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

31st May 2016

Abstract

Zusammenfassung

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* [16].

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 [13], identifying key infrastructure nodes in computer networks, analyzing the effects of human land use to organism movement [11], or finding so-called super-spreaders in disease transmission networks [8].

One of the most widely-known centrality measures is *PageRank*, proposed by the founders of Google, Sergey Brin and Larry Page [17]. 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. 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. This is because 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 centrality* which has shown to have nice properties on arbitrary graphs [4]. Instead of summing all distances and then computing the inverse, the harmonic 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 [9] 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. Moreover, this effort can currently not be avoided if the application requires an exact ranking of all nodes by their exact closeness centrality.

Cite survey of different centrality measures

For some applications, however, it is enough to compute a list of the k most central nodes. This problem is called *Top-k closeness*. Limiting the problem to the k most central nodes can decrease the required computational effort significantly. It is only necessary to compute the exact closeness centrality of the k most central nodes to rank them. For all other nodes, it is sufficient to obtain an upper bound for their closeness that is smaller than the exact closeness of the k-th most central node.

Mention algorithms for dynamic closeness

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 target 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 [7]. Since the problem is shown to be NP-hard, no efficient exact algorithm exists at this point. However, there are approximative greedy algorithms [7, 21] for the problem. Bergamini and Meyerhenke improve on the work in [7] by reducing the memory requirements and total number of operations .

Cite unpublished work

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 [5], and the additional optimizations for street networks proposed by Bergamini et al. in [3]. 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.

We also contribute an algorithm to update the group of nodes with the highest group closeness after an edge insertion. It is based on Bergamini and Meyerhenke's improved version of the algorithm by Chen et al. The basic idea is to verify whether the choices of the greedy algorithm are still valid on the modified graph with as little computational expense 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.

Both dynamic algorithms are based on existing implementations in NetworKit which is an open-source toolkit for high-performance network analysis [18].

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 $V \times V$. 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*

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

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

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

$$h(v) = \sum_{u \in V} \frac{1}{d(v,u)} \text{ [4].}$$

If there is no path between u and v, the inverse distance is set to 0. Unless stated otherwise, *closeness centrality* refers to *harmonic 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 $\mathcal{O}(n^3)$ [12]; Johnsons's algorithm for weighted graphs without negative cycles has a time complexity of $\mathcal{O}(|V| \cdot (|V| \log |V| + |E|))$ [14]. There are also algorithms that are based on Fast Matrix Multiplication [20, 22] 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 more memory.

3.1 Efficient algorithms for closeness centrality

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

3.1.1 Approximation of closeness centrality

Eppstein and Wang propose a fast approximation algorithm in [10] 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 $\mathcal{O}\left(\frac{\log n}{\epsilon^2} \cdot (n + m)\right)$.

3.1.2 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 [5] which works especially well on complex networks. Angriman presents an adaptation of the algorithms to work with harmonic centrality [1]. Bergamini et al. present some modifications for street networks (i.e. graphs with large diameter) in [3]. 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 centrality to avoid redundancy and since it is easily applicable to disconnected graphs. However, the adaptation of these algorithms to closeness centrality is fairly straightforward.

3.1.2.1 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 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 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.

$$\begin{aligned}
h'(v) &\leq h_d(v) + \frac{\widetilde{\gamma}_{d+1}}{d+1} + \frac{r(v) - n_d - \widetilde{\gamma}_{d+1}}{d+2} && (3.3)\\
&= h_d(v) + \frac{(d+2)\cdot\widetilde{\gamma}_{d+1} + (d+1)\cdot(r(v) - n_d - \widetilde{\gamma}_{d+1})}{(d+1)\cdot(d+2)} && (3.4)\\
&= h_d(v) + \frac{\widetilde{\gamma}_{d+1} + (d+1)\cdot(r(v) - n_d)}{(d+1)\cdot(d+2)} && (3.5)\\
\widetilde{h}(v) &= h_d(v) + \frac{\widetilde{\gamma}_{d+1}}{(d+1)\cdot(d+2)} + \frac{r(v) - n_d}{d+2} && (3.6)
\end{aligned}$$

3.1.2.2 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 computes 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 [19]. 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 $u \in C \land v \in D$
- 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.

3.1.2.3 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

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

Algorithm 1: Computing connected components in an undirected graph

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).

3.1.2.4 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 3.1.2.2). 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.

3.1.2.5 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 [3]. The basic idea is to always run a complete breadth-first search from a source node and then use the number of nodes

```
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 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 $(\widetilde{h},$ 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 2: BFScut()

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.

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.

