Edge Sparsification for Centrality Computation

Marie-Christin Litzinger

December 1, 2023

3654112

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisor:
Ernestine Großmann
Marcelo Fonseca Faraj

Co-Referee:
Prof. Dr. Michael Gertz

Acknowledgments

I would like to thank Prof. Dr. Christian Schulz for supervising this thesis, providing valuable guidance and fostering a supportive academic environment. My special thanks go to Ernestine Grossmann and Marcelo Fonseca Faraj for their dedicated assistance. I thank them for their prompt responses and professional expertise, which was essential to the completion of this thesis. The joint efforts of Prof. Schulz and the research team contributed greatly to the outcome of this work. I would also like to thank my family and friends for their ongoing support throughout my entire academic journey.

I hereby certify that I have written the work myself and that I have not used any sources or aids other than those specified and that I have marked what has been taken over from other people's works, either verbatim or in terms of content, as foreign. I also certify that the electronic version of my thesis transmitted completely corresponds in content and wording to the printed version. I agree that this electronic version is being checked for plagiarism at the university using plagiarism software.

Heidelberg, December 1, 2023

[illegible]

Marie-Christin Litzinger

Abstract

A fundamental concept of Network Analysis are centrality measures, which assign a score to each vertex, representing its structural importance. Especially the relative ranking and the identification of the most central vertices provide insights into the underlying data set and are applied in various domains. Even though there exist exact algorithms in polynomial time, these do not scale well and lots of approximation algorithms have been developed. In this work, an edge sparsification, which is applied to graphs before centrality computations, is proposed. It estimates how often each edge occurs on shortest path within the graph. The reduced graph is then constructed by including only edges that are part of many shortest paths. The experiments show that the edge sparsification works best for distance-based centrality measures. For betweenness centrality strongly correlated rankings and an overlap of 83% between the most central vertices are achieved with an average speedup of 26%. For closeness centrality, similar results are obtained with an average speedup of 21%.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In various domains, data is collected to capture the connections or relationships among arbitrary objects. Examples include social networks, where individual users are interconnected, biological networks modeling protein interactions, or transportation networks connecting different locations through routes. By representing these data sets as graphs, they can be further investigated using techniques from the field of Network Analysis, involving algorithms and statistical tools. The information and insights derived, such as non-trivial relations between entities, can be used in practice to optimize processes or strategies [101].

One very important problem of Network Analysis is identifying the most important vertices of a graph, or put differently, the concept of centrality. A centrality measure is a function, which expresses the relative importance of a vertex [81]. Since the definition of importance may vary for each application, lots of centrality measures have been developed. These measures capture diverse aspects, ranging from local to global graph properties, and exhibiting varying degrees of computational complexity [81]. Some well-known centrality measures include betweenness, closeness, and eigenvector centrality.

A major application area of centrality measures are biological networks, such as protein interaction, metabolic and gene regulatory networks. Wuchty and Stadler [103] analyze three such network types with respect to their centers, while in [47] lethality in protein networks is investigated. Jothi [48] combines edge betweenness centrality with clustering and shows that this can be used to identify cancerous tissues. Further work can be found in [26, 52]. In addition to biological networks, centrality measures are applied across various domains and contexts, including:

- Terrorist networks [54]
- Collaboration networks [71, 104]
- Transportation and road networks [38, 44]

- Social networks and measurement of influence in social networks [60, 83]
- Partitioning and community detection [28, 39]

For each centrality measure exist algorithms, which compute the exact scores on arbitrary graphs. However, they are often computationally expensive and can only be applied to smaller graphs. Given that today's graphs can contain millions of edges, improving the scalability of centrality measures has become a main focus in current research [101]. Consequently, heuristics and approximation algorithms have been developed to compute approximate instead of exact results while speeding up the overall computation times. In practice however, it has been shown that not the score of a vertex but rather the relative ranking and the identification of vertices with the highest centrality scores are most important. Thus, lots of algorithms of today's research focus on these aspects.

While many of these approaches have shown to work well in practice, they reach their limits when dealing with very large graphs. To overcome this challenge, these algorithms are often augmented with supplementary techniques like parallelism or reductions [4, 78, 87, 94]. This thesis is situated in precisely this context, providing a novel edge sparsification. Instead of computing centrality scores directly on the entire graph, the graph is reduced first. The results obtained on the reduced graph then serve as approximations for the corresponding results on the full graph. This yields a speedup for centrality computations, but comes with a potential loss of quality. Since shortest paths are part of many centrality measures and also reflect the topological structure of a graph, they play a major role in the sparsification. Especially edges that frequently appear in the shortest paths within a graph are of particular importance. The core concept of the sparsification is therefore, to keep these edges in the graph while removing unimportant ones, i.e., edges which do not occur sufficiently often on shortest paths. Thereby, the order of shortest paths is preserved.

1.2 Our Contribution

In this thesis an edge sparsification based on shortest paths is proposed. In order to determine the edges included in the reduced graph and those to be excluded, so-called edge counts are computed. These counts represent how frequently an edge is part of shortest paths and can be computed exactly using all-pairs shortest path (APSP). Since this is computationally expensive, three different breadth-first search (BFS) variants are presented, which conduct single-source shortest path (SSSP) computations only for a subset of vertices. Based on the edge counts, the reduced graph is constructed. Thereby, two maximum-weight spanning forest (MWSF) methods are proposed, which ensure the connectivity of the reduced graph while integrating edges with high counts.

In extensive parameter tuning experiments, two parameter configurations are obtained for each combination of BFS and MWSF method – one focussing on the relative ranking of vertices and the other one on identifying the most central vertices. Moreover, insights into the influence of parameters on the approach and on the resulting reduced graphs are provided. Finally, the results on the reduced graphs are compared to the ones obtained without edge sparsification. It is shown that betweenness and closeness centrality outperform other centrality measures. The top 100 centrality vertices are identified with an accuracy of 73% and 83%, achieving a speedup of 14% and 28% for closeness and betweenness centrality, respectively. Vertex rankings for these centralities are computed with a speedup of 14% and 25%, exhibiting correlation coefficients of 0.91 and 0.84.

1.3 Structure

The subsequent chapters of this thesis are organized as follows: In Chapter 2 fundamental concepts of graphs are provided, followed by the definitions and exact algorithms for the centrality measures. An overview of state-of-the-art approximation algorithms, heuristics and other techniques related to each centrality measures is given in Chapter 3. In the subsequent Chapter 4, the edge sparsification, along with all its components and approaches, is explained. Its experimental evaluation is given in Chapter 5. First, two sets of parameter tuning experiments are presented, one optimized for the vertex ranking and the other one for the most central vertices. The properties of reduced graphs and the influence of parameters are then empirically investigated. The final part of Chapter 5 compares centrality computations and its results between full and reduced graphs. In Chapter 6 a summary of the findings is given and aspects, which can be investigated in future research, are discussed.

CHAPTER 2

Fundamentals

In the first section of this chapter, graph terminology as well as foundational concepts and algorithms are introduced. Subsequently, different centrality measures are defined along with algorithms for their exact calculation.

2.1 Graph Preliminaries

A graph G is an abstract structure used to model pairwise relationships between objects [27]. It is defined as $G = (V, E)$ where V is a finite set of objects known as vertices and $E \subseteq \binom{V}{2}$ is the set of edges. The number of vertices and edges are denoted by $n := |V|$ and $m := |E|$ respectively. The vertex set of a graph G is denoted as $V(G)$, and its set of edges is represented as $E(G)$ or, if the context is clear, simply as V and E.

Two vertices $u, v \in V$ are called adjacent or neighbors if there exists an edge connecting them, i.e., $e = \{u, v\} \in E$. These edges do not have a direction and the graph is thus called undirected. The set of neighbors of a vertex v is denoted as $N(v) = \{u \mid \{v, u\} \in E\}$. The degree of a vertex is defined as the number of neighbors $d(v) = |N(v)|$. Additional information can be encoded by assigning a numeric value to each edge using a mapping $\omega : E \to \mathbb{R}$. Every graph that exhibits such weights or costs is then called a weighted graph [65].

Graphs can be represented by a $n \times n$ matrix $A = (a_{ij})_{n \times n}$ which is known as adjacency matrix [27]. Assuming that the vertices of G are labeled from 1 to n, the matrix is defined as

$$a_{ij} := \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

In the weighted case, $a_{ij} = \omega(\{v_i, v_j\})$ if the respective edge $\{v_i, v_j\}$ exits. Alternatively, an adjacency list can be used to represent a graph which stores for every vertex v_i a list of its neighbors at the respective index.

Algorithm 1 Kruskal's algorithm for computing a MWSF [65]

```
    Input Weighted graph $G = (V, E, \omega)$ where $\omega$ is the edge weight function
    Output MWSF $G_F = (V, E_F)$
1: $E_F := \emptyset$
2: for each $\{u, v\} \in E$ in descending order of cost do
3:     if $u$ and $v$ are in different subtrees of $E_F$ then
4:         $E_F := E_F \cup \{\{u, v\}\}$
5:     end if
6: end for
7: $G_F := (V, E_F)$
8: return $G_F$
```

A graph $G' = (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$ [27]. If G' further contains all edges $\{u, v\} \in E$ with $u, v \in V'$, G' is called an induced subgraph of G. Since every induced subgraph is uniquely defined by its vertex set, this is denoted by $G[V']$. A subgraph $G' \subseteq G$ is called a spanning subgraph of G if $V' = V$.

A path $p = \langle v_0, \dots, v_k \rangle$ is a sequence of vertices in which consecutive vertices are connected by edges $\{v_0, v_1\}, \{v_1, v_2\}, \dots \{v_{k-1}, v_k\} \in E$ [27, 65]. A cycle is a path with common first and last vertex, i.e., $c = \langle v_0, \dots, v_k, v_0 \rangle$. For an unweighted graph, the length of a path or cycle is its number of edges. In a weighted graph, it is defined as the sum of the edge weights along that path or cycle. A graph without cycles is called acyclic or a forest.

If there exists a path between any two vertices, a graph is called connected, otherwise disconnected [27]. A maximal connected subgraph of G, which is automatically also an induced subgraph, is called (connected) component of G. A connected forest is called a tree and degree-1 vertices are called leaves. For a graph $G = (V, E)$ a spanning tree G' is defined by a set of edges $T \subseteq E$ such that $G' = (V, T)$ is a tree [65]. A maximum-weight spanning tree (MWST) is then a spanning tree where $\omega(T) := \sum_{e \in T} \omega(e)$ is maximized. Consequently, a MWSF is defined as the union of MWSTs for each connected component of G.

Originally, Kruskal's algorithm is proposed for computing the minimum weight spanning tree of a graph, which is defined analogue to the MWST [65]. As demonstrated in Algorithm 1, it can also be adapted to compute a MWSF by either inverting the edge weights or by modifying the order of edges. The algorithm iterates over the edges in descending order with respect to their weights. For each edge it is checked whether it connects two vertices u, v in either the same or different components in G_F. If u and v belong to the same component, then there exists already a path connecting them and adding the edge $\{u, v\}$ to E_F creates a cycle, violating the requirement that G_F is a forest. Conversely, if the vertices are part of different components, the edge $\{u, v\}$ is a maximum weight edge in the cut and is added to E_F.

In order to efficiently determine whether an edge connects two vertices from different components, a union-find data structure can be used [65]. It maintains a partition of the set $\{1, \dots, n\}$ and provides the operations

- *find*, which returns for a given $i \in \{1, \dots n\}$ the representative of the subset the index belongs to and
- *union*, which combines two given subsets.

Each subset can be represented as a rooted tree, using the root as representative [65]. By additionally using the optimizations *union by rank*, which ensures that the depth of the trees does not exceed $\log n$, and *path compression*, which decreases the amount of vertices needed to traverse in *find*, both operations need amortized constant time. In Kruskal's algorithm, the connected components in (V, E_F) are the subsets in the union-find data structure. Due to the efficiency of the latter, the overall running time of Kruskal mainly depends on the sorting of edges and is thus $O(m \log m)$.

The distance $d(u, v)$ between two vertices $u, v \in V$ is the length of a shortest uv-path in G [27, 65]. If the graph is disconnected and no such path exists $d(u, v) := \infty$. The diameter of G is then defined as the greatest distance between any two vertices

$$\operatorname{diam}(G) = \max_{u,v \in V} d(u, v). \tag{2.2}$$

To determine the distance between a pair of vertices, BFS and Dijkstra's algorithm can be used for unweighted and weighted graphs, respectively. These algorithms are also applicable to compute the distances from a single vertex $s \in V$ – the source – to all other vertices in a graph, known as SSSP. Additionally, the APSP problem can be solved by applying these algorithms to every vertex in a graph. BFS traverses a graph layer by layer while Dijkstra scans vertices in order of increasing shortest path distances. The actual paths can be retrieved by recursively iterating through the predecessor array. The runtime complexity of Dijkstra is $O(m + n \log n)$ (using a Fibonacci heap) and BFS runs in $O(n + m)$ time. Both algorithms construct a shortest path tree rooted in s. Throughout this thesis graphs are assumed to be undirected, unweighted and connected.

2.2 Centrality Measures

The idea behind centrality measures is to assign a degree of importance to each vertex, represented by a real number [19]. These numbers are based on structural properties of a graph. Such measures are a fundamental tool in network analysis and widely applied in the study of various network types. The first centrality indices were defined in the 1940s and since then, various ones have been added, each focusing on different properties of a graph [11]. However, a commonly accepted general definition of a centrality measure does not exist [19]. This is mainly caused by the fact that importance heavily depends on the context of the application and the problem at hand.

However, based on which features and characteristics they use to define importance, centrality measures are divided into groups [19]. Still, there exists no general classification, but a large overlap between different publications. The following presents a subset of the groups presented in [19, 24].

1. Reachability
   Using a vertex' degree or "the cost it takes to reach all other vertices" [19, p. 20], centrality measures belonging to this group evaluate the ability of a vertex to reach other vertices. Representatives are, e.g., degree, closeness, eccentricity and k-core centrality as well as centroid values.

2. Shortest Paths
   In many use cases, information flows through graphs in shortest paths. Thus, there exist many centrality measures which base a vertex importance on those paths, for example betweenness, stress and reach centrality [19, 101].

3. Feedback
   For these centrality measures, "a node is the more central the more central its neighbors are" [19, p. 46]. Apart from eigenvector and Katz centrality, which are discussed below, the Hubbell index, PageRank and Bargaining centrality belong to this group.

Centrality measures can also be extended to groups of vertices or turned into edge centralities. In the following, the centrality measures relevant for this thesis – degree, closeness, betweenness, eigenvector, Katz and k-core centrality – are described in detail. For the remaining groups and other centrality measures, the reader is referred to [11, 19].

2.2.1 Degree Centrality

A widely applied centrality measure is degree centrality, where each vertex is assigned a centrality score equal to its degree. It is based on the notion that the more connections a vertex has, the more important it is [19, 24, 95]. For a given vertex $v \in V$, degree centrality is defined as

$$c_D(v) = d(v) \tag{2.3}$$

and can further be normalized by $n - 1$ or the maximum degree in a graph [95]. For a single vertex, degree centrality is computed in $O(n)$, and for all vertices of a graph in $O(m)$ [95]. The mayor limitation of degree centrality is its restricted perspective of the graph's topology [50]. Only local information about a vertex is used to determine its importance. However, in various practical applications further information is necessary to differentiate between vertices with nearly identical degree.

2.2.2 Closeness Centrality

Initially introduced by Bavelas [8], closeness centrality is a measure that assesses the importance of a vertex within a graph by considering the shortest paths to all other vertices [84]. The underlying concept is that vertices, that are closer to all other vertices, are considered more important. In other words, vertices that can quickly receive information sent from any part of the graph are considered most central [95]. Mathematically, this is expressed by using the inverse of the (average) distances to all other vertices. Given a graph $G = (V, E)$ with $|V| = n$ vertices, closeness centrality is defined as

$$c_C(u) = \frac{1}{\sum_{v \in V} d(u, v)} \tag{2.4}$$

which can be normalized with $n - 1$, resulting in

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

It follows from this definition that closeness centrality can only be applied to connected graphs [24, 95]. In a disconnected graph, for every vertex $u \in V$, there exists another vertex $v \neq u, v \in V$ such that u and v belong to different components. Thus, $d(u, v) = \infty$ and both $c_C(u)$ and $c_C(v)$ are equal to zero. Nevertheless, an extension of closeness centrality, known as harmonic centrality deals with pairs of unreachable vertices [11, 89]. Instead of utilizing the average distance, harmonic centrality employs the harmonic mean of all distances. For a given vertex $u \in V$, harmonic centrality is defined as follows:

$$c_H(u) = \sum_{v \neq u, v \in V} \frac{1}{d(u, v)} \tag{2.6}$$

Closeness centrality can be exactly computed by solving the APSP problem. This is achieved in $O(n^{2.373})$ using fast matrix multiplication or in $O(nm)$ by performing BFS n times [9]. However, fast matrix multiplication involves large hidden constants, making BFS-based approaches commonly preferred in practical applications.

2.2.3 k-core Centrality

Seidman proposed a centrality measure based on the degree of a vertex, or more precisely on k-cores. The k-core of a graph is the largest induced subgraph in which every vertex has at least degree k [25]. The core number of a vertex is then the highest value of k, such that the vertex belongs to a k-core. Note, that the $(k + 1)$-core is a subset of the k-core. The k-core centrality of a vertex is then defined as the core number of that vertex [7, 25]. This can be further normalized, for instance, by the maximum degree of the graph.

A k-shell of a graph is the subgraph induced by the vertices whose core number is k and thus consists of the vertices that are part of the k-core but do not belong to the $(k + 1)$-core [24]. k-core centrality is sometimes also referred as k-shell centrality.

A k-core decomposition algorithm, which computes the core numbers of all vertices in a graph, is outlined in Algorithm 2. Vertices with degree k are removed from the graph and their core number is set to k. Their removal decreases the degree of neighboring vertices,

Algorithm 2 k-core decomposition algorithm [24, 25]

```
    Input A graph $G = (V, E)$
    Output Core numbers for all vertices in $G$
 1: Set $k = 0$
 2: $core\_numbers \leftarrow$ empty array of size $n$
 3: repeat
 4:     repeat
 5:         Remove all vertices with $d(v) = k$ from $G$
 6:         $core\_numbers[v] \leftarrow k$
 7:     until $\forall v \in V : d(v) > k$
 8:     $k \leftarrow k + 1$
 9: until $V = \emptyset$
10: return $core\_numbers$
```

which then also may have degree k. Therefore, this is iteratively repeated until the graph contains only vertices with a degree larger than k. In order to compute the core numbers of all vertices, k is set to zero initially and increased until all vertices have a core number and are removed from the graph [24, 25].

2.2.4 Betweenness Centrality

Betweenness centrality originates from the field of social studies and was first defined in the 1970s [3, 36]. It measures the influence or the amount of control a vertex holds over connections between pairs of vertices [29, 87]. For a vertex $u \in V$ it is defined as the fraction of all pairwise shortest paths that go through u [19]. Consequently, a high betweenness centrality score indicates that a significant number of shortest paths traverse that vertex and its removal would result in longer paths between pairs of vertices [63].

For two vertices $s, t \in V$ the number of shortest paths between them is defined as σ_{st} [32, 19]. Furthermore, $\sigma_{st}(u)$ denotes the number of such paths that traverse through the intermediary vertex $u \in V$. Note, that in an undirected graph it holds $\sigma_{st} = \sigma_{ts}$. Using these definitions, the following types of dependencies are established:

- Pair-dependency of a vertex pair $s, t \in V$ on a vertex $u \in V$

  $$\delta_{st}(u) = \frac{\sigma_{st}(u)}{\sigma_{st}}, \tag{2.7}$$

  which quantifies the fraction of shortest paths between s and t that pass through v [19].

- Dependency of a source vertex s on a vertex u, with $s, u \in V$

  $$\delta_{s\bullet}(u) = \sum_{t \in V} \delta_{st}(u), \tag{2.8}$$

which summarizes the collective influence of vertex u on all paths starting from vertex s [19].

Betweenness centrality of a vertex $u \in V$ is then defined as the sum of pair-dependencies, i.e.,

$$c_B(u) = \sum_{s \neq u \in V} \delta_{s\bullet}(u) \tag{2.9}$$

$$= \sum_{s \neq u \neq t \in V} \delta_{st}(u) = \sum_{s \neq u \neq t \in V} \frac{\sigma_{st}(u)}{\sigma_{st}} \tag{2.10}$$

and can be normalized by multiplying with $\frac{1}{n(n-1)}$ [13]. In a disconnected graph, the number of shortest paths between vertices belonging to different components is zero and consequently, such vertex pairs do not influence the centrality score of a vertex [18]. Moreover, $\frac{0}{0} = 0$ by convention and therefore, betweenness centrality can also be applied to disconnected graphs.

