilde{h}(v) = {} & 1 \cdot deg(v) \\ & + \frac{1}{2} \cdot (|\{w \in R(s) : |d(s, w) - d(s, v)| \leq 1\}| - deg(v) - 1) \\ & + \sum_{\substack{w \in R(s) \\ |d(s,w) - d(s,v)| > 1}} |\frac{1}{d(s, w)} - \frac{1}{d(s, v)}|. \end{aligned} \tag{3.7}$$

After the BFS, the number of nodes γ_j on each level j is known. With that information Equation 3.7 can be rewritten to

$$\widetilde{h}(v) = \frac{1}{2} \cdot \left(\sum_{|j-d(s,v)|\leq 1} \gamma_j \right) + \left(\sum_{|j-d(s,v)|>1} \gamma_j \cdot \left| \frac{1}{j - d(s,v)} \right| \right) - \frac{1}{2} + \frac{1}{2} \cdot deg(v). \quad (3.8)$$

For all nodes v with the same distance from s, the first three terms of the equation are the same. Therefore, a preliminary *level bound* can be computed once for each level. For each individual node v, only $\frac{1}{2} \cdot deg(v)$ needs to be added to get $\widetilde{h}(v)$.

For directed graphs, Equation 3.7 does not hold. Consider two nodes v and w with $l(w) < l(v)$. In the undirected case, it is possible to infer a lower bound for the distance between the two nodes. This is not possible in the directed case because there could be a shortcut between v and w as shown in Figure 3.1. For all nodes w with $l(w) < l(v)$ and $w \notin N(v)$, we can only assume that the distance must be at least 2. This leads to slightly less tight upper bounds for directed graphs. Modifying Equation 3.7 accordingly leads to

$$\begin{aligned}\widetilde{h}_{directed}(v) = {} & \frac{1}{2} \cdot deg(v) \\ & + \frac{1}{2} \cdot (|\{w \in R(s) : d(s,w) - d(s,v) \leq 1\}|) \\ & + \sum_{\substack{w \in R(s) \\ d(s,w)-d(s,v)>1}} \frac{1}{d(s,w)} - \frac{1}{d(s,v)}. \end{aligned} \quad (3.9)$$

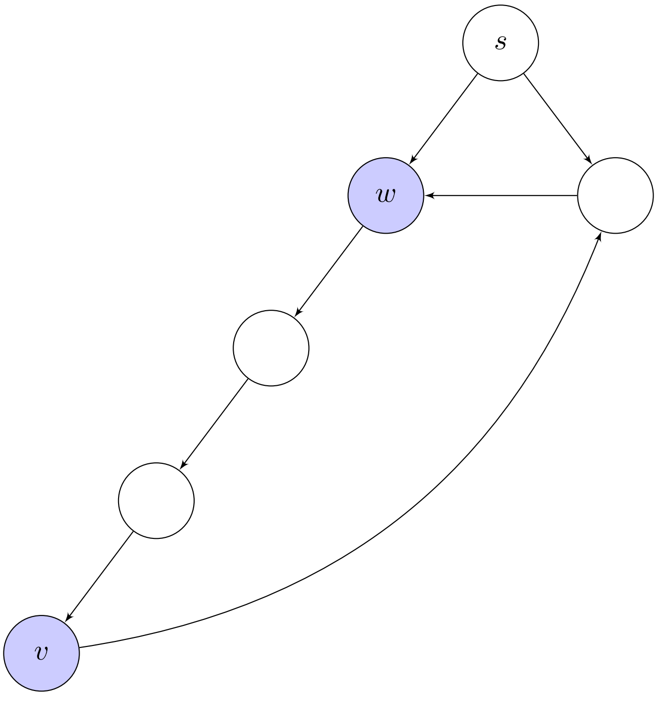

Figure 3.1: BFS tree
Distances in directed graphs are not symmetric. The distance between w and v is 3 in the BFS tree rooted in s, while the distance in the actual directed graph is 2.

Algorithm outline

The algorithm to compute the k nodes with highest closeness in networks with large diameters is outlined in Algorithm 4. The algorithm uses a priority queue `Q` which contains all unprocessed nodes, sorted by decreasing priority. It is required that the priority queue makes it possible to modify the priorities of existing elements efficiently. Initially, the degree of each node can be used for its priority. Alternatively, Bergamini et al. propose a preprocessing algorithm which computes an initial upper bound for the closeness centrality of each node based on its extended neighborhood, that is, all nodes up to a configurable distance l. The algorithm maintains an array `score` which contains the closeness centralities of each node. Each entry is initialized to ∞ (this means `std::numeric_limits<double>::max()` in a C++ implementation). The priority queue `Top` maintains the k nodes with highest closeness in increasing order.

The algorithm processes the nodes in `Q` in decreasing order of priority (Line 8). In each iteration, the node v with the maximum priority is extracted from `Q` (Line 9). We call v the *active* node. If the known upper bound for the closeness of the node is lower than the exact known value for the k-th most central node, the algorithm can be aborted and the list with the k nodes is returned. Otherwise, the `updateBounds` function is called with v as the source node (Line 13).

The `updateBounds` function is outlined in Algorithm 5. It computes a complete BFS from the supplied source node s and counts the number of nodes γ_i on each level i. The complete BFS from s makes it possible to compute the exact closeness of s. The resulting upper bound for each other node in the graph is computed using Equation 3.7 in the undirected case and Equation 3.9 in the directed case.

After the call to `updateBounds` returns, Algorithm 4 updates `score`. If the upper bound for a node u returned by the function is smaller than the previous upper bound stored in `score[`u`]`, the array is updated with the new, tighter upper bound (Line 15). In order to keep the list of the k most central up-to-date, the active node v is inserted with its exact closeness centrality as the priority into `Top` if the exact closeness centrality of v is larger than that of the k-th node in `Top` (Line 20). If `Top` contains more than k elements, the node with the smallest closeness is removed from the list.

Algorithm 4: Static computation of the k nodes with the highest closeness centrality in networks with large diameter

```
Data: $G = (V, E)$
Result: A list with the $k$ nodes with the highest closeness
Preprocessing($G$)
Q $\leftarrow V$, sorted by decreasing degree or a precomputed upper bound
Top $\leftarrow$ []
score $\leftarrow$ array indexed by node ID storing the current upper bounds for all nodes
for $v \in V$ do
    score[$v$] $\leftarrow \infty$
end
while $Q$ is not empty do
    $v \leftarrow$ Q.extractMax()
    if $|Top| \geq k \wedge$ score[$v$] $\leq$ Top[$k$] then
        return Top
    end
    levelBasedBounds $\leftarrow$ updateBounds($v$)
    score[v] $\leftarrow$ levelBasedBounds[v]
    forall $w \in V$ do
        if levelBasedBounds[$w$] $<$ score[$w$] then
            score[$w$] $\leftarrow$ levelBasedBounds[$w$]
        end
    end
    if score[$v$] $>$ Top[$k$] then
        add $v$ to Top
        sort Top by score and reduce it to at most $k$ elements
    end
    re-order Q according to the new values in score
end
```

Algorithm 5: The `updateBounds` function computes the exact closeness centrality of the supplied source node s and provides upper bounds for the closeness centrality of all other nodes in the graph.

```
Data: $G = (V, E), s \in V$
Result: The exact closeness centrality of $s$, upper bounds for the closeness of all other nodes
$d \leftarrow$ BFSfrom($s$)
$d_{max} \leftarrow max_{v \in V} d(s, v)$
$\forall i \in [1..d_{max}] : \gamma_i \leftarrow |\{v : d(s, v) = i\}|$
forall $v \in V$ do
    $S[v] \leftarrow \frac{1}{2} \cdot \left( \sum_{|j-d(s,v)| \leq 1} \gamma_j \right) + \left( \sum_{|j-d(s,v)|>1} \gamma_j \cdot \left| \frac{1}{j-d(s,v)} \right| \right) - \frac{1}{2} + \frac{1}{2} \cdot deg(v)$
end
$S[s] \leftarrow \sum_{u \in R(s)} \frac{1}{d(s,u)}$
return $S$
```

4. Top-k closeness centrality for dynamic graphs

Many algorithms from the field of network analysis, including the algorithms from Chapter 3, are designed to be run only once on a static graph. This does not match the requirements of many real-world applications. In social networks like Facebook or Twitter, the underlying social graph changes constantly. There are always new users who join a social network, some others leave; existing users befriend other users or end virtual friendships. The structure of the underlying social graph always changes, and therefore every metric in the network also changes.

The naïve approach to update these metrics is to simply run the corresponding static algorithm again after the graph has changed. The problem with this approach is that information obtained in previous runs is disregarded and the runtime of the algorithm will be roughly the same as before. In practice, modifying a single edge in the graph will often not affect all nodes in the graph. Depending on the metric and the corresponding algorithm, nodes unaffected by a modification can then simply be skipped to reduce the overall runtime.

In this thesis, we will present two new dynamic algorithms for Top-k closeness centrality. The first one is based on the static algorithm for complex networks by Borassi et al. (see Section 3.2). The second one is based on the static algorithm for networks with large diameter by Bergamini et al. (see Section 3.2.5).

4.1 Preliminaries

Let $G(V, E)$ denote an unweighted graph G with the set of nodes V and the set of edges E. We can define an *edge insertion* as a function f_i, such that

$$f_i(G(V,E),(u,v)) = G'(V, E \cup (u,v)) \text{ with } (u,v) \notin E. \tag{4.1}$$

An *edge removal* is defined as a function f_r, such that

$$f_r(G(V,E),(u,v)) = G'(V, E \setminus (u,v)) \text{ with } (u,v) \in E. \tag{4.2}$$

We will use *edge modification* as a collective term for both edge insertions and edge removals. We do not consider node modifications in this thesis because adding or removing isolated nodes does not affect the closeness centralities of the other nodes in the graph. Further, removing a node that is not isolated is equivalent to first removing all of its incident edges one-by-one and then removing the isolated node. Inserting a node with a set of incident edges can also be split into two parts: adding an isolated node and then adding the incident edges one-by-one.

We also define the *distance to the edge modification* as $d_G^f(w,(u,v)) := \min\left(d_G(w,u), d_G(w,v)\right)$ for undirected graphs. In directed graphs, only the distance to the source node u of the edge is relevant and we get $d_G^f(w,(u,v)) = d_G(w,u)$.

4.2 Dynamic closeness centrality

Sarıyüce et al. propose a dynamic algorithm to update the closeness centralities of all nodes in an unweighted graph after edge insertions or edge removals in [22]. The algorithm performs a breadth-first search starting from each node to compute the pairwise distances between all nodes. During this search, the algorithm also computes the total distance of the start node to all other nodes. The total distance is used to compute the closeness centrality of the start node.

The authors present several optimizations that make the dynamic algorithm more efficient on complex real-world networks. Let $G = (V, E)$ denote the base graph and $G' = (V, E \cup \{u, v\})$ the modified graph. The authors show that any node $s \in V$ with $|d_G(s, u) - d_G(s, v)| \leq 1$ is unaffected by the edge insertion. The recomputation of the closeness centrality can be skipped for these nodes.

The algorithm also maintains the biconnected components of the graph. A biconnected component is a subgraph of G from which any node v can be removed while the graph still stays connected. Each edge belongs to only one biconnected component, but a node can belong to multiple biconnected components. A node v is called *articulation vertex* if the graph $G - v$ has more connected components than G. After an edge insertion, the biconnected components of the graph are updated and the new edge is assigned to one of the biconnected components. The authors show that is possible to update the closeness centralities of all nodes by running a new BFS only for each node in the biconnected component of the new edge. The new closeness centrality for nodes of the biconnected component is computed with the new BFS. For each of the other nodes, a representative node from the biconnected component can be chosen. The closeness centrality of these other nodes can be updated based on the change of the total distance of its representative node.

The third optimization is based on identifying nodes with equal or similar neighborhoods. The authors propose two kinds of identical nodes: *type-I-identical* nodes have the exact same neighborhood. Nodes are *type-II-identical* if $\{u\} \cup N(u) = \{v\} \cup N(v)$. In both cases, the closeness centralities of all nodes in the same class are identical. Therefore, it is enough to compute the closeness centrality for one representative of a class of nodes.

At last, the authors propose a more efficient algorithm for breadth-first searches in complex networks. They call the approach *SSSP hybridization*. In complex networks, there often is a spike-shaped distribution of the distances in the network. The standard algorithm for a breadth-first search visits the node in a *top-down* manner, that is, the nodes on level k are used to discover the nodes on level $k + 1$. At some point, the number of remaining unvisited nodes is smaller than the number of nodes on level k. It will then be faster to check which of the remaining nodes have a neighbor on level k and set their level to $k + 1$.

The worst-case time complexity of the algorithm is $\mathcal{O}(n \cdot (n + m))$, which is the same as recomputing the closeness centralities from scratch. The algorithm keeps track of all pairwise distances in the graph, therefore the space complexity is $\mathcal{O}(n^2)$. Since it is necessary that the algorithm computes the exact closeness centralities for all nodes at least once, this algorithm does not scale to large networks.

4.3 Dynamic Top-k closeness centrality

We will now describe two new algorithms to update the list of the k most central nodes in a graph after an edge modification. The first algorithm is based on the static algorithm by Borassi et al. presented in Section 3.2. It works especially well on networks with small diameter. The second algorithm is based on the static algorithm for networks with large diameter by Bergamini et al. which is described in Section 3.2.5. The adaptations of both algorithms for the dynamic case both follow the static version closely. However, the dynamic versions will try to use previously collected information in order to skip some of the work the static versions have to do.

In any case, our dynamic algorithms only use a linear amount of additional memory. It is not required to store all pairwise distances.

4.3.1 Affected nodes

In general, it makes sense to only process nodes which are actually *affected* by an edge modification. Transforming a graph G to G' by inserting or removing edges will change the distances between some node pairs, and thus change the closeness centralities of these nodes.

4.3.1.1 Definition and observations

Definition 4.1. *Let $G = (V, E)$ denote an unweighted graph and $G' = f(G)$ a modified version of G with $f \in \{f_i, f_r\}$. Then the nodes in*

$$\begin{aligned} A_G^f(u,v) &= \{s \in V \mid \exists t \in V : d_G(s,t) \neq d_{G'}(s,t)\} && (4.3)\\ &= \{s \in V \mid d_G(s,v) \neq d_{G'}(s,v) \lor d_G(s,u) \neq d_{G'}(s,u)\} && (4.4)\end{aligned}$$

*are called **affected nodes**.*

First, we consider the insertion of the edge (u, v). In undirected graphs, both u and v will obviously be affected by this edge insertion because their new distance will be 1 while it had to be at least 2 before the insertion. In directed graphs, only u will be affected. Now consider two nodes s and t with $d_G(s,t) > d_G(s,u) + d_G(v,t) + 1$. After inserting (u, v), the new shortest path will use the new edge. The length of the new path is then $d_{G'}(s,t) = d_G(s,u) + d_G(v,t) + 1$. In general, every new shortest path has to contain the edge (u, v).

In the case of an edge removal, the distance between two nodes s and t only changes if all the shortest paths between the two nodes contain the removed edge. This makes both operations symmetrical. After an edge insertion, all new shortest paths contain the inserted edge. If the edge is subsequently removed, only these shortest paths cease to exist.

We noted earlier that each new shortest path or each erased shortest path must always contain the modified edge. We will call these shortest paths *affected shortest paths.* Each affected shortest path must contain the node sequence $u - v$. In undirected graphs, we can always switch the direction of a path such that u and v appear in this order.

For any node $s \in A^f_G(u, v)$, we only need to know whether there is at least one affected shortest path which starts in that node. Each affected shortest path contains the sequence $u - v$ and the distance between both nodes changes after modifying the edge (u, v). Therefore, the distance to either u or v will change for each affected node. Therefore, Equation 4.3 can be rewritten to Equation 4.4. We note that in the directed case, there are no nodes s for which $d_G(s, u) \neq d_{G'}(s, u)$. Equation 4.4 can be rewritten for directed graphs as

$$A^f_G(u, v) = \{s \in V \mid d_G(s, v) \neq d_{G'}(s, v)\}. \tag{4.5}$$

Figure 4.1 shows an example of an edge modification. The nodes affected by either removing or inserting the dashed edge between u and v are marked in red.