```
Data: $G = (V, E)$
Result: A list with the $k$ nodes with the highest closeness centrality
Compute $r(v) \forall v \in V$
Top $\leftarrow$ 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
```

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

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 3.1.2.1). 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. With this discovery, 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 $\left| \frac{1}{d(s,w)} - \frac{1}{d(s,v)} \right| \leq 1$ 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

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

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

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

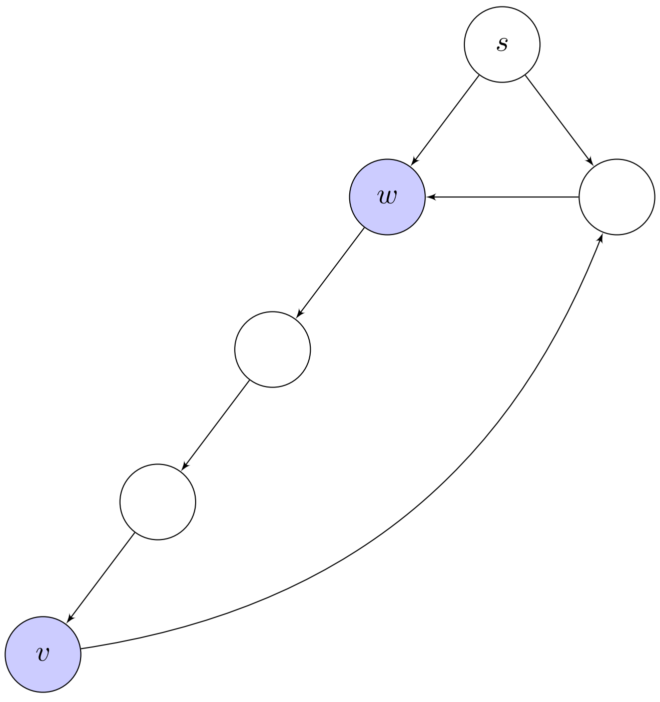

Figure 3.1: BFS tree
Distances in directed graphs are not symmetric. The distance between w and v is 3 in the BFS tree rooted in s, while the distance in the actual directed graph is 2.

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.

Mention parallelization

```
Data: $G = (V, E)$
Result: A list with the $k$ nodes with the highest closeness
1  Preprocessing($G$)
2  Q ← $V$, sorted by decreasing degree or a precomputed upper bound
3  Top ← []
4  score ← array indexed by node ID storing the current upper bounds for all nodes
5  for $v \in V$ do
6      score[$v$] ← $\infty$
7  end
8  while $Q$ is not empty do
9      $v$ ← Q.extractMax()
10     if $|Top| \geq k \wedge$ score[$v$] $\leq$ Top[$k$] then
11         return Top
12     end
13     levelBasedBounds ← updateBounds($v$)
14     score[v] ← levelBasedBounds[v]
15     forall $w \in V$ do
16         if levelBasedBounds[$w$] < score[$w$] then
17             score[$w$] ← levelBasedBounds[$w$]
18         end
19     end
20     if score[$v$] > Top[$k$] then
21         add $v$ to Top
22         sort Top by score and reduce it to at most $k$ elements
23     end
24     re-order Q according to the new values in score
25 end
```

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

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

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.

4. Top-k closeness 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. The dynamic algorithm for Top-k closeness in complex networks (Section 3.1.2) builds on the static algorithm.

4.1 Preliminaries

Let $G(V, E)$ denote an unweighted graph G with the set of nodes V and the set of edges E. For simplicity, we will use the notation (u, v) for an edge between two nodes u and v interchangeably for directed and undirected edges. We can define an *edge insertion* as

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

An *edge removal* is defined as

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

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 an 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 $d_G^f(w,(u,v)) := \min(d_G(w,u), d_G(w,v))$.

4.2 Dynamic Top-k closeness

We will now describe two algorithms to update the list of the k most central nodes in a graph after an edge modification. The first algorithm is based on static algorithm by

Borassi et al. presented in Section 3.1.2. 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 3.1.2.5. The adaptations of both algorithms for the dynamic case both follow the static version closely. However, the dynamic versions will try use previously collected information in order to skip some of the work the static versions have to do.

In general, it makes sense to only process nodes which are actually affected by an edge modification.

4.2.1 Affected nodes

Transforming a graph G to G' by inserting or removing edges will reduce the distances between some node pairs, and thus change the closeness centralities of these nodes.

4.2.1.1 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*

$$A_G^f(u, v) = \{s \mid \exists t \in V : d_G(s, t) \neq d_{G'}(s, t)\}$$

*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.

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.

4.2.1.2 Computing affected nodes

We noted in the previous section 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.

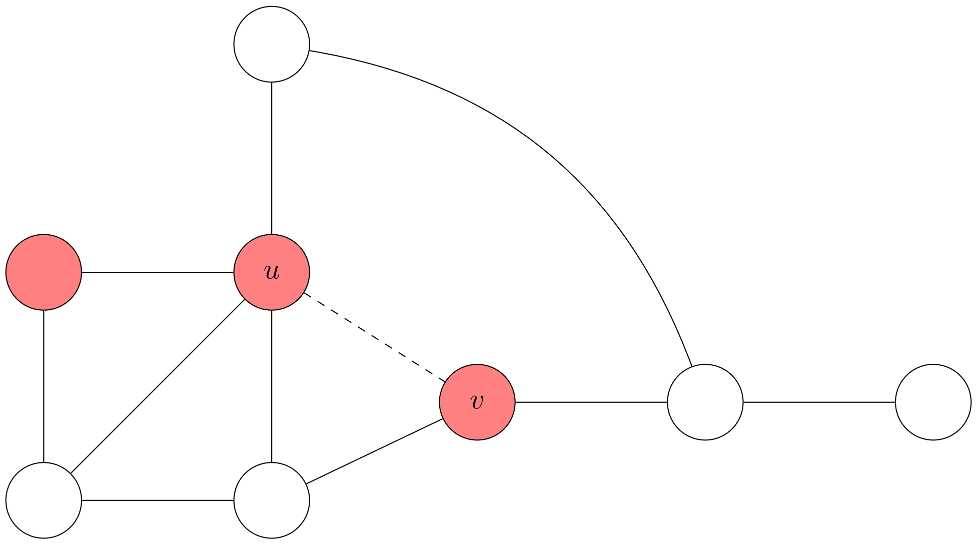

Figure 4.1: Affected and unaffected nodes
The nodes marked in red will be affected if the edge between u and v is modified.

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. Definition 4.1 can be rewritten to

$$A^f_G(u,v) = \{s \in V \mid d_G(s,v) \neq d_{G'}(s,v) \vee d_G(s,u) \neq d_{G'}(s,u)\}. \tag{4.3}$$

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.

We first run a reverse BFS (following incoming edges) from v and store the distances to each node in the graph (Line 1). During that first reverse BFS, we always assume that the modified edge is not part of the graph. For simplicity, the pseudocode in Algorithm 6 explicitly constructs a new graph instance G' (Line 2). In practice, however, the dynamic algorithm that manages the closeness centralities only holds one instance of the graph G and does not modify it. Instead, the dynamic algorithm gets notified about changes and can react to them. This means that we have to handle edge insertions and edge removals separately. For edge insertions, we simply ignore the inserted edge during the first BFS. For edge removals, we can run a standard reverse BFS on the graph since the edge is no longer part of the graph when the dynamic algorithm updates the closeness centralities.

We then execute a second reverse BFS starting in v, this time on a graph which contains the modified edge. This pruned reverse BFS is shown in Algorithm 7. It is supplied with the graph G, the source node for the reverse BFS s and an array d_{old} containing the distances between s and each other node in the graph that does not contain the modified edge. 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 15). 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.

In practice, we operate on a virtual graph that either already contains an inserted edge or does no longer contain a removed edge. In the case of an edge insertion, the underlying graph will already contain the new edge, so no additional adjustments are required. For edge removals, the underlying graph will no longer contain the edge, so it has to be added back virtually. This is done by adding the old neighbor u to the search queue before the search starts. This is shown in Lines 6-8 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. On undirected graphs, we first run a normal BFS from both u and v on the graph without the modified edge and store the distances. Then we run a second pruned BFS on the graph which includes the modified edge. The set of the affected nodes is the union of all nodes whose distance to either u or v is smaller in the graph that does include the modified edge.