To compute betweenness centrality scores of all vertices in a graph, it is necessary to determine the dependencies, which, in turn, requires the calculation of both $\sigma_{st}(u)$ and σ_{st} for all possible triples of vertices s, t, u [19, 32]. Using the fact that u is contained in a shortest path between s and t if and only if $d(s,t) = d(s,u) + d(u,t)$, it follows that shortest paths from s to t through u can be written as concatenation of shortest paths from s to u and from u to t. This property directly transfers to the number of shortest paths, leading to the equation

$$\sigma_{st}(u) = \sigma_{su}\sigma_{ut}. \tag{2.11}$$

The predecessor set of u with respect to s is then defined as

$$P_s(u) = \{v \in V \mid (u,v) \in E, d(s,u) = d(s,v) + 1\} \tag{2.12}$$

and it holds

$$\sigma_{su} = \sum_{v \in P_s(u)} \sigma_{sv}. \tag{2.13}$$

For a given source s, σ_{su} can be computed for all $u \in V$ by running a BFS. Together with Equation (2.11) this implies that computing $c_B(u)$ requires $O(n^2)$ for a single vertex u and $O(n^3)$ for all vertices [19, 32, 87].

Theorem 1
For the dependency $\delta_{s\bullet}(u)$ of a source vertex $s \in V$ and any other $u \in V$ it holds

$$\delta_{s\bullet}(u) = \sum_{w : u \in P_s(w)} \frac{\sigma_{su}}{\sigma_{sw}}(1 + \delta_{s\bullet}(w)). \tag{2.14}$$

The previous statement has been proven in [17] where the author also provides a faster algorithm, which computes betweenness centrality scores for all vertices in $O(nm)$ time for unweighted graphs. The key idea of that algorithm is to compute accumulated dependencies recursively using Equation (2.14) [17, 19, 32, 87]. The algorithm operates in two phases: First, a shortest path tree for every $s \in V$ is computed to obtain σ_{su} and $P_s(u)$ for every $u \in V$. In the second phase, the shortest path trees are traversed backwards to compute $\delta_{s\bullet}(u)$ for all $s \neq u$ using Equation (2.14). The final betweenness score of a vertex is then the sum of all dependency values computed during the n SSSP runs. The space requirements can further be reduced from $O(n^2)$ to $O(n+m)$ by directly summing up the dependency values [19].

Brandes' algorithm has been shown to be almost optimal [13, 15]. For sparse graphs computing betweenness centrality in $O(mn^{1-\epsilon}), \epsilon > 0$ would contradict some commonly held complexity assumptions like the Strong Exponential Time Hypothesis or the Orthogonal Vector conjecture. For dense, weighted graphs it has been shown that computing exact betweenness centrality scores is as hard as computing APSP and determining an approximation with a specified relative error is equally challenging as computing the diameter [1, 13]. For these algorithms, no algorithm running in $O(n^{3-\epsilon}), \epsilon > 0$ exists.

2.2.5 Eigenvector Centrality

Bonacich proposed a centrality measure that relies on the eigenvector associated with the largest eigenvalue of the adjacency matrix [24, 95]. This centrality score quantifies a vertex importance proportional to the sum of neighboring centrality scores, implying that a vertex is important if its neighbors are important. Let A be the adjacency matrix of a graph G, i.e., $a_{ij} = 1$ if vertex v_i is connected to vertex v_j and $a_{ij} = 0$ otherwise. The eigenvector centrality of v_i is then defined as

$$c_E(v_i) = \frac{1}{\lambda} \sum_{j=1}^{n} a_{ij} x_j, \; i = 1, 2, \ldots n \tag{2.15}$$

which can also be written as $Ax = \lambda x$, with λ being the greatest eigenvalue of A [19, 24, 95]. This can be computed exactly by using the power iteration method [102]. Starting with an initial vector $\boldsymbol{x^{(0)}} \in \mathbb{R}^{n \times 1}$, $\|\boldsymbol{x^{(0)}}\|_2 = 1$ the $(k+1)$-th iterate is defined as

$$\boldsymbol{x^{(k+1)}} = \frac{A\boldsymbol{x^{(k)}}}{\|A\boldsymbol{x^{(k)}}\|_2} = \frac{A^k\boldsymbol{x^{(0)}}}{\|A^k\boldsymbol{x^{(0)}}\|_2} \tag{2.16}$$

where $\boldsymbol{x^{(k)}}, \boldsymbol{x^{(k+1)}} \in \mathbb{R}^{n \times 1}$. For $k \to \infty$, $\boldsymbol{x^{(k)}}$ converges to the eigenvector associated with the largest eigenvalue. In practice, the method is terminated after a sufficient number of iterations or a given error tolerance is reached [79].

2.2.6 Katz Centrality

To assess the significance of a vertex, Katz centrality considers the number of paths between vertex pairs [24, 59, 95]. It was initially introduced by Katz and can be viewed as a generalization of eigenvector centrality [59]. Assuming that the vertices are labeled from 1 to n, the Katz centrality of a vertex $v_i \in V$ is defined as

$$c_K(v_i) = \sum_{k=1}^{\infty} \sum_{j=1}^{n} \alpha^k (A^k)_{ij}. \tag{2.17}$$

Here, $(A^k)_{ij}$ denotes the number of paths of length k from v_i to v_j and $\alpha \geq 0$ is a damping factor, which ensures that longer paths between two vertices v_i and v_j have less influence on the centrality of v_i than shorter paths [19, 95]. To guarantee convergence α must be chosen such that $\lambda_1 < \frac{1}{\alpha}$ holds, where λ_1 is the largest eigenvalue of A [19]. Assuming convergence, Equation 2.17 can be rewritten as follows:

$$\begin{aligned} c_K(v_i) &= \sum_{k=1}^{\infty} \sum_{j=1}^{n} \alpha^k (A^k)_{ij} && (2.18) \\ &= \sum_{k=1}^{\infty} \alpha^k (A^T)^k \mathbf{1}_n && (2.19) \\ & && (2.20) \end{aligned}$$

which is equivalent to

$$\begin{aligned} \boldsymbol{c_K} &= ((I - \alpha A^T)^{-1})\mathbf{1}_n && (2.21) \\ (I - \alpha A^T)\boldsymbol{c_K} &= \mathbf{1}_n. && (2.22) \end{aligned}$$

This reformulation into an inhomogeneous system of linear equations highlights the feedback nature of Katz centrality, namely that $c_k(v_i)$ depends on $c_k(v_j), j \neq i$ [19]. The exact computation of Equation (2.22) is computationally infeasible, since the inverse of a matrix is required [68, 69]. Using Cholesky decomposition is impractical as well, due to its computational complexity of $O(n^2)$. Thus, iterative methods which require $O(m)$ time, if the number of iterations is not very large, are commonly used in practice.

Such a method approximates the solution $\boldsymbol{x}$ of a linear system $M\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{b}$ and M are given [68, 69]. The procedure starts with an initial guess $\boldsymbol{x^{(0)}}$ and in each iteration the current guess $\boldsymbol{x^{(k)}}$ is improved until a stopping criterion is reached. Such a criterion could, for example, be a predefined number of iterations or an error tolerance. In the case of Katz centrality, $M = I - \alpha A$ and $b = \mathbf{1}_n$. This is then iteratively solved for $\boldsymbol{x}$ or $\boldsymbol{c_K}$.

CHAPTER 3

Related Work

For each centrality presented in the previous section, it has been shown that polynomial time and space algorithms for computing the exact centrality scores exist. However, these only work well on smaller graphs. On today's real world instances their computation takes prohibitively long, making them inapplicable in practice. Furthermore, it is important to note that practical applications do not necessarily require exact centrality values. Often approximations or the ranking of vertices with respect to a given centrality measure are sufficiently informative. Therefore, lots of research is dedicated to the development of approximation algorithms and heuristics which make trade-offs between accuracy and computational speed. The key emphasis lies on scalability and minimal runtime dependency on the graphs's size. In the following, different such approaches are discussed for all previously introduced centrality measures. Note, that these are limited to static graphs, for algorithms on dynamic graphs the reader is referred to [45].

3.1 Degree Centrality

Since the exact computation is already very efficient algorithms, which approximate vertex degrees are not needed [95]. However, an approach to estimate the rank of a vertex by using local information is presented in [90]. In a preprocessing step, graph characteristics such as the maximum and average degree are computed. Subsequently, the rank of a given vertex is estimated either by utilizing characteristics related to the power law degree distribution or by applying a sampling based approach. In the latter method, the estimated global rank is obtained by extrapolation of the exact local rank with respect to the vertices in the samples. The authors show that using random walk for collecting samples yields the best results on real-world graphs.

3.2 Closeness Centrality

A randomized approximation algorithm for weighted graphs was firstly proposed by [31]. Instead of computing APSP, their idea is to select a random set of vertices, also known as samples. For each of those vertices SSSPs are computed, and the remaining centrality scores are obtained by extrapolation from those shortest path computations. The algorithm runs in $O(\frac{\log n}{\epsilon^2}(n \log n + m))$ and approximates the scores within an additive error $\epsilon\Delta$ with high probability, where ϵ is a constant and Δ is the graphs' diameter. Different strategies for selecting the samples are investigated in [20], showing that random selection works best.

However, since the sample average is a poor estimator of the average distance, the algorithm is improved in [22]. Similar to the algorithm of [31], a set of k vertices is selected for which SSSP computations are conducted. For each of the remaining vertices, the centrality score is then approximated either by using the sample average or the centrality score of the closest vertex in the sample to the current vertex. The decision on either of the two variants is based on the distance of a vertex to the set of samples. This hybrid estimator achieves a small relative error and provides probabilistic guarantees for all graphs and vertices.

While approximation algorithms yield reasonably accurate centrality scores, they may still not be able to achieve the correct ranking since closeness centrality scores tend to be distributed within a narrow interval [73, 101]. Moreover, in practice often only the top-k centrality vertices are needed. Such approaches are presented in [9, 14, 56, 74].

Due to the fact that even for small sample set sizes, exact centrality computations become computationally expensive, machine learning techniques have been explored [40, 42, 66]. Mendonça et al. [66] propose an algorithm based on neural networks and graph embeddings. The encoder uses the adjacency matrix and the degree centrality values of all vertices as input variables and transforms them to a lower dimensional space using weight matrices. The resulting embeddings serve as inputs for iterative supervised machine learning. Through backpropagation the weight matrices are updated such that the approach can be used to predict the closeness centrality scores of vertices.

Another machine learning based approach was presented by Grando and Lamb [40]. Their idea is to use computationally efficient centrality measures as input variables for neural networks which then learn to predict more complex measures. Here, eigenvector and degree centrality are used as input and in the experimental evaluation it is shown that they result in high quality regression models for closeness and betweenness centrality.

Alternative approaches for estimating closeness centrality are presented in [80, 88, 91]. Saxena et al. [91] observe that the reverse ranking of closeness centrality follows a sigmoid curve, which can be described by a logistic curve whose parameters have to be estimated for a given graph. Once estimated, the closeness centrality rank of a vertex is computed in $O(1)$ time. The overall complexity of the algorithm is $O(m)$, however it does not provide any guarantees. Rattigan et al. [80] introduce a network structure index that uses annotations on vertices to estimate the distance between pairs of vertices. These estimates are used to compute closeness centrality scores. Although this approach does not offer guarantees, it performs well in experiments. In [88] the centrality preserving compressions and graph splits, which are used to reduce the complexity of the graph for faster centrality computations, were extended from betweenness to closeness centrality by the same authors. A more detailed description can be found in Section 3.4 and [87].

In their paper, Bader and Madduri [4] offer parallel implementations of both, the exact and the closeness approximation algorithm of [31] on high-end shared memory symmetric multiprocessor and multithreaded architectures. Furthermore, a parallel algorithm for closeness and betweenness centrality is introduced by Shi and Zhang [94]. They employ an efficient APSP algorithm in conjunction with graphics processing units (GPUs). Algorithms for computing closeness centrality on dynamic graphs are presented in [75, 86]. For a more detailed analysis the reader is referred to [45].

3.3 k-core Centrality

Batagelj and Zaversnik [7] propose a linear time algorithm for k-core decomposition. The key concept of the algorithm is, that by removing all vertices with a degree smaller than k as well as their incident edges, the resulting graph is a k-core (see Section 2.2.3). Using bin-sort, the algorithm runs in $O(n + m)$ but requires reading and writing to three different arrays in each iteration, which becomes expensive for large graphs due to memory latency [25]. The synchronization overhead makes this algorithm also challenging to parallelize.

A distributed k-core decomposition algorithm is given in [67]. The graph is partitioned and a subset of vertices is assigned to each processor. Additionally, for every vertex a list of its neighbors along with their corresponding core numbers is maintained. Initially, the core number of every vertex is equivalent to its degree. As local coreness estimates are updated, the changes are communicated to its neighbor processors. If necessary, they update their own core numbers and communicate those changes further. The algorithm is converged if there are no changes to communicate. However, this approach only works well if lots of processors are available, each responsible for a small set of vertices [25, 49]. Moreover, in the worst case the algorithm requires $O(nm)$ time.

The algorithm of Dasari et al. [25] computes the core numbers of each vertex by processing in levels. This means that all vertices within the l-core are identified before those in the subsequent $(l + 1)$-core are computed [49]. Initially, the core number of all vertices is set to the respective degree. In the scan phase, all vertices belonging to level l, i.e., those with core number l, are collected. In the process phase, the neighboring vertices are examined, which potentially results in a decrease of their core numbers. This in turn leads to more vertices with core number l which are processed in the next iteration. The whole process is repeated until no more vertices belong to level l [49]. The computational complexity is $O(k_{max}n + m)$ where k_{max} is the largest value of k for which a k-core exists in the graph. While this complexity is worse than the one of [7], it achieves much better running times in practice. The algorithm can further be parallelized by distributing all vertices of a level or sublevel among available threads, processing them independently and synchronizing at the end of each sublevel. In [49] the authors introduce an enhanced version of the latter approach, taking advantage of the fact that the majority of graph vertices have low core values. Additionally, they use less synchronization calls and atomic operations.

3.4 Betweenness Centrality

Due to its definition and versatile applicability, betweenness centrality is one of the most popular centrality measures. Various methods for computing exact and approximate results have been developed. They are categorized into six groups based on the underlying strategies [23, 62]:

1. Computation of fewer SSSP using sampling
2. Exploitation of parallelism
3. Exploitation of structural graph properties to compress or partition a graph
4. Reduction of complexity through decomposition into clusters
5. Exploitation of correlation with centrality measures that are easier to compute
6. Application of Machine Learning

An overview of the approaches for each group, recent advances and the state-of-the-art algorithms are presented. It's worth noting that some approaches can not be clearly assigned to one group but fit into multiple ones.

Sampling based Approaches. Inspired by [31], Brandes and Pich [20] use a set of sampled vertices to extrapolate the centrality scores of all other vertices. Different strategies for selecting the samples are investigated, and it is shown that random selection performs better than deterministic strategies. However, this approach leads to an overestimation of betweenness centrality scores for vertices which are close to the samples [38]. Thus, Geisberger et al. [38] propose a different estimator, which is not affected by this issue and performs well in experiments. The algorithm provides good approximations even for less important vertices and can also be parallelized. However, it does not have a theoretical guarantee [2, 38, 62].

Instead of a fixed number of samples, the algorithm introduced in [5] uses an adaptive approach where the number of samples depends on the information obtained from each sample. It approximates the centrality score of a single vertex and provides a theoretical guarantee, which however only holds for high centrality vertices.

In [81], the authors introduce an alternative sampling based approach. Instead of sampling vertices, they sample vertex pairs $(s,t) \in V \times V$ with $s \neq t$ uniformly at random. For each such pair, a shortest st-path is also sampled at random. Repeating this τ times results in a sequence of shortest paths $\pi_1, \pi_2, \ldots \pi_\tau$ [99, 100]. The betweenness centrality of a vertex v is then approximated by calculating the fraction of those shortest paths that go through v and is defined as

$$\tilde{c}_B = \frac{1}{\tau}\sum_{i=1}^{\tau} x_i(v), \qquad x_i(v) = \begin{cases} 1 & \text{if } v \in \pi_i \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

Using this technique, the betweenness centrality scores of all and of the top-k vertices are within an additive and a multiplicative factor $\varepsilon \in (0,1)$ from the exact ones with probability at least $(1-\delta), \delta > 0$, respectively. Additionally, a lower bound for the number of samples required to achieve a given error guarantee is given. However, this involves the computation of an upper bound on the vertex-diamter, the maximum number of vertices on any shortest path [82].

The ADaptive Algorithm for Betweenness via Random Approximation (KADABRA) algorithm of Borassi and Natale [13] further improves the approach of [81] by

1. Using a balanced bidirectional BFS, which runs in $O(|E|^{\frac{1}{2}+o(1)})$ on many real-world networks
2. Using an adaptive sampling technique which decreases the required amount of sampled shortest paths

The algorithm stops if $f(v), g(v) < \varepsilon, \ \forall v \in V$ where $f(v) = (\tilde{c}_B(v), \delta_L(v), \omega, \tau)$ and $g = (\tilde{c}_B(v), \delta_U(v), \omega, \tau)$. ω is the maximal number of samples and $\delta_L(v), \delta_U(v)$ are per-vertex failure probabilities. Consequently, the KADABRA algorithm is divided into the following phases [99]:

1. Computation of the diameter to determine the value of ω
2. Calibration of δ_L and δ_U using a fixed number of samples
3. Computation of betweenness centrality scores using adaptive sampling of shortest paths

A similar adaptive sampling approach called Approximating Betweenness with Rademacher Averages (ABRA) is presented in [82]. It uses Rademacher averages for the stopping criterion instead of the vertex diameter and a scheduler to decide when this stopping criterion should be tested. Even though both, KADABRA and ABRA provide the same theoretical guarantees, it is shown in [13] that KADABRA exceeds ABRA in terms of approximation quality and runtime. This has also been confirmed in the empirical comparison of [62] and thus, KADABRA is the state-of-the-art algorithm for approximation of betweenness centrality scores [99].

The shared-memory parallelization of KADABRA presented by van der Grinten et al. [100] makes the approach applicable to graphs with hundreds of millions of edges with an error of $\varepsilon = 0.01$. This is further improved by the same authors in [99] where they provide a Message Passing Interface (MPI)-based implementation of KADABRA, which outperforms other implementations in terms of speed and also enables the computation of betweenness centralitiy scores for graphs with billions of edges.

Exploitation of Structural Graph Properties. Another technique to speed up betweenness centrality computations involves reducing the size of the graph using reductions or partitioning. The authors of [78] combine two heuristics to achieve a speedup over Brandes' algorithm. They firstly identify structurally equivalent vertices, i.e., vertices that have the same set of neighbors and in particular the same centrality scores, and then contract these in linear time. In the second heuristic, the graph is partitioned into bi-connected components. Finally, the exact centrality scores of all vertices are calculated component-wise. The idea of contracting structurally equivalent vertices is also used by [87], together with compressing the graph by, e.g., removing vertices of degree 1. The graph is split along bridges and articulation vertices, i.e., edges and vertices which increase the number of connected components when removed. These two steps – compressing and splitting – are iteratively repeated until the graph does not change anymore. The exact betweenness centrality scores are computed on the different parts of the graph. Since information about pair and source dependencies is kept throughout the reduction process, the exact centrality scores of all vertices are determined.

Decomposition into Clusters. The idea of these approaches is to divide the graph into clusters, compute the betweenness centrality scores for the vertices inside the clusters and then on the remaining part of the graph [23]. The computations on the sub-graphs should be faster than on the original graph, leading to an overall speedup of the approach. The algorithm of [32] first partitions the graph, runs SSSP computations on the different subgraphs and then uses the centrality scores obtained on the skeleton – a simplified hierarchical representation of the graph – to compute all betweenness centrality values. However, these centrality scores are computed only with respect to a subset S of vertices and for $S = V$, the algorithm converges towards Brandes' algorithm with longer computation times [23].

In [23] clustering is used to identify the equivalence classes in a graph. An equivalence class consists of all vertices that, when used as sources for SSSP computations, yield identical dependency scores for all vertices outside the cluster [23]. The calculation of betweenness centrality consists of computing local and global dependencies, which are pair dependencies between vertices of the same and of different clusters, respectively. By selecting a representative for each equivalence class, the amount of SSSP computations and the runtime is reduced. To avoid errors, such as intra-cluster errors, hierarchical sub-network corrections are used, resulting in exact betweenness centrality scores. The authors provide a sequential and parallel map-reduce version of their algorithm. More cluster based approaches can be found in [58, 97].

Exploitation of Correlation. In the last years, lots of variants of betweenness centrality have been introduced. Firstly, because they focus on different aspects of importance, and secondly, because they are used to approximate betweenness centrality if the two measures correlate and the other one is easier to compute. One such measure is k-betweenness centrality where the length of the shortest paths is limited to k [16, 77]. Pfeffer and Carley [77] showed that for graphs with low average degree, k-betweenness centrality scores can be computed in $O(n)$ while achieving high qualitative approximate results. In [53] the correlation between betweenness centrality and k-path betweenness centrality, which considers simple paths of length at most k, is investigated. The authors provide a randomized algorithm that runs in $O(k^3 n^{2-2\alpha} \log n)$ with probabilistic guarantees for computing the latter and show experimentally that vertices with high k-path centrality have high betweenness centrality scores. Other variants of betweenness centrality include, e.g., flow betweenness centrality, which considers paths of all lengths without cycles [37], routing betweenness centrality, which includes arbitrary loop-free routing schemes [29] and random-walk betweenness centrality, which is based on an electrical current model and random walks [72].