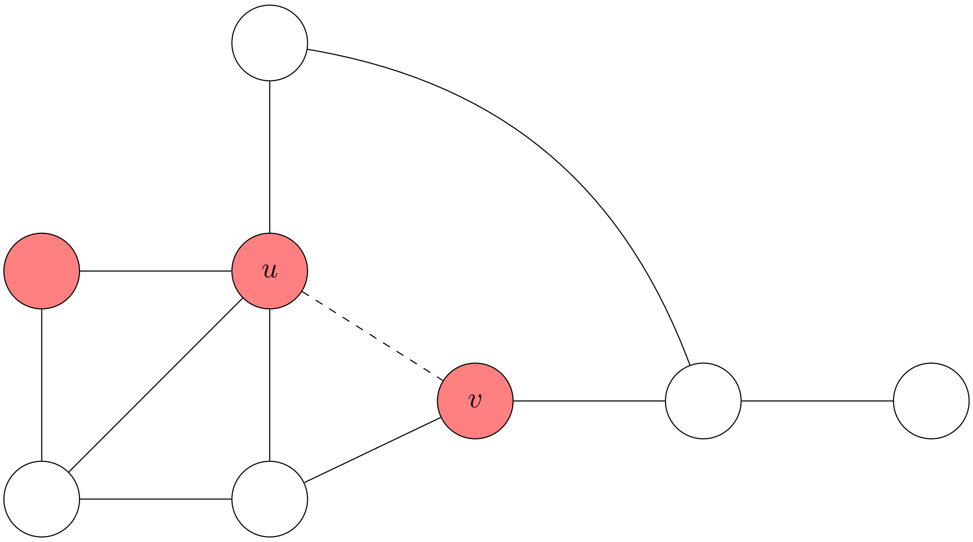

Figure 4.1: Affected and unaffected nodes
The nodes marked in red will be affected if the edge between u and v is modified.

4.3.1.2 Computing affected nodes

Directed graphs

In directed graphs, each affected shortest path contains the exact sequence $u-v$. Therefore, we only need to find those nodes for which the distance to v changes (see Figure 4.3). Algorithm 6 computes this set of nodes for an edge insertion on a directed graph.

Let G^- denote the graph G without the edge (u, v), let G^+ denote the graph G that contains the edge (u, v). We first run a reverse BFS (following incoming edges) on G^- starting from v and store the distances to each node in the graph (Line 8).

We then execute a second reverse BFS starting in v, this time on G^+. This pruned reverse BFS is shown in Algorithm 7. It is supplied with the graph G^+, the source node s for the reverse BFS and an array d^- containing the distances between s and each other node in G^-. We use a modified version of the standard algorithm which adds subtree pruning to avoid a full search. If we visit a new node w, we check if its distance to the source node is smaller than the old distance (Line 12). If the new distance is smaller than the old distance, we add w to the set of affected nodes and to the search queue. This technique allows us to skip subtrees that have not changed due to the edge modification.

It is not practical to construct separate instances G^- and G^+ of the graph when dealing with real-world networks. It is reasonable to assume that the underlying application only holds one instance of the graph. For instance, in the case of an edge removal, the underlying graph will no longer contain the removed edge when the update algorithm is called. This means that the underlying graph is equal to G^-. Therefore, the first reverse BFS can be executed without any further adjustments. However, the removed edge has to be added back virtually for the second BFS which is run on G^+. This is done by adding the old neighbor u to the search queue before the search starts. This is shown in Lines 5-8 of Algorithm 8.

Undirected graphs

Finding the affected nodes on undirected graphs requires more computational effort. As illustrated in Figure 4.2, those nodes can either have a different distance to u or to v. In the undirected case, we basically run the algorithm from the directed case twice. Once we start the breadth-first searches from u and and once from v. The set of the affected nodes is the union of all nodes whose distance to either u or v is different in G^- and G^+.

Algorithm 6: Computing affected nodes in directed graphs.

```
Data: $G = (V, E), (u, v)$: the modified edge
Result: $A_G^f(u, v)$
/* Construct $G^-$ and $G^+$ */
if $(u, v) \in E$ then
    $G^- \leftarrow (V, E \setminus \{(u, v)\})$
    $G^+ \leftarrow (V, E)$
else
    $G^- \leftarrow (V, E)$
    $G^+ \leftarrow (V, E \cup \{(u, v)\})$
end
/* like a normal BFS, but following incoming edges */
$d^- \leftarrow$ ReverseBFS($G^-, v$)
$A_G^f(u, v) \leftarrow$ PrunedReverseBFS($G^+$, $d^-$, $v$)
```

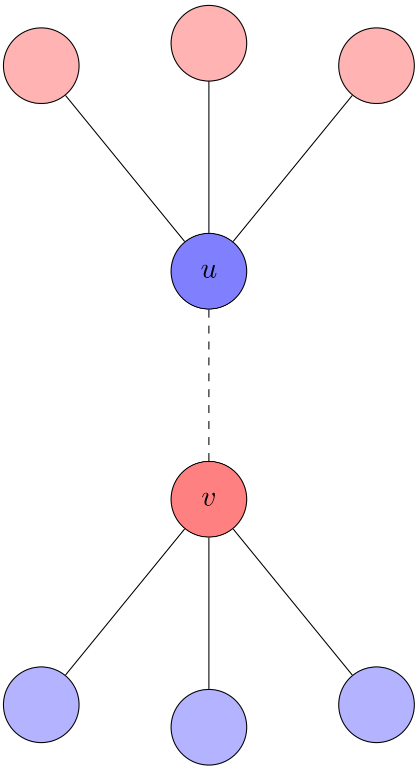

Figure 4.2: Computing affected nodes in undirected graphs

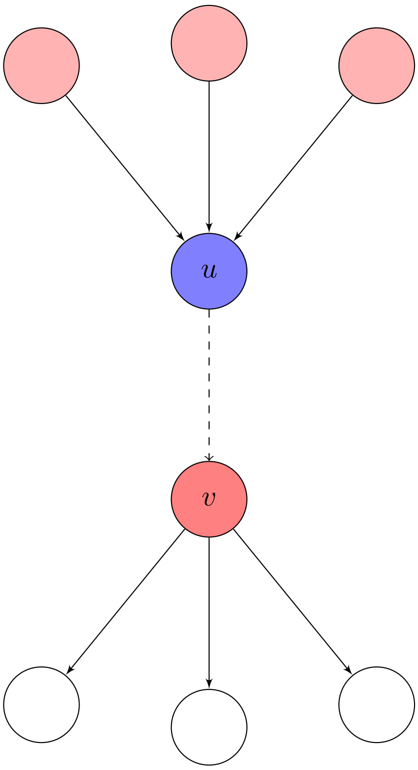

Figure 4.3: Computing affected nodes in directed graphs

Algorithm 7: `PrunedReverseBFS`

```
Data: $G = (V, E), d^{-}, s$
Result: $A_G^f(u, v)$
$A_G^f(u, v) \leftarrow \emptyset$
/* FIFO queue                                                    */
$Q \leftarrow \{s\}$
$d \leftarrow$ array storing the distances of the nodes
$d[s] \leftarrow 0$
Mark $s$ as visited
while $Q$ is not empty do
    $u \leftarrow$ first node from $Q$
    forall $w \in N^{\leftarrow}(u)$ do
        if $w$ is not marked as visited then
            $d[w] \leftarrow d[u] + 1$
            Mark $w$ as visited
            if $d^{-}[w] < d[w]$ then
                $A_G^f(u, v) \leftarrow A_G^f(u, v) \cup \{w\}$
                Enqueue $w$ in $Q$
            end
        end
    end
end
return $A_G^f(u, v)$
```

Algorithm 8: `PrunedReverseBFS` for edge removals

```
Data: $G = (V, E), d^-, s, (u, v) \notin E$
Result: $A_G^f(u, v)$
/* FIFO queue */
$Q \leftarrow \{s\}$
$d \leftarrow$ array storing the distances of the nodes
$d[s] \leftarrow 0$
Mark $s$ as visited
/* Virtually add back the old neighbor $u$ */
Enqueue $u$ in $Q$
Mark $u$ as visited
$d[u] = 1$
$A_G^f(u, v) \leftarrow \{u\}$
while $Q$ is not empty do
    $u \leftarrow$ first node from $Q$
    forall $w \in N^{\leftarrow}(u)$ do
        if $w$ is not marked as visited then
            $d[w] \leftarrow d[u] + 1$
            Mark $w$ as visited
            if $d_{old}[w] < d[w]$ then
                $A_G^f(u, v) \leftarrow A_G^f(u, v) \cup \{w\}$
                Enqueue $w$ in $Q$
            end
        end
    end
end
return $A_G^f(u, v)$
```

4.3.2 Dynamic Top-k closeness in complex networks

In the following sections, we will describe how our dynamic algorithm reproduces the results of the static algorithm in a more efficient manner by using existing knowledge of the graphs.

4.3.2.1 Recomputing the number of reachable nodes

After an edge modification, it is possible that the number of reachable nodes changes for some nodes in the graph. The dynamic instance of the algorithm stores the connected component each node belongs to. In the case of an edge insertion in undirected graphs, the number of reachable nodes does not change for any node if u and v are in the same component. If they are in different components, the two components are merged and the number of reachable nodes is set to the total number of nodes in the merged component. In all other cases, including edge insertions in directed graphs, we compute the number of reachable nodes (or an upper bound in directed graphs) from scratch. There are algorithms that maintain strongly-connected components in the case of edge removals [8, 17]. However, we have not implemented them for this thesis.

4.3.2.2 Preserving still-valid information

We keep a global data structure that stores either the exact closeness centrality or an upper bound for each node like in the static algorithm. During an update operation, we keep a priority queue with the k most central nodes. The exact closeness centralities or computed upper bounds of nodes that are unaffected by an edge modification are still valid after the modification is applied. Therefore, nodes that have previously belonged to the k most central nodes can be re-inserted into the priority queue that manages the Top-k list.

Edge insertions

In the case of an edge insertion, the closeness centrality of affected nodes can only increase. We recall that in the static algorithm, the breadth-first search for a given node is aborted if it is clear that the closeness centrality of this node cannot be larger than the closeness centrality of the k-th most central node, also denoted as x_k in this thesis. If there is a new member of the Top-k list after the edge insertion, its closeness centrality must be larger than x_k of the old graph. The closeness centrality (or the upper bound) of each affected node in the graph is initially marked as invalid.

Edge removals

In the case of an edge removal, the exact closeness centralities of affected nodes become invalid, but upper bounds of affected nodes are still valid. Consider a shortest path $u{-}v{-}w$ which is also the only shortest path of length 2 between u and w. If the edge (u, v) is removed, the new shortest path will at least have length 3. Now consider the harmonic closeness centrality $h(u)$. The contribution of the node pair (u, w) is $\frac{1}{d_G(u,w)} = \frac{1}{2}$ in the unmodified graph. After the edge is removed, the contribution is only $\frac{1}{d_{G'}(u,w)} = \frac{1}{3}$. Since an edge removal does not create any new shorter shortest paths between any pair of nodes, the exact closeness centrality of affected nodes will always be smaller than before. Therefore, all upper bounds of affected nodes are still valid, but now less tight than before. This also allows the following optimization in some cases: if none of the k most central nodes is affected by an edge removal, the update algorithm can be aborted after the computation of the affected nodes.

4.3.2.3 Recomputing closeness centralities after edge insertions

In this section, we will describe Algorithm 10 which updates the list of the k most central nodes and their exact closeness centrality after an edge insertion.

The dynamic algorithm iterates over the affected nodes and recomputes their closeness centralities or an upper bound for their closeness centralities. We can expect that running the algorithm on the initial graph produces upper bounds instead of exact values for at least some of the nodes in the graph. In the following, we will present optimizations that allow us to update these upper bounds without running a new pruned BFS from these nodes. Another optimization allows us to skip nodes if they are far away from the edge insertion. If we cannot apply any of the optimizations, the dynamic algorithm simply performs a new pruned BFS starting from the given node. The algorithm that performs the pruned breadth-first searches in the dynamic case is similar to the one we have presented for the static case. It is outlined in Algorithm 9. The only difference to the static algorithm is that the dynamic algorithm also returns the level at which a pruned BFS has been aborted as part of the result tuple.

Algorithm 9: `BFSCut` in the dynamic case

Data: $G = (V, E), v, x_k$

Result: A tuple $(h, \texttt{isExact}, d_{cutoff})$ with $\texttt{isExact} = \texttt{false}$ if h is only an upper bound for the exact harmonic closeness centrality.

```
Create queue $Q$
$Q$.enqueue($v$)
Mark $v$ as visisted
$d \leftarrow 0; h \leftarrow 0; \widetilde{\gamma} \leftarrow 0; nd \leftarrow 0$
while $!Q.isEmpty$ do
    $u \leftarrow$ Q.dequeue()
    if $d(v, u) > d$ then
        $d \leftarrow d + 1$
        $r \leftarrow r(u)$
        $\widetilde{h} \leftarrow h + \frac{\widetilde{\gamma}}{(d+1)\cdot(d+2)} + \frac{r-n_d}{d+2}$
        if $\widetilde{h} \leq x_k$ then
            return $(\widetilde{h}, \texttt{false}, d)$
        end
    end
    forall $w \in N(u)$ do
        if $w$ is not marked as visited then
            Mark $w$ as visited
            $Q$.enqueue($w$)
            $n_d \leftarrow n_d + 1$
            $pred[w] \leftarrow u$
            $d(v, w) \leftarrow d(v, u) + 1$
            $h \leftarrow h + \frac{1}{d(v,w)}$
            if $G$ is directed then
                $\widetilde{\gamma} \leftarrow \widetilde{\gamma} + outdegree(w) - 1$
            else
                $\widetilde{\gamma} \leftarrow \widetilde{\gamma} + outdegree(w)$
            end
        else if $d(v, w) > 1 \wedge pred[u] \neq w$ then
            $\widetilde{h} \leftarrow \widetilde{h} - \frac{1}{d+1} + \frac{1}{d+2}$
            if $\widetilde{h} \leq x_k$ then
                return $(\widetilde{h}, \texttt{false}, d)$
            end
    end
    return $(h, \texttt{true}, d)$
end
```

Skipping far-away nodes

Let w be an arbitrary node in G for which only an upper bound for its closeness centrality is known. We recall that the static algorithm computes the upper bound for the closeness centrality with a pruned breadth-first search. Let $d_{cutoff}(w)$ denote the level at which the previous search was aborted. If the distance of w to the edge modification $d^f_G(w,(u,v))$ is larger than the cutoff level (see Figure 4.4), it is not necessary to run a new pruned BFS. In this case, the search on the modified graph would be completely identical to the previous search on the old graph because the search would be aborted without visiting the modified edge (u, v). Notice that this assumption is only valid if the number of reachable nodes does not change for w due to the edge insertion. If these conditions are met, the dynamic algorithm can mark the old upper bound as valid and skip the expensive recomputation of the upper bound.

Notice that it is possible that the closeness centralities of some nodes will increase after an edge insertion. This can result in a larger closeness centrality of the k-th most central node. In this case, a new pruned BFS could be aborted earlier than before because the new threshold is higher. However, $d_{cutoff}(w)$ will never increase after an edge insertion. Therefore, we will sometimes run a new pruned BFS for a node where the new cutoff level is smaller than the distance to the edge insertion.

If we remove edges from the graph, the closeness centralities of some nodes, potentially including the k-th most central node, will decrease. This could in theory lead to a later cutoff point for a new pruned BFS. As we will see later, the algorithm for edge removals either keeps the old upper bound or recomputes a new (lower) bound. If the bound is recomputed, we get a new cutoff level. Since we remove edges from the graph in this case, the upper bound obtained with a new pruned BFS up until the cutoff level can never be larger than the old one.

Updating bounds of boundary nodes

Let w be a node in G for which only an upper bound is known and whose distance to an edge insertion is exactly equal to the cutoff level of the BFS. We then call w *boundary node.* Without loss of generality, we assume that u is always the node closer to the chosen node w and therefore that $d(w, u) = d_{cutoff}(w)$. This scenario is shown in Figure 4.5.

We will now show that the level difference between u and v was at least 2 before the edge insertion. In consequence, there is one additional node at distance $d_{cutoff}(w) + 1$ after the edge insertion that was previously assumed to be at least at distance $d_{cutoff}(w) + 2$.

Proposition 4.1.

$$d_G(w, u) - d_G(w, v) \geq 2$$

Proof. If the distance in the old graph from w to u was the same as the distance from w to v, w would not have been affected by the edge insertion. This is the case because a new path between w and v using the edge (u, v) will always have length $d(w, u) + 1$. This is contradicts the assumption that $d(w, u) = d(w, v)$.

Secondly, if $d_G(w, u) - d_G(w, v) = 1$, then w would also not have been affected by the edge insertion. Similar to the first case, a potential new shortest path between w and v using the edge (u, v) will always have the length $d(w, u) + 1$. Therefore, this new shortest path will at best be as short as an already known shortest path in the old graph. Since the distance between w and v is not decreased by the inserted edge, w would not be affected by the edge insertion. This is again a contradiction of our initial assumption that w is affected. □

The proposition can also be written as

$$d_G(w, v) \geq d_G(w, u) + 2 = d_{cutoff}(w) + 2. \tag{4.6}$$

We recall that Equation 3.6 is used to compute the upper bounds for the closeness centrality of a node during the pruned BFS starting from that node. Using Proposition 4.1, we can analyze how both u and v contribute to both the old and the new upper bound. Since we assume that $d(w, u) = d_{cutoff}(w)$, the contribution of u to $\widetilde{h}(w)$ is $\frac{1}{d_{cutoff}(w)}$ and will not change due to the edge insertion.