```
Data: $G = (V, E), (u, v) \notin E$
Result: $A^f_G(u, v)$
/* like a normal BFS, but following incoming edges                    */
1 $d \leftarrow$ ReverseBFS(G, v)
2 $G' \leftarrow (V, E \cup \{(u, v)\})$
3 $A^f_G(u, v) \leftarrow$ PrunedReverseBFS($G'$, $d$, $v$)
```
Algorithm 6: Computing affected nodes in directed graphs.

4.2.2 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. Algorithm 9 updates the k most central nodes and their closeness centralities after an edge insertion.

4.2.2.1 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 should store 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 from scratch. There are algorithms that maintain strongly-connected

```
Data: $G = (V, E), d_{old}, s$
Result: $A^f_G(u, v)$
1  $A^f_G(u, v) \leftarrow \emptyset$
   /* FIFO queue */
2  $Q \leftarrow \{s\}$
3  $d \leftarrow$ array storing the distances of the nodes
4  $d[s] \leftarrow 0$
5  Mark $s$ as visited
6  while $Q$ is not empty do
7      $u \leftarrow$ first node from $Q$
8      forall $w \in N^{\leftarrow}(u)$ do
9          if $w$ is not marked as visited then
10             $d[w] \leftarrow d[u] + 1$
11             Mark $w$ as visited
12             if $d_{old}[w] < d[w]$ then
13                 $A^f_G(u, v) \leftarrow A^f_G(u, v) \cup \{w\}$
14                 Enqueue $w$ in $Q$
15             end
16         end
17     end
18 end
19 return $A^f_G(u, v)$
```

Algorithm 7: `PrunedReverseBFS`

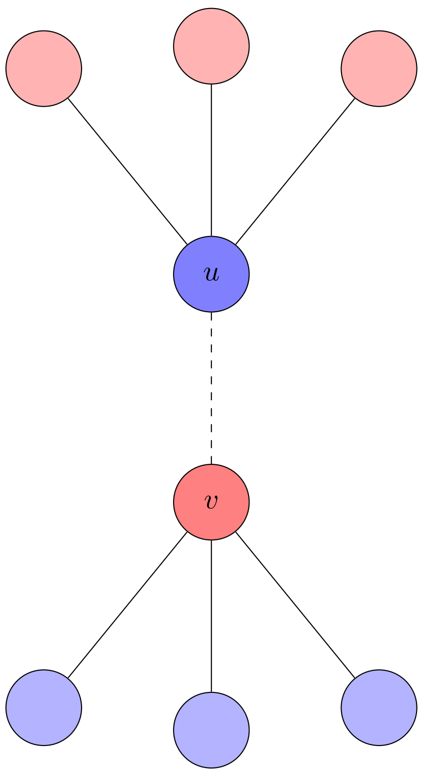

Figure 4.2: Computing affected nodes in undirected graphs

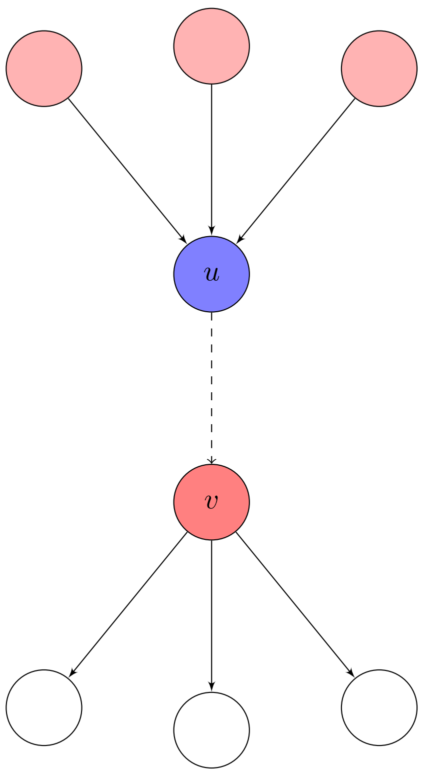

Figure 4.3: Computing affected nodes in directed graphs

components in the case of edge deletions [6, 15]. However, we have not implemented them