Exploitation of Parallelism. Other research focuses on the development of parallel computing approaches, multicore architectures, and GPU technology to increase the scalability of betweenness centrality calculations. Bader and Madduri [4] use shared memory symmetric multiprocessor and multithreaded architectures to implement the exact algorithm in parallel. This is further enhanced by the same authors through reducing synchronization overhead and optimized cache locality [61]. Edmonds et al. [30] propose an algorithm which not only parallelizes shortest path computations but also maintains low space complexity through a hybrid approach that combines different level of parallelism and distributes vertices among processors. A fully distributed algorithm is presented in [10]. The authors provide a distributed preprocessing algorithm, which removes degree-1 vertices before using bi-dimensional decomposition to split the graph onto multi-GPU systems. Exact betweenness centrality scores are computed by combining coarse- and fine-grained parallelism with a data-thread mapping based on prefix sum operations.

Sariyüce et al. [85] introduce a heterogeneous approach, which combines different levels of parallelism as well as central processing unit (CPU) and GPU architectures. They utilize a vertex-based approach combined with virtualization of high degree vertices in order to avoid load imbalance and reduce the graph by iteratively removing degree-1 vertices. The combination of all those techniques makes the approach very efficient. However, the runtime depends on the diameter of the graph and is in the worst case quadratic in the number of vertices. Betweenness centrality scores of graphs with large diameters can be efficiently calculated using [64]. The authors provide a work-efficient method on the GPU, which works well for these types of graphs and an edge-parallel approach that is applicable for graphs with smaller diameter. Depending on the structure of the graph, an online hybrid approach decides on one of the two.

Machine Learning. In [63] the authors introduce a graph neural network (GNN) based approach with constrained message passing scheme, which estimates betweenness centrality scores. The focus is thereby set on the relative ranking rather than exact scores. By aggregating features within a multi-hop neighborhood, the model learns to predict how many vertices a given vertex can reach. The approach yields qualitative promising results, while requiring less computation time.

Fan et al. [33] use an encoder-decoder framework that maps each vertex to an embedding vector before estimating its centrality score. For the encoder, a GNN is used. It aggregates for each vertex information about its neighbors instead of computing shortest paths, similar to the previous approach. The decoder consists of a multi-layer perceptron which maps each embedding vector to a centrality score. This achieves overall good accuracy. Further machine learning techniques for betweenness centrality are provided in [40, 42, 66]. Since these also include the estimation of closeness centrality scores, a detailed description has already been presented in Section 3.2.

3.5 Eigenvector Centrality

As described in Section 2.2.5, eigenvector centrality scores are computed using the power iteration method. This can be speedup by adjusting the error tolerance such that the algorithm stops earlier or by using parallelism and GPUs [93, 96].

Rakaraddi and Pratama [79] use the adjacency matrix and the degree of each vertex as inputs for an encoder-decoder model, which generates centrality scores for each vertex. They show that the proposed algorithm works best with unsupervised learning and performs well in experiments, particularly at identifying the most central vertices. A detailed description for a similar machine learning approach can be found in Section 3.2 and [66].

3.6 Katz Centrality

Solving or approximating the system of linear equations in Section 2.2.6 with numerical iterative solvers yields good results. However, it takes lots of iterations to converge and is thus not applicable for large graphs. In [35] the authors propose to compute Katz centrality iteratively by computing the recurrence $c_{K,i+1} = \alpha A c_{K,i} + \vec{I}$ until a fixed point or a predefined number of iterations is reached. This approach runs in $O(n+m)$ time and performs well in practice but does not provide guarantees.

In order to obtain the correct ranking of vertices, the calculated centrality scores do not necessarily have to be fully correct, they just have to yield the correct order. Therefore, Nathan et al. [70] derive error bounds on the centrality ranking and show that using these bounds, vertices with high Katz centrality can be found without computing all centrality scores. They develop a stopping criterion, which can be used with any iterative solver and decreases the amount of iterations needed to identify most central vertices. For the full ranking, however, this does not lead to a speedup [43].

A sampling based approximate algorithm is presented in [59]. For a given vertex, a set of vertices and edges is sampled first. An approximate Katz centrality score is then obtained by counting the number of walks in that subgraph. These steps are repeated multiple times to obtain the desired accuracy. The authors show that the theoretical estimator is unbiased and that estimated values can be bounded with high probability. Working with subsets makes the approach even more efficient, which is demonstrated by the experiments. For a single vertex the algorithm requires $O((l(d-1)\frac{r^2}{n^2}m + ls))$ time, where r is the size of sampled subset, s is the size of the successors set for the given vertex and l, d are small constants. The approach can be efficiently extended to multiple vertices by reusing intermediate results.

The algorithm in [43] also computes approximate Katz centrality scores and guarantees the correct ranking by iteratively improving upper and lower bounds on the centrality score of each vertex. Different ranking criterions can be used, e.g., the

- *top-k criterion* which ensures that the top-k vertices are ranked correctly or the
- *ranking criterion* which ranks all vertices correctly.

A ranking is considered correct, if the respective vertices are ϵ-seperated, i.e., for the lower bound $l_r(w)$ of a vertex w and the upper bound $u_r(v)$ of a vertex v it holds $l_r(w) > u_r(v) - \epsilon, \epsilon > 0$. The algorithm runs in $O(r|E| + r\mathcal{C})$, where $\mathcal{C}$ is the complexity of the *top-k criterion* which requires $O(|V| + k\log(k))$ time. A speedup of at least 50% compared to numerical methods is achieved and in the same paper, an extension to dynamic graphs as well as parallel CPU and GPU implementations are provided. For more implementations of Katz centrality on dynamic graphs, the reader is referred to [45, 69].

CHAPTER 4

Edge Sparsification

In this chapter, the shortest path based edge sparsification is explained. Firstly, the overall structure as well as associated parameters are provided. Then, the two primary components – the computation of edge counts using a BFS approach and the construction of the reduced graph based on MWSFs – are described in detail. Finally, the time and space complexity of the algorithm is analyzed.

4.1 Overview

In order to speedup centrality computations, the graph is first reduced using the edge sparsification. Edges that are part many shortest paths, are of particular importance for both path-based centrality measures and the overall topological structure of a graph. Consequently, these edges are preserved in the reduced graph, while others are removed. The core components of the approach are the following:

1. Computation of edge counts
   The edge count is defined for each edge and approximates how often this edge occurs on shortest paths. These counts are computed using one of the three BFS-based variants detailed in Section 4.2.

2. Construction of the reduced graph
   Based on the computed edge counts, the reduced graph is constructed using a MWSF. Two such approaches are described in Section 4.3.

The entire procedure of the edge sparsification is outlined in Algorithm 3. Using the parameters described in Table 4.1, an undirected, unweighted graph $G = (V, E)$ is reduced to a graph $G' = (V, E')$ with $E' \subset E$. Initially, every edge has count zero and thus e_{counts} is initialized accordingly, containing $m = |E|$ elements in total. A set of vertices, whose size depends on the parameter *set_size*, is selected randomly to compute the edge counts by

Algorithm 3 Edge Sparsification of Graph G

```
    **Input** The graph $G = (V, E)$ and the parameter list in Table 4.1
    **Output** The reduced graph $G' = (V, E')$
 1: $e_{counts} \leftarrow$ empty array of size $m$
 2: **for** edge $e$ in $E(G)$ **do**
 3:     $e_{counts}[e] \leftarrow 0$
 4: **end for**
 5: **for** $i \leftarrow 1$ to *bfs_repetitions* **do**
 6:     $subset\_vertices \leftarrow$ select *set_size* vertices at random from $V(G)$
 7:     $calculate\_edge\_counts(e_{counts}, subset\_vertices,$ *bfs_method*$)$
 8: **end for**
 9: $G_w \leftarrow G$ with edge weights $e_{counts}$
10: $G' \leftarrow construct\_reduced\_graph(G_w,$ *edge_percentage*, *mwsf_method*$)$
11: **return** $G'$
```

one of three BFS variants, i.e., *bfs_set*, *bfs_single* and *bfs_percentage*. Each of them produces shortest path trees, which are used to increment the associated edge counts (see Section 4.2). These two steps are repeated as often as specified in parameter *bfs_repetitions*. Since exactly one count is assigned to each edge, these can also be viewed as edge weights, transforming the originally unweighted graph into a weighted one. In the next step, this graph is then used to construct the reduced graph G', employing one of the two MWSF-based methods described in Section 4.3. The amount of edges added to the reduced graph is in both variants restricted by the parameter *edge_percentage*, which defines how many edges the reduced graph may contain with respect to the full one. The resulting graph G' is a spanning subgraph of G.

Parameter	Description
set_size	size of the subset of vertices
bfs_method	defines which approach is used to run the BFS
bfs_percentage	specifies the percentage of vertices that has to be reached in each iteration of the *bfs_percentage* approach
repetitions	number of repetitions of edge count calculations
edge_percentage	amount of edges in percent which should be part of the reduced graph
mwsf_method	defines which MWSF-based approach should be used to construct the reduced graph

Table 4.1: Input parameters of the edge sparsification algorithm

4.2 BFS Variants

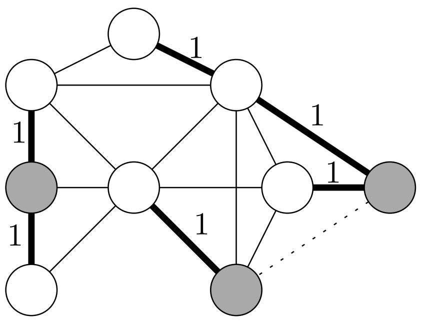
The objective of all the methods outlined below is to determine edge counts, which estimate how often each edge is included in shortest paths within the graph. This process involves the following steps: Starting from a set of vertices, a BFS tree is constructed. The edge counts of all edges belonging to that tree, i.e., which are part of some shortest paths, are increased by one. Thereby, the methods vary in how they construct the shortest path tree. In the following, each approach is explained and illustrated using an example graph.

4.2.1 BFS-Set

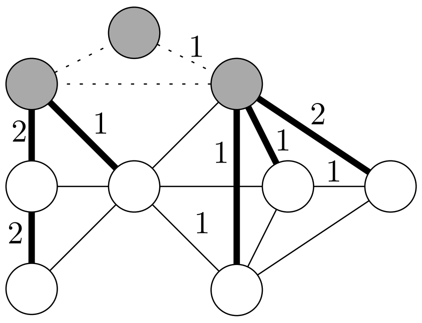
Instead of initiating SSSP computations from a single vertex, this approach uses multiple ones – specifically, all vertices contained in the $subset\_vertices$ variable of Algorithm 3. This is achieved by setting these vertices as ’visited’ and inserting them directly into the queue (see Algorithm 4). Similar to a normal BFS, the graph is then traversed in levels,

Algorithm 4 Computation of edge counts using BFS-set

```
    Input Array $e_{counts}$ and $subset\_vertices$ containing the edge counts and a subset of
    vertices respectively
    Output Incremented edge counts for all edges belonging to the shortest path tree
 1: Initialize an array $visited$ with all elements set to false
 2: $Q$ := a queue data structure
 3: for each vertex $v$ in $subset\_vertices$ do
 4:     $visited[v] \leftarrow true$
 5:     insert $v$ to the end of $Q$
 6: end for
 7: while $Q$ is not empty do
 8:     $u \leftarrow$ remove and return vertex from $Q$
 9:     $neighbors \leftarrow$ empty array
10:     for each neighbor $v$ of $u$ do
11:         if $visited[v]$ is $false$ then
12:             $visited[v] \leftarrow true$
13:             $e_{counts}[\{u, v\}] \leftarrow e_{counts}[\{u, v\}] + 1$
14:             insert $v$ into $neighbors$
15:         end if
16:     end for
17:     shuffle the elements of $neighbors$ at random
18:     add elements of $neighbors$ to the end of $Q$
19: end while
20: return $e_{counts}$
```



(a) BFS tree and incremented edge counts using the gray colored vertices as subset.



(b) Incremented edge counts after the second BFS with a different subset

Figure 4.1: Two iterations of the BFS-set approach with subsets consisting of three vertices. Dotted lines connect vertices within the subset, while edges of the BFS tree are depicted in bold. Unnumbered edges have an edge count of zero.

starting from these very vertices. For each vertex in the queue the following steps are executed. First, its neighbors are investigated. If a neighboring vertex has not been visited yet, a shortest path to that vertex is found and simultaneously, an edge of the shortest path tree is identified. The edge count of the respective edge is then incremented by one. All newly visited neighbors are then added to the queue in random order. These steps are repeated for every vertex and the algorithm terminates when the queue is empty and all vertices of the graph have been visited.

This process is illustrated in Figure 4.1. The subset of vertices is colored in gray and the edge counts are represented as edge weights. Edges without any number have count zero. After running Algorithm 4, edges which are part of the shortest path tree – highlighted in bold – have count 1 while the others still have count zero. Due to the nature of this approach, edges connecting vertices within the subset are not considered and thus illustrated as dotted lines (see Figure 4.1a). This also implies that the BFS does not construct a tree but instead results in disconnected components. Only if the vertices within the subset are treated as a single, large starting vertex, the result is a tree. If multiple iterations are specified, the previous steps are repeated with a new at random selected set of vertices (see Figure 4.1b). Since the edge counts are not reset between individual runs of Algorithm 4, they range from zero to one after one iteration and from zero to *bfs_repetitions* after *bfs_repetitions* many iterations, respectively.

4.2.2 BFS-Single

Similar to the previous approach, the BFS-single method receives the edge counts array, initialized with zeros and a randomly selected subset of vertices as input. However, instead of just one BFS run, this variant initiates individual runs for each vertex in the subset. The traversal for the graph and the incrementation of edge counts follow the same procedure

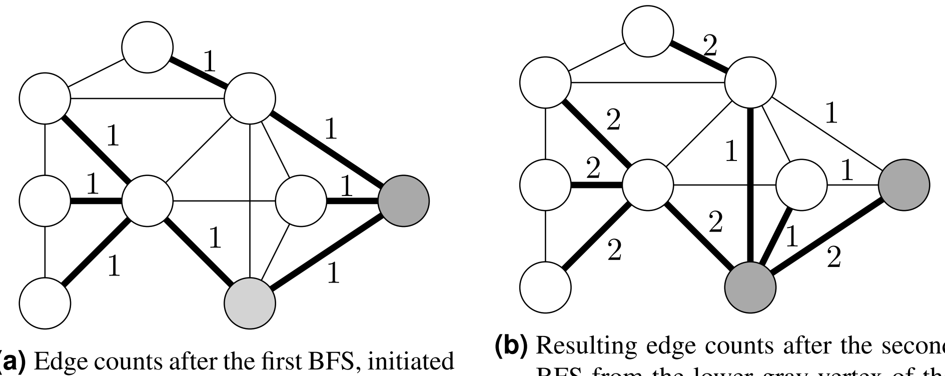
Algorithm 5 Computation of edge counts using BFS-single

```
    Input Arrays $e_{counts}$ and $subset\_vertices$ containing the edge counts and a subset of vertices, respectively
    Output Incremented edge counts of the edges belonging to the shortest path trees
1: for each vertex $v$ in $subset\_vertices$ do
2:     $e_{counts} \leftarrow$ BFS_SET($e_{counts}$, $[v]$)        ▷ Call Algorithm 4 with just one vertex in the subset
3: end for
4: return $e_{counts}$
```

as in the BFS-set approach. Thus Algorithm 4 is invoked with just one vertex at a time (see Algorithm 5). In Figure 4.2 one iteration of the BFS-single approach is shown. The gray colored vertices in Figure 4.2a are again the ones belonging to the randomly selected subset. In the first run, the BFS tree is computed starting from the right dark gray vertex, indicated with bold edges. Consequently, the counts of all edges belonging to that tree are incremented by one. In contrast to the previous method, this approach naturally encompasses all edges and as a result, the shortest path tree is an actual tree, which also contains a path to the other start vertex. Using Algorithm 4, a BFS tree is then also computed for the second vertex in the subset (see Figure 4.2b). In total, *set_size* SSSP computations are performed in each iteration, leading to edge counts ranging from 0 to *set_size* for one and from 0 to *set_size* · *bfs_repetitions* for all iterations.

4.2.3 BFS-Percentage

In this approach, the BFS is not terminated when all vertices of the graph are visited. Instead, it is stopped earlier if a certain percentage of vertices, determined by the parameter



(a) Edge counts after the first BFS, initiated from the right gray vertex

(b) Resulting edge counts after the second BFS from the lower gray vertex of the subset

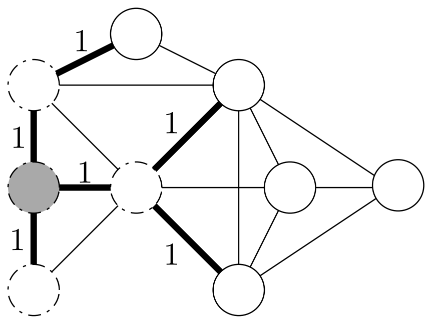
Figure 4.2: Steps of the BFS-single approach for a subset of two vertices. Edges which belong to the BFS tree are drawn bold and edges without number have edge count zero.

Algorithm 6 Computation of edge counts using BFS-percentage

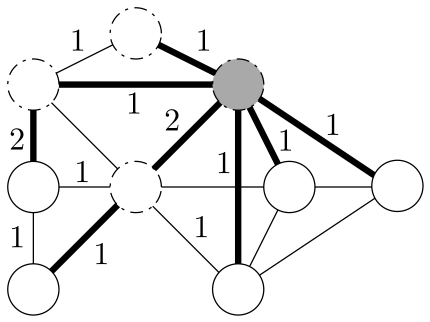
```
    **Input** Array $e_{counts}$, containing the edge counts per edge, one randomly selected start vertex $s$ and the parameter *bfs_percentage*
    **Output** Incremented edge counts for all edges belonging to the shortest path trees
 1: Initialize an array $reached$ of size $n$ with all elements set to false
 2: $unreached \leftarrow [s]$
 3: while $unreached$ is not empty do
 4:     $u \leftarrow$ select one vertex of $unreached$ at random
 5:     $e_{counts} \leftarrow$ BFS_SET$(e_{counts}, [u], reached, bfs\_percentage)$
 6:     remove all elements from $unreached$
 7:     for each vertex $v$ of $V$ do
 8:         if $reached[v]$ is $false$ then
 9:             insert $v$ into $unreached$
10:         end if
11:     end for
12: end while
13: return $e_{counts}$
```

bfs_percentage, is reached. The overall procedure is outlined in Algorithm 6. First, the vertices of the subset are added to the $unreached$ list. Note, that for this approach the subset is assumed to contain just one vertex. An extension to multiple ones is discussed in Section 6.2. From the unreached vertices, one is then selected at random. Starting from this very vertex, a run of a slightly modified BFS-set algorithm is initiated. This algorithm differs from the one presented in Section 4.2.1 in the following aspects. A vertex is considered reached if it is removed from the queue during the BFS, implying that all its outgoing edges have been explored. The $reached$ array has to be updated accordingly. Secondly, the stopping criterion of the loop must be adjusted, since the BFS has to be stopped if *bfs_percentage*$\cdot n$ vertices of the graph are reached. Note, that this criterion refers to the vertices reached in the current BFS run. This finally results in calculated edge counts and an updated $reached$ array, which is then, in turn, used to update the list of unreached vertices. The procedure is repeated until all vertices of the graph have been reached in at least one of the BFS runs.

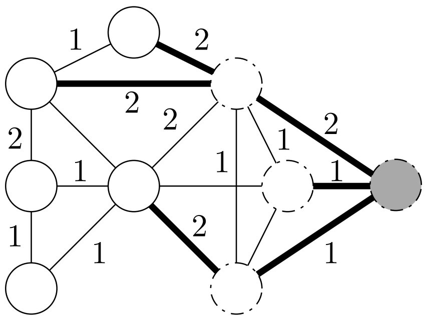
In Figure 4.3a, the randomly selected vertex is colored in gray and let *bfs_percentage* be set to 0.5, i.e., $\lfloor 4.5 \rfloor = 4$ vertices. Starting from the gray vertex, all its outgoing edges are explored, the neighbors are added to the queue and the edge counts are incremented. Additionally, the gray vertex is set to reached, represented by the dotted outline. In the subsequent iterations, all neighbors of the gray vertex are processed similarly and they, too, are marked as reached. Since this accumulates to a total of four vertices, the BFS terminates. Subsequently, a new vertex is selected from the unreached vertices, initiating another BFS until again four vertices are reached (see Figure 4.3b). After the second iteration, not all vertices are reached and thus, a third BFS is run from the gray vertex in Figure 4.3c.



(a) Incremented edge counts and partial BFS tree after reaching 50% of the vertices



(b) Resulting edge counts after a second BFS from an unreached vertex



(c) Resulting edge counts after running a third BFS from an unreached vertex. Now, all vertices have been reached.

Figure 4.3: Steps of the BFS-percentage approach with *bfs_percentage*$= 0.5$. Start vertices of the BFS are colored gray and vertices which have been reached are drawn with dotted outline. The BFS tree is highlighted with bold edges and edge counts. Edges without numbers have edge count zero.