With $d(w, v) \geq d_{cutoff}(w) + 2$, we can deduce that the original contribution of v to $\widetilde{h}(w)$ was $\frac{1}{d_{cutoff}(w)+2}$. Since we insert (u, v) into the graph, the new distance between w and v will be $d(w, u) + 1$. Thus, the new contribution of v to $\widetilde{h}(w)$ is $\frac{1}{d_{cutoff}(w)+1}$. In summary, we get

$$\widetilde{h}_{new}(w) = \widetilde{h}_{old}(w) - \frac{1}{d_{cutoff}(w) + 2} + \frac{1}{d_{cutoff}(w) + 1}. \tag{4.7}$$

This allows us to update the upper bound for the closeness centrality of w cheaply and without a new pruned BFS if the updated bound is smaller than the k-th largest closeness centrality. Notice that this optimization can only be applied if the number of reachable nodes from w does not increase due to the edge insertion.

Algorithm 10: Recomputation of the k most central nodes after an edge insertion.

```
Data: $G = (V, E), (u, v) \notin E$
Result: A list with the $k$ nodes with the highest closeness
$r_{old}(v) \leftarrow r(v) \quad \forall v \in V$
Update $r(v) \quad \forall v \in V$
Compute the set of affected nodes $A(u, v)$
$d^f_G \leftarrow$ array with the distances to the edge modification for each affected node
$x_k \leftarrow$ Top.getMin()
forall $w \in$ Top do
    if $w \in A(u, v)$ then
        Top.remove($w$)
    end
end
forall $w \in A(u, v)$ do
    /* Initially, we assume that we need a new pruned BFS for this node */
    newBfs $\leftarrow$ true
    if $d_{cutoff}(w) < d^f_G(w) \wedge$ !isExact$(w) \wedge r_{old}(w) = r(w)$ then
        /* The inserted edge is beyond the cutoff threshold of the original
           pruned BFS from $w$. We do not need to do anything to recompute
           $h(w)$ in this case. */
        newBfs $\leftarrow$ false
    else if $d^f_G(w) = d_{cutoff}(w) \wedge$!isExact$(w) \wedge r_{old}(w) = r(w)$ then
        /* $w$ is a boundary node and $h(w)$ can be updated cheaply with
           Equation 4.7. */
        $h(w) \leftarrow h(w) - \frac{1}{d_{cutoff}(w)+2} + \frac{1}{d_{cutoff}(w)+1}$
        if $h(w) < x_k$ then
            newBfs $\leftarrow$ false
        end
    if newBfs then
        /* No optimization is applicable, we have to perform a new pruned
           BFS- */
        $(h,$ isExact$, d_{cutoff}) \leftarrow$ BFSCut$(w, x_k)$
        $h(w) \leftarrow h$
        isExact$(v) \leftarrow$ isExact
        $d_{cutoff}(w) \leftarrow d_{cutoff}$
    end
    if $isExact \wedge h(w) > x_k$ then
        Top.insert($h$, $w$)
        if Top.size() $> k$ then
            Top.removeMin()
        end
        if Top.size() $= k$ then
            $x_k \leftarrow$ Top.getMin()
        end
    end
end
```

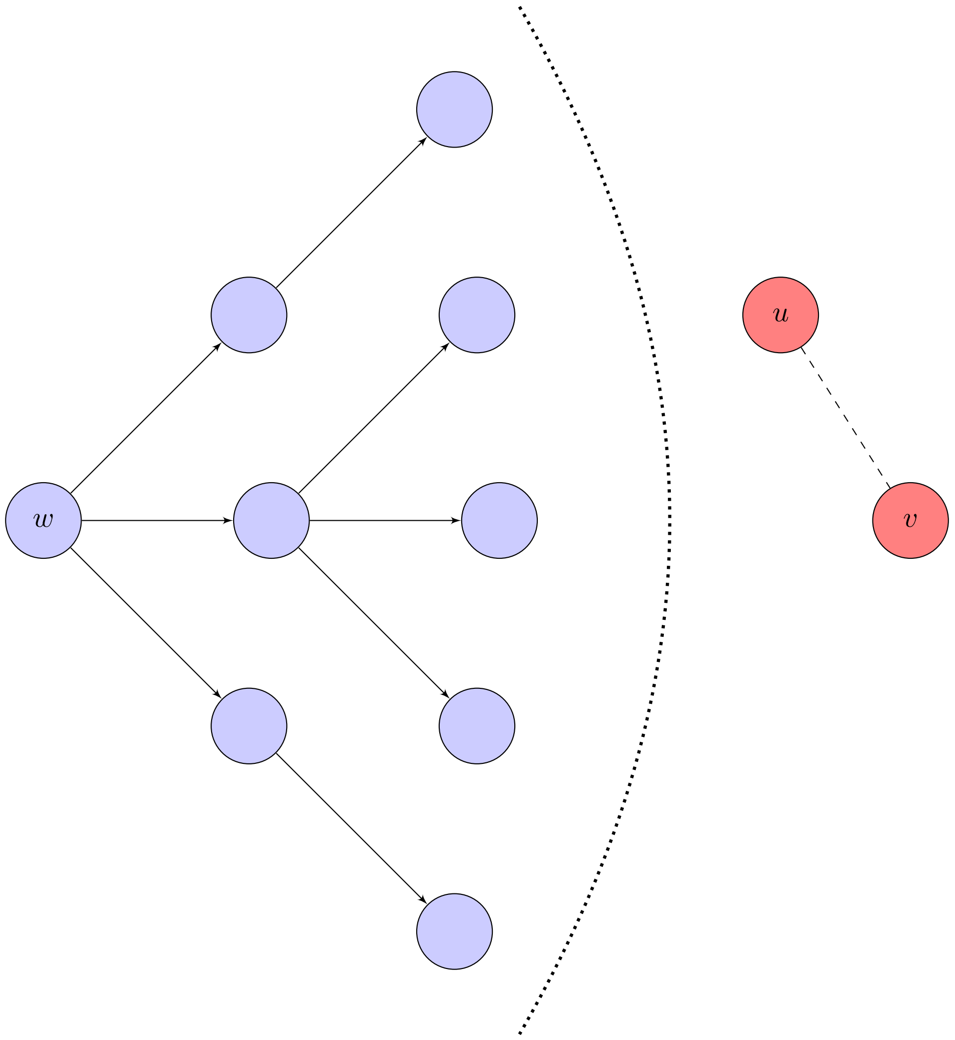

Figure 4.4: Skipping far-way nodes
The dotted line divides the nodes which were visited during the pruned BFS from w from the nodes whose distance was only estimated with a lower bound. If the modified edge is beyond the dotted line, it does not reduce the distance of w to any node for which an exact distance was computed with the pruned BFS.

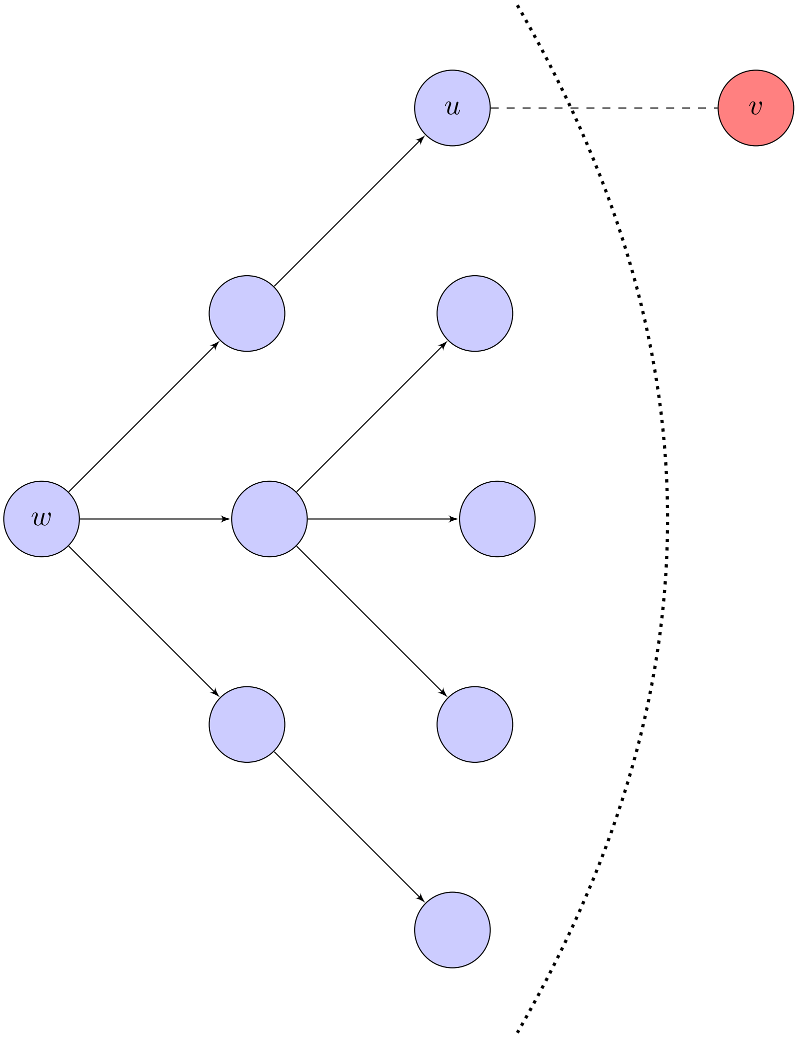

Figure 4.5: Improving upper bounds without an additional BFS
The dotted line divides the nodes which were visited during the pruned BFS from w from the nodes whose distance was only estimated with a lower bound. If u is exactly on the last level before the boundary and v is beyond the boundary, the impact of that edge insertion can be computed without performing a new pruned BFS.

4.3.2.4 Recomputing closeness centralities after edge removals

As we have already discussed earlier, the upper bounds of affected nodes remain valid after an edge is removed from the graph. Algorithm 11 outlines the steps necessary to update the k most central nodes and their closeness centralities after an edge removal. As in the edge insertion case, the algorithm first computes the set of affected nodes. Since the closeness centralities of affected nodes can only decrease, existing upper bounds stay valid and exact closeness centralities become upper bounds. Therefore, we mark the current closeness centrality of each affected node as an upper bound in Line 4. The algorithm starts with an empty priority queue `Top` which manages the k most central nodes (Line 7).

In contrast to the algorithm for edge insertions, the algorithm for edge removals iterates over all the nodes in the graph in decreasing order of their previous closeness centrality (Line 8). If `Top` already contains k nodes and the closeness centrality of the k-th node is larger than the old closeness centrality or upper bound of w, the loop can be aborted (Line 9). This is possible since edge removals can only reduce the closeness centralities of nodes in the graph. Therefore, there cannot be any nodes that “jump the queue” as in the case of an edge insertion.

If the there is an exact value for the closeness centrality of w and, it can be added to `Top` immediately and it is not necessary to start a new pruned BFS. Otherwise, the algorithm executes a pruned BFS to obtain a new upper bound or the exact closeness centrality for w. If the BFS yields the exact closeness centrality, w is added to `Top`.

Algorithm 11: Recomputation of the k most central nodes after an edge removal.

```
Data: $G = (V, E), (u, v) \notin E$
Result: A list with the $k$ nodes with the highest closeness
Update $r(v) \quad \forall v \in V$
Compute the set of affected nodes $A(u, v)$
foreach $w \in A(u, v)$ do
    isExact[$w$] $\leftarrow$ false
end
$x_k \leftarrow 0$
Top $\leftarrow$ empty priority queue
forall $w \in V$ in decreasing order of $\widetilde{h}_{old}(w)$ do
    if Top.size() $= k \wedge h(w) < x_k$ then
        break;
    end
    if !isExact[$w$] then
        Top.insert($h(w)$, $w$)
        if Top.size() $> k$ then
            Top.removeMin()
        end
        if Top.size() $= k$ then
            $x_k \leftarrow$ Top.getMin()
        end
        continue;
    end
    /* Recompute the closeness centrality of the node */
    $(h, \text{isExact}, d_{cutoff}) \leftarrow$ BFSCut($w, x_k$)
    $h(w) \leftarrow h$
    isExact($v$) $\leftarrow$ isExact
    $d_{cutoff}(w) \leftarrow d_{cutoff}$
    if $isExact \wedge h > x_k$ then
        Top.insert($h$, $w$)
        if Top.size() $> k$ then
            Top.removeMin()
        end
        if Top.size() $= k$ then
            $x_k \leftarrow$ Top.getMin()
        end
    end
end
```

4.3.3 Dynamic closeness centrality in networks with large diameters

We recall that the algorithm for complex networks performs a pruned BFS from every single node in the graph. On the contrary, the algorithm for street networks always performs a complete BFS for some nodes in order to compute upper bounds for all the other nodes in the graph (Section 3.2.5, Algorithm 4). Since there is either a complete BFS or no BFS at all for each node, there is no cutoff level that could be used to skip far-away nodes after an edge insertion or to update the upper bounds of boundary nodes cheaply. However, it is possible to compute an upper bound for the *improvement* of the closeness centrality of each node after an edge insertion. In the case of edge removals, we use the same algorithm for street networks as we do for complex networks (see Section 4.3.2.4).

4.3.3.1 Level-based improvement bounds

Let (u, v) denote an edge that is inserted into a graph G to create G'. We will at first only consider directed graphs, but the optimization can be easily adapted to undirected graphs by executing the same steps for both u and v. Let $S_x = \{t : d_{G'}(x, t) < d_G(x, t)\}$ denote the set of *affected sink nodes* of x, i.e. the set of endpoints of paths starting in x that are shorter in G' than in G. Let $\Phi^i_G = \{t : d_G(u, t) = i\}$ denote the set of nodes with distance i from node u in G, $\Phi_{G'}$ denotes the same set in G'.

As we have already noted earlier, each new shortest path in the graph must contain the edge (u, v). For any new path $x - ... - u - v - ... - t$ the subpath $u - v - ... - t$ is also a new shortest path. On the other hand, there could be a node x which already has a shorter path to at least one of the sink nodes of u that does not use the edge (u, v). Therefore, $S_x \subseteq S_u$ for all $x \in V \setminus \{u\}$.

Definition 4.2. *The improvement for the closeness centrality of u after the insertion of the edge (u, v) is*

$$\begin{aligned} h_{impr}(u) &= h_{new}(u) - h_{old}(u) \\ &= \sum_{t \in S_u} \frac{1}{d_{G'}(u,t)} - \frac{1}{d_G(u,t)} \\ &= \sum_i \frac{1}{i} \cdot \left(\Phi^i_{G'} - \Phi^i_G\right). \end{aligned} \tag{4.8}$$

Definition 4.3. *The improvement of the contribution of the node pair (x, w) to the closeness centrality of x after the insertion of the edge (u, v) is*

$$h_{impr}(x, w) = \frac{1}{d_{G'}(x,w)} - \frac{1}{d_G(x,w)} \tag{4.9}$$

We will now show that $h_{impr}(u)$ is an upper bound for the improvement of the closeness centrality of *all* affected nodes. First, we consider the contribution to the improvement of an affected node x and a corresponding sink node w. We want to show that

$$\begin{aligned} & h_{impr}(u, w) \geq h_{impr}(x, w) \\ \iff\ & \frac{1}{d_{G'}(u,w)} - \frac{1}{d_G(u,w)} \geq \frac{1}{d_{G'}(x,w)} - \frac{1}{d_G(x,w)}. \end{aligned} \tag{4.10}$$

In the old graph, a shortest path between x and w either contains u or it does not. In either case, $d_G(x, w) > d_G(u, w)$ because otherwise x would not be an affected node. Since the new shortest path will contain the edge (u, v), we also know that $d_{G'}(x, w) > d_{G'}(u, w)$. However, we cannot guarantee that $d_G(u, w) < d_G(x, w)$, since there could be a shorter path from x to w in G that does not contain u.