```
Data: $G = (V, E), d_{old}, s, (u, v) \notin E$
Result: $A^f_G(u, v)$
1  $A^f_G(u, v) \leftarrow \emptyset$
   /* FIFO queue                                              */
2  $Q \leftarrow \{s\}$
3  $d \leftarrow$ array storing the distances of the nodes
4  $d[s] \leftarrow 0$
5  mark $s$ as visited
   /* Virtually add back the old neighbor $u$                  */
6  Enqueue $u$ in $Q$
7  Mark $u$ as visited
8  $d[u] = 1$
9  while $Q$ is not empty do
10     $u \leftarrow$ first node from $Q$
11     forall $w \in N^{\leftarrow}(u)$ do
12         if $w$ is not marked as visited then
13             $d[w] \leftarrow d[u] + 1$
14             Mark $w$ as visited
15             if $d_{old}[w] < d[w]$ then
16                 $A^f_G(u, v) \leftarrow A^f_G(u, v) \cup \{w\}$
17                 Enqueue $w$ in $Q$
18             end
19         end
20     end
21 end
22 return $A^f_G(u, v)$
```

Algorithm 8: `PrunedReverseBFS` for edge removals

because the time to compute the number of reachable nodes is small compared to the total runtime for an update operation in most cases.

4.2.2.2 Preserving still-valid information

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 previously belonged to the k most central nodes can be re-inserted into the priority queue which manages the Top-k list.

Edge insertions

In the case of an edge insertion, the harmonic centrality of affected nodes can only increase. If there is a new member of the Top-k list after the edge insertion, its harmonic centrality must be larger than the old centrality of the old k-th most central node. The cutoff threshold for pruned breadth-first searches is set to this old centrality initially. The closeness centrality (or the upper bound) of each affected node is 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 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.

4.2.2.3 Recomputing closeness centralities after edge insertions

The dynamic algorithm has the same main loop that computes either the closeness centrality or an upper bound for each node. However, it only iterates over affected nodes in the first place. The static algorithm already only computes an upper bound for the closeness centrality for many nodes. It is possible that the computed upper bound for these nodes would not change due to the edge modification. In this section, we will describe Algorithm 9 which updates the list of the k most central nodes and their exact closeness centrality after an edge insertion.

Skipping far-away nodes

Let w be an arbitrary node in G for which only an upper bound for its closeness centrality is known. Since the upper bounds are computed with a pruned breadth-first search, it is possible to store the cutoff level of this search. If the distance of w to the edge modification $d_G^f(w, (u, v))$ is larger than the cutoff level (see Figure 4.4), it is not necessary to run another 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). Therefore, the dynamic algorithm can mark the old upper bound as valid and skip the expensive recomputation of the upper bound.

Updating bounds of boundary nodes

This optimization can only be 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. Without loss of generality, we assume that u is always the node closer to the chosen node w. This also means that $d(w, u) = d_{cutoff}(w)$. The scenario is shown in Figure 4.5.

First, we show that

$$\begin{aligned} & d_G(w, u) - d_G(w, v) \geq 2 \\ \iff & d_G(w, v) \geq d_G(w, u) + 2 \\ & \qquad\qquad = d_{cutoff}(w) + 2. \end{aligned}$$

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.

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. With our preconditions that $d(w, u)$ is equal to the cutoff level and $d(w, v) \geq d(w, u) + 2 = d_{cutoff}(w) + 2$, 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

$$\widetilde{h}_{new}(w) = \widetilde{h}_{old}(w) - \frac{1}{d_{cutoff}(w) + 2} + \frac{1}{d_{cutoff}(w) + 1}. \tag{4.4}$$

This allows us to update the upper bound for the closeness centrality of w cheaply and without a new pruned BFS.

4.2.2.4 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 and marks their closeness centralities as invalid. The algorithm always keeps a list of all nodes in the graph sorted by their closeness centrality or an upper bound in decreasing order. The algorithm iterates over this list (Line 6). The algorithm starts with an empty priority queue `Top` which manages the k most central nodes (Line 5).

The algorithm iterates over each node w in the graph. 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 7). 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 if it still valid, it can be added to `Top` immediately and it is not necessary to start a new pruned BFS. Otherwise,