4.3 Construction of the Reduced Graph

The last step of Algorithm 3 consists of constructing the reduced graph G' based on $G_{weighted}$, i.e., the original graph G and the edge counts computed by one of the BFS variants (see Section 4.2). The goal is thereby to construct G' such that it is a connected spanning subgraph primarily composed of edges with high counts. Moreover, G' may not contain arbitrary many edges, but only a certain percentage of those contained in G, which is specified by the parameter *edge_percentage*. Note, that all edges are added to the reduced graph without any associated weights, resulting in an unweighted, connected graph. Two such approaches are discussed in the following.

Algorithm 7 Construction of the reduced graph G' using a single MWST

```
    Input The graph $G_w$ and the parameter edge_percentage
    Output The reduced graph $G'$ of $G$
1: $G' \leftarrow (V, E')$ where $E'$ is empty
2: $G_{MWST} \leftarrow Kruskal(G_w)$
3: add edges $E(G_{MWST})$ to $G'$
4: $e_{sorted} \leftarrow$ sort edges $E(G_w) \setminus E(G')$ in descending order
5: add the first edge_percentage$\cdot|E(G_w)|$ elements of $e_{sorted}$ to $G'$
6: return $G'$
```

4.3.1 MWST and Sorted Edges

In the first step of Algorithm 7 a MWST is computed using Kruskal's algorithm where edges with equal weights are considered in random order. Since G, and consequently, G_w are connected, such a MWST is guaranteed to exist in G_w. The edges of that tree are then added to the reduced graph G', ensuring its connectivity. The remaining edges, which are not part of the MWST, are sorted in descending order with respect to their weight, i.e., their edge count. Finally, the top k edges with the highest counts are added to the reduced graph. k depends on the parameter *edge_percentage* and can be computed by $k =$ *edge_percentage*$\cdot|E(G_w)|$. As this approach is based on a single MWST or MWSF, it is referred to as MWSF-single.

4.3.2 Iterative Computation of the Reduced Graph

The second approach iteratively computes G' using the steps outlined in Algorithm 8. First, Kruskal's algorithm is applied to $(V, E(G_{weighted}) \setminus E(G'))$. Since G' does not contain any edges in the first iteration, the result of that is a MWST by the same argument as in the previous approach. The edges from this tree are then incorporated into the reduced graph. In the next iteration, Kruskal's algorithm is called again, but since $E(G')$ is now non-empty, $(V, E(G_{weighted}) \setminus E(G'))$ does not necessarily have to be connected and thus Kruskal's algorithm computes a MWSF, indicated by G_{MWSF}. The edges are added to G' and the next iteration follows. This procedure is repeated until at least k edges, where $k =$ *edge_threshold*$\cdot|G_{weighted}(E)|$, are contained in the reduced graph. Due to the iterative computation of MWSFs, this method is referred to as MWSF-iterative.

4.4 Complexity Analysis

Theorem 2
The BFS variants BFS-set, BFS-single and BFS-percentage require $O(bfs\_repetitions(n + m))$, $O(bfs\_repetitions \cdot set\_size(n + m))$ and $O(bfs\_repetitions(n^2 + nm))$ time. Constructing the reduced graph can either be done in $O(m \log m)$ using MWSF-single or

Algorithm 8 Iterative construction of the reduced graph G' using MWSFs

```
    Input The graph $G_w$ and the parameter edge_percentage
    Output The reduced graph $G'$ of $G$
1: $G' \leftarrow (V, E')$ where $E'$ is empty
2: $num\_edges\_reduced = edge\_percentage \cdot |E(G_w)|$
3: while $|E(G')| < num\_edges\_reduced$ do
4:     $G_{MWSF} \leftarrow Kruskal((V, E(G_w) \setminus E(G')))$
5:     add edges $E(G_{MWSF})$ to $G'$
6: end while
7: return $G'$
```

in $O(nm \log m)$ with MWSF-iterative. By combining the latter with BFS-percentage, the worst case complexity of Algorithm 3 is therefore $O(bfs\_repetitions(n^2 + nm) + nm \log m) = O(bfs\_repetitions \cdot n^2 + nm \log m)$.

Proof. In order to determine the complexity of Algorithm 3, each BFS and MWSF variant is analyzed separately. The BFS-set approach is a BFS with minor modifications, requiring $O(n + m)$ time (see Algorithm 4). In the BFS-single approach, Algorithm 4 is executed for each vertex of the subset separately (see Algorithm 5) which implies an overall complexity of $O(set\_size(n + m))$. The initialization steps of BFS-percentage shown in Algorithm 6 can be computed in $O(n)$. Inside the loop, Algorithm 4 is called for a randomly selected vertex and afterwards unreached vertices are determined by iterating over the $reached$ array. The latter can be computed in $O(n)$ and thus, the complexity of BFS-percentage depends on how often Algorithm 4 is executed.

One might assume that the highest complexity is reached when Algorithm 4 has to be executed for every vertex. However, this would imply that the algorithm terminates after reaching only one vertex. The complexity of Algorithm 4 then reduces to $O(1 + d(v))$ for a single vertex v, resulting in a total runtime of $O(n + m)$ for BFS-percentage. Instead, the worst-case complexity occurs when, in each call of Algorithm 4, only already reached vertices, except for the starting vertex, are visited. The more vertices are newly reached, the longer the runtime of Algorithm 4, but at the same time, the number of necessary iterations decreases. If always the same 50% of the vertices in a graph are reached, with only the starting vertex being previously unvisited, then Algorithm 4 must be called for each of the remaining 50% of the vertices. If the majority of edges in the graph runs between the repeatedly visited 50%, not only the number of iterations but also the runtime of Algorithm 4 is maximized.

Consider a graph G with n vertices and m edges with the following structure. Let $\frac{n}{2}$ of the vertices form a complete graph and let each of the remaining $\frac{n}{2}$ vertices be connected to exactly one vertex from that complete graph. Note, that two such vertices may not be connected to the same vertex in the complete subgraph. The resulting graph G then contains $\frac{n(n-1)}{2} + \frac{n}{2}$ edges. An example graph with 10 vertices is shown in Figure 4.4. Let

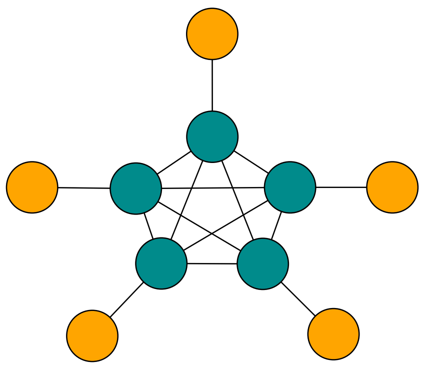


Figure 4.4: Example of a graph with 10 vertices where the five blue vertices form a complete subgraph and the orange vertices have degree 1.

the parameter *bfs_percentage* then be defined such that exactly $\frac{n}{2} + 1$ vertices are reached in each run and let further the first randomly selected vertex be a vertex of degree one, i.e., a vertex that does not belong to the complete subgraph.

Algorithm 4 then traverses the vertices in the following order: When the start vertex is dequeued, its only neighbor is enqueued. Since this vertex is part of the complete subgraph, all its neighbors, i.e., all vertices from the complete graph, are inserted into the queue and traversed accordingly. Eventually, also unreached vertices will be contained in the queue. However, since *bfs_percentage* is chosen such that only $\frac{n}{2} + 1$ are reached, the algorithm terminates after the last vertex belonging to the complete subgraph is dequeued. Consequently, in this iteration, no additional previously unreached vertices, apart from the starting vertex, are reached. Thus, all other degree-1 vertices remain unreached, and inevitably, a degree-1 vertex will be used as the starting vertex in the next iteration. The previously described steps are repeated, and consequently, Algorithm 4 must be invoked separately for all degree-1 vertices.

In a single run of Algorithm 4, $\frac{n}{2} + 1$ vertices and $\frac{n(n-1)}{2} + 1$ edges are traversed. Since $\frac{n(n-1)}{2} + 1 \approx m$, this requires $O(\frac{n}{2} + 1 + m)$ time. The overall complexity of BFS-percentage is then $O(\frac{n}{2}(\frac{n}{2} + 1 + m)) = O(n(n + m))$ because there are $\frac{n}{2}$ degree-1 vertices. In total, the complexity for computing the edge counts is

- $O($*bfs_repetitions*$(n + m))$ for BFS-set,
- $O($*bfs_repetitions* $\cdot$ *set_size*$(n + m))$ for BFS-single and
- $O($*bfs_repetitions*$(n^2 + nm))$ for BFS-percentage.

Using a union-find data structure with union by rank and path compression as described in Chapter 2, the complexity of Kruskal's algorithm is $O(m \log m)$ [65]. For the MWSF-single approach outlined in Section 4.3.1, which computes one MWST and then adds edges with high counts to the reduced graph, the sorting of edges is the most expensive part and thus, the overall complexity remains $O(m \log m)$.

It can be seen in Algorithm 8 that for MWSF-iterative the complexity depends on the amount of times Kruskal's algorithm is executed. Assume that the maximum number of iterations is required, i.e., the parameter *edge_percentage* is set to one. In each iteration, either $n-1$ or $n-c$ edges, where c is the amount of connected components, are removed from the graph. However, c depends on the structure of the graph and can also vary between iterations. An upper bound on the number of iterations is obtained by considering a complete graph. This graph can be constructed by the union of edge-disjoint spanning cycles (when n is odd) or it can be factored in paths [21, 76]. Thus, there exist $\lfloor \frac{n}{2} \rfloor$ disjoint spanning trees, implying that at most $n-1$ iterations are possible. Even though it is possible that in an arbitrary graph fewer edges are added per iteration, the total number of required iterations is then smaller since the graph contains fewer edges. The overall complexity of MWSF-iterative is therefore $O(\lfloor \frac{n}{2} \rfloor m \log m) = O(nm \log m)$. By combining BFS-percentage and MWSF-iterative, the statement follows.

Theorem 3
Algorithm 3 requires $O(n+m)$ space for all combinations of BFS and MWSF approaches.

Proof. Storing the graph using an adjacency list requires $O(n+m)$ space. This also holds for the edge counts array and the subset of vertices. As for a normal BFS without any modifications, BFS-set requires $O(n)$ space for the queue, the visited and the neighbors array (see Algorithm 4) in total. Since BFS-single calls Algorithm 4 multiple times, its space complexity is also $O(n)$. In addition to $O(n)$ space for Algorithm 4, BFS-percentage stores reached and unreached vertices. This also requires $O(n)$ space. Using a union-find data structure, Kruskal's algorithm needs only $O(n)$ memory. Thereby, no additional weighted graph G_w has to be created in order to implement Kruskal's algorithm. Moreover, edges can be added directly to the reduced graph without constructing the intermediate graph G_{MWSF}. Thus, both MWSF approaches require $O(n)$ space. Combining these arguments leads to the statement.

CHAPTER 5

Experimental Evaluation

In this chapter, an experimental evaluation of the edge sparsification algorithm, described in Chapter 4, is provided. First, the general structure of the experiments as well as evaluation measures are presented, followed by the utilized graph instances. To determine suitable parameter configurations a series of tuning experiments is conducted. Finally, the results on reduced and unreduced graphs are compared.

5.1 Hardware and Implementation

The edge sparsification is implemented in C++ and compiled with gcc version 9.4. All experiments are performed on an Intel(R) Xeon(R) Silver $4{,}216$ @ 2.10GHz with 16 cores and 93 GB main memory under Ubuntu 20.04.1 LTS and Linux kernel version 5.4.0-152-generic. The computation of the evaluation measures is done in python 3.10.12 using the libraries numpy (version 1.25.2) and scipy (version 1.11.2).

The code strongly relies on the open-source toolkit `NetworKit`[1] [2] version 10.1, which includes a variety of efficient algorithms for network analysis. Graphs are represented using an adjacency array data structure which requires $O(n + m)$ memory where n and m are the number of vertices and edges, respectively. Each vertex is represented by a 64 bit integer and within a graph, these IDs form a consecutive range. Edge IDs can be assigned optionally. `NetworKit` provides multiple exact and approximate algorithms for centrality computations. The following ones are used in the experiments[2]:

- `DegreeCentrality`: This algorithm computes the degree centrality of each vertex, normalized by the maximum degree of a graph in $O(n)$.

[1] `https://networkit.github.io/` (Accessed Dec. 1, 2023)

[2] `https://networkit.github.io/dev-docs/python_api/centrality.html` (Accessed Dec. 1, 2023)

- `ApproxCloseness`: This function implements the algorithm of [22]. The scores are normalized by $n-1$ and the parameters $nSamples = 1000$ and $\epsilon = 0.03$ are used.
- `CoreDecomposition`: The algorithm is an implementation of the ParK-algorithm of [25] with normalization by the maximum degree of a graph.
- `KadabraBetweenness`: Using the improvements of [100], this algorithm computes betweenness centrality based on the KADABRA algorithm [13]. The centrality scores are normalized by $n(n-1)$ per default.
- `EstimateBetweenness`: This function implements the algorithm of [38], which provides normalized estimated betweenness centrality scores without guarantees.
- `EigenvectorCentrality`: Using parallel power iteration, eigenvector centrality scores are computed up to a certain error tolerance [96].
- `KatzCentrality`: This algorithm calculates approximate Katz centrality scores based on [43].

For the remaining parameters, the default values of `NetworKit` are chosen. All these algorithms originate from current research and belong to the state-of-the-art for their respective centrality measure. More detailed information about each approach is provided in Chapter 3. Most of these algorithms provide parallel implementations, however, for a fair comparison between the sparsification and centrality computations on unreduced graphs, the algorithms of `NetworKit` are executed sequentially.

5.2 Methodology

In order to evaluate the effectiveness of the approach, the experiments are conducted as follows. Firstly, the graph is reduced using the edge sparsification. Subsequently, centrality scores for each vertex are calculated using the previously described algorithms of `NetworKit`. The obtained results, i.e., centrality scores and computation times, are then compared to the ones on the full graph. Note, that computation times on the reduced graph include the time required for the edge sparsification.

Each experiment is repeated four times per graph instance, initialized with different random seeds. Across these repetitions, computation times and centrality scores are averaged using the geometric mean and arithmetic average, respectively. These results are further averaged over all graphs. Analogous to [34], let the runtime of a specific centrality algorithm on the full graph be denoted as σ_{full} and on the reduced graph as $\sigma_{reduced}$. The *time ratio* over computations on the full graph is defined as

$$\frac{\sigma_{full}}{\sigma_{reduced}}. \tag{5.1}$$

Similarly, the *speedup* or *time improvement* is then defined as

$$(\frac{\sigma_{full}}{\sigma_{reduced}} - 1) \cdot 100\%. \tag{5.2}$$

To assess the quality of approximate centrality scores on the reduced graphs, two evaluation measures are used: the Kendall τ_b correlation coefficient and the centrality similarity. The correlation coefficient τ_b measures the similarity between two rankings given the same set of vertices [41]. Since it is not affected by variations in normalization and distribution, it is particularly suitable for centrality computations. The coefficient τ_b varies between ± 1, indicating a perfect association for values close to ± 1 and no relation for values close to 0.

Centrality similarity was first introduced in [98] and can be defined as follows. Let $A = [a_{(1)}, a_{(2)}, \ldots, a_{(n)}]$ and $B = [b_{(1)}, b_{(2)}, \ldots, b_{(n)}]$, where $a_{(i)}, b_{(i)} \in V,\ i \in \{1, \ldots, n\}$, be two vertex rankings with respect to a given centrality measure. The centrality similarity $M_{A,B}(\delta)$ is the percentage of vertices in $[a_{(1)}, a_{(2)}, \ldots, a_{(\lfloor \delta n \rfloor)}]$ that also occur in $[b_{(1)}, b_{(2)}, \ldots, b_{(\lfloor \delta n \rfloor)}]$ where $\delta \in [0, 1]$. It thus measures the overlap of vertices between the rankings of the top $100\delta\%$ nodes. This can be extended to absolute values $\Delta \in \mathbb{N}, \Delta < n$ such that $M_{A,B}(\Delta)$ is then the overlap of rankings $[a_{(1)}, a_{(2)}, \ldots, a_{(\Delta)}]$ and $[b_{(1)}, b_{(2)}, \ldots, b_{(\Delta)}]$. In the experiments, centrality similarity is used to compare rankings on the full graph A_{full} to ones on the reduced graph $B_{reduced}$, i.e., $M_{f,r}(\cdot)$.

5.3 Graph Instances

For the experiments a diverse set of undirected graphs, originating from various classes and of varying sizes, is utilized in order to assess the edge sparsifications' performance. These graph instances have been obtained from the Stanford Network Analysis Platform (SNAP) [57] and Koblenz Network Collection (KONECT) dataset [55] as well as from the 10th DIMACS implementation challenge [6, 46]. For each graph, the following preprocessing steps are applied:

- Reduction to its largest connected component to ensure connectivity, such that closeness centrality can be calculated
- Removal of self-loops
- Assignment of edge IDs for computing edge counts

The information in Tables 5.1 and 5.2 refers to the preprocessed graphs. The set of graphs is further divided into two disjoint sets: one designated for the parameter tuning experiments and the other set is used for the comparison against computations without prior sparsification in Section 5.5.

Graph	n	m	Type
out.dbpedia-occupation	127,569	250,934	Affiliation
out.dbpedia-team	901,132	1,366,463	Affiliation
out.flickr-groupmemberships	395,978	8,537,702	Affiliation
oregon1_010526	11,174	23,408	Autonomous
out.higgs-twitter-social	456,289	12,508,221	Online Social
out.hyves	1,402,672	2,777,419	Online Social
out.livemocha	104,102	2,193,082	Online Social
UGA50	24,380	1,174,051	Online Social
Villanova62	7,754	314,980	Online Social
Rutgers89	24,568	784,596	Online Social
FSU53	27,731	1,034,798	Online Social
out.citeseer	365,154	1,721,980	Citation
out.patentcite	3,764,117	16,511,739	Citation
CA-AstroPh	17,902	196,972	Collaboration
CA-CondMat	21,363	91,285	Collaboration
CA-HepPh	11,203	117,619	Collaboration
out.cit-HepTh	27,399	352,020	Collaboration
Email-EuAll	224,832	339,925	Communication
WikiTalk	2,388,953	4,656,681	Communication
out.arenas-email	1,133	5,451	Communication
HR_edges	54,573	498,202	Social
Slashdot0902	82,167	504,230	Social
deezer_europe_edges	28,281	92,752	Social
large_twitch_edges	168,114	6,797,556	Social
soc-pokec-relationships	1,632,803	22,301,964	Social
twitter_combined	81,306	1,342,296	Social
out.dimacs9-COL	435,666	521,200	Infrastructure
out.dimacs9-FLA	1,070,375	1,343,951	Infrastructure
out.opsahl-powergrid	4,941	6,593	Infrastructure
roadNet-PA	1,087,562	1,541,513	Infrastructure
roadNet-TX	1,351,137	1,879,201	Infrastructure
out.flickrEdges	105,721	2,316,668	Miscellaneous
shipsec5.mtx	179,860	4,966,617	Miscellaneous
GaAsH6.mtx	61,349	1,660,230	Miscellaneous
web-BerkStan	654,782	6,581,870	Web
web-Google	855,801	4,291,351	Web
web-NotreDame	325,728	1,090,107	Web

Table 5.1: Graph instances for the experiments against unreduced graphs, obtained from the 10th DIMACS implementation challenge, as well as the SNAP and KONECT data set.

Graph	n	m	Type
out.dimacs10-as-22july06	22,963	48,435	Autonomous
out.dbpedia-recordlabel	168,261	233,259	Affiliation
CA-GrQc	4,158	13,421	Collaboration
CA-HepTh	8,637	24,806	Collaboration
Email-Enron	33,695	180,811	Communication
out.subelj_cora_cora	23,165	89,157	Citation
out.com-amazon	334,862	925,872	Miscellaneous
pdb1HYS.mtx	36,417	2,154,174	Miscellaneous
out.petster-friendships-dog-uniq	426,485	8,543,320	Online Social
out.petster-cat-household	68,315	494,561	Online Social
UConn91	17,205	604,866	Online Social
Smith60	2,970	97,133	Online Social
Michigan23	30,105	1,176,488	Online Social
out.dimacs9-NY	264,345	365,050	Infrastructure
musae_facebook_edges	22,469	170,822	Social
Slashdot0811	77,359	469,179	Social
com-dblp.ungraph	317,079	1,049,866	Social
facebook_combined	4,038	88,233	Social
soc-Epinions1	75,877	405,738	Social
rgg_n_2_17_s0	131,067	728,749	Random
web-Stanford	255,264	1,941,926	Web

Table 5.2: Graph instances for the tuning experiments, obtained from the 10th DIMACS implementation challenge, as well as the SNAP and KONECT data set.

Parameter	Value Range
set_size	$[1, n)$
bfs_method	$\{bfs\_set, bfs\_single, bfs\_percentage\}$
bfs_percentage	$(0, 1)$
bfs_repetitions	$\mathbb{Z}^+$
edge_percentage	$(0, 1)$
mwsf_method	$\{mwsf\_single, mwsf\_iterative\}$

Table 5.3: Ranges of the parameters included in the edge sparsification algorithm (see Algorithm 3).