We define $\Delta(u) := d_G(u,w) - d_{G'}(u,w)$ and $\Delta(x) := d_G(x,w) - d_{G'}(x,w)$. We rewrite Equation 4.10:

$$\frac{1}{d_{G'}(u,w)} - \frac{1}{d_{G'}(u,w)+\Delta(u)} \geq \frac{1}{d_{G'}(x,w)} - \frac{1}{d_{G'}(x,w)+\Delta(x)} \tag{4.11}$$

Since it is possible that the shortest path from x to w in G does not contain u, we can state that $\Delta(x) \leq \Delta(u)$. In practical terms, this means that the improvement of the distance of all affected nodes x is at most as large as it is for u. If the shortest path in the old graph contains u, the distance improvement for u and x to w is exactly the same. Otherwise, it is possible that the distance improvement for x is smaller since $d_G(x,w) < d_G(x,u) + d_G(u,w)$.

Since $\Delta(x)$, $\Delta(u)$ and all the distances are larger than 0, we only have to show that the inequality holds for $\Delta(x) \in [0, \Delta(u)]$. For $\Delta(x) = 0$ we get the inequality

$$\begin{aligned}\frac{1}{d_{G'}(u,w)} - \frac{1}{d_{G'}(u,w)+\Delta(u)} &\geq \frac{1}{d_{G'}(x,w)} - \frac{1}{d_{G'}(x,w)+0}\\ \frac{1}{d_{G'}(u,w)} - \frac{1}{d_{G'}(u,w)+\Delta(u)} &\geq 0\end{aligned}$$

which trivially holds. In the other case, we set $\Delta(x) = \Delta(u)$ and get

$$\frac{1}{d_{G'}(u,w)} - \frac{1}{d_{G'}(u,w)+\Delta(u)} \geq \frac{1}{d_{G'}(x,w)} - \frac{1}{d_{G'}(x,w)+\Delta(u)}$$

which also holds if all the terms are positive.

Since we now know that $h_{impr}(x,w) \leq h_{impr}(u,w)$ for any affected node x and a given sink node w, and that $|S_x| \leq |S_u|$, we can conclude that $h_{impr}(x) \leq h_{impr}(u)$ for any affected node x.

We can improve this upper bound for $h_{impr}(x)$ further. We know that the new shortest path between x and w contains the inserted edge (u,v), and thus $d_{G'}(x,w) = d_{G'}(x,u) + d_{G'}(u,w)$. Equation 4.8 can be used to compute the improvement of u. However, this bound is not as tight as it could be for nodes which are further away from the edge insertion than u. For instance, it does make a larger difference for the harmonic closeness centrality if the distance between two nodes changes from 3 to 2 than it does if the distance changes from 8 to 7, since $\frac{1}{2} - \frac{1}{3} > \frac{1}{7} - \frac{1}{8}$. We can adapt Equation 4.8 to provide an upper bound for the improvement for nodes with distance j to the edge insertion:

$$h_{impr,LB}(j) = \sum_i \frac{1}{i+j} \cdot \left(\Phi^i_{G'} \cdot - \Phi^i_G\right). \tag{4.12}$$

Figure 4.6 shows an example how level-based improvement bounds work. We insert an edge between u and v into a directed graph. Sink nodes are marked in blue, affected nodes in a shade of yellow (excluding u, which is also affected). While node u has a new shortest path to each of the sink nodes, there are affected nodes like u_1 and u_3 which already have a shorter path to some of the sink nodes. The further an affected node is from the inserted edge, the smaller is the maximum improvement. This is indicated by a lighter shade of yellow in the figure.

Computing level-based improvement bounds

In order to use level-based improvement bounds, we need to compute Φ^i_G and $\Phi^i_{G'}$. This can be done with two breadth-first searches on G and G' starting in u. On undirected graphs, the algorithm to compute the affected nodes of an edge insertion already computes the distances of all nodes to u in both G and G'. Knowing the distances $d_G(u, w)$ and $d_{G'}(u, w)$ for each node w allows us to count the number of nodes with each distinct distance. In the directed case, we need to to perform an additional forward BFS from u in order to compute the distances and then Φ^i_G and $\Phi^i_{G'}$.

Algorithm 12 outlines the steps to compute the level-based improvement bounds in a directed graph.

Complex networks

We have also integrated this optimization in our dynamic algorithm for edge insertions in complex networks. We add the level-based improvement bound to the old closeness centrality of a node if the other two optimizations, skipping far-away nodes and cheaply updating boundary nodes, cannot be applied. If the resulting upper bound is smaller than the closeness centrality of the k-th most central node, we do not need to run a new pruned BFS for this node.

4.3.3.2 Recomputing closeness centralities after edge insertions

The algorithm that updates closeness centralities in networks with large diameters after edge insertions is based on the static algorithm for the problem which was presented in Section 3.2.5. Algorithm 13 outlines the dynamic algorithm. First, the affected nodes and the maximum possible improvement of the closeness centrality of each affected node are computed (Line 1 and 2). As in the dynamic algorithm for complex networks, affected nodes are removed from the priority queue `Top` which manages the k most central nodes (Line 7). The computed maximum improvement for each node is added in Line 9.

The algorithm then iterates over each affected node w (Line 11). If the current known upper bound in `score` is smaller than the exact closeness centrality of the k-th most central node, the recomputation step can be skipped (Line 14). Otherwise, the algorithm handles w exactly as in the static case. It computes the exact closeness of w (Line 16) and updates the upper bounds of other nodes in the graph if possible (Line 18). At last, the `Top` queue is updated if w has a larger exact closeness centrality than the current k-th most central node.

Algorithm 12: Computation of the level-based improvement bound for each affected node in a directed graph

```
Data: $G = (V, E), (u, v) \notin G$
Result: $h_{impr}(w) \forall w \in A(u, v)$
$G' \leftarrow G \cup \{u, v\}$
$A(u, v) \leftarrow$ AffectedNodes$(u, v)$
$d_G \leftarrow$ BFS from $u$ in $G$
$d_{G'} \leftarrow$ BFS from $u$ in $G'$
$\Delta \leftarrow max\ d^-$
/* Count the number of nodes on each level from $u$ */
for $i \leq \Delta$ do
    $\Phi^i_G \leftarrow |\{w \in V \mid d_G(w) = i\}|$
    $\Phi^i_{G'} \leftarrow |\{w \in V \mid d_{G'}(w) = i\}|$
end
/* Compute the improvement bound for each level */
for $1 \leq i \leq \Delta$ do
    $h_{impr,LB}(i) \leftarrow 0$
    for $1 \leq j \leq \Delta$ do
        $h_{impr,LB}(i) \leftarrow h_{impr,LB}(i) + \frac{1}{i+j} \cdot (\Phi^j_{G'} - \Phi^j_G)$
    end
end
/* Select the correct bound for each affected node based on its distance
   to $u$ */
$d_{reverse} \leftarrow$ reverse BFS from $u$ foreach $w \in A(u, v)$ do
    $h_{impr}(w) \leftarrow h_{impr,LB}(d_{reverse}(w))$
end
```

Algorithm 13: Dynamic recomputation of the k nodes with the highest closeness centrality in networks with large diameter after an edge insertion.

```
Data: $G = (V, E), (u, v) \notin G$
Result: A list with the $k$ nodes with the highest closeness
Compute the set of affected nodes $A(u, v)$
maxImprovement ← level-based improvement for each node with Algorithm 12
Q ← $A(u, v)$, sorted by decreasing previous upper bound for the closeness centrality
score ← array indexed by node ID storing the current upper bounds for all nodes
forall $w \in$ Top do
    if $w \in A(u, v)$ then
        Top.remove($w$)
    end
    score[$w$] ← score[$w$] + maxImprovement[$w$]
end
while $Q$ is not empty do
    $v$ ← Q.extractMax()
    if score[$v$]$\leq$ Top[$k$] then
        continue;
    end
    levelBasedBounds ← updateBounds($v$)
    score[v] ← levelBasedBounds[v]
    forall $w \in V$ do
        if levelBasedBounds[$w$] $<$ score[$w$] then
            score[$w$] ← levelBasedBounds[$w$]
        end
    end
    if score[$v$] $>$ Top[$k$] then
        add $v$ to Top
        sort Top by score and reduce it to at most $k$ elements
    end
    re-order Q according to the new values in score
end
```

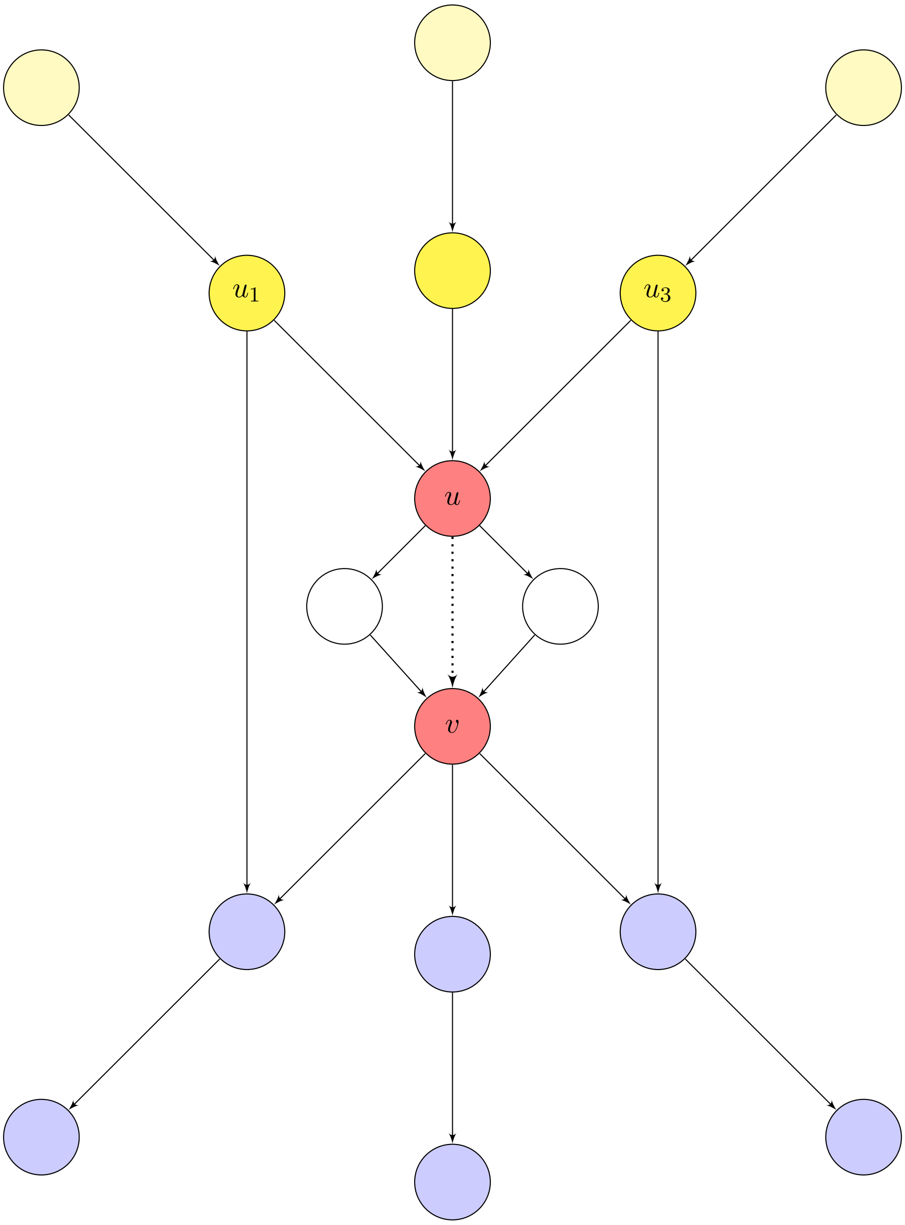

Figure 4.6: Level-based improvement bounds

5. Group closeness

The concept of closeness centrality can be extended to groups of nodes. Sometimes, an application does not require a list of important nodes, but rather a group of nodes that in combination has a large influence. The most important nodes of a graph are often close to each other, so their “spheres of influence” overlap. In other words, the most important nodes might have similar distances to most nodes in the graph. In a group of the most important nodes, the individual contribution of some nodes might therefore be minimal. Instead, it might make more sense to “spread out” the nodes of the group over the whole graph in order to minimize the distances to as many nodes as possible.

Chen et al. call this problem *Maximum Closeness Centrality Group Identification* (MCGI) in [9]. They show the problem to be NP-hard.

5.1 Problem definition

Let $G = (V, E)$ denote an undirected connected graph. Let $S \subseteq V$ denote a group of k nodes. We define the distance of a node v to the group as $d_S(v) := min_{s \in S} d(s, v)$. The *group closeness* of S is defined as

$$c(S) = \frac{|V|}{\sum_{v \in V \setminus S} d_S(v)}. \tag{5.1}$$

The goal of the Maximum Closeness Centrality Group Identification problem is to find the group S^* which maximizes $c(S^*)$ among all groups S^* of size k. Figure 5.1 illustrates the problem. The yellow nodes belong to a group of size 3.

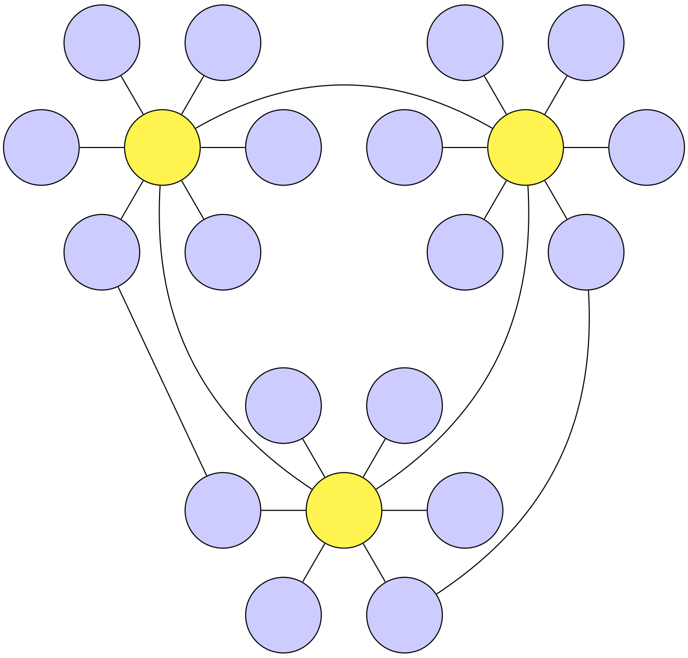

Figure 5.1: Group closeness

5.2 Computing group closeness

The group closeness function $c(S)$ has two properties that can be exploited to design an approximative algorithm for the problem: monotonicity and submodularity.

- Monotonicity: Given any two sets $S, T \subseteq V$ with $S \subseteq T$, a function $f(x)$ is monotonic, if $f(S) \leq f(T)$.
- Submodularity: Given any two sets $S, T \subseteq V$ with $S \subseteq T$ and $v \in V \setminus T$, a function $f(x)$ is submodular, if $f(S \cup \{v\}) - f(S) \geq f(T \cup \{v\}) - f(T)$.

Chen et al. provide a proof that the group closeness function has these properties in [9].

5.2.1 Greedy algorithm by Chen et al.

For any submodular and monotonic function, there is a greedy algorithm that computes a $1 - \frac{1}{e}$ approximation of the optimal solution [19]. Since the group closeness function is such a function, there exists a greedy algorithm that computes a set S with k nodes that is a $1 - \frac{1}{e}$ approximation of the optimal solution S^* [9]. It has a space complexity of $\mathcal{O}(n^2)$ and a time complexity of $\mathcal{O}(n^3 + k \cdot n^2)$. The authors use the Floyd-Warshall algorithm to compute the pairwise distances of all nodes which explains the n^3 term in the time complexity.

The greedy algorithm first computes the exact closeness centralities of all nodes in the graph and chooses the node with the highest closeness centrality as the first group member. After that, it always chooses the node that improves the group closeness centrality the most in each iteration.

Algorithm 14 outlines the greedy algorithm. It maintains a set S which contains the nodes already selected in previous iterations. The matrix M stores the distance $d_{S \cup \{u\}}(v)$ ($v \in V$) for each node u. For instance, if a node u has distance 4 to a node w and the group S has distance 5 to w, $M[u, w]$ would store 4. *Score* is a vector that stores the group closeness if a node u is added to S, i.e. $c(S \cup \{u\})$. Initially each entry in *Score* is set to the closeness centrality of the corresponding node (Line 3). In the main loop (Line 4), the algorithm selects the node with the highest value in *Score* in each iteration for the group S (Line 5). After the node is selected, the distances stored in M are updated to reflect any new shortest paths between the group S and other nodes in the graph (Line 9). At last, the *Score*$[u]$ is updated for each node $u \in V \setminus S$ (Line 13).

5.2.2 Improvements by Bergamini et al.

Bergamini et al. improve on the algorithm by Chen et al. with an algorithm that scales to large real-world networks by reducing the memory requirements and skipping unnecessary operations. Algorithm 5 outlines the necessary steps.