```
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 $A(u, v)$
$d_G^f \leftarrow$ array with the distances to the edge modification for each affected node
Mark $h(w)$ as invalid for $w \in A(u, v)$
$x_k \leftarrow$ 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
    if $d_{cutoff}[w] < d_G^f[w]$ then
        Mark $h(w)$ as valid
    else if $d_G^f[w] = d_{cutoff}[w] \wedge !isExact[w] \wedge cutoffIsExact[w]$ then
        $h(w) \leftarrow h(w) - \frac{1}{d_{cutoff}[w]+2} + \frac{1}{d_{cutoff}[w]+1}$
        Mark $h(w)$ as valid
    else
        $(h,$ isExact, cutoffIsExact, $d_{cutoff}) \leftarrow$ BFSCut($w, x_k$)
        $h(w) \leftarrow h$
        isExact($v$) $\leftarrow$ isExact
        cutoffIsExact($v$) $\leftarrow$ cutoffIsExact
        $d_{cutoff}(w) \leftarrow d_{cutoff}$
    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
```

Algorithm 9: Recomputation of the k most central nodes after an edge insertion.

the algorithm executes a pruned BFS to obtain a new upper bound or – in most cases – the exact closeness centrality for w. If the BFS yields the exact closeness centrality, w is added to `Top`.

Data: $G = (V, E), v, x_k$
Result: A tuple $(h, \texttt{isExact}, \texttt{cutoffIsExact}, d_{cutoff})$ with $\texttt{isExact} = \texttt{false}$ if h is only an upper bound for the exact harmonic centrality. `cutoffIsExact` is `true` if the search is aborted when a new level is reached (Line 7).

```
1  Create queue $Q$
2  $Q$.enqueue($v$)
3  Mark $v$ as visited
4  $d \leftarrow 0; h \leftarrow 0; \widetilde{\gamma} \leftarrow 0; nd \leftarrow 0$