5.4 Parameter Tuning Experiments

As shown in Table 5.3, the edge sparsification algorithm involves several parameters for which an optimal configuration must be determined. Thus, parameter tuning experiments have to be conducted first. These experiments aim to reveal the influence of parameters by exploring various combinations. The strategy is thereby the following: one parameter is altered at a time, while keeping the others constant. The value for which the best results are obtained is adopted for the corresponding parameter. To determine whether a result is good or better than another, various criteria can be employed. In the present thesis, two criteria are pursued simultaneously:

- Maximization of quality
- Minimum speedup of 10%

A threshold of 10% has been chosen to avoid over-optimization on the tuning set and to ensure that the approach maintains practical relevance by offering a substantial speedup. Regarding centrality measures, there are two primary goals, namely (i) identifying the most central nodes and (ii) obtaining a ranking of nodes based on a specific measure (see Chapter 3). As these objectives differ significantly from each other, separate parameter configurations are determined in the parameter tuning experiments, indicated by Sections 5.4.1 and 5.4.2. These are referred to as the ranking and top configuration, respectively.

The parameters of the edge sparsification include two categorical ones, namely the BFS and MWSF approach, represented by *bfs_method* and *mwsf_method*, respectively (see Table 5.3). Comparing these components provides important insights into their impact on the reduced graph, computation times, and the quality of centrality scores. Thus, the parameter tuning experiments of each objective are conducted for all possible combinations of these two parameters. Furthermore, the tuning experiments are limited to betweenness centrality, specifically the `EstimateBetweenness` algorithm. This decision is motivated by the following reasons:

- The optimization for multiple centrality measures may yield suboptimal configurations, which do not show the potential of the sparsification.
- Throughout the experiments, it has been shown that the edge sparsification achieves the best results for `EstimateBetweenness`. This will be further discussed in Section 5.5.
- In an experimental evaluation, computation time becomes a major concern. Especially distance based centrality measures are computationally more expensive, and thus focussing on a single measure reduces the required time for each experiment.

5.4.1 Ranking of Vertices

In the following, the results of the parameter tuning experiments for maximizing ranking quality are presented. This section is organized by the individual parameters, and corresponding results are explained and interpreted. The baseline configurations can be found in Table A.1 and are adjusted according to the optimal values identified for each parameter. Evaluation measures include time ratios for assessing the computational speed and the Kendall τ_b correlation coefficient for reviewing the ranking quality. It is important to note that not all BFS variants share the same parameters, resulting in variations across subsections.

Set Size

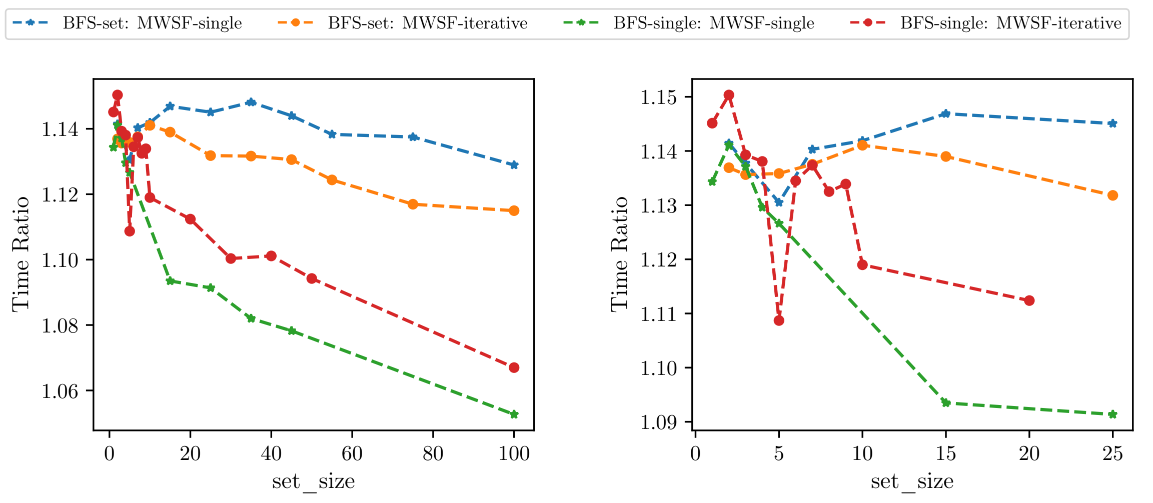
The first parameter of the tuning experiments for BFS-set and BFS-single is the set size. Both, time ratios and correlation coefficients, are displayed in joined plots in Figure 5.1. The results with respect to the time ratio are split into two plots in order to show long term trends (Figure 5.1a) as well as variations for smaller set sizes (Figure 5.1b). The set size in BFS-percentage is always equivalent to one and therefore does not have to be determined in the tuning experiments.

Regarding BFS-set, both MWSF approaches exhibit comparable speedups for single-digit set sizes (see Figure 5.1b). Only for set sizes greater than 10, the curves begin to diverge, resulting in MWSF-single being one average faster than MWSF-iterative. As set sizes increase, a downward trend is observed for both approaches. However, these variations are only marginal. For MWSF-iterative, the speedup decreases from set size two to set size 100 by just 2%, while for MWSF-single, it reduces by only one percent. Additionally, the difference between the two MWSF approaches remains marginal, ranging from one to two percent.

The rather constant computation times for varying set sizes are caused by the fact that the parameter does not affect the amount of BFS in the sparsification algorithm (see Section 4.2.1). In the BFS-set approach only one BFS is executed for the whole subset of vertices, regardless of its size. Therefore, for a fixed amount of *bfs_repetitions*, no drastic changes with respect to the speedup are determined.

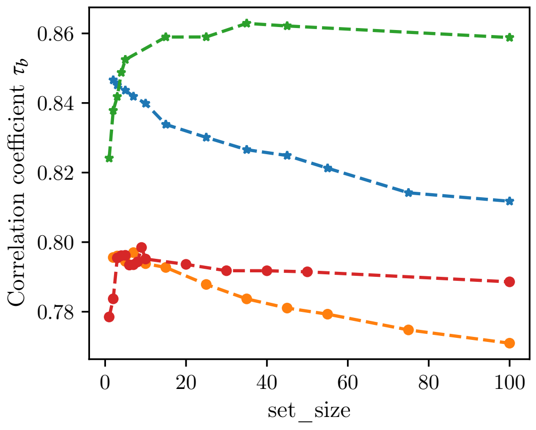
Concerning BFS-single, the results in Figure 5.1b show that MWSF-iterative achieves higher speedups than MWSF-single, apart from an outlier observed at 5 vertices. Both approaches are faster for smaller set sizes and reach a maximum at 2 nodes. In contrast to BFS-set, BFS-single executes a BFS for every vertex in the subset, and thus the subset size directly affects the number of BFS. This explains the strong decrease of speedup for increasing set sizes; for instance, MWSF-single drops below the 10% threshold with a set size of 15 vertices, MWSF-iterative at about 50 vertices. For comparison, BFS-set still maintains a speedup of around 12% with a set size of 100.

The quality of vertex rankings obtained on the reduced graphs is depicted in Figure 5.1c. It is observed that (i) MWSF-single achieves better results over all set sizes and (ii) the



(a) Time ratios of BFS-set and BFS-single for varying set sizes.

(b) Time ratios of BFS-set and BFS-single for set sizes up to 25 vertices.



(c) Correlation coefficients τ_b for BFS-set and BFS-single.

Figure 5.1: Results of the tuning experiments for the parameter *set_size*. Time ratios and coefficients τ_b are computed with respect to the full graphs. For BFS-single set sizes 5 and 3, and for BFS-set 2 are chosen for MWSF-single and MWSF-iterative, respectively.

shapes of the curves within a BFS method are similar and differ only by a qualitative gap. For BFS-set, the correlation coefficients are highest for smaller set sizes and then decrease continuously. The difference between the two correlation coefficients of the MWSF approaches is around 5%. While MWSF-single reaches its maximum at 2 vertices, indicating strong correlation with $\tau_b = 0.84$, MWSF-iterative's correlation coefficient is equivalent to 0.79 for 5 vertices. In general, both approaches exhibit moderate to strong correlations with a clear tendency towards smaller set sizes.

This can be explained by the fact that the BFS-set approach does not consider edges that run between vertices of the subset (see Section 4.2.1). The larger the number of vertices in the subset, the greater the number of edges that are not considered in a BFS. Since their edge count is thus smaller, these edges are less likely to be part of the reduced graph even though they are potentially important.

For BFS-single, a strong increase of the correlation coefficients can be observed for both MWSF approaches. However, this only applies to small set sizes: MWSF-iterative reaches a plateau with $\tau_b = 0.79$ already at 3 vertices and more vertices do not yield significant changes. For MWSF-single the initial increase is significantly larger than for MWSF-iterative and even after the first peek at 5 nodes with $\tau_b = 0.85$, the correlation coefficient still increases. However, these variations become marginal and remain at $\tau_b \approx 0.86$.

These results are counterintuitive in the sense that one might expect larger subsets, and consequently more BFS calculations, to result in an improvement in quality since more information about shortest paths within a graph is obtained. However, these observations could be explained with the following argument. While more BFS executions lead to a few edges with high edge counts, they also result in numerous edges with the same counts, making it difficult to distinguish between them. Hence, edges, especially those with intermediate ranks, are more frequently selected at random. This particularly affects the iterative computation of MWSFs.

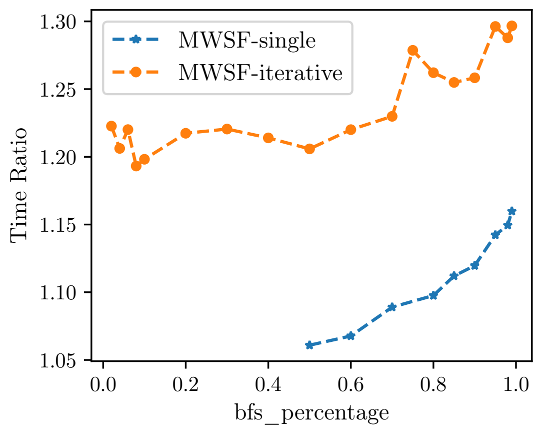
For both, BFS-set and BFS-single qualitative peaks are reached for small subset sizes, while achieving a speedup of over 10%. Thus, the parameter *set_size* is set to 2 for both MWSF approaches related to BFS-set. Concerning BFS-single, set sizes 3 and 5 are chosen for MWSF-iterative and MWSF-single.

BFS Percentage

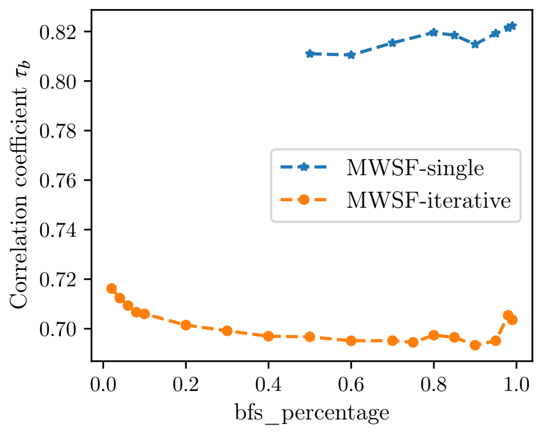
The results of the tuning experiments for parameter *bfs_percentage* are shown in Figure 5.2. This only applies to BFS-percentage since it is the only BFS variant, where the subset of vertices is not pre-selected but depends on the amount of BFS needed to reach all vertices. In general, it holds that for smaller percentages, fewer vertices are reached within a BFS and therefore more runs are needed overall. This leads to increasing computation times.

For MWSF-single only high percentages achieve a comparable runtime (see Figure 5.2a). Starting from the maximum at 99%, the speedup decreases rapidly and at 85%, it is already less than 10%. For MWSF-iterative the results look quite different. Although the speedup also decreases along with the percentages, it remains on average above 20%. Even for single-digit values, a significant speedup is achieved.

As shown in Table A.3, the number of required BFS steadily increases as the percentages decrease, reaching a maximum of on average $2{,}473$ runs for 2%. For comparison, with BFS-single, the speedup for MWSF-iterative already dropped below 10% for 50 nodes. This can be explained by examining the computation times for centralities on the reduced graph, excluding the runtime of the sparsification algorithm (see Figure A.1). MWSF-iterative exhibits a speedup of over 30%. This is significantly higher than the speedup



(a) Time ratios of BFS-percentage.



(b) Correlation coefficients τ_b for BFS-percentage.

Figure 5.2: Results of the tuning experiments for the parameter *bfs_percentage*. Time ratios and correlation coefficients are computed with respect to the full graphs. The percentages 0.99 and 0.02 are chosen MWSF-single and MWSF-iterative, respectively.

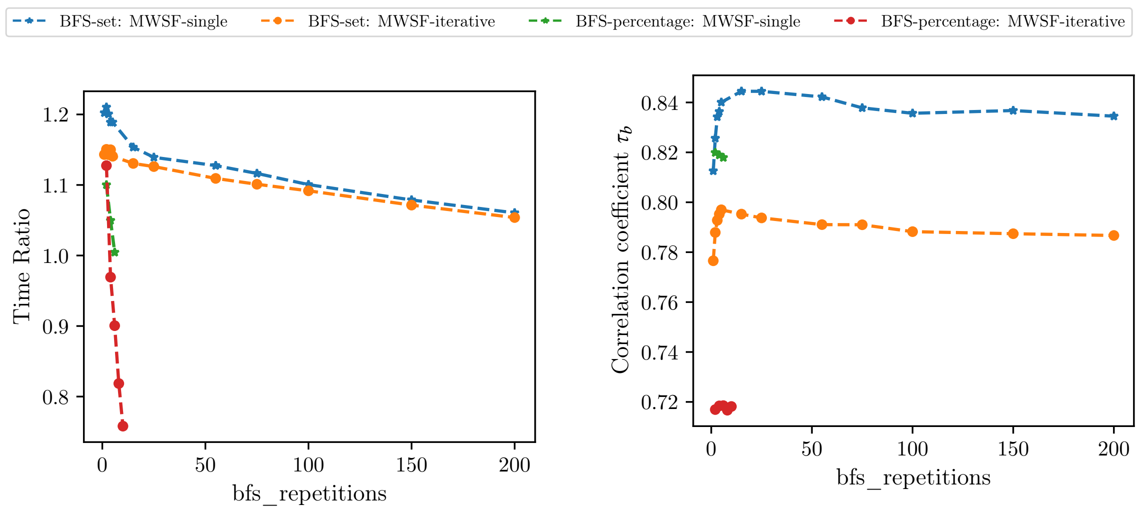
of MWSF-single and the speedup of MWSF-iterative combined with BFS-single. This compensates for the increased computational effort required for more BFS. The increased speedup of centrality computations for MWSF-iterative results from the amount of edges in the graphs. Due to the baseline configuration with *edge_percentage*$= 0.30$, the reduced graphs contain on average 38% of the edges, significantly fewer than for MWSF-single with 52%. Since these graphs are smaller, the centrality algorithms run faster.

Interestingly, the increasing amount of BFS using MWSF-iterative does not lead to significant variations in quality. For both, small and large percentages, peaks are observed, reaching nearly 72%, but for moderate values, only minor fluctuations are evident. Most importantly, the quality of MWSF-iterative lags behind that of the MWSF-single approach by approximately 10%. The latter achieves the best results at 99%, decreasing along with the parameter *bfs_percentage*. Given that the highest correlation coefficients are observed for 2% and 99% with sufficient speedup, the parameter *bfs_percentage* is adjusted accordingly for MWSF-iterative and MWSF-single, respectively.

BFS Repetitions

For BFS-single, the repeated selection of a random subset with subsequent edge count calculation is equivalent to selecting one subset of larger size. Therefore, this parameter does not have to be considered for this approach. It is implicitly part of the previous experiments for the set size. The results for BFS-set and BFS-percentage are depicted in Figure 5.3.

Since the parameter *bfs_repetitions* determines the amount of BFS runs, it has a higher influence on the speedup of BFS-set than the parameter *set_size*. As expected, the highest



(a) Time ratios of BFS-set and BFS-percentage.

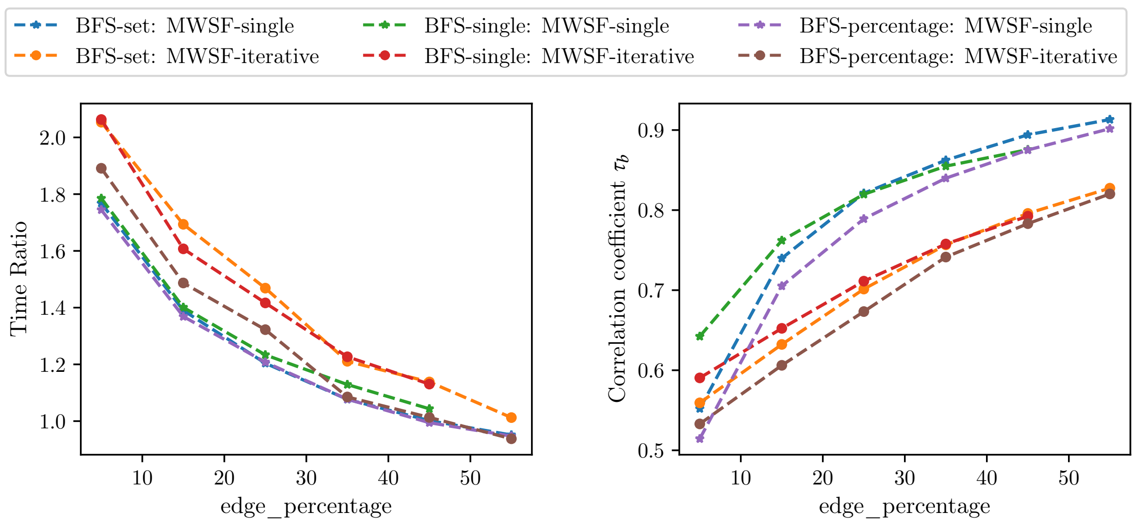
(b) Correlation coefficients τ_b for BFS-set and BFS-percentage.

Figure 5.3: Results of the tuning experiments for the parameter *bfs_repetitions*. Time ratios and correlation coefficients are computed with respect to the full graphs. For BFS-set repetition counts 25 and 5, and for BFS-percentage repetition count 1, are chosen for MWSF-single and MWSF-iterative, respectively.

speedup is achieved for few repetitions, with MWSF-single surpassing MWSF-iterative by about 6% (see Figure 5.3a). While MWSF-iterative consistently decreases with increasing repetitions, MWSF-single exhibits a steep drop of 6% in the first 25 repetitions. As both MWSF curves are decreasing, they start to converge, until there is less than one percent difference between them.

With increasing number of repetitions, the correlation coefficients of both MWSF approaches increase, as depicted in Figure 5.3b. One could assume that these continue to rise, as the repeated calculation of SSSP should increase the quality and accuracy of the edge counts and thus, the overall quality of the ranking. However, the results in Figure 5.2b show that after a strong increase for relatively few repetitions, the correlation coefficients stay rather constant and even start to decrease. This shows that for both MWSF approaches quality does not automatically improve with more repetitions. On the contrary, a few are sufficient to achieve adequate improvements; too many reduce the speedup and sometimes even the quality. A similar behavior was already noted during the *set_size* experiments for BFS-single in Section 5.4.1. No significant improvement in quality was determined for more vertices in the subset, i.e., for more BFS runs. Consequently, it is also assumed this behavior is caused by many edge counts sharing the same value.

Since the BFS-percentage approach executes significantly more BFS in a single repetition compared to BFS-set, fewer parameter values are tested (see Figure 5.3a). Even for these limited repetitions, the speedup drops significantly below 10% or fully disappears,



(a) Time ratios of BFS-set, BFS-single and BFS-percentage.

(b) Correlation coefficients τ_b for BFS-set, BFS-single and BFS-percentage.

Figure 5.4: Results of the tuning experiments for the parameter *edge_percentage*. Time ratios and correlation coefficients are computed with respect to the full graphs. The following values are chosen for MWSF-single and MWSF-iterative: 0.25 and 0.45 for BFS-set, 0.35 and 0.45 for BFS-single and 0.25 for BFS-percentage.

making the approach slower than computations on the full graph. Also in terms of quality, multiple repetitions do not yield significant improvements and correlation coefficients stay constant (see Figure 5.3b). Although more repetitions would have to be tested in order to derive a definitive conclusion, the experiments for *bfs_repetitions* and *set_size* for BFS-set and BFS-single suggest that the correlation coefficients would vary only marginally with more repetitions. Consequently, one repetition is used for both MWSF approaches with respect to BFS-percentage. For BFS-set, 5 and 25 repetitions are used for MWSF-iterative and MWSF-single in the subsequent experiments.