The algorithm by Chen et al. first computes the exact closeness centralities of each node in the graph and then picks the node with the highest closeness centrality as “start node” for

Algorithm 14: Greedy algorithm to approximate the group with maximum group closeness

```
   Data: $G = (V, E)$: a social network, $k$: a positive integer
   Result: $S$: a set of $k$ nodes
 1 $S \leftarrow \emptyset$
 2 $M \leftarrow$ all shortest path distances
 3 $Score \leftarrow \{c(u) \mid u \in V\}$
 4 while $|S| < k$ do
 5     $v \leftarrow arg\,max_{w \in V \setminus S} Score[w]$
 6     $S \leftarrow S \cup \{v\}$
 7     foreach $u \in V \setminus S$ do
 8         foreach $w \in V$ do
 9             if $d(u, w) > d(v, w)$ then
10                 $M[u, w] = d(v, w)$
11             end
12         end
13         $Score[u] \leftarrow c(S \cup \{u\})$
14     end
15 end
16 return $S$
```

the group. However, it is only necessary to find the node with the highest closeness centrality, not the closeness centrality of each node in the graph. For that purpose, Bergamini et al. use the algorithm for Top-k closeness centrality from Section 3.2 to compute the first node of the group (Line 1).

The algorithm stores the distance of the group to each node in the graph in an array d_S. The resulting group closeness centrality of S if a node u was added to S is stored in *Score*. In each iteration of the main loop (Line 5), the algorithm selects one node for S and updates the distances from S to each node (Line 15).

Pruned SSSP

The *Score* for a node u is updated in Line 8 of the algorithm. Since $d_S[w]$ is constant in each iteration, we only need to compute the distances to nodes w where $d(u, w) < d_S[w]$. For this purpose, we use a variant of the single-source-shortest-path (SSSP) algorithm that aborts the search for a subtree if it reaches a node w with $d(u, w) \geq d_S[w]$. Consider a node x in the subtree of w. The shortest path between u and x contains w and therefore $d(u, x) = d(u, w) + d(w, x)$. Since there is a shorter path from S to w, it is clear that $d_S[w] + d(w, x) \leq d(u, w) + d(w, x)$. In this case, w is not added to the search queue.

Submodularity improvement

Another optimization exploits the submodularity of the group closeness function to reduce the number of pruned SSSP invocations in Lines 6-9 of the algorithm. As stated earlier, a function f is submodular, if $f(S \cup \{v\}) - f(S) \geq f(T \cup \{v\}) - f(T)$ with $S \subseteq T$.

Let S_i with $|S_i| = i$ denote the set computed in the i-th iteration of the main loop. Since we only ever add nodes to S and never remove any, we always have $S_i \subset S_{i+1}$. Due to submodularity, we get $c(S_i \cup \{u\}) - c(S_i) \geq c(S_{i+1} \cup \{u\}) - c(S_{i+1})$ for any node $u \in V$. We call $\Delta_i(u) := c(S_i \cup \{u\}) - c(S_i)$ the *marginal gain* of u with respect to S_i. It follows that in each iteration i, the marginal gain of a node u can only decrease. Note that finding the node with the highest marginal gain is equivalent to finding the node with the highest score since $\Delta_i(u) = c(S_i \cup \{u\}) - c(S_i)$ and $c(S_i)$ is not dependent on u.

If we now find a node s with $\Delta_i(s)$, we can skip any node u with $\Delta_{i-1}(u) \leq \Delta_i(s)$ in the i-th iteration (since $\Delta_i(u) \leq \Delta_{i-1}(u) \leq \Delta_i(s)$). For the optimization to be most efficient, it is important to find nodes with high marginal gains early. It is only necessary to keep track of the marginal gain in the previous iteration for each node. The nodes of the graph can be inserted into a priority queue sorted by their previous marginal gain. As soon as the previous marginal gain of an extracted node is smaller than the maximum marginal gain in the current iteration, the loop can be aborted and the node with the maximum marginal gain can be chosen for the group S.

Algorithm 15: Efficient greedy algorithm to approximate the group with maximum group closeness

```
Data: $G = (V, E)$: a graph, $k$: a positive integer
Result: $S$: a set of $k$ nodes
$s_0 \leftarrow$ TopKCloseness(G, 1)
$S \leftarrow \{s_0\}$
SSSP($s_0$)
$d_S[u] \leftarrow d(s_0, u) \quad \forall u \in V$
while $|S| < k$ do
    foreach $u \in V \setminus S$ do
        PrunedSSSP($u$, $d_S$)
        /* $Score[u]$ is set to $c(S \cup \{u\})$ */
        $t \leftarrow \sum_{w \in V \setminus S} min\{d(u,w), d_S[w]\}$
        $Score[u] \leftarrow \frac{(n-|S|-1)}{t}$
    end
    $s \leftarrow arg\,max_{w \in V \setminus S} Score[w]$
    $S \leftarrow S \cup \{s\}$
    SSSP($s$)
    foreach $u \in V$ do
        $d_S[u] \leftarrow min\{d_S[u], d(s,u)\}$
    end
end
return $S$
```

5.3 Incremental group closeness

We will now present an algorithm that recomputes a set S with high group closeness after an edge insertion. Our algorithm is based on the work by Bergamini and Meyerhenke. The algorithm tries to replicate the work of the static algorithm as long as possible with as little computational effort as possible.

Basically, the algorithm is split into two phases which we will call *replication* (Algorithm 16) and *recomputation* (Algorithm 17). There are two data structures that are initialized during the first run of the algorithm on the initial graph. S is the set of nodes selected for the group. Let S_i denote the set of selected nodes after the i-th node s_i is selected. $Score$ is a two-dimensional array that stores for each iteration i and each node u the resulting score $c(S_{i-1} \cup u)$. The algorithm is still correct if $Score$ stores the marginal gain $\Delta_i(u)$ for each iteration i. Both S and $Score$ persist between edge insertions.

The dynamic algorithm for group closeness requires $\mathcal{O}(k \cdot n)$ additional memory because it stores the marginal gains for each of the k iterations instead of only keeping the marginal gains of the last iteration.

5.3.1 Replicating the static algorithm

In the first phase of the algorithm, we try to replicate the results of the static algorithm. First of all, we create a copy S_{old} of the previous group in order to keep track of any changes to the group (Line 2).

To update the first node of S, we use our dynamic algorithm for Top-k closeness centrality from Section 4.3.2.3 (Line 3).

Similar to the dynamic algorithm for Top-k closeness centrality, we compute the set of nodes affected by the edge insertion (Line 7). As long as the set S_i is identical to S_{old} in iteration i, we recompute the marginal gains for these affected nodes and update $Score[i][w]$ ($w \in A_G^f(u,v)$) accordingly. For all unaffected nodes, the existing marginal gains stored in $Score$ become upper bounds (Lines 8 and 15). Consider a node u with marginal gain $\Delta_i(u)$. After an edge insertion, it is possible that a node s_j chosen in any iteration $j < i$ has shorter path to a node x that was previously covered by u. This results in a smaller marginal gain for u.

After the old marginal gains of unaffected nodes are marked as upper bounds and the marginal gains of affected nodes have been computed, our algorithm selects the new node s for S. For this purpose, it selects the node s with the highest marginal gain from $Score[i]$ for the current iteration. If the marginal gain is the exact value, s is added to S (Line 29). Otherwise, the exact marginal gain of s is computed with a pruned SSSP (Line 25). This procedure is repeated until the maximum of $Score[i]$ is an exact marginal gain and not an upper bound. As in the static algorithm, the distances from S to each node in the graph are updated after a new node is added to the group.

The replication phase ends once the algorithm chooses a different group member than before. Since the marginal gain of a node is dependent on the nodes contained in S, a different group S will yield to different marginal gains for the individual node. Once the dynamic algorithm chooses a different node, the marginal gains computed in previous iterations become invalid.

5.3.2 Recomputation

If the group changes after an edge insertion, the dynamic algorithm falls back to an extended variant of the static algorithm as soon as the first node has changed. The recomputation phase is outlined in Algorithm 17. The only difference to the static algorithm is that the scores or marginal gains for each node are stored for each individual iteration and not only stored for the last iteration.

Algorithm 16: Incremental greedy algorithm to approximate the group with maximum group closeness after an edge insertion.

```
Data: $G = (V, E)$: a graph, $k$: a positive integer, $(u, v) \notin E$: the inserted edge
Result: $S$: a set of $k$ nodes
$Score[j][u]$ with $j < k$ and $u \in V$ and $S$ are persistent between two edge insertions
$S_{old} \leftarrow S$
$s_0 \leftarrow$DynTopKCloseness($G$, 1, $(u, v)$)
$S \leftarrow \{s_0\}$
SSSP($s_0$)
$d_S[u] \leftarrow d(s_0, u) \quad \forall u \in V$
$A_G^f(u, v) \leftarrow$AffectedNodes($G$, $(u, v)$)
scoreIsExact[$j$][$u$] $\leftarrow$ false $\quad \forall j < k \quad \forall u \in V$
while $|S| < k$ do
    $i \leftarrow |S|$
    foreach $u \in A_G^f(u, v) \setminus S$ do
        PrunedSSSP($u$, $d_S$)
        /* $Score[i][u]$ is set to $c(S \cup \{u\})$ */
        $t \leftarrow \sum_{w \in V \setminus S} min\{d(u, w), d_S[w]\}$
        $Score[i][u] \leftarrow \frac{(n-|S|-1)}{t}$
        scoreIsExact[$i$][$u$] $\leftarrow$ true
    end
    maximumIsExact $\leftarrow$ false
    do
        $s \leftarrow arg\,max_{w \in V \setminus S} Score[i][w]$
        if scoreIsExact[$i$][$s$] then
            maximumIsExact $\leftarrow$ true
        else
            SSSP($s$, $d_S$)
            /* $Score[i][s]$ is set to $c(S \cup \{u\})$ */
            $t \leftarrow \sum_{w \in V \setminus S} min\{d(s, w), d_S[w]\}$
            $Score[i][s] \leftarrow \frac{(n-|S|-1)}{t}$
            scoreIsExact[$i$][$s$] $\leftarrow$ true
        end
    while !maximumIsExact;
    $S \leftarrow S \cup \{s\}$
    SSSP($s$)
    foreach $u \in V$ do
        $d_S[u] \leftarrow min\{d_S[u], d(s, u)\}$
    end
    /* If a group member has changed, we abort the replication phase */
    if $s \neq S_{old}[i]$ then
        break
    end
end
if $|S| < k$ then
    Fall back to an extended version of the static algorithm (Algorithm 17)
end
return $S$
```

Algorithm 17: Extended version of the main loop of the static algorithm for the fallback case

Data: $G = (V, E)$: a graph, k: a positive integer, $(u, v) \notin E$: the inserted edge

Result: S: a set of k nodes

```
while $|S| < k$ do
    $i \leftarrow |S|$
    foreach $u \in V \setminus S$ do
        PrunedSSSP($u$, $d_S$)
        /* $Score[i][u]$ is set to $c(S \cup \{u\})$ */
        $t \leftarrow \sum_{w \in V \setminus S} min\{d(u, w), d_S[w]\}$
        $Score[i][u] \leftarrow \frac{(n-|S|-1)}{t}$
    end
    $s \leftarrow arg\,max_{w \in V \setminus S} Score[i][w]$
    $S \leftarrow S \cup \{s\}$
    SSSP($s$)
    foreach $u \in V$ do
        $d_S[u] \leftarrow min\{d_S[u], d(s, u)\}$
    end
end
```

6. Implementation in NetworKit

We have implemented the dynamic algorithms for Top-k closeness centrality and group closeness from this thesis in NetworKit. NetworKit is a self-described "open-source toolkit for high-performance network analysis" [23]. Most of the code is written in C++, while a Cython [3] wrapper provides a Python interface. This arrangement allows for implementations with minimal overhead while still providing a simple interface for end users.

6.1 Existing components

NetworKit provides tools to import graphs from files in several formats. There is a unified interface for graphs that provides (parallel) iterators for nodes, edges and neighbors. On top of that, NetworKit provides data structures that are often required for graph algorithms, such as priority queues.

In NetworKit, dynamic algorithms hold a reference to the underlying graph. Once the graph is modified externally, the dynamic algorithm is notified with a `GraphEvent` that contains the necessary information. For edge modifications, the event signals whether it was an edge insertion or an edge removal and contains the pair of nodes that is part of the edge.

There are already implementations of the static algorithms for Top-k closeness centrality in complex networks and street networks in NetworKit. There also is an implementation for the algorithm for group closeness by Bergamini et al. Our dynamic algorithms for these problems are based on these implementations.

6.2 New components

We add four new algorithms to NetworKit. `DynTopClosenessHarmonic` manages the list of the k most central nodes in a dynamic graph. `DynGroupCloseness` approximates the group with the highest group closeness among all groups with k members in a dynamic graph.

The class `AffectedNodes` is used by both dynamic algorithms to compute the set of nodes affected by an edge modification. On top of that, `AffectedNodes` computes an upper bound for the improvement of the closeness centrality for each affected node. Recall from Section 4.3.3.1 that computing these upper bounds only requires knowing the distances to the edge modification for each node. Since computing the set of affected nodes already requires a complete BFS from the affected nodes incident to the modified edge, it is possible to compute the distance to the edge modification for each reachable node.

At last, we added a class `DynConnectedComponents` which keeps track of the connected components of an undirected graph. In the case of an edge insertion, we check whether the two nodes are in the same connected component. If this is the case, we have to do nothing else. If the nodes are in different components, we merge the components. In the case of an edge removal, we simply re-run Algorithm 1. We do not aim to solve the problem of keeping track of (strongly) connected components in dynamic graphs in general, but rather optimize the easiest case.

6.3 Implementation details and parallelism

Top-k closeness centrality

The algorithm for Top-k closeness centrality in complex networks is almost *embarrassingly parallel*. Recall that the algorithm runs a pruned BFS from each (affected) node in the graph. These breadth-first searches are all independent from each other and can be run in parallel. The cutoff level for these searches depends on the closeness centrality x_k of the k-th most central nodes. The k most central nodes are managed by a shared priority queue. Accesses to this priority queue must be protected by a mutex. Since multiple pruned searches are run in parallel, it is possible that the k-th most central node changes while other searches are still running. It makes sense to keep track of x_k for each thread individually and update it with the global value after each local iteration. While it would be possible to share x_k between all threads, this would introduce a lot of synchronization overhead. In practice, x_k rarely changes by a huge amount and thus the gains of updating x_k “live” are negligible. As a result, the parallel implementation usually computes the exact closeness centrality and tighter upper bounds for more nodes than a single-threaded implementation because x_k is only updated roughly every n iterations (with n being the number of threads).

The nodes for which a pruned BFS has yet to be run are also managed by a priority queue. In our implementation, we sort nodes by their degree for the run on the initial graph. After edge modifications, we build a queue of affected nodes sorted by their previous closeness centrality (upper bound). These heuristics aim to process nodes with a potentially high closeness centrality first. The higher the closeness centralities of the early nodes, the larger x_k will be and the earlier the breadth-first searches of less-central nodes can be cut off. The algorithm for large diameters actually requires reordering the priority queue when a tighter upper bound is obtained for any node. In any case, the priority queue is shared among all threads and accesses to it have to be synchronized with a mutex.

Additionally, our algorithm keeps track of the current upper bounds and exact closeness centralities for each node in a global vector. It also keeps track of the cutoff level of the last pruned BFS executed for each node and whether the stored value is only an upper bound for the closeness centrality. In the algorithm for complex networks, one node is only processed by one corresponding thread. Therefore, synchronization is not required when accessing these shared data structures. The algorithm for networks with large diameters, however, can obtain upper bounds for multiple nodes with only one BFS. In this case, we protect accesses to any of these data structures with a single mutex in order to ensure consistency.

Group closeness

Recall that the algorithm that approximates the group with the highest group closeness computes $\Delta_i(u) = c(S_{i-1} \cup \{u\})$ for $u \in V \setminus S_{i-1}$ in each iteration i. For the dynamic algorithm, we need to keep a list of all nodes u sorted by $\Delta_i(u)$ for each iteration. In case $\Delta_i(u)$ is recomputed after an edge insertion, we need to update this list for all iterations. We use k priority queues, one for each iteration.

We first compute $\Delta_i(u)$ in parallel and store the results locally. After all the nodes are processed, we update the priority queue for the corresponding iteration with the new values. After an edge insertion, we only need to recompute $\Delta_i(u)$ for unaffected nodes u if u is actually the first node in the priority queue, that is the node with the largest improvement among all nodes. Our algorithm extracts nodes from the priority queue, recomputes their exact improvement if necessary and then adds the node back to the queue. This is repeated until the exact value for $\Delta_i(u)$ is known for the first node in the queue.