5  while $!Q.isEmpty$ do
6      $u \leftarrow$ Q.dequeue()
7      if $d(v, u) > d$ then
8          $d \leftarrow d + 1$
9          $r \leftarrow r(u)$
10         $\widetilde{h} \leftarrow h + \frac{\widetilde{\gamma}}{(d+1)\cdot(d+2)} + \frac{r-n_d}{d+2}$
11         if $\widetilde{h} \leq x_k$ then
12             return $(\widetilde{h}, \texttt{false}, \texttt{true}, d)$
13         end
14     end
15     forall $w \in N(u)$ do
16         if $w$ is not marked as visited then
17             Mark $w$ as visited
18             $Q$.enqueue($w$)
19             $n_d \leftarrow n_d + 1$
20             $pred[w] \leftarrow u$
21             $d(v, w) \leftarrow d(v, u) + 1$
22             $h \leftarrow h + \frac{1}{d(v,w)}$
23             if $G$ is directed then
24                 $\widetilde{\gamma} \leftarrow \widetilde{\gamma} + outdegree(w) - 1$
25             else
26                 $\widetilde{\gamma} \leftarrow \widetilde{\gamma} + outdegree(w)$
27             end
28         else if $d(v, w) > 1 \wedge pred[u] \neq w$ then
29             $\widetilde{h} \leftarrow \widetilde{h} - \frac{1}{d+1} + \frac{1}{d+2}$
30             if $\widetilde{h} \leq x_k$ then
31                 return $(\widetilde{h}, \texttt{false}, \texttt{false}, d)$
32             end
33     end
34     return $(h, \texttt{true}, \texttt{true}, d)$
35 end
```

Algorithm 10: `BFSCut` in the dynamic case

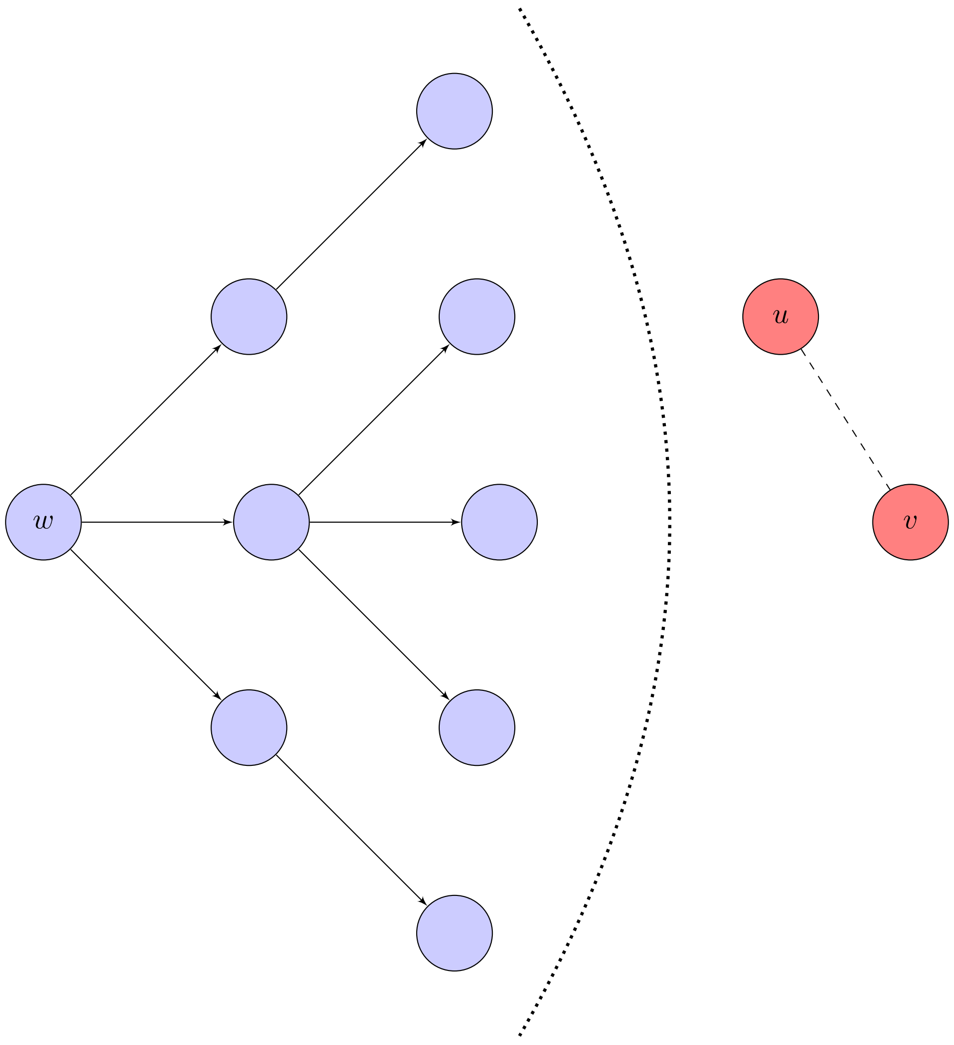

Figure 4.4: Skipping far-way nodes

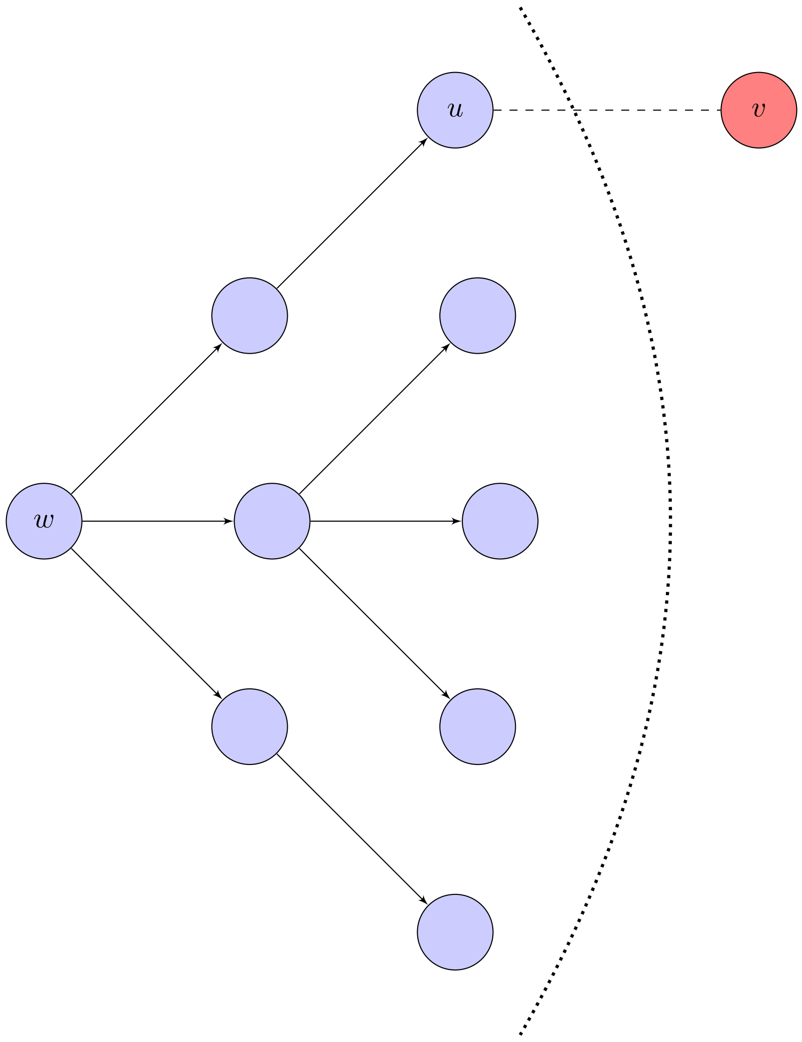

Figure 4.5: Improving upper bounds without an additional BFS