Percentage of Edges

The last parameter to be determined in the tuning experiments is the edge percentage, which is included in all BFS variants. The time ratios and correlation coefficients of all approaches are depicted in Figure 5.4. As expected, all combinations of BFS- and MWSF-method achieve their highest speedups with very low percentages. The reduced graph then contains fewer edges, increasing the speed of the centrality computations. With increasing percentages, all speedups continuously decrease until at around 55%, the computations on reduced and full graphs are equally fast. The reason for this is not the sparsification, but the centrality computation algorithms, which require nearly as much time for an edge threshold of 55% as they do on the full graphs. Using BFS-single as an example, this is stated in Tables A.4 and A.5. This indicates that computations are not necessarily faster if the graph contains fewer edges. There is a certain quantity of edges beyond which the calculations take as long as on the full graph.

The plot in Figure 5.4a also shows, that for varying percentages, MWSF-iterative achieves generally better results with respect to the speedup than MWSF-single. For smaller percentages, the speedup of MWSF-iterative is on average 0.2 bigger. However, with increasing edges in the graph the difference decreases, and both approaches perform similarly (see Figure 5.4b). This can be explained by looking at the time improvements of just the centrality computations and the average amount of edges in the graphs. Centrality computations are faster on the reduced graphs obtained from MWSF-iterative than MWSF-single, most likely because these graphs contain fewer edges (see Tables A.4 and A.5). When comparing the speedups of the two MWSF approaches that generate a similar number of edges in the reduced graph, the difference between them is already significantly smaller.

For all MWSF-single approaches a fast increase of τ_b can be observed from 5% to 15% (see Figure 5.4b). The correlation coefficients continue to further increase, however, with continuously less steep slopes. Beyond an edge percentage of 0.45, the rankings on the reduced and unreduced graphs exhibit a strong correlation, indicated by values of 0.85 or higher for τ_b. The results for MWSF-iterative display a similar trend: as the number of edges in the graph increases, the quality improves. However, this improvement is less strong, reaching a maximum correlation coefficient of approximately 0.8. As discussed for the time improvement, the reason for the qualitatively higher results of MWSF-single is most likely the fact, that the reduced graphs contain more edges than the ones constructed with MWSF-iterative. Especially for lower percentages, the comparison in Tables A.4 and A.5 demonstrates that both approaches perform equally well. However, for higher values of *edge_percentage*, MWSF-single clearly outperforms MWSF-iterative.

In general, the correlation coefficients shown in Figure 5.4b imply that with more edges, the reduced graphs exhibit better ranking quality. This is caused by the fact that the reduced graphs converge closer to the full graphs. Besides the slopes of all curves decrease with increasing values for *edge_percentage*. This suggests that, for a perfect correlation, the majority of edges must be present in the reduced graph.

Looking solely at the correlation coefficients, the best choice would be to use higher percentages. However, the speedup strongly decreases with increasing amount of edges and thus the best possible choices for MWSF-single and MWSF-iteartive, which align with the strategy presented in Section 5.2, are the following: 0.25 and 0.45 for BFS-set, 0.35 and 0.45 for BFS-single and 0.25 for BFS-percentage.

Summary

In the following, the insights gained throughout the previous experiments are shortly summarized, and the resulting parameter configurations are listed in Table 5.4. For BFS-set and BFS-single, smaller set sizes result in the highest correlation coefficients. Only a

BFS Variant	**MWSF**	**Set Size**	**BFS Percentage**	**Repetitions**	**Edge** [%]	τ_b	**Time Improvement** [%]
Set	Single	2	–	25	25	0.82	20.4
	Iterative	2	–	5	45	0.8	13.8
Single	Single	5	–	–	35	**0.86**	12.89
	Iterative	3	–	–	45	0.79	13.05
Percentage	Single	1	0.99	1	25	0.79	20.78
	Iterative	1	0.2	1	25	0.67	**32.23**

Table 5.4: Resulting parameter configurations of the ranking tuning experiments.

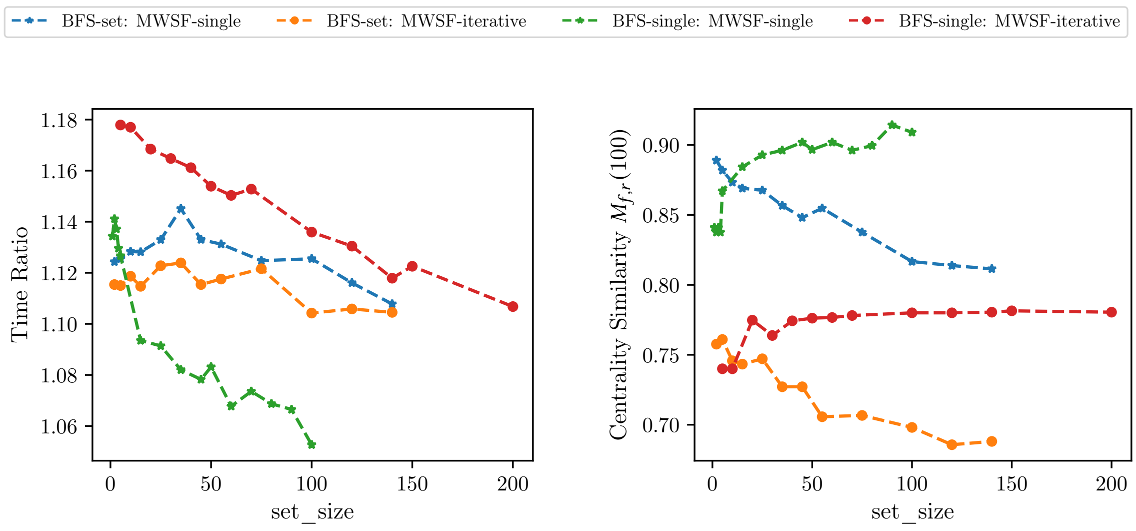
certain amount of repetitions is beneficial for BFS-set, as too many do not lead to qualitative improvements. Concerning BFS-percentage, it has been shown that variations of the parameter *bfs_percentage* are only possible if the reduced graphs contain fewer edges. However, these perform qualitatively worse. Multiple repetitions do not yield improvements. Regarding the parameter *edge_percentage*, it is observed, that across all variants the quality increases with more edges in the graph at the cost of a decreasing speedup. Generally, MWSF-single achieves qualitatively better results, while MWSF-iterative yields the highest speedup. With respect to the correlation coefficient τ_b, the combination of BFS- and MWSF-single yields the best results. BFS-percentage achieves the highest speedups (see Table 5.4). To assess the effectiveness of the sparsification, the configuration with the best quality, i.e., the one involving BFS-single, is utilized for the experiments in Section 5.5.

5.4.2 Identification of Most Central Vertices

In this section, the outcomes of the parameter tuning experiments for identifying the most central vertices are presented. Similar to the previous section, the results are described, interpreted, and organized by each involved parameter. The baseline configurations are listed in Table A.2. Evaluation measures include time ratios as well as the centrality similarity with $\Delta = 100$.

Set Size

The results for the tuning experiments of the parameter *set_size* are shown in Figure 5.5. Note, that since some approaches required fewer data points than others to capture general trends effectively, the number of data points in the plots differs between the curves. Concerning BFS-single, a consistent downward trend is observable for both MWSF approaches. Starting with a speedup of approximately 18%, MWSF-iterative steadily diminishes, loosing around 2% of speedup every 40 vertices. However, even for 200 vertices, a speedup over 10% is still achieved. The results for MWSF-single, however, exhibit a different behavior. After an initial speedup of approximately 14% for a set size of up to 5



(a) Time ratios for BFS-set and BFS-single.

(b) Centrality Similarity $M_{f,r}(100)$ for BFS-set and BFS-single in percent.

Figure 5.5: Results of the tuning experiments for the parameter *set_size*. Time ratios and centrality similarity values are computed with respect to the full graphs. For BFS-single set sizes 10 and 150, and for BFS-set 2 and 5 are chosen for MWSF-single and MWSF-iterative, respectively.

vertices, the speedup sharply declines, falling below 10% for 15 vertices already. As the set size increases, the speedup continues to decrease.

In terms of quality, MWSF-single distinctly surpasses MWSF-iterative. An offset of approximately 10% is evident between the two approaches, even for small single-digit values. As the number of vertices increases, the centrality similarity grows for both methods. However, for MWSF-iterative, the maximum of 77% is reached at around 20 vertices and remains relatively constant up to a set size of 150. This behavior is similar to the one observed during the ranking tuning experiments (see Section 5.4.1). In contrast to MWSF-iterative, the centrality similarity of MWSF-single continues to increase. Although the rate of increase diminishes, the shape of the curve still shows an upward trend. In contrast to the findings of the ranking tuning experiments, these results suggest that an increasing number of vertices in the subset improves the quality.

For a set size of 2 vertices, both approaches of BFS-set achieve similar speedups of around 12.5% and 11.5%. The curves initially begin to rise, with MWSF-single showing a stronger ascent than MWSF-iterative, and reach a peak at a subset size of 35 vertices. The speedups then consistently decline as the number of vertices increases, falling below the 10% threshold at 100 and 140 vertices. Similar to the previous experiments in Section 5.4.1, differences between both MWSF approaches and the variations of the time improvements are rather small and can be attributed to the fact that the number of BFS does not change.

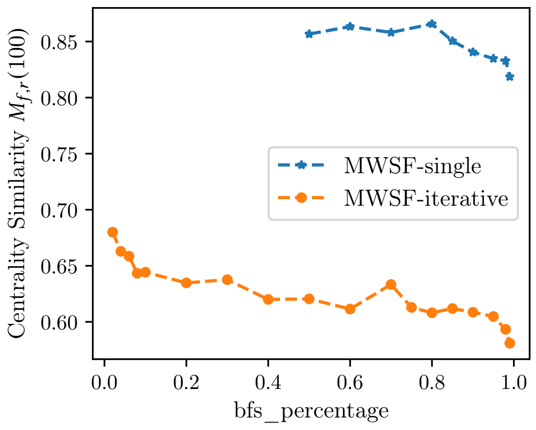


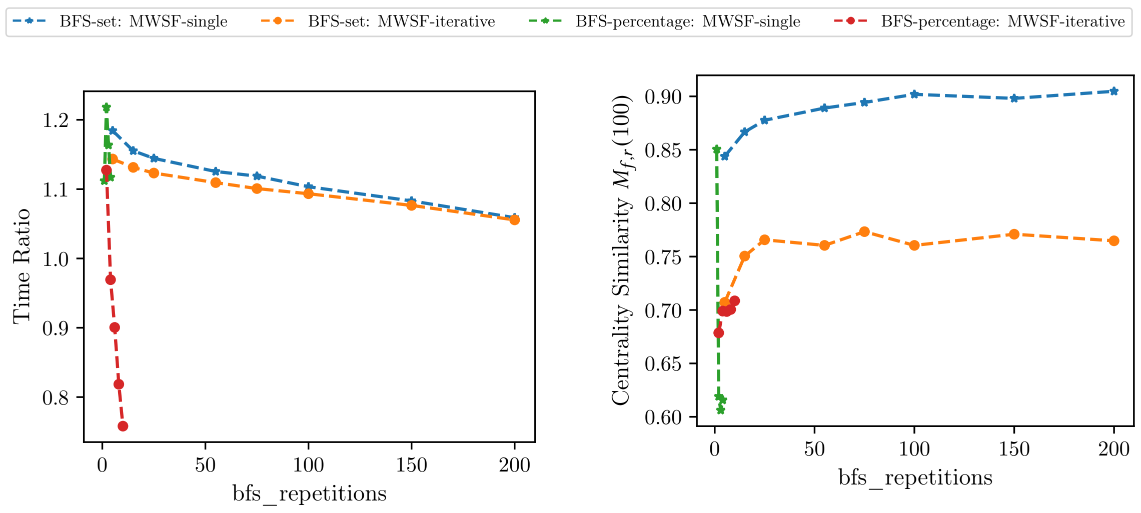
Figure 5.6: Results of the tuning experiments for the parameter *bfs_percentage*. The time ratios are equivalent to the ones shown in Figure 5.3a. The centrality similarities are computed with respect to the full graphs. The percentages 0.85 and 0.02 are chosen MWSF-single and MWSF-iterative, respectively.

In terms of centrality similarity, the two MWSF methods differ by approximately 12%, which persists throughout varying set sizes. Similar to the experiments on ranking optimization, smaller subsets yield qualitatively better results and maxima are reached at 2 and 5 vertices with $M_{f,r}(100) = 0.88$ and $M_{f,r}(100) = 0.76$. This is caused by the fact that for subsets with an increased number of vertices, the loss of information from the edges unconsidered between these vertices becomes too substantial. Consequently, the parameter *set_size* is set to 2 and 5 for MWSF-single and MWSF-iterative. With respect to BFS-single, set sizes 10 and 150 are used for the following experiments.

BFS Percentage

Since the same baseline configuration is used as for the ranking tuning experiments, the results in terms of time ratios are identical to the ones described in Figure 5.2a and summarized briefly: With decreasing values for *bfs_percentage*, more BFS runs have to be performed, resulting in smaller speedups. While MWSF-iterative still attains a substantial speedup for single-digit percentages, MWSF-single drops below 10% for 85% already.

In contrast to the outcomes of the ranking experiments, the results show that as percentages decrease, the quality improves, indicating the identification of a growing number of most central nodes. For MWSF-single, a significant increase is observed between *bfs_percentage* values of 0.99 and 0.8. Beyond this point, the centrality similarity only marginally deviates, despite an increase of conducted BFS. For higher percentages, this pattern is also observed for MWSF-iterative. However, with a more extensive range of percentage values tested, it becomes apparent that the centrality similarity starts to gradually rise again from 50%. For single-digit values it further increases, reaching a maximum of



(a) Time ratios for BFS-set and BFS-percentage.

(b) Centrality Similarity $M_{f,r}(100)$ for BFS-set and BFS-single in percent.

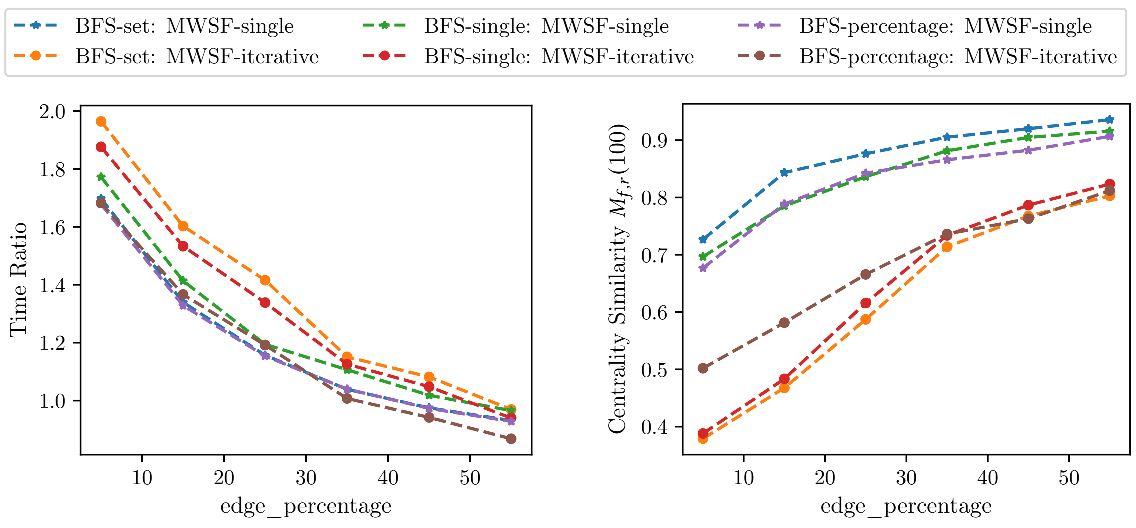
Figure 5.7: Results of the tuning experiments for the parameter *bfs_repetitions*. Time ratios and centrality similarities are computed with respect to the full graphs. For BFS-set repetition counts 100 and 75 and for BFS-percentage repetition counts 1 and 2 are chosen for MWSF-single and MWSF-iterative, respectively.

68% at 0.02. Nevertheless, the qualitative performance of this approach is lower than the one of MWSF-single, which attains up to 85% centrality similarity.

Since maximum quality and a speedup of at least 10% is guaranteed at *bfs_percentage* values 0.02 and 0.85, these values are adopted for MWSF-iterative and MWSF-single in the following experiments.

BFS Repetitions

The impact of varying the number of BFS repetitions on both BFS-set and BFS-percentage is illustrated in Figure 5.7. Concerning BFS-set, the results closely resemble those obtained during the previous tuning experiments. For few repetitions, the difference between both MWSF approaches is largest, while at the same time both achieve their highest speedups. However, as the repetition count increases, this improvement diminishes and the curves converge. The results for BFS-percentage also exhibit similarities to those of the ranking tuning experiments. For single-digit values, the speedup of MWSF-iterative dramatically decreases, resulting in no improvement at all. Regarding MWSF-single, however, a peak with over 20% speedup is achieved with just two repetitions. Interestingly, this time ratio peak does not coincide with a quality peak. Instead, the centrality similarity decreases from 85% to 62%. For MWSF-iterative, multiple iterations only marginally improve the observed quality.



(a) Time ratios for BFS-set, BFS-single and BFS-percentage.

(b) Centrality similarity $M_{f,r}(100)$ for BFS-set, BFS-single and BFS-percentage.

Figure 5.8: Results of the tuning experiments for the parameter *edge_percentage*. Time ratios and centrality similarities are computed with respect to the full graphs. The following edge percentages are chosen for MWSF-single and MWSF-iterative for BFS-set, BFS-single and BFS-percentage: 0.25 and 0.35, 0.35 and 0.25.

Similar to the quality results of Figure 5.2b, a steep increase of centrality similarities is observed for both BFS-set approaches with only few repetitions. In contrast to the ranking experiments, no downward trend for further repetitions can be observed. Particularly for MWSF-single, the centrality similarities do not stagnate but continue to exhibit a slight upward trend, implying that multiple repetitions improve the ability to identify the most central vertices.

Concerning BFS-percentage, multiple repetitions only achieve a marginal increase in quality for MWSF-iterative. Thus, repetition counts 1 and 2 are used for MWSF-single and MWSF-iterative in the subsequent experiments. For BFS-set, an increased count of repetitions has positive effects and thus 100 repetitions are chosen for MWSF-single and 75 for MWSF-iterative.

Edge Percentage

In terms of time ratios, varying the percentage of edges in the reduced graphs produces results similar to those observed in the ranking experiments. The greater the number of edges included in the graph, the smaller the speedup (see Figures 5.4b and 5.8a). For BFS-set and BFS-single, MWSF-iterative consistently outperforms MWSF-single. Only for BFS-percentage, MWSF-iterative eventually falls below the speedup of MWSF-single. This can be attributed to the two BFS repetitions.

BFS Variant	**MWSF**	**Set Size**	**BFS Percentage**	**Repetitions**	**Edge** [%]	$M_{f,r}(100)$	**Time Improvement** [%]
Set	Single	2	–	100	25	$\mathbf{0.88}$	15.67
	Iterative	5	–	75	35	0.71	15.11
Single	Single	10	–	–	35	$\mathbf{0.88}$	10.66
	Iterative	150	–	–	35	0.73	12.58
Percentage	Single	–	0.85	1	25	0.84	15.4
	Iterative	–	0.02	2	25	0.67	$\mathbf{19.19}$

Table 5.5: Resulting parameter configurations of the tuning experiments regarding the most central vertices.

Concerning quality, it is again observed that an increase in the number of edges in the graph yields improvements and the curves of the MWSF approaches exhibit notable similarity (see Figure 5.8b). In contrast to the previous *edge_percentage* experiments, all MWSF-single approaches already reach a centrality similarity $M_{f,r}(100)$, of 70% at 5% edges. The quality then increases by another 10% up to *edge_percentage*$=$ 0.15, after which the rate of increase levels off. At 55%, all three BFS variants achieve an overlap of 86-90% among the top 100 centrality vertices.

For BFS-percentage, the progression of MWSF-iterative closely mirrors that observed in the preceding *edge_percentage* experiments (see Figure 5.4b). However, for BFS-set and BFS-single, some differences are evident: Initially, both approaches display a centrality similarity of 40% which is significantly lower than the initial quality results in Figure 5.4b. For increasing edge percentages, the curves start to increase, particularly strong between 20 and 30%, until at 55%, a centrality similarity of approximately 75% is reached.

Despite different initial quality, all MWSF approaches reach similar qualitative results as in the ranking tuning experiments. Since variations of *edge_percentage* induce significant changes with respect to the time ratios and quality, it highlights the influential role of this parameter in the sparsification. Ensuring a minimum speedup of 10%, the following edge percentage values are chosen for MWSF-single and MWSF-iterative: 0.25 and 0.35 for BFS-set, 0.35 for BFS-single and 0.25 for BFS-percentage.