Affected nodes

When computing the set of affected nodes of an edge modification (u, v) in undirected graphs, the breadth-first searches starting in u and v are independent from each other and can also be run in parallel.

7. Experimental evaluation

In this chapter, we want to evaluate our dynamic algorithms for Top-k closeness centrality and group closeness.

7.1 Graphs

We evaluate our dynamic algorithms on complex networks from the *Stanford Large Network Dataset Collection* [18] and on street networks from several countries. We have generated the undirected street networks from the directed source file by assuming directed edges to be undirected and then filtering out duplicate edges.

Graph	Nodes	Edges
facebook_combined	4039	88234
ca-GrQc	5242	14496
as20000102	6474	13895
ca-HepTh	9877	25998
oregon1_010526	11174	23409
oregon2_010526	11461	32730
ca-HepPh	12008	118521
ca-AstroPh	18772	198110
ca-CondMat	23133	93497
as-caida20071105	26475	53381
email-Enron	36692	183831
loc-brightkite_edges	58228	214078
loc-gowalla_edges	196591	950327
com-dblp	317080	1049866
com-amazon	334863	925872
com-youtube	1134890	2987624
as-skitter	1696415	11095298

Table 7.1: Undirected complex networks

Graph	Nodes	Edges
wiki-Vote	7115	103689
cit-HepTh	27770	352807
cit-HepPh	34546	421578
p2p-Gnutella31	62586	147892
soc-Epinions1	75879	508837
twitter_combined	81306	1768149
soc-Slashdot0902	82168	948464
email-EuAll	265214	420045
web-Stanford	281903	2312497
web-NotreDame	325729	1497134
web-Google	875713	5105039
wiki-Talk	2394385	5021410
cit-Patents	3774768	16518948

Table 7.2: Directed complex networks and web graphs

Graph	Nodes	Edges	Diameter
faroe-islands	31097	31974	1464
liechtenstein	54972	56616	2176
isle-of-man	61082	63793	1089
malta	91188	101437	591
belize	96977	103198	3242
azores	237174	243185	1804

Table 7.3: Street networks

7.2 Dynamic Top-k closeness centrality

First, we want to evaluate our dynamic algorithm for Top-k closeness centrality. We compare our algorithm to the naïve approach of simply running the static algorithm again after the graph has changed. We have made our algorithms available through the Python interface of NetworKit and use Python scripts to collect the data for our experiments. For these experiments, we have used two identical machines with 48 AMD Opteron 6172 cores (clocked at 2.10 GHz) and 256 GB of main memory.

7.2.1 Methodology

Our experiments cover several “dimensions”:

- Edge insertions and edge removals
- Directed and undirected graphs
- Complex networks and street networks
- Different values for k

Edge insertions

For our edge insertion experiments, we select 100 edges at random from the base graph and remove them. The resulting graph is the initial graph on which we initialize the dynamic algorithm. We then add back the removed edges one-by-one and measure the update time of the dynamic algorithm. We also create a separate instance of the algorithm and measure the time it requires to run the static algorithm after each edge insertion as a reference. With the update time and the reference time, we can compute the speedup of the dynamic algorithm compared to the static algorithm. In the process, we also collect information about the effectiveness of the various optimization strategies for complex networks we have proposed in Section 4.3.2. In all our experiments, we compare our dynamic algorithm with the static one with $k \in \{1, 10, 100\}$.

Edge removals

To measure the speedup of our dynamic algorithm for edge removals, we start with the base graph. We select 100 edges at random and remove them one-by-one. After each edge removal, we perform the update of the dynamic algorithm and measure the time it takes. We also measure the reference time of the static algorithm and compute the speedup of the dynamic algorithm.

7.2.2 Edge insertions in complex networks

The results for edge insertions in undirected complex networks are shown in Table 7.6, the results for directed complex networks in Table 7.7. We always yield a double-digit speedup in the geometric mean in undirected networks with $k \in \{1, 10, 100\}$.

We achieve smaller speedups on directed networks. One reason for this is the computation of the number of reachable nodes. In the undirected case, computing the number of reachable nodes does not take a significant amount of time in the first place. On top of that, we have a simple optimization for edge insertions which is faster than recomputing the connected components of a graph from scratch. However, in the directed case, computing the upper bound for the number of reachable nodes takes a significant amount of time.

For instance, it takes about 2 seconds to do so on the graph `wiki-Talk`. Since we have not implemented any optimizations for directed graphs, this effectively results in a lower bound for the update time in these graphs, even if almost no node is actually affected by an edge insertion.

Update time distribution

Figures 7.1 and 7.2 contain the update times for all 100 inserted edges in the graphs `as20000102` and `com-amazon`. In general, the graphs for the undirected complex network we have tested look quite similar. In each case, every update of the dynamic algorithm is faster than running the static algorithm again. For many nodes, the dynamic update is significantly faster. However, there are pathological cases where the update time is close to the static reference time for many graphs.

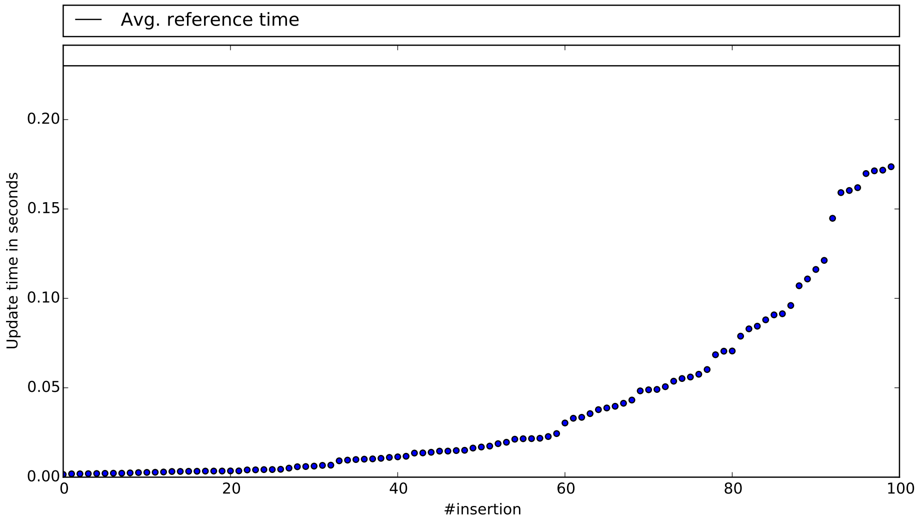

Figure 7.1: Update times of 100 random edge insertions on `as20000102` with $k = 10$

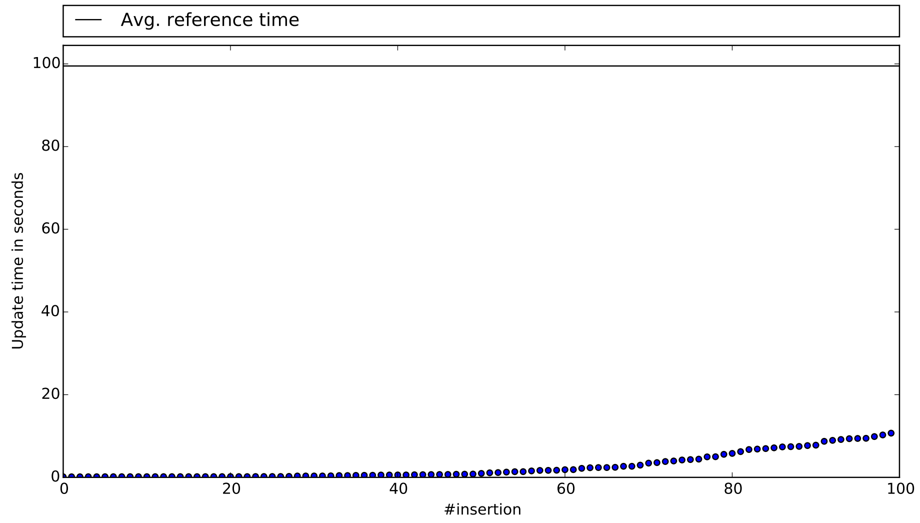

Figure 7.2: Update times of 100 random edge insertions on `com-amazon` with $k = 100$

Optimizations

We now want to examine the effect of the various optimizations we have proposed for the dynamic algorithm for complex networks. Table 7.4 contains the average number of affected nodes and the percentage of nodes skipped due to each of the optimizations in undirected complex networks. In any case, 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 level-based improvement bounds, we need to run a new pruned BFS for less than 1% of the affected nodes for most of the graphs.

Table 7.5 contains the same analysis for directed complex networks. In contrast to the undirected complex networks, nodes cannot be skipped entirely or updated cheaply as often. Edge insertions more often change the upper bound for the number of reachable nodes in these graphs. Therefore, only the level-based improvement bounds are applicable in these cases. Depending on the graph, we also need to run a new pruned BFS for a higher percentage of the affected nodes. However, the number of affected nodes is sometimes extremely small compared to the number of nodes in the graph. For instance, only 75 of the 3774768 nodes in the graph `cit-Patents` are affected by an edge insertion on average. In this specific case, the bottleneck of recomputing the upper bounds for the number of reachable nodes in directed graphs is very apparent.

Graph	affected	% affected	far-away	boundary	impr. bound	pruned BFS
as-caida20071105	4871	18.399%	78.020%	9.426%	11.993%	0.561%
as-skitter	85367	5.032%	94.076%	4.993%	0.149%	0.782%
as20000102	1300	20.087%	75.659%	15.294%	7.199%	1.849%
ca-AstroPh	488	2.599%	85.000%	3.599%	1.881%	9.520%
ca-CondMat	2036	8.801%	91.423%	5.569%	1.810%	1.198%
ca-GrQc	631	12.040%	69.443%	11.565%	15.326%	3.666%
ca-HepPh	408	3.397%	89.808%	3.299%	2.228%	4.665%
ca-HepTh	1483	15.014%	72.514%	13.195%	11.862%	2.429%
com-amazon	76651	22.890%	90.137%	4.948%	4.546%	0.369%
com-dblp	55686	17.562%	96.502%	2.930%	0.432%	0.136%
com-youtube	210217	18.523%	58.377%	3.587%	37.940%	0.096%
email-Enron	1779	4.849%	91.916%	6.374%	1.366%	0.344%
facebook_combined	153	3.782%	53.577%	15.101%	24.723%	6.598%
loc-brightkite_edges	4198	7.209%	83.519%	13.770%	1.514%	1.196%
loc-gowalla_edges	14711	7.483%	75.705%	6.343%	15.188%	2.764%
oregon1_010526	1984	17.755%	73.074%	22.725%	2.719%	1.482%
oregon2_010526	1306	11.398%	75.920%	20.450%	2.636%	0.994%

Table 7.4: Impact of optimizations in undirected complex networks
Analysis of the various optimization strategies, averaged over 100 random edge insertions in undirected complex networks with $k = 10$. The column "affected" contains the average number of affected nodes, "far-away" the number of nodes skipped because their previous cutoff level was smaller than the distance to the edge insertion, "boundary" the number of boundary nodes and "impr. bound" the number of nodes that could be updated with their maximum improvement without surpassing the k-th most central node.

Graph	affected	% affected	far-away	boundary	impr. bound	pruned BFS
cit-HepPh	724	2.096%	84.854%	3.066%	3.745%	8.335%
cit-HepTh	406	1.463%	31.217%	5.163%	42.573%	21.047%
cit-Patents	75	0.002%	0.000%	0.000%	99.933%	0.067%
email-EuAll	9375	3.535%	19.879%	1.057%	79.057%	0.007%
p2p-Gnutella31	3781	6.042%	47.083%	8.261%	32.584%	12.073%
soc-Epinions1	1696	2.235%	31.853%	1.717%	66.078%	0.351%
soc-Slashdot0902	3986	4.851%	25.621%	2.715%	71.430%	0.235%
twitter_combined	904	1.112%	87.040%	9.103%	1.596%	2.261%
web-Google	48512	5.540%	62.906%	0.735%	36.342%	0.017%
web-NotreDame	4924	1.512%	58.379%	2.108%	39.496%	0.017%
web-Stanford	14683	5.208%	53.691%	5.562%	37.511%	3.236%
wiki-Talk	3512	0.147%	59.850%	2.824%	37.317%	0.009%
wiki-Vote	19	0.269%	48.930%	40.157%	5.796%	5.117%

Table 7.5: Impact of optimizations in directed complex networks
Analysis of the various optimization strategies, averaged over 100 random edge insertions in directed complex networks with $k = 10$.

Affected nodes

We now want to analyze the number of affected nodes after edge modifications. Table 7.4 suggests that, on average, a clear majority of the nodes ($> 80\%$) in the tested graphs are unaffected by a random edge modification. In directed graphs (Table 7.5), fewer than 6% of all nodes are affected by an edge insertion on average for all graphs.

As an example, we have computed the number of affected nodes for each edge in the undirected graph `oregon1_010526`. For this purpose, we remove each edge individually from the graph, compute the number of affected nodes and then add the edge back. Figure 7.3 contains a histogram of the data. The median number of affected nodes is 1208 for this graph. The median is represented by the dashed line in the histogram. We can see that for over half of the edges in the graph, only a tenth of all nodes will be affected if the edge is modified. However, there is also a large peak at the other end of the spectrum, where a considerable number of edges affects almost every node in the graph if they are modified. Most of these node have only a single neighbor and removing the edge between them splits the connected component, therefore affecting every node in the connected component.

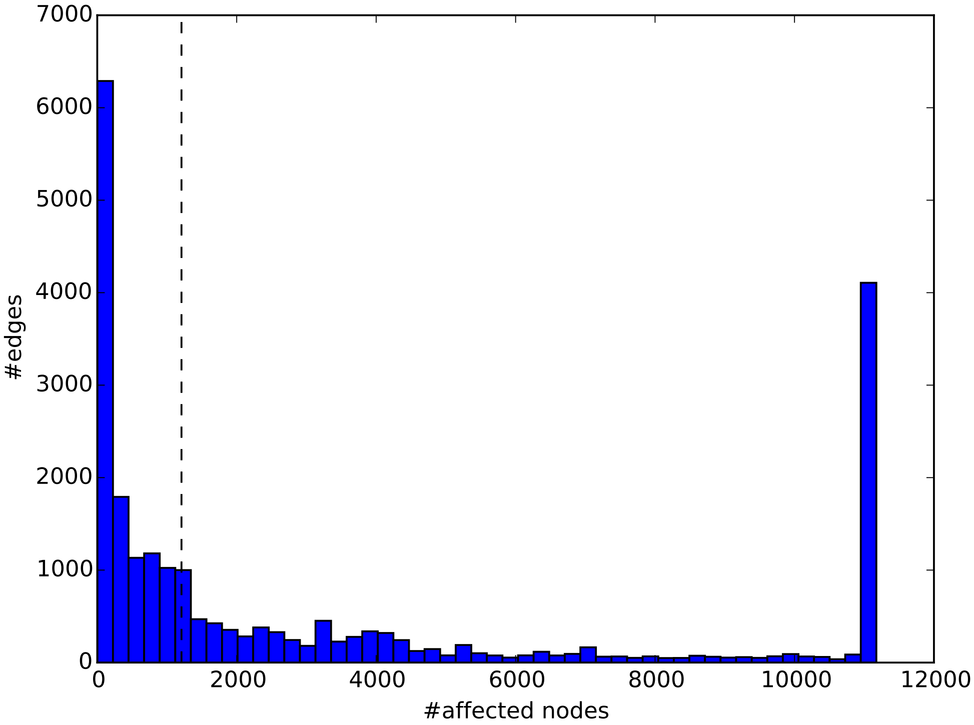

Figure 7.3: Distribution of the number of affected nodes in `oregon1_010526`

Graph	k = 1				k = 10				k = 100			
	ref [s]	gmean	min	max	ref [s]	gmean	min	max	ref [s]	gmean	min	max
facebook_combined	0.132	52.354	1.729	92.226	0.13	52.961	1.671	87.422	0.13	58.617	1.519	78.683
ca-GrQc	0.182	31.932	1.909	223.006	0.187	34.259	1.919	212.018	0.18	32.648	1.905	164.15
as20000102	0.225	18.163	1.34	175.157	0.23	14.219	1.182	163.056	0.22	17.367	1.349	149.471
ca-HepTh	0.335	27.743	1.56	218.97	0.341	20.706	1.483	298.27	0.33	24.422	1.482	258.957
oregon1_010526	0.384	16.217	1.397	154.007	0.397	14.997	1.319	152.956	0.388	18.393	1.428	169.635
oregon2_010526	0.408	31.605	1.391	178.563	0.412	39.461	1.361	174.984	0.397	33.568	1.312	148.284
ca-HepPh	0.445	60.461	2.767	331.736	0.46	60.483	1.932	123.843	0.449	58.843	1.782	114.023
ca-AstroPh	0.675	43.052	1.376	104.393	0.703	46.843	4.032	97.165	0.762	44.18	1.992	382.766
ca-CondMat	0.804	30.455	1.473	263.041	0.821	28.09	1.455	304.707	0.81	38.934	1.495	247.589
as-caida20071105	0.955	17.172	1.283	122.889	0.967	20.043	1.451	123.456	0.933	18.087	1.236	126.564
email-Enron	1.302	56.122	2.613	303.076	1.328	49.113	1.506	126.504	1.403	58.462	1.471	278.372
loc-brightkite_edges	2.051	44.699	1.389	116.031	1.982	32.469	1.398	109.521	2.075	35.809	1.536	118.759
loc-gowalla_edges	7.023	52.144	1.277	111.11	8.027	53.864	1.458	112.168	22.242	107.676	1.99	322.198
com-dblp	11.601	17.413	1.344	112.883	12.568	31.8	1.332	117.433	17.569	33.821	1.784	151.297
com-amazon	27.709	38.652	2.561	214.115	33.834	45.213	2.89	220.926	99.487	88.985	9.37	726.038
com-youtube	29.568	20.466	1.127	84.857	88.891	68.17	2.251	190.132	124.343	68.219	3.025	290.741
as-skitter	242.991	114.884	2.35	291.879	279.276	146.953	4.849	368.249	320.279	159.128	4.519	418.5
(geometric) mean	19.223	34.231	1.631	166.282	25.327	37.686	1.798	160.411	34.823	43.355	1.932	207.488

Table 7.6: Update times for 100 random edge insertions with $k \in \{1, 10, 100\}$ in undirected complex networks
The column “ref” contains the average reference time in seconds, “gmean” the geometric mean of the achieved speedups, “min” and “max” the minimum and maximum speedup.