```
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 $A(u, v)$
Mark $h(w)$ as invalid for $w \in A(u, v)$
$x_k \leftarrow 0$
Top ← 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 $h(w)$ is still valid 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 */
    $(h,$ isExact, cutoffIsExact, $d_{cutoff}) \leftarrow$ BFSCut($w, x_k$)
    $h(w) \leftarrow h$
    isExact($v$) ← isExact
    cutoffIsExact($v$) ← cutoffIsExact
    $d_{cutoff}(w) \leftarrow d_{cutoff}$
    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
```

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

4.2.3 Dynamic closeness centrality in networks with large diameters

Edge modifications in networks with large diameter often affect almost every node in the graph. On top of that, the algorithm from Section 3.1.2.5 always computes the exact closeness centrality of each processed node. Therefore, there is no cutoff level for nodes 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.

4.2.3.1 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*, 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 the node u in G, $\Phi_{G'}$ denotes the same set in G'.

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

The improvement for the closeness centrality of u is

$$\begin{aligned} h_{impr}(u) &= h_{new}(u) - h_{old}(u) \\ &= \sum_{t \in S_u} \frac{1}{d_{G'}(u,t)} - \frac{1}{d_G(u,t)} \\ &= \sum_i \frac{1}{i} \cdot \left(\Phi^i_{G'} \cdot - \Phi^i_G\right). \end{aligned} \tag{4.5}$$

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. Let $h_{impr}(x,w) = \frac{1}{d_{G'}(x,w)} - \frac{1}{d_G(x,w)}$ denote the improvement of the contribution of the node pair (x, w) to the closeness centrality of x. We want to show that

$$\begin{aligned} & h_{impr}(u,w) \geq h_{impr}(x,w) \\ \iff \quad & \frac{1}{d_{G'}(u,w)} - \frac{1}{d_G(u,w)} \geq \frac{1}{d_{G'}(x,w)} - \frac{1}{d_G(x,w)}. \end{aligned} \tag{4.6}$$

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 that does not contain u. We rewrite Equation 4.6

$$\begin{aligned} \frac{1}{d_{G'}(u,w)} - \frac{1}{d_G(u,w)} &\geq \frac{1}{d_{G'}(x,w)} - \frac{1}{d_G(x,w)} \\ \frac{1}{o} - \frac{1}{o+p} &\geq \frac{1}{o+q} - \frac{1}{o+q+r} \end{aligned} \tag{4.7}$$

with $o, p, q, r > 0$. We set $o := d_{G'}(u,w)$. Since we know that $d_G(u,w) > d_{G'}(u,w)$, we can write $d_G(u,w)$ as $o+p$. We also know that $d_{G'}(x,w) > d_{G'}(u,w)$, so we write $d_{G'}(x,w)$ as $o+q$. Since $d_G(x,w) > d_{G'}(x,w)$, we write $d_G(x,w)$ as $o+q+r$.

Since o and q are positive, $\frac{1}{o} > \frac{1}{o+q}$. In order to prove that the inequality holds, we have to show that $\frac{1}{o+q+r} \geq \frac{1}{o+p}$. Since it is possible that the shortest path from x to w in G does not contain u, we can state that $d_G(x,w) - d_{G'}(x,w) \leq d_G(u,w) - d_{G'}(u,w)$. 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 (p in Equation 4.7) and x (r in Equation 4.7) to w is exactly the same. Otherwise, it is possible that the distance improvement for x is smaller since $d_G(x,w) < d_G(x,u) + d_G(u,w)$. It follows that $r \leq p$ in Equation 4.7.

Finally, we get

$$\frac{1}{o} - \frac{1}{o+p} \geq \frac{1}{o+q} - \frac{1}{o+q+p} \tag{4.8}$$

which holds for $o, p, q > 0$. 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 4.5 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 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 4.5 to provide an upper bound for the improvement for nodes with distance j to the edge insertion:

$$h_{impr,LB}(j) = \sum_i \frac{1}{i+j} \cdot \left(\Phi^i_{G'} \cdot - \Phi^i_G\right). \tag{4.9}$$

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 $\Phi^i_{G'}$.

Reference Figure 4.6

4.2.3.2 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 3.1.2.5. Algorithm 12 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 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.