Summary

The findings of the tuning experiments, focused on the most central vertices, are summarized in the following. Concerning the set size, few vertices yield the best qualitative results for BFS-set, whereas an increased amount is beneficial for BFS-single. However, this significantly reduces the speedup. Smaller values for *bfs_percentage* improve the centrality similarity and multiple repetitions quickly become infeasible due to the lack of speedup. For BFS-set, an increased repetition count yields qualitative better results. With more edges in the reduced graph, the most central vertices are identified better, at the cost of increasing computation time. As for the previous experiments, the results shown in Table 5.5 imply

Centrality Algorithm	Time Ratio	τ_b	$M_{f,r}(100)$
KadabraBetweenness	0.949	0.67	0.84
EstimateBetweenness	**1.042**	0.87	**0.86**
DegreeCentrality	0.002	0.90	0.80
ApproxCloseness	1.018	**0.93**	0.71
EigenvectorCentrality	0.455	0.88	0.62
KatzCentrality	0.187	0.89	0.80
CoreDecomposition	0.095	0.89	0.39

Table 5.6: Time ratios, correlation coefficients τ_b and the centrality similarity for the top 100 vertices obtained by using the configuration optimized for the vertex ranking.

that MWSF-single yields results with the highest quality, while MWSF-iterative achieves higher speedups. With a centrality similarity of 0.88, BFS-set combined with MWSF-single produces the best results with the highest speedup and is therefore applied in the experiments of Section 5.5 (see Table 5.5).

5.5 Comparison Against Unreduced Graphs

To evaluate the performance of the edge sparsification and explore the characteristics of the resulting graphs, experiments are conducted on the second group of graph instances (see Table 5.1). For each of the primary goals, i.e., the identification of most central vertices and the ranking of vertices, the best parameter configurations of the tuning experiments are utilized (see Tables 5.4 and 5.5). The scores of all centrality measures discussed in Chapter 2 are calculated using the algorithms described in Section 5.2.

Using the ranking configuration, time improvements are achieved for two algorithms, namely `ApproxCloseness` and `EstimateBetweenness` (see Table 5.6). With averaged speedups of 1% and 4%, the edge sparsification slightly outperforms computations on the full graph while achieving coefficients τ_b of 0.87 and 0.83, indicating a strong correlation with respect to the ranking. For `DegreeCentrality`, `CoreDecomposition`, `KatzCentrality` and `EigenvectorCentrality`, no speedup is obtained. On the contrary, the sparsification requires at least 50% more time compared to unreduced graphs. The reason for that is the fact that these centrality algorithms are already highly efficient on full graphs. While the calculations on the reduced graphs show a slight improvement in speed, this marginal gain is not enough to offset the additional time required. The edge sparsification is therefore not suitable for these centrality algorithms.

Even though `KadabraBetweenness` is known as the fastest betweenness approximation algorithm, a slowdown and even the worst correlation coefficient is obtained for both parameter configurations (see Tables 5.6 and 5.7). The results do not provide a clear explanation for these outcomes. In early experiments, it was observed that `KadabraBetweenness` exhibits highly fluctuating performances on graphs of the same

Centrality Algorithm	Time Ratio	τ_b	$M_{f,r}(100)$
KadabraBetweenness	0.594	0.64	0.84
EstimateBetweenness	**1.055**	0.83	**0.86**
DegreeCentrality	0.0002	0.76	0.73
ApproxCloseness	0.925	**0.87**	0.52
EigenvectorCentrality	0.105	0.74	0.33
KatzCentrality	0.027	0.76	0.72
CoreDecomposition	0.012	0.70	0.11

Table 5.7: Time ratios, correlation coefficients τ_b and the centrality similarity for the top 100 vertices obtained by the configuration focused on the most central vertices.

size. A possible reason for this could be the diameter. However, an analysis on this aspect is still open.

Similar findings are obtained for the top configuration in terms of time ratios. `EstimateBetweenness` achieves the overall best results with an average speedup of 5% and a centrality similarity of 86%. In comparison to the results in Table 5.6, the `KadabraBetweenness` algorithm is even slower, and no speedup is achieved for `ApproxCloseness`. This holds true for other centrality algorithms as well, which take nearly twice as long as on the full graphs (reasoning explained above).

In terms of quality, the ranking configuration produces more accurate rankings compared to the top configuration (see Tables 5.6 and 5.7). `ApproxCloseness` yields the highest correlation coefficient of $\tau_b = 0.93$ but also for the other centrality measures, promising results are obtained. Except for `KadabraBetweenness` with $\tau_b = 0.66$, the configuration consistently achieves coefficients exceeding 0.8, indicating a strong correlation between rankings in full and sparsified graphs. Interestingly, the ranking configuration also yields impressive results for identifying the most central nodes. Both configurations perform equally well for the two betweenness algorithms. However, for other centralities, the ranking configuration clearly surpasses the top configuration with improvements ranging from 7% to 28%.

To analyze the properties of the resulting graphs and the differences between the two configurations, we investigate the number of edges in the reduced graphs. Figure 5.9 shows that the ranking configuration produces graphs with more edges than the top configuration. This is a direct result of the edge percentage parameter, which is set to 0.35 for the ranking and 0.25 for the top configuration. At the same time, it is observed that many reduced graphs contain slightly more edges than specified. This is because in the MWSF-single approach, the edges are added in addition to the tree edges.

The majority of graphs contains less than 60% of edges; however, there are a couple of graphs that include significantly more, or nearly all edges. These graphs mainly correspond to infrastructure graphs, or graphs for which on average $\frac{m}{(n-1)} \approx 1.5$, i.e., sparse graphs. Thus, when applying the MWSF-single approach, which involves inserting a MWST and an additional 25% or 35% of edges, the reduced graph automatically includes all or close to

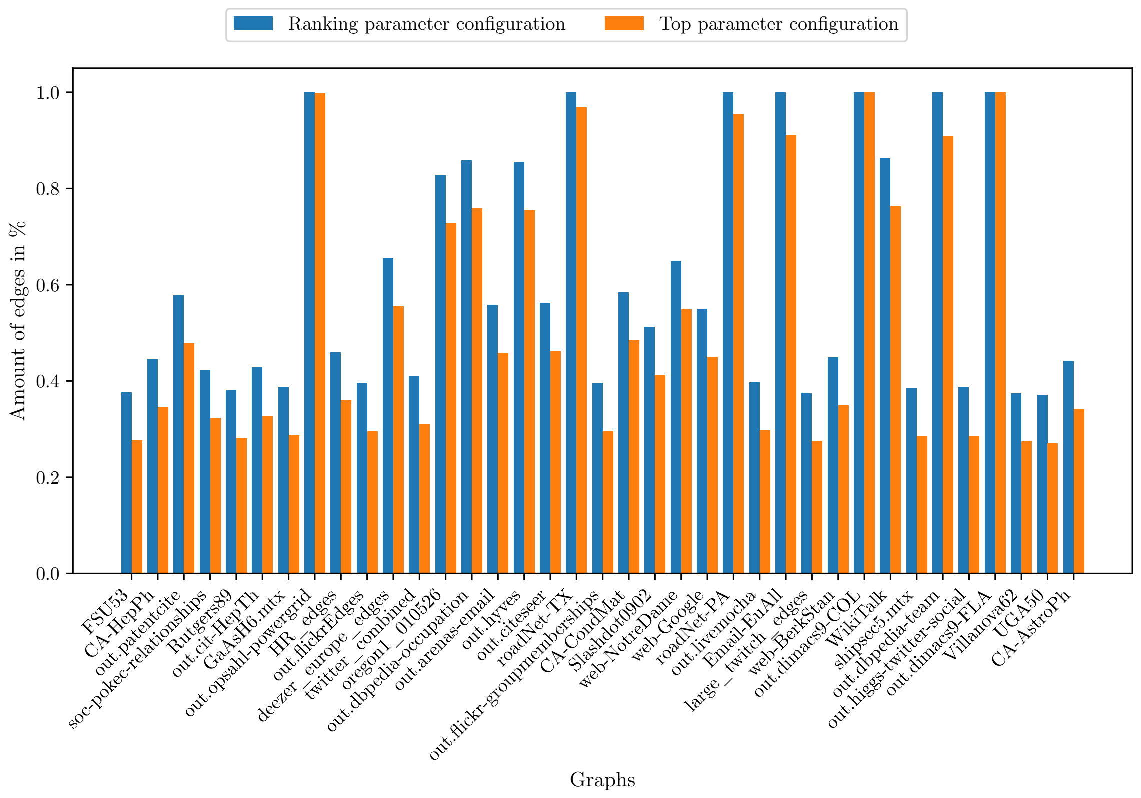


Figure 5.9: Comparison of the amount of edges in reduced graphs obtained by the ranking and top parameter configuration. The percentage is calculated with respect to the full graph.

all edges. Therefore, further sparsification is not useful for graphs that are already sparse.

Consequently, the results on the other graphs are now investigated, excluding sparse graphs. In comparison to the previous findings, the speedup for all centrality algorithms increases for the ranking configuration (see Table 5.8). As expected, `DegreeCentrality`, `EigenvectorCentrality`, `KatzCentrality` and `CoreDecomposition` still lag significantly behind the computation times on full graphs. For `EstimateBetweenness` and `ApproxCloseness`, however, the speedup is notably higher with 20% and 13%, respectively. The corresponding correlation coefficients are only marginally smaller, leading to still consistently qualitative good results. With the exception of `EigenvectorCentrality`, `KatzCentrality` and `CoreDecomposition`, τ_b and $M_{f,r}(100)$ are in general smaller than those in Table 5.6. However, it is expected that tuning the parameters exclusively on denser graphs improves these results.

An increased speedup of 13% and 23% is also observed for `ApproxCloseness` and `EstimateBetweenness` using the top configuration (see Table 5.9). The sparsification, however, does not yield improvements for the other centrality measures; in fact, the speedup for `KadabraBetweenness` decreases significantly. In terms of quality, the

Centrality Measure	Time Ratio	τ_b	$M_{f,r}(100)$
KadabraBetweenness	1.0	0.63	0.80
EstimateBetweenness	$\mathbf{1.253}$	0.84	$\mathbf{0.83}$
Degree	0.001	0.87	0.74
Closeness	1.148	$\mathbf{0.91}$	0.73
Eigenvector	0.493	0.87	0.55
Katz	0.193	0.86	0.73
k-core	0.096	0.85	0.24

Table 5.8: Time ratios, correlation coefficients τ_b and the centrality similarity for the top 100 vertices of denser graphs for the ranking configuration.

correlation coefficients are on average higher, while the centrality similarity experiences a slight decline.

On average, the ranking configuration produces qualitatively higher results in terms of both correlation coefficients and centrality similarity, even though only 5 BFS – significantly fewer than the 100 executed for the top configuration – are performed. Since both configurations have a similar subset size, this is primarily attributed to the two parameters *bfs_repetitions* and *edge_percentage*. Despite executing only 5 BFS, the higher amount of edges in the graphs seems to be sufficient to achieve qualitative good and also better results than with 100 BFS. This emphasizes the insights gained from the tuning experiments that number of edges highly influences the quality of the approach.

The best results are achieved for `EstimateBetweenness` as well as `ApproxCloseness`: For `EstimateBetweenness`, an approximate vertex ranking with a correlation coefficient of 0.84 is computed 25% faster, and the top 100 vertices are calculated with 28% speedup and an accuracy of 83%. With a speedup of 14% each, an approximate closeness centrality ranking with $\tau_b = 0.91$, and the top 100 vertices with an accuracy of 73% are obtained. The combination of sparsification and centrality computations results in speedups in the double-digit range for `EstimateBetweenness`

Centrality Measure	Time Ratio	τ_b	$M_{f,r}(100)$
KadabraBetweenness	0.591	0.60	0.81
EstimateBetweenness	$\mathbf{1.285}$	0.80	$\mathbf{0.83}$
Degree	0.001	0.69	0.66
Closeness	1.051	$\mathbf{0.85}$	0.53
Eigenvector	0.103	0.72	0.27
Katz	0.03	0.69	0.65
k-core	0.015	0.60	0.04

Table 5.9: Time ratios, correlation coefficients τ_b and the centrality similarity for the top 100 vertices of denser graphs for the top configuration.

and `ApproxCloseness`, making it significantly faster than computations on full graphs. The vertex rankings exhibit strong correlation, with correlation coefficients surpassing 0.8. Moreover, good results are achieved for the overlap of the most central vertices. It is expected that a speedup will be evident even for larger graphs, highlighting the potential of the approach.

CHAPTER 6

Conclusion

In this work, a novel edge sparsification algorithm is presented. Edges that occur frequently on shortest paths are preserved, while less relevant edges are removed. To determine their importance, three variants of BFS are introduced, which perform SSSP computations on subsets of vertices. Subsequently, the reduced graph is constructed using MWSFs for which two approaches are proposed. The parameters are tuned for two distinct objectives – the ranking of vertices and the identification of the most central vertices. Thereby, all combinations of BFS variants and MWSF methods are investigated. The goal is to achieve maximum quality with a minimum speedup of 10%.

The tuning experiments indicate that for identifying the most central vertices, an increased number of BFS computations enhance the quality. However, for the ranking, a plateau is already reached after a few BFS executions and more repetitions do not lead to qualitative improvements. Simultaneously, the number of edges in the reduced graph plays an important role. As the number of edges decreases, the approach becomes faster, at the cost of decreasing quality. Concerning the most central vertices, this can be improved by increasing the number of BFS executions. The increased computational effort is then compensated by the speedup of centrality computations on graphs with fewer edges. As the number of edges increases, the quality improves steadily, but at the same time, the speedup decreases. Therefore, a trade-off must be found between the number of BFS executions and removed edges. If more than 60% of edges are preserved during edge sparsification, no difference in computation time between reduced and full graphs is achieved.

Based on the parameter configurations of the tuning experiments, the approach is evaluated on a diverse graph collection using multiple centrality measures. The results for path-based centralities clearly outperform the ones for degree, eigenvector, Katz and k-core centrality. No speedup is observed for these measures; on the contrary, the computation times surpass those on full graphs. The edge sparsification introduces a computational overhead that the centrality measures can not compensate for, because they are already highly efficient on unreduced graphs. Betweenness and closeness centrality lead to speedups of 28% and 14% on average.

Path-based centralities achieve notably higher qualitative outcomes. The best ranking results are obtained for closeness centrality, showing a strong correlation between reduced and unreduced graphs. Moreover, there is a large overlap concerning the most central vertices. While betweenness centrality is faster, its ability to determine the correct ranking lacks behind closeness centrality. However, most central vertices are identified with a higher accuracy. The remaining centrality measures can not compete with these findings – neither qualitatively nor with respect to the runtime.

6.1 Discussion

Throughout the experiments, the variation of parameters has shown that the results follow clear tendencies. It is expected that between data points no significant outliers occur. Consequently, employing linear interpolation between two data points provides valid parameter selections. Choosing the mean value of two consecutive parameters is, therefore, a practical approach, particularly in situations where an optimal value is assumed to lie between two tested options. This can be repeated iteratively to eventually converge to a local optimum.

Moreover, the results demonstrated that the amount of removed edges is an important parameter of the proposed approach. It was observed that the number of edges in the reduced graph varies significantly between the two MWSF methods. MWSF-single first computes a MWST and then inserts a certain amount of edges. In contrast, MWSF-iterative conducts iterations until the reduced graph contains at least a specified amount of edges. Consequently, MWSF-single could be altered, such that the reduced graph does not exceed the specified percentage.

Each centrality algorithm exhibits an error tolerance that ensures the accuracy of computed centrality scores. This directly affects the quality and runtime of the approach. For certain centralities, the scores often fall into a narrow interval [101, 73], making them hard to distinguish. In order to compare and rank them, a high level of accuracy must be ensured. In the case of betweenness centrality, the majority of vertices have tiny scores with only a few exhibiting higher values. Consequently, the error tolerance must be chosen very small, leading to a significantly higher runtime. It is thus worth exploring how different values influence the outcome of the edge sparsification. Determining suitable error tolerances for each individual method is expected to further improve the results.

6.2 Future Work

Through the experiments, a lot of insights have been gained regarding the various methods and parameters of the edge sparsification algorithm. Several extensions are yet unexplored and could be investigated further. In all BFS variants, the vertices in the subset are chosen randomly. Alternative strategies, such as selecting vertices depending on certain criteria, potentially improve the performance and more accurately capture the importance of edges within a graph. The set could be chosen, such that the vertices correspond to a distribution, for example, in terms of the degree or any other attributes associated with vertices, like labels or weights.

Concerning the calculation of edge counts, not only one but multiple BFS methods could be combined. They either operate on the same subset or select different sets independently. Moreover, the BFS-percentage approach can be extended to sets of vertices. This, can be further divided into two methods, depending on whether a BFS is executed for all vertices or for each vertex individually.

The experimental evaluation demonstrated that the ranking quality does not improve with an increased number of BFS executions. A possible explanation is the presence of many edges with the same count. To tackle this issue, an additional criterion could be developed, such that a differentiation among these edges is possible. This could be an interesting future research direction.

The combination of MWSF approaches could also be an aspect of further research. For instance, MWSF-iterative is initially applied until the graph contains a certain amount of edges. An additional percentage of edges with the highest counts is subsequently inserted into the graph. Moreover, different criteria can be explored for selecting edges with MWSF-single. Instead of choosing solely important edges, a certain number of edges could be sampled randomly, which potentially improves the performance of the approach. To further reduce the overall running time of the edge sparsification, components can be parallelized, such as the BFS and centrality computations.

APPENDIX A

Further Results

BFS Variant	MWSF	Set Size	BFS Percentage	Repetitions	Edge [%]
Set	Single	5	–	25	30
	Iterative	5	–	5	45
Single	Single	5	–	–	35
	Iterative	3	–	–	45
Percentage	Single	1	0.99	1	30
	Iterative	1	0.02	1	30

Table A.1: Baseline parameter configuration for the tuning experiments with focus on the ranking of vertices.

BFS Variant	MWSF	Set Size	BFS Percentage	Repetitions	Edge [%]
Set	Single	10	–	35	30
	Iterative	5	–	35	45
Single	Single	50	–	–	35
	Iterative	40	–	–	40
Percentage	Single	1	0.99	1	30
	Iterative	1	0.02	1	30

Table A.2: Baseline parameter configuration for the tuning experiments with focus on the most central vertices.

bfs_percentage	Average number of BFS
0.02	2473.88
0.04	1911.33
0.06	1603.6
0.08	1404.44
0.1	1258.94
0.2	836.61
0.3	654.35
0.4	528.87
0.5	442.73
0.6	354.93
0.7	284.22
0.75	251.62
0.8	219.13
0.85	182.8
0.9	143.33
0.95	91.79
0.98	52.31
0.99	34.83

Table A.3: Average number of BFS runs for increasing values of *bfs_percentage* during the tuning experiments of Section 5.4.1.

	0.05	0.15	0.25	0.35	0.45	0.55
Amount of edges in %	0.24	0.29	0.34	0.44	0.52	0.62
Time Ratio for centrality computations	2.08	1.72	1.47	1.24	1.17	1.04
Correlation coefficient τ_b %	0.56	0.63	0.7	0.76	0.8	0.83

Table A.4: Amount of edges, time ratios and correlation coefficients for varying values of the parameter *edge_percentage* for BFS-single combined with MWSF-iterative during the ranking tuning experiments Section 5.4.1.

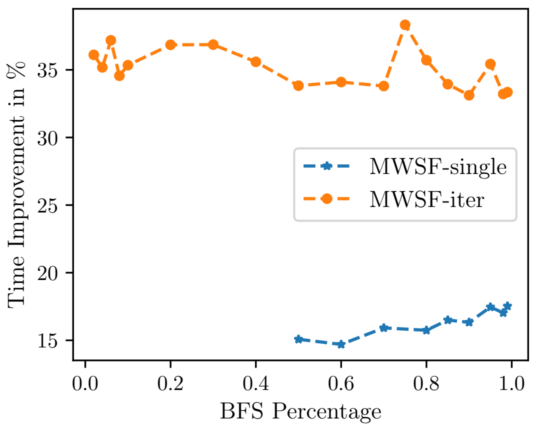


Figure A.1: Time improvement in percent for the centrality computations over varying values for *bfs_percentage*. Note that the speedups depend solely on the centrality computations, excluding the computation time of the sparsification algorithm. On the reduced graphs obtained with MWSF-iterative centrality computations run on average 25% faster than on the ones resulting from MWSF-single.

	0.05	0.15	0.25	0.35	0.45	0.55
Amount of edges in %	0.28	0.38	0.48	0.57	0.66	0.75
Time Ratio for centrality computations	1.81	1.41	1.22	1.09	1.01	0.96
Correlation coefficient τ_b %	0.55	0.74	0.82	0.86	0.89	0.91

Table A.5: Amount of edges, time ratios and correlation coefficients for varying values of the parameter *edge_percentage* for BFS-single combined with MWSF-single during the ranking tuning experiments Section 5.4.1.

Zusammenfassung

Ein grundlegendes Konzept der Graphenanalyse sind Zentralitätsmetriken, die jedem Knoten eine Wertigkeit zuweisen, die seine Relevanz darstellt. Insbesondere die relative Rangfolge und die Identifizierung der wichtigsten Knoten bieten Einblicke in den zugrunde liegenden Datensatz und finden Einsatz in verschiedenen Anwendungsbereichen. Obwohl es exakte Algorithmen in polynomialer Laufzeit gibt, sind diese nicht gut skalierbar und daher wurden zahlreiche Approximationsalgorithmen entwickelt. In dieser Arbeit wird eine Edge Sparsification vorgestellt, die vor den Zentralitätsberechnungen auf Graphen angewendet wird. Dabei wird ermittelt, wie oft jede Kante auf kürzesten Wegen innerhalb eines Graphen vorkommt. Der reduzierte Graph wird dann konstruiert, indem nur Kanten einbezogen werden, die Teil vieler kürzester Pfade sind. Die Experimente zeigen, dass die Edge Sparsification am besten für distanzbasierte Zentralitätsmetriken funktioniert. Für Betweenness Centrality werden stark korrelierte Rangfolgen und eine Überlappung von 83% zwischen den wichtigsten Knoten mit einer durchschnittlichen Verbesserung der Laufzeit von 26% erreicht. Für Closeness Centrality werden ähnliche Ergebnisse mit einer durchschnittlichen Beschleunigung von 21% erzielt.