Graph	k = 1				k = 10				k = 100			
	ref [s]	gmean	min	max	ref [s]	gmean	min	max	ref [s]	gmean	min	max
wiki-Vote	0.216	19.041	11.95	22.189	0.214	18.465	12.556	21.38	0.215	18	8.984	21.75
cit-HepTh	0.763	8.025	1.933	10.882	0.915	11.042	1.606	14.45	0.995	11.368	4.016	14.837
cit-HepPh	0.94	9.679	1.085	13.301	0.991	10.246	1.281	14.191	1.141	11.076	1.277	15.681
p2p-Gnutella31	1.941	12.975	3.905	23.593	1.772	11.148	3.713	18.847	2.146	11.46	3.046	21.722
soc-Epinions1	2.273	18.292	1.279	30.288	2.311	15.941	1.442	22.304	2.419	16.37	1.215	22.423
soc-Slashdot0902	2.566	12.678	1.038	15.689	2.762	15.008	1.208	22.42	3.84	17.313	1.585	24.955
twitter_combined	2.686	11.931	1.262	15.763	2.67	12.052	1.302	14.759	4.284	17.021	1.716	20.459
email-EuAll	7.93	22.904	1.221	34.001	8.02	24.681	1.335	38.4	8.13	24.459	1.177	40.373
web-NotreDame	10.214	22.515	4.144	36.852	9.952	23.611	4.156	32.553	13.374	32.776	5.658	48.457
wiki-Talk	23.181	5.921	2.786	31.057	99.18	27.564	17.174	31.612	143.048	37.472	5.996	43.777
web-Stanford	23.433	39.419	1.319	93.015	34.845	64.707	1.277	131.088	54.918	116.648	7.278	207.119
web-Google	26.529	12.369	1.13	19.26	39.589	17.832	1.584	28.016	108.569	40.943	1.588	84.751
cit-Patents	151.871	9.931	8.204	11.597	153.478	9.483	7.457	11.18	159.299	10.498	8.188	13.225
(geometric) mean	19.58	13.988	2.2	22.835	27.438	17.231	2.647	24.225	38.644	21.326	3.022	31.058

Table 7.7: Update times for 100 random edge insertions with $k \in \{1, 10, 100\}$ in directed complex networks
The column “ref” contains the average reference time in seconds, “gmean” the geometric mean of the achieved speedups, “min” and “max” the minimum and maximum speedup.

7.2.3 Edge insertions in street networks

We now want to evaluate our dynamic algorithm on the six street networks from Table 7.3. We run a similar experiment for street networks as we have for complex networks. The only difference is that we only compute the reference time for ten of the intermediate graphs. We then calculate an average of these reference times and compute the speedups based on the average reference time. The results are listed in Table 7.10 for the undirected street networks and in Table 7.11 for the directed street networks. We observe that the speedups are generally higher for our street networks than for the complex networks.

One reason for this is the large diameter of street networks. While many nodes might be affected by an edge insertion, many of them can be pretty far away from the edge insertion. We have already seen earlier that edge modifications in close proximity of a node have a larger effect than far-away edge modification. Therefore, the closeness centrality of these far-away nodes will not change by a great amount. As a result, applying the level-based improvement bounds only rarely results in a new upper bound for a node outside the Top-k that is larger than the old k-th largest closeness centrality. It is often enough to apply the level-based improvement bounds for all nodes except the affected nodes among the k most central ones. In Table 7.8 we compare the number of complete breadth-first searches our dynamic algorithm performs with the static algorithm. It is clear that the dynamic algorithm performs a significantly smaller number of searches. This also explains the large speedups we observe.

We also observe higher average speedups in directed street networks than in undirected street networks. The reason for this is the lower number of affected nodes in directed networks. This can be seen when comparing the values in Table 7.8 and Table 7.9.

Graph	affected	|V|	Reference BFS	Dynamic BFS
azores	40900	237174	3863	47
belize	82040	96977	4243	108
faroe-islands	11923	31097	1404	64
isle-of-man	55205	61082	2703	132
liechtenstein	51295	54972	2328	124
malta	56096	91188	5099	113

Table 7.8: Average number of complete breadth-first searches in undirected street networks with $k = 100$

Graph	affected	|V|	Reference BFS	Dynamic BFS
azores	20254	237174	7527	12
belize	35745	96977	5148	43
faroe-islands	9471	31097	2610	54
isle-of-man	28141	61082	3107	63
liechtenstein	23626	54972	3328	50
malta	26248	91188	6134	69

Table 7.9: Average number of complete breadth-first searches in directed street networks with $k = 100$

Graph	k = 1				k = 10				k = 100			
	ref [s]	gmean	min	max	ref [s]	gmean	min	max	ref [s]	gmean	min	max
faroe-islands	3.694	57.555	11.809	1018	3.462	64.882	6.235	923.889	3.475	65.796	10.89	1028.51
liechtenstein	8.965	36.632	14.685	1226.48	8.661	32.673	10.749	742.558	9.444	20.744	7.339	1443.12
isle-of-man	11.262	90.369	16.119	1499.57	11.706	74.057	22.495	282.317	11.571	53.184	14.042	1450.95
malta	13.06	122.5	39.621	1080.39	12.82	118.39	27.176	1046.01	13.603	97.763	23.099	1443.97
belize	20.541	62.072	21.121	1622.32	21.712	50.048	17.789	1661.14	25.035	39.859	11.616	2019.76
azores	27.199	142.13	40.248	326.176	27.247	152.377	55.461	332.258	27.915	141.379	20.682	389.911
(geometric) mean	14.12	76.845	21.329	1011.4	14.268	72.208	18.526	694.093	15.174	58.478	13.564	1161

Table 7.10: Update times for 100 random edge insertions with $k \in \{1, 10, 100\}$ in undirected street networks
The column “ref” contains the average reference time in seconds, “gmean” the geometric mean of the achieved speedups, “min” and “max” the minimum and maximum speedup.

Graph	k = 1				k = 10				k = 100			
	ref [s]	gmean	min	max	ref [s]	gmean	min	max	ref [s]	gmean	min	max
faroe-islands	2.604	157.908	30.16	832.939	2.728	124.944	27.203	967.669	2.866	95.685	20.759	880.479
liechtenstein	8.063	71.494	30.384	1303.71	7.826	82.943	30.561	1426.39	8.538	85.119	26.906	1588.3
isle-of-man	9.379	130.253	42.72	1531.27	9.665	145.912	22.306	1866.73	9.668	104.827	38.813	1512.3
malta	11.947	191.444	27.376	1281.98	12.332	144.149	12.984	1382.22	12.538	156.519	19.632	1091.53
belize	13.906	122.54	40.854	1336.08	15.154	96.587	5.801	1007.36	16.357	120.394	29.523	1581.21
azores	28.644	239.576	110.293	708.208	27.807	228.899	99.023	493.405	28.861	227.81	92.464	493.907
(geometric) mean	12.424	142.191	41.113	1124.05	12.585	129.965	22.741	1099.86	13.138	124.17	32.423	1103.21

Table 7.11: Update times for 100 random edge insertions with $k \in \{1, 10, 100\}$ in directed street networks
The column “ref” contains the average reference time in seconds, “gmean” the geometric mean of the achieved speedups, “min” and “max” the minimum and maximum speedup.

7.2.4 Edge removals in complex networks

At last, we want to evaluate our dynamic algorithm for edge removals. We remove 100 random edges from the graph one-by-one and measure the update times. Due to time constraints, we have only computed the reference time of the static algorithm after every ten edge removals. The results for undirected complex networks are listed in Table 7.12, the results for undirected street networks in Table 7.13 and the results for directed street networks in Table 7.14.

For undirected complex networks, we observe double-digit speedups in the geometric mean on all graphs. For some graphs and some edges, the speedup of the dynamic algorithm can be up to three orders of magnitude.

In street networks, we again observe much larger speedups. Recall that we use the same algorithm for both complex networks and street networks. We iterate over every node in the graph in decreasing order of their previous closeness centrality or upper bound until we reach a node whose old upper bound is smaller than the exact closeness centrality of the k-th most central node. Therefore, we will mostly run complete breadth-first searches for the affected nodes among the k most central nodes. As we have seen in Section 7.2.3, the relative change of the closeness centralities after edge modifications in street networks are rather small. Therefore, there often is little change in the ranking of the nodes. It is often enough to recompute the exact closeness centralities of the affected nodes among the k most central nodes.

Graph	k = 1				k = 10				k = 100			
	ref [s]	gmean	min	max	ref [s]	gmean	min	max	ref [s]	gmean	min	max
facebook_combined	0.112	11.728	3.089	15.198	0.121	12.838	4.341	16.538	0.13	13.023	3.478	17.738
ca-GrQc	0.154	47.108	10.856	921.962	0.157	41.522	8.545	915.638	0.151	33.438	8.024	760.947
as20000102	0.188	33.583	15.427	43.933	0.194	26.907	13.988	46.148	0.183	19.545	11.064	43.828
ca-HepTh	0.282	35.877	11.908	852.242	0.28	30.011	7.964	847.234	0.276	23.076	7.976	784.087
oregon1_010526	0.321	31.173	11.265	40.368	0.32	22.73	9.516	40.944	0.318	19.128	8.211	39.713
oregon2_010526	0.332	28.051	13.129	34.747	0.346	22.474	9.019	36.018	0.334	18.171	10.194	34.048
ca-HepPh	0.386	19.669	3.364	849.561	0.408	19.875	3.684	888.877	0.396	16.635	2.901	816.004
ca-AstroPh	0.568	17.244	4.772	767.619	0.577	17.019	3.945	709.664	0.669	16.377	2.612	829.106
ca-CondMat	0.675	24.785	9.003	737.27	0.711	23.31	7.069	733.503	0.683	18.828	6.552	698.119
as-caida20071105	0.76	22.871	10.245	31.764	0.767	18.889	11.36	32.13	0.757	14.617	9.69	31.569
email-Enron	1.053	23.66	7.163	678.557	1.075	21.952	9.441	645.415	1.173	21.828	8.032	704.578
loc-brightkite_edges	1.674	16.77	8.169	20.462	1.689	13.874	5.888	21.335	1.803	12.232	5.648	22.23
loc-gowalla_edges	5.764	12.352	5.15	15.834	6.421	12.121	5.815	17.186	19.233	21.904	1.703	50.859
com-dblp	9.632	12.491	3.694	18.524	10.192	10.809	3.534	19.029	15.289	11.486	3.136	28.805
com-amazon	31.525	33.324	5.763	57.895	30.694	21.648	2.658	49.403	83.667	42.646	7.778	134.251
com-youtube	37.569	14.913	5.938	20.414	95.859	18.927	1.353	54.741	133.456	20.08	1.529	74.309
as-skitter	336.102	53.084	1.033	84.169	313.04	35.342	1.073	76.443	492.071	39.964	1.08	121.05
(geometric) mean	25.123	23.338	6.418	95.477	27.226	20.437	5.263	101.12	44.152	19.873	4.741	121.101

Table 7.12: Update times for 100 random edge removals with $k \in \{1, 10, 100\}$ in undirected complex networks
The column "ref" contains the average reference time in seconds, "gmean" the geometric mean of the achieved speedups, "min" and "max" the minimum and maximum speedup.

Graph	k = 1				k = 10				k = 100			
	ref [s]	gmean	min	max	ref [s]	gmean	min	max	ref [s]	gmean	min	max
faroe-islands	3	174.525	46.465	2119.15	3.03	176.409	60.66	2415.04	2.991	162.026	37.017	2076.44
liechtenstein	7.842	76.118	50.133	1744.71	7.884	76.006	54.175	2248.93	8.836	73.512	54.79	2950.11
isle-of-man	10.281	84.939	67.748	3055.96	9.977	78.024	64.428	2563.11	10.207	65.256	52.595	2689.05
malta	11.597	93.482	51.771	1543.31	12.172	93.48	51.991	1430.01	12.253	83.264	47.082	1467.43
belize	17.508	119.023	25.744	1950.99	17.523	112.325	20.783	2142.72	19.817	101.253	13.732	2394.16
azores	25.804	220.592	38.487	890.16	25.567	230.051	39.725	1122.2	25.241	209.382	36.414	1138.82
(geometric) mean	12.672	118.504	44.81	1765.5	12.692	116.708	45.682	1905.49	13.224	105.413	36.868	2009.8

Table 7.13: Update times for 100 random edge removals with $k \in \{1, 10, 100\}$ in undirected street networks
The column “ref” contains the average reference time in seconds, “gmean” the geometric mean of the achieved speedups, “min” and “max” the minimum and maximum speedup.

Graph	k = 1				k = 10				k = 100			
	ref [s]	gmean	min	max	ref [s]	gmean	min	max	ref [s]	gmean	min	max
faroe-islands	2.984	307.814	49.063	3755.06	3.05	312.551	54.554	3881.59	3.119	305.6	41.437	3899.65
liechtenstein	8.664	166.661	60.866	3983.57	8.64	174.874	68.587	6251.16	9.812	163.476	55.419	3834.49
isle-of-man	10.859	155.951	63.212	366.018	11.103	173.229	65.429	374.513	11.873	163.354	63.113	402.725
malta	12.527	202.493	49.013	4710.48	12.535	189.714	52.262	4548.08	12.787	174.614	48.63	4333.87
belize	13.146	160.658	55.867	4233.6	13.417	164.816	50.796	4483	14.241	150.264	45.17	4870.38
azores	32.032	565.52	43.012	2162.45	31.112	666.809	44.899	3347.73	30.644	578.381	36.675	3259.14
(geometric) mean	13.369	229.779	53.027	2486.1	13.31	241.302	55.486	2920.32	13.746	223.262	47.63	2730.32

Table 7.14: Update times for 100 random edge removals with $k \in \{1, 10, 100\}$ in directed street networks
The column “ref” contains the average reference time in seconds, “gmean” the geometric mean of the achieved speedups, “min” and “max” the minimum and maximum speedup.

7.3 Group closeness

We want to evaluate our dynamic algorithm for group closeness in this section. The experiment is similar to our experiments with dynamic Top-k closeness centrality. We first remove 100 edges randomly from the graph and compute the initial group on the resulting graph. Then, we add these edges back one-by-one and measure the time it takes to recompute the group. As a reference, we run the static algorithm after every ten edge insertions and compute an average of these runtimes.

The results are listed in Table 7.15 for $k \in \{10, 25, 100\}$. Recall that our algorithm tries to minimize the work it takes to replicate the results of the static algorithm as long as the group is unchanged after the edge insertion. After one node in the group has changed, the algorithm falls back to the static algorithm. We expect smaller speedups for larger group sizes because it becomes more likely that any of the nodes has changed, the larger the group is. Our experimental results confirm this expectation. For $k = 10$, we observe a speedup of about 15 in the geometric mean, while the speedup is only about 8 for $k = 100$ in the geometric mean.

Cutoff points

We have also listed the average “cutoff” point for each graph. This is either the point where the group is completely computed or when the dynamic algorithm falls back to the static algorithm. We observe that the dynamic algorithm almost never falls back to the static algorithm for $k = 10$. It is much more likely that the group changes for $k = 100$ since the average cutoff point is after about 74 group members.