```
Data: $G = (V, E), (u, v) \notin G$
Result: A list with the $k$ nodes with the highest closeness
1  Compute the set of affected nodes $A(u, v)$
2  Compute an upper bound for the improvement for each node maxImprovement
3  Q $\leftarrow A(u, v)$, sorted by decreasing previous upper bound for the closeness centrality
4  score $\leftarrow$ array indexed by node ID storing the current upper bounds for all nodes
5  forall $w \in$ Top do
6      if $w \in A(u, v)$ then
7          Top.remove($w$)
8      end
9      score[$w$] $\leftarrow$ score[$w$] + maxImprovement[$w$]
10 end
11 while $Q$ is not empty do
12     $v \leftarrow$ Q.extractMax()
13     if score[$v$] $\leq$ Top[$k$] then
14         continue;
15     end
16     levelBasedBounds $\leftarrow$ updateBounds($v$)
17     score[v] $\leftarrow$ levelBasedBounds[v]
18     forall $w \in V$ do
19         if levelBasedBounds[$w$] $<$ score[$w$] then
20             score[$w$] $\leftarrow$ levelBasedBounds[$w$]
21         end
22     end
23     if score[$v$] $>$ Top[$k$] then
24         add $v$ to Top
25         sort Top by score and reduce it to at most $k$ elements
26     end
27     re-order Q according to the new values in score
28 end
```

Algorithm 12: Dynamic recomputation of the k nodes with the highest closeness centrality in networks with large diameter after an edge insertion.

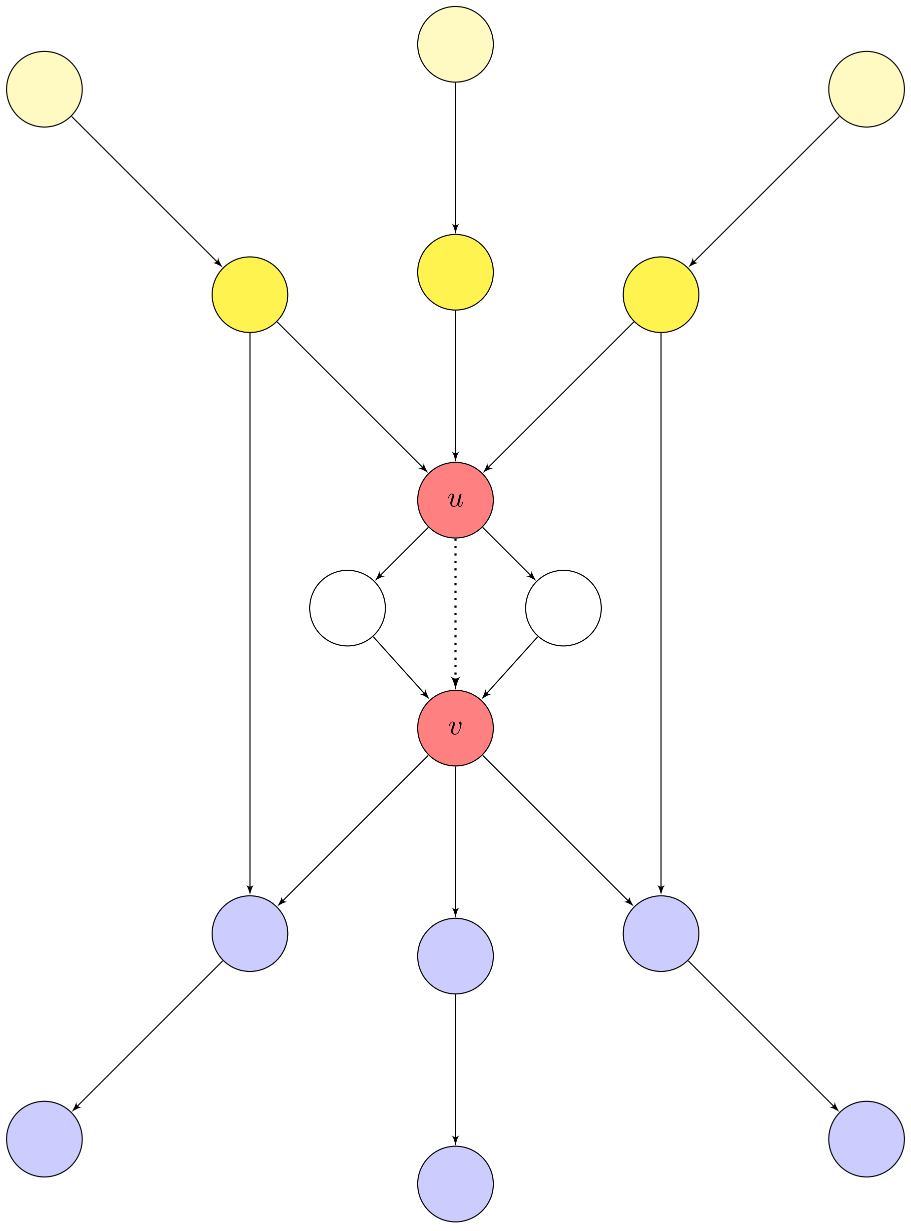

Figure 4.6: Level-based improvement bounds

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 important nodes, the individual contribution of some nodes might be minimal.

5.1 Preliminaries

Let $S \subseteq V$ denote a group of k nodes. We define the distance of a node v to the group as $d_S(v) := min_{s \in S} d(s, v)$.

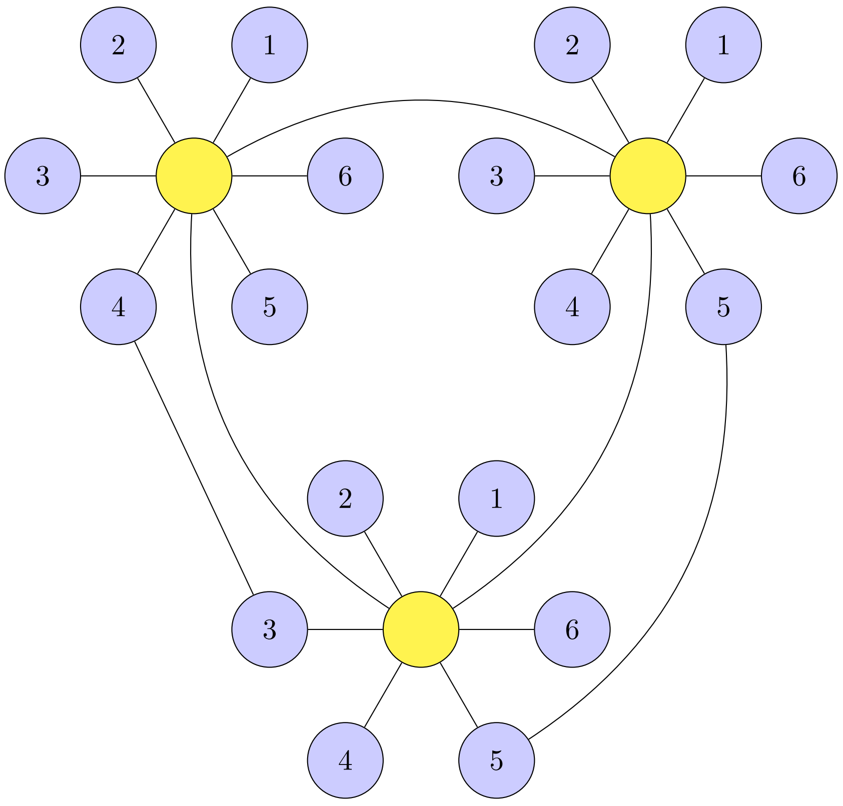

Figure 5.1: Group closeness

6. Conclusion

7. 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 31. Mai 2015 ________________________________

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] 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.

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

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

[6] 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.

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

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

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

[10] 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.

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

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

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

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

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

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

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

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

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

[20] 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.

[21] 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.

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

Todo list