Bibliography

[1] Amir Abboud, Fabrizio Grandoni, and Virginia Vassilevska Williams. Subcubic Equivalences between Graph Centrality Problems, APSP, and Diameter. *ACM Transactions on Algorithms*, 19(1):1–30, January 2023. ISSN 1549-6325, 1549-6333. doi: 10.1145/3563393.

[2] Eugenio Angriman, Alexander van der Grinten, Michael Hamann, Henning Meyerhenke, and Manuel Penschuck. Algorithms for Large-Scale Network Analysis and the NetworKit Toolkit. In Hannah Bast, Claudius Korzen, Ulrich Meyer, and Manuel Penschuck, editors, *Algorithms for Big Data*, volume 13201, pages 3–20. Springer Nature Switzerland, Cham, 2022. ISBN 978-3-031-21533-9 978-3-031-21534-6. doi: 10.1007/978-3-031-21534-6_1.

[3] Jac M. Anthonisse. The rush in a directed graph. *Stichting Mathematisch Centrum, Amsterdam, Netherlands*, Technical Report BN 9/71:1–10, 1971.

[4] D.A. Bader and K. Madduri. Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 539–550, Columbus, OH, USA, 2006. IEEE. ISBN 978-0-7695-2636-2. doi: 10.1109/ICPP.2006.57.

[5] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating Betweenness Centrality. In Anthony Bonato and Fan R. K. Chung, editors, *Algorithms and Models for the Web-Graph*, volume 4863, pages 124–137. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-77003-9. doi: 10.1007/978-3-540-77004-6_10.

[6] David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for Graph Clustering and Partitioning. In Reda Alhajj and Jon Rokne, editors, *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer New York, New York, NY, 2014. ISBN 978-1-4614-6169-2 978-1-4614-6170-8. doi: 10.1007/978-1-4614-6170-8_23.

[7] V. Batagelj and M. Zaversnik. An O(m) Algorithm for Cores Decomposition of Networks, October 2003.

[8] Alex Bavelas. A Mathematical Model for Group Structures. *Human Organization*, 7(3):16–30, July 1948. ISSN 0018-7259, 1938-3525. doi: 10.17730/humo.7.3.f4033344851gl053.

[9] Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke. Computing top- k Closeness Centrality Faster in Unweighted Graphs. *ACM Transactions on Knowledge Discovery from Data*, 13(5):1–40, October 2019. ISSN 1556-4681, 1556-472X. doi: 10.1145/3344719.

[10] Massimo Bernaschi, Giancarlo Carbone, and Flavio Vella. Scalable betweenness centrality on multi-GPU systems. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 29–36, Como Italy, May 2016. ACM. ISBN 978-1-4503-4128-8. doi: 10.1145/2903150.2903153.

[11] Paolo Boldi and Sebastiano Vigna. Axioms for centrality. *Internet Mathematics*, 10 (3-4):222–262, 2014. doi: 10.1080/15427951.2013.865686.

[12] Phillip Bonacich. Factoring and weighting approaches to status scores and clique identification. *The Journal of Mathematical Sociology*, 2(1):113–120, January 1972. ISSN 0022-250X, 1545-5874. doi: 10.1080/0022250X.1972.9989806.

[13] Michele Borassi and Emanuele Natale. KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation. *ACM Journal of Experimental Algorithmics*, 24:1–35, December 2019. ISSN 1084-6654, 1084-6654. doi: 10.1145/3284359.

[14] Michele Borassi, Pierluigi Crescenzi, and Andrea Marino. Fast and simple computation of top-k closeness centralities. *CoRR*, abs/1507.01490, 2015.

[15] Michele Borassi, Pierluigi Crescenzi, and Michel Habib. Into the Square: On the Complexity of Some Quadratic-time Solvable Problems. *Electronic Notes in Theoretical Computer Science*, 322:51–67, April 2016. ISSN 15710661. doi: 10.1016/j.entcs.2016.03.005.

[16] Stephen P. Borgatti and Martin G. Everett. A Graph-theoretic perspective on centrality. *Social Networks*, 28(4):466–484, October 2006. ISSN 03788733. doi: 10.1016/j.socnet.2005.11.005.

[17] Ulrik Brandes. A faster algorithm for betweenness centrality*. *The Journal of Mathematical Sociology*, 25(2):163–177, June 2001. ISSN 0022-250X, 1545-5874. doi: 10.1080/0022250X.2001.9990249.

[18] Ulrik Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2):136–145, May 2008. ISSN 03788733. doi: 10.1016/j.socnet.2007.11.001.

[19] Ulrik Brandes and Thomas Erlebach, editors. *Network Analysis: Methodological Foundations*. Number 3418 in LCNS, Tutorial. Springer, Berlin ; New York, 2005. ISBN 978-3-540-24979-5.

[20] Ulrik Brandes and Christian Pich. Centrality Estimation in Large Networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, July 2007. ISSN 0218-1274, 1793-6551. doi: 10.1142/S0218127407018403.

[21] Gary Chartrand and Linda M. Lesniak. *Graphs and Digraphs (2. Ed.)*. Wadsworth & Brooks / Cole Mathematics Series. Wadsworth, 1986. ISBN 978-0-534-06324-5.

[22] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F. Werneck. Computing Classic Closeness Centrality, at Scale. In *Proceedings of the Second ACM Conference on Online Social Networks*, pages 37–50, October 2014. doi: 10.1145/2660460.2660465.

[23] Cecile Daniel, Angelo Furno, Lorenzo Goglia, and Eugenio Zimeo. Fast cluster-based computation of exact betweenness centrality in large graphs. *Journal of Big Data*, 8(1):92, December 2021. ISSN 2196-1115. doi: 10.1186/s40537-021-00483-1.

[24] Kousik Das, Sovan Samanta, and Madhumangal Pal. Study on centrality measures in social networks: A survey. *Social Network Analysis and Mining*, 8(1):13, December 2018. ISSN 1869-5450, 1869-5469. doi: 10.1007/s13278-018-0493-2.

[25] Naga Shailaja Dasari, Ranjan Desh, and M. Zubair. ParK: An efficient algorithm for k-core decomposition on multicore processors. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 9–16, Washington, DC, USA, October 2014. IEEE. ISBN 978-1-4799-5666-1. doi: 10.1109/BigData.2014.7004366.

[26] A. Del Sol, H. Fujihashi, and P. O'Meara. Topology of small-world networks of protein-protein complex structures. *Bioinformatics*, 21(8):1311–1315, April 2005. ISSN 1367-4803, 1460-2059. doi: 10.1093/bioinformatics/bti167.

[27] Reinhard Diestel. *Graph Theory*. Number 173 in Graduate Texts in Mathematics. Springer, Berlin, fifth edition, first softcover printing edition, 2018. ISBN 978-3-662-53621-6 978-3-662-57560-4.

[28] Mark Ditsworth and Justin Ruths. Community detection via katz and eigenvector centrality. *CoRR*, abs/1909.03916, 2019.

[29] Shlomi Dolev, Yuval Elovici, and Rami Puzis. Routing betweenness centrality. *Journal of the ACM*, 57(4):1–27, April 2010. ISSN 0004-5411, 1557-735X. doi: 10.1145/1734213.1734219.

[30] Nick Edmonds, Torsten Hoefler, and Andrew Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *2010 International Conference on High Performance Computing*, pages 1–10, Goa, India, December 2010. IEEE. ISBN 978-1-4244-8518-5. doi: 10.1109/HIPC.2010.5713180.

[31] David Eppstein and Joseph Wang. Fast approximation of centrality. *Journal of Graph Algorithms and Applications*, 8:39–45, 2004. doi: 10.7155/JGAA.00081.

[32] Dóra Erdős, Vatche Ishakian, Azer Bestavros, and Evimaria Terzi. A Divide-and-Conquer Algorithm for Betweenness Centrality. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 433–441. Society for Industrial and Applied Mathematics, June 2015. ISBN 978-1-61197-401-0. doi: 10.1137/1.9781611974010.49.

[33] Changjun Fan, Li Zeng, Yuhui Ding, Muhao Chen, Yizhou Sun, and Zhong Liu. Learning to Identify High Betweenness Centrality Nodes from Scratch: A Novel Graph Neural Network Approach. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 559–568, Beijing China, November 2019. ACM. ISBN 978-1-4503-6976-3. doi: 10.1145/3357384.3357979.

[34] Marcelo Fonseca Faraj and Christian Schulz. Buffered Streaming Graph Partitioning. *ACM Journal of Experimental Algorithmics*, 27:1–26, December 2022. ISSN 1084-6654, 1084-6654. doi: 10.1145/3546911.

[35] Kurt C. Foster, Stephen Q. Muth, John J. Potterat, and Richard B. Rothenberg. A Faster Katz Status Score Algorithm. *Computational & Mathematical Organization Theory*, 7(4):275–285, 2001. ISSN 1381298X. doi: 10.1023/A:1013470632383.

[36] Linton C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35, March 1977. ISSN 00380431. doi: 10.2307/3033543.

[37] Linton C. Freeman, Stephen P. Borgatti, and Douglas R. White. Centrality in valued graphs: A measure of betweenness based on network flow. *Social Networks*, 13(2):141–154, June 1991. ISSN 03788733. doi: 10.1016/0378-8733(91)90017-N.

[38] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better Approximation of Betweenness Centrality. In J. Ian Munro and Dorothea Wagner, editors, *2008 Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 90–100. Society for Industrial and Applied Mathematics, Philadelphia, PA, January 2008. ISBN 978-1-61197-288-7. doi: 10.1137/1.9781611972887.9.

[39] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, June 2002. ISSN 0027-8424, 1091-6490. doi: 10.1073/pnas.122653799.

[40] Felipe Grando and Luis C. Lamb. On approximating networks centrality measures via neural learning algorithms. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 551–557, Vancouver, BC, Canada, July 2016. IEEE. ISBN 978-1-5090-0620-5. doi: 10.1109/IJCNN.2016.7727248.

[41] Felipe Grando, Diego Noble, and Luis C. Lamb. An Analysis of Centrality Measures for Complex and Social Networks. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Washington, DC, USA, December 2016. IEEE. ISBN 978-1-5090-1328-9. doi: 10.1109/GLOCOM.2016.7841580.

[42] Felipe Grando, Lisandro Z. Granville, and Luis C. Lamb. Machine Learning in Network Centrality Measures: Tutorial and Outlook. *ACM Computing Surveys*, 51 (5):1–32, September 2019. ISSN 0360-0300, 1557-7341. doi: 10.1145/3237192.

[43] Alexander Grinten van der, Elisabetta Bergamini, Oded Green, David A. Bader, and Henning Meyerhenke. Scalable Katz Ranking Computation in Large Static and Dynamic Graphs. *ACM Journal of Experimental Algorithmics*, 27:1–16, December 2022. ISSN 1084-6654, 1084-6654. doi: 10.1145/3524615.

[44] R. Guimerà, S. Mossa, A. Turtschi, and L. A. N. Amaral. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *Proceedings of the National Academy of Sciences*, 102(22):7794–7799, May 2005. ISSN 0027-8424, 1091-6490. doi: 10.1073/pnas.0407994102.

[45] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent Advances in Fully Dynamic Graph Algorithms – A Quick Reference Guide. *ACM Journal of Experimental Algorithmics*, 27:1–45, December 2022. ISSN 1084-6654, 1084-6654. doi: 10.1145/3555806.

[46] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12. IEEE, 2010. doi: 10.1109/IPDPS.2010.5470485.

[47] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, May 2001. ISSN 0028-0836, 1476-4687. doi: 10.1038/35075138.

[48] R. Jothi. A Betweenness Centrality Guided Clustering Algorithm and Its Applications to Cancer Diagnosis. In Ashish Ghosh, Rajarshi Pal, and Rajendra Prasath,

editors, *Mining Intelligence and Knowledge Exploration*, volume 10682, pages 35–42. Springer International Publishing, Cham, 2017. ISBN 978-3-319-71927-6 978-3-319-71928-3. doi: 10.1007/978-3-319-71928-3_4.

[49] Humayun Kabir and Kamesh Madduri. Parallel k-Core Decomposition on Multicore Platforms. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1482–1491, Orlando / Buena Vista, FL, USA, May 2017. IEEE. ISBN 978-1-5386-3408-0. doi: 10.1109/IPDPSW.2017.151.

[50] U Kang, Spiros Papadimitriou, Jimeng Sun, and Hanghang Tong. Centralities in Large Networks: Algorithms and Observations. In *Proceedings of the 2011 SIAM International Conference on Data Mining*, pages 119–130. Society for Industrial and Applied Mathematics, April 2011. ISBN 978-0-89871-992-5 978-1-61197-281-8. doi: 10.1137/1.9781611972818.11.

[51] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18 (1):39–43, March 1953. ISSN 0033-3123, 1860-0980. doi: 10.1007/BF02289026.

[52] Dirk Koschützki and Falk Schreiber. Centrality Analysis Methods for Biological Networks and Their Application to Gene Regulatory Networks. *Gene Regulation and Systems Biology*, 2:GRSB.S702, January 2008. ISSN 1177-6250, 1177-6250. doi: 10.4137/GRSB.S702.

[53] Nicolas Kourtellis, Tharaka Alahakoon, Ramanuja Simha, Adriana Iamnitchi, and Rahul Tripathi. Identifying high betweenness centrality nodes in large social networks. *Social Network Analysis and Mining*, 3(4):899–914, December 2013. ISSN 1869-5450, 1869-5469. doi: 10.1007/s13278-012-0076-6.

[54] Vladis E Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002.

[55] Jérôme Kunegis. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*, pages 1343–1350, 2013.

[56] Erwan Le Merrer, Nicolas Le Scouarnec, and Gilles Trédan. Heuristical top-k: Fast estimation of centralities in complex networks. *Information Processing Letters*, 114 (8):432–436, August 2014. ISSN 00200190. doi: 10.1016/j.ipl.2014.03.006.

[57] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014.

[58] Yong Li, Wenguo Li, Yi Tan, Fang Liu, Yijia Cao, and Kwang Y. Lee. Hierarchical Decomposition for Betweenness Centrality Measure of Complex Networks. *Scientific Reports*, 7(1):46491, April 2017. ISSN 2045-2322. doi: 10.1038/srep46491.

[59] Mingkai Lin, Wenzhong Li, Lynda J. Song, Cam-Tu Nguyen, Xiaoliang Wang, and Sanglu Lu. SAKE: Estimating Katz Centrality Based on Sampling for Large-Scale Social Networks. *ACM Transactions on Knowledge Discovery from Data*, 15(4): 1–21, August 2021. ISSN 1556-4681, 1556-472X. doi: 10.1145/3441646.

[60] Jing-Kai Lou, Shou-de Lin, Kuan-Ta Chen, and Chin-Laung Lei. What Can the Temporal Social Behavior Tell Us? An Estimation of Vertex-Betweenness Using Dynamic Social Information. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 56–63, Odense, Denmark, August 2010. IEEE. ISBN 978-1-4244-7787-6. doi: 10.1109/ASONAM.2010.46.

[61] Kamesh Madduri, David Ediger, Karl Jiang, David A. Bader, and Daniel Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, Rome, Italy, May 2009. IEEE. ISBN 978-1-4244-3751-1. doi: 10.1109/IPDPS.2009.5161100.

[62] John Matta, Gunes Ercal, and Koushik Sinha. Comparing the speed and accuracy of approaches to betweenness centrality approximation. *Computational Social Networks*, 6(1):2, December 2019. ISSN 2197-4314. doi: 10.1186/s40649-019-0062-5.

[63] Sunil Kumar Maurya, Xin Liu, and Tsuyoshi Murata. Fast Approximations of Betweenness Centrality with Graph Neural Networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2149–2152, Beijing China, November 2019. ACM. ISBN 978-1-4503-6976-3. doi: 10.1145/3357384.3358080.

[64] Adam McLaughlin and David A. Bader. Scalable and High Performance Betweenness Centrality on the GPU. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 572–583, New Orleans, LA, USA, November 2014. IEEE. ISBN 978-1-4799-5500-8 978-1-4799-5499-5. doi: 10.1109/SC.2014.52.

[65] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, Berlin, 2008. ISBN 978-3-540-77977-3 978-3-540-77978-0.

[66] Matheus R. F. Mendonça, André M. S. Barreto, and Artur Ziviani. Approximating Network Centrality Measures Using Node Embedding and Machine Learning. *IEEE Transactions on Network Science and Engineering*, 8(1):220–230, January 2021. ISSN 2327-4697, 2334-329X. doi: 10.1109/TNSE.2020.3035352.

[67] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. In Cyril Gavoille and Pierre Fraigniaud, editors, *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC*

2011, San Jose, CA, USA, June 6-8, 2011, pages 207–208. ACM, 2011. doi: 10.1145/1993806.1993836.

[68] Eisha Nathan and David A. Bader. Approximating Personalized Katz Centrality in Dynamic Graphs. In Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, volume 10777, pages 290–302. Springer International Publishing, Cham, 2018. ISBN 978-3-319-78023-8 978-3-319-78024-5. doi: 10.1007/978-3-319-78024-5_26.

[69] Eisha Nathan and David A. Bader. Incrementally updating Katz centrality in dynamic graphs. *Social Network Analysis and Mining*, 8(1):26, December 2018. ISSN 1869-5450, 1869-5469. doi: 10.1007/s13278-018-0504-3.

[70] Eisha Nathan, Geoffrey Sanders, James Fairbanks, Van Emden Henson, and David A. Bader. Graph Ranking Guarantees for Numerical Approximations to Katz Centrality. *Procedia Computer Science*, 108:68–78, 2017. ISSN 18770509. doi: 10.1016/j.procs.2017.05.021.

[71] M. E. J. Newman. Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality. *Physical Review E*, 64(1):016132, June 2001. ISSN 1063-651X, 1095-3787. doi: 10.1103/PhysRevE.64.016132.

[72] M. E. J. Newman. A measure of betweenness centrality based on random walks. *Social Networks*, 27(1):39–54, 2005. doi: 10.1016/J.SOCNET.2004.11.009.

[73] Mark E. J. Newman. *Networks*. Oxford University Press, Oxford, second edition edition, 2018. ISBN 978-0-19-880509-0. doi: 10.1093/oso/9780198805090.001.0001.

[74] Paul W. Olsen, Alan G. Labouseur, and Jeong-Hyon Hwang. Efficient top-k closeness centrality search. In *2014 IEEE 30th International Conference on Data Engineering*, pages 196–207, Chicago, IL, USA, March 2014. IEEE. ISBN 978-1-4799-2555-1. doi: 10.1109/ICDE.2014.6816651.

[75] Rasmus Pagh and Suresh Venkatasubramanian, editors. *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, January 2018. ISBN 978-1-61197-505-5. doi: 10.1137/1.9781611975055.

[76] Edgar M. Palmer. On the spanning tree packing number of a graph: A survey. 230 (1-3):13–21, 2001. doi: 10.1016/S0012-365X(00)00066-2.

[77] Jürgen Pfeffer and Kathleen M. Carley. K-Centralities: Local approximations of global measures based on shortest paths. In *Proceedings of the 21st International Conference on World Wide Web*, pages 1043–1050, Lyon France, April 2012. ACM. ISBN 978-1-4503-1230-1. doi: 10.1145/2187980.2188239.

[78] Rami Puzis, Polina Zilberman, Yuval Elovici, Shlomi Dolev, and Ulrik Brandes. Heuristics for Speeding Up Betweenness Centrality Computation. In *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*, pages 302–311, Amsterdam, Netherlands, September 2012. IEEE. ISBN 978-1-4673-5638-1 978-0-7695-4848-7. doi: 10.1109/SocialCom-PASSAT.2012.66.

[79] Appan Rakaraddi and Mahardhika Pratama. Unsupervised Learning for Identifying High Eigenvector Centrality Nodes: A Graph Neural Network Approach. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 4945–4954, Orlando, FL, USA, December 2021. IEEE. ISBN 978-1-66543-902-2. doi: 10.1109/BigData52589.2021.9671902.

[80] Matthew J. Rattigan, Marc Maier, and David Jensen. Using structure indices for efficient approximation of network properties. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 357–366, Philadelphia PA USA, August 2006. ACM. ISBN 978-1-59593-339-3. doi: 10.1145/1150402.1150443.

[81] Matteo Riondato and Evgenios M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, pages 413–422, New York New York USA, February 2014. ACM. ISBN 978-1-4503-2351-2. doi: 10.1145/2556195.2556224.

[82] Matteo Riondato and Eli Upfal. ABRA: Approximating