Update time distribution

The distribution of the update times for group closeness looks similar to the distribution for Top-k closeness centrality. This can be seen in Figure 7.4 for $k = 100$ and the graph `com-dblp`. The update times for about a third of inserted edges is absolutely minimal compared to the reference time. Another third of the edges have more or less equal update times on a higher level. In the last third, there is a constant increase. For some edges, the update time is close to the reference time. In these cases, there is a large number of affected nodes and the dynamic algorithm falls back to the static algorithm relatively early.

Iteration times

At last, we want to analyze the time it takes to select the i-th member of the group and compare our dynamic algorithm to the static algorithm. For this purpose, we measure the iteration times during each update operation and compute an average for each iteration. We do the same for each run of the static algorithm to get a reference. We have observed two types of graphs: in some graphs, selecting the first node takes the longest (see Figure 7.5); in other graphs, selecting the second node takes longer (see Figure 7.6). Later iterations are significantly faster than the first two iterations for all the graphs we have tested.

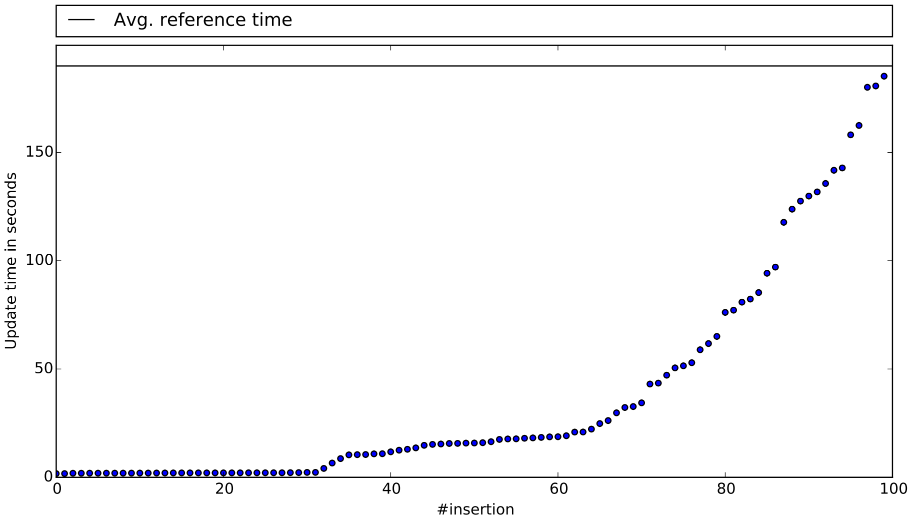

Figure 7.4: Update time distribution for group closeness on the graph `com-dblp` with $k = 100$

Since our dynamic algorithm for Top-k closeness centrality is faster than re-running the static algorithm, the first iteration of our dynamic algorithm for group closeness is also significantly faster than the corresponding static algorithm. Recall that our dynamic algorithm for group closeness only has to recompute the marginal gains for affected nodes in later iterations as long as the group has not changed. The old marginal gains of unaffected nodes become upper bounds and do not have to be recomputed unless the node is not the node with the highest known exact marginal gain. As we have seen earlier, the group often does not change at all or only in a late iteration which leads to a late average cutoff point. Therefore, we also observe much smaller average iteration times for later iterations compared to the static algorithm.

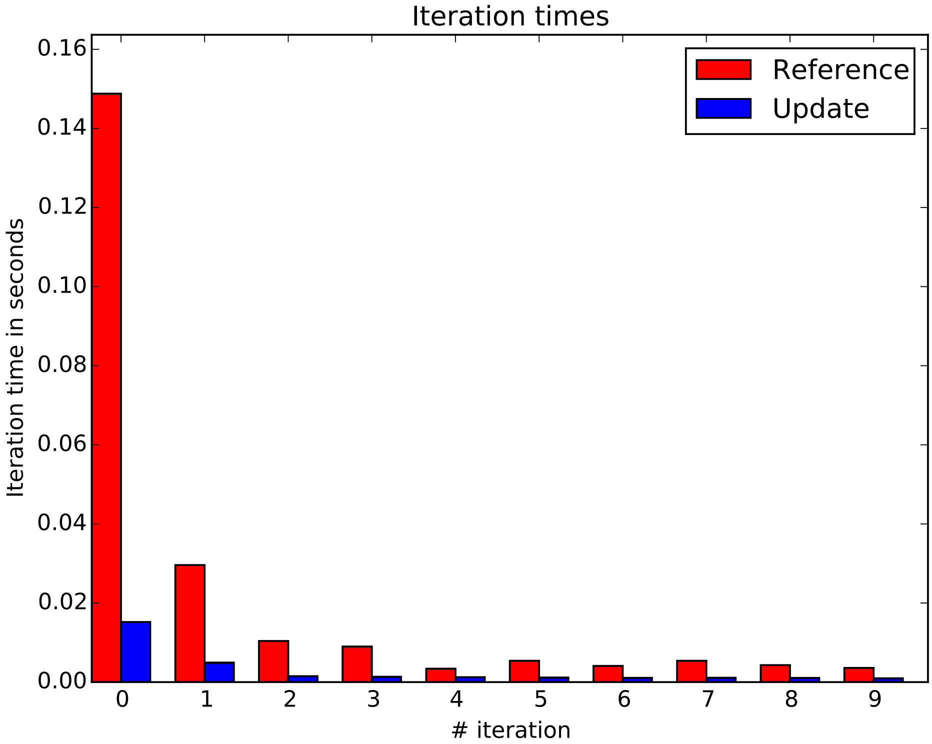

Figure 7.5: Iteration times for group closeness on the graph `ca-GrQc` with $k = 10$

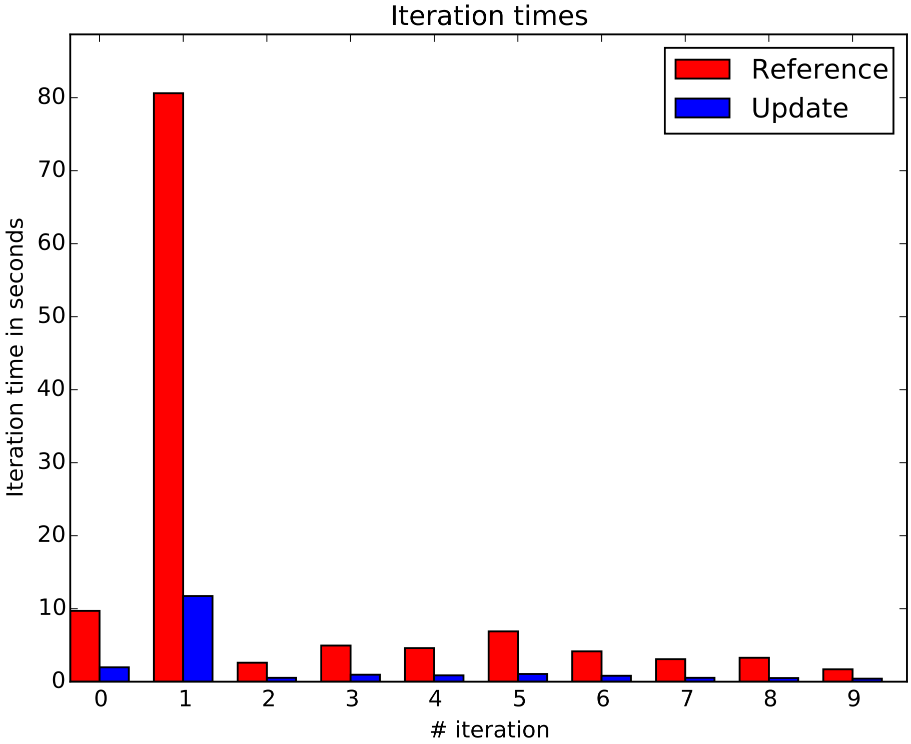

Figure 7.6: Iteration times for group closeness on the graph `com-dblp` with $k = 10$

Graph	k = 10				k = 25				k = 100			
	ref [s]	gmean	max	cutoff	ref [s]	gmean	max	cutoff	ref [s]	gmean	max	cutoff
facebook_combined	0.185	13.425	16.775	10	0.214	12.396	16.6	22.4	0.272	4.556	6.248	26.35
ca-GrQc	0.23	11.949	44.743	9.96	0.282	9.014	32.801	24.3	0.576	4.527	20.536	62.2
as20000102	0.235	6.408	27.596	9.76	0.271	5.371	24.983	23.72	0.43	3.342	7.412	36.44
oregon1_010526	0.411	7.501	26.866	10	0.466	9.468	27.517	24.93	0.698	4.563	10.365	61.13
oregon2_010526	0.45	10.827	31.441	10	0.532	10.046	26.392	24.87	0.772	5.692	18.078	65
ca-HepTh	0.471	9.901	44.656	10	0.572	7.448	39.717	24.47	0.885	4.328	27.774	71.19
ca-HepPh	0.611	17.87	26.56	10	0.749	16.41	34.307	24.64	1.136	9.119	21.565	75.1
as-caida20071105	1.12	6.538	24.239	10	1.236	6.732	24.731	24.99	1.76	5.159	20.681	77.94
ca-CondMat	1.144	10.41	29.263	10	1.401	11.819	34.703	24.77	2.138	6.015	21.897	73.71
ca-AstroPh	1.184	13.345	23.295	10	1.352	12.816	26.111	24.92	1.995	8.062	22.698	80.97
email-Enron	1.643	15.63	26.138	10	1.933	14.199	30.854	25	2.734	9.355	22.631	91.51
loc-brightkite_edges	3.018	13.49	28.242	10	3.468	10.922	24.963	25	5.14	6.519	24.864	80.48
loc-gowalla_edges	17.426	16.947	32.629	10	19.323	16.118	35.97	25	26.993	8.785	32.544	96.85
com-dblp	121.719	21.881	147.72	10	141.551	14.791	148.484	25	189.959	13.911	109.104	97.55
com-amazon	286.456	17.746	318.317	10	305.592	18.609	288.659	24.93	346.771	10.653	193.967	98.23
com-youtube.	746.916	29.85	282.806	10	788.207	34.138	275.085	25	979.787	16.64	136.334	99.46
as-skitter	3243.8	102.091	628.851	10	4259.74	130.412	807.081	25	5121.79	66.371	584.498	100
(geometric) mean	260.413	14.521	49.708	9.984	325.111	13.726	49.708	24.644	393.167	7.826	31.751	76.124

Table 7.15: Update times for group closeness for 100 random edge insertions with $k \in \{1, 10, 100\}$ in undirected street networks The column "ref" contains the average reference time in seconds, "gmean" the geometric mean of the achieved speedups, "max" the maximum speedup. The column "cutoff" contains the average cutoff point.

8. Conclusion

We have developed and implemented dynamic algorithms for Top-k closeness centrality for both complex networks and street networks. We initially limit the recomputation of closeness centralities or upper bounds thereof to nodes actually affected by an edge modification. Additionally, we can exploit the fact that we only need to compute the exact closeness centralities for the k most central nodes. We can keep the upper bounds of nodes which are far away from an edge insertion in place and can update the upper bounds of so-called boundary nodes cheaply. At last, we can compute the maximum improvement of the closeness centrality for each node. If the resulting upper bound is smaller than the k-th largest closeness centrality, we can use the improvement bound and do not need a new breadth-first search.

Our dynamic algorithm is between 35 and 45 times faster on average than the static algorithm for edge insertions on undirected complex networks. On directed complex networks, the dynamic algorithm is between 15 and 20 times faster. In the best case, our dynamic algorithm is up to 700 times faster for some edges.

We achieve even higher speedups on street networks, improving by a factor of more than 100 on average with directed street networks. In some cases, our dynamic algorithm is up to three orders of magnitudes faster than the static algorithm. Our analysis shows that the speedup is the result of a much lower number of complete breadth-first searches compared to the static algorithm.

We have also implemented an incremental algorithm based on the greedy algorithm for the Maximum Closeness Centrality Group Identification problem. We make use of our dynamic algorithm for Top-k closeness centrality and try to replicate the results of the static algorithm with as little computational effort as possible. We avoid recomputing the marginal gains for as many nodes as possible. Once any node in the group has changed, we fall back to the static algorithm.

The dynamic algorithm is between 8 and 15 times faster than the static algorithm on average, and up to 800 times faster for some inserted edges on the larger networks we have tested.

Future work

We have already implemented a simple optimization to speed up the computation of the number of reachable nodes for each node in undirected graphs after an edge insertion. As we have seen in our experimental evaluation, recomputing an upper bound for the number of reachable nodes in directed graphs after edge modifications is a significant bottleneck. There are algorithms that maintain reachability information in directed graphs after edge removals [8, 17]. Since these algorithms maintain exact reachability information for single source nodes, it might be impractical to use them for our purposes. However, it might be practical to maintain upper bounds for the number of reachable nodes.

Our dynamic algorithm is currently designed to update the ranking of the k most central nodes after every single edge modification. However, it might not be necessary to update the Top-k nodes after each edge modification. Then it might be possible to coalesce multiple edge insertions or edge removals into a single update operation and still be faster than running the static algorithm. This is called *batch update*. One could compute the set of affected nodes individually for each edge in the batch and compute the union of all these sets. Even far-away nodes could still be skipped for edge insertions if their minimum distance to any of the edges is larger than their previous cutoff level.

In our dynamic algorithm for group closeness, we perform a pruned BFS for each affected node in each iteration of the algorithm. It might be possible to adapt the idea of the level-based improvement bounds for the group closeness algorithm. If it is possible to compute an upper bound for the increase of the marginal gain for all nodes, we could skip the expensive pruned BFS for nodes where we are sure that the new marginal gain will be smaller than the currently known maximum.

9. Declaration

Ich versichere hiermit wahrheitsgemäß, die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie (KIT) zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 03. Mai 2017 ____________________________
Patrick Bisenius

Bibliography

[1] Eugenio Angriman. Efficient computation of harmonic centrality on large networks: theory and practice. 2016.

[2] Alex Bavelas. Communication patterns in task-oriented groups. *The Journal of the Acoustical Society of America*, 22(6):725–730, 1950.

[3] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[4] Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke. Computing top-k closeness centrality faster in unweighted graphs. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 68–80. SIAM, 2016.

[5] Elisabetta Bergamini, Tanya Gonser, and Henning Meyerhenke. Scaling up group closeness maximization. In submission, 2017.

[6] Paolo Boldi and Sebastiano Vigna. Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262, 2014.

[7] Michele Borassi, Pierluigi Crescenzi, and Andrea Marino. Fast and simple computation of top-k closeness centralities. *arXiv preprint arXiv:1507.01490*, 2015.

[8] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F Italiano, Jakub Łącki, and Nikos Parotsidis. Decremental single-source reachability and strongly connected components in $\mathcal{O}(m\sqrt{n})$ total update time. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 315–324. IEEE, 2016.

[9] Chen Chen, Wei Wang, and Xiaoyang Wang. *Efficient Maximum Closeness Centrality Group Identification*, pages 43–55. Springer International Publishing, Cham, 2016.

[10] AH Dekker. Network centrality and super-spreaders in infectious disease epidemiology.

[11] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[12] David Eppstein and Joseph Wang. Fast approximation of centrality. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 228–229. Society for Industrial and Applied Mathematics, 2001.

[13] Ernesto Estrada and Örjan Bodin. Using network centrality measures to manage landscape connectivity. *Ecological Applications*, 18(7):1810–1825, 2008.

[14] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[15] Linton C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215 – 239, 1978.

[16] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, January 1977.

[17] Jakub Łącki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Transactions on Algorithms (TALG)*, 9(3):27, 2013.

[18] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[19] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, 14(1):265–294, 1978.

[20] Mark Newman. Networks: an introduction. 2010. *United Slates: Oxford University Press Inc., New York*, pages 1–2.

[21] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[22] Ahmet Erdem Sariyuce, Kamer Kaya, Erik Saule, and Umit V Catalyurek. Incremental algorithms for network management and analysis based on closeness centrality. *arXiv preprint arXiv:1303.0422*, 2013.

[23] Christian Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: An interactive tool suite for high-performance network analysis. *CoRR, abs/1403.3005*, 2014.

[24] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[25] Xiao Fan Wang and Guanrong Chen. Complex networks: small-world, scale-free and beyond. *IEEE circuits and systems magazine*, 3(1):6–20, 2003.

[26] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 887–898. ACM, 2012.

[27] Junzhou Zhao, John Lui, Don Towsley, and Xiaohong Guan. Measuring and maximizing group closeness centrality over disk-resident graphs. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 689–694. ACM, 2014.

[28] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM (JACM)*, 49(3):289–317, 2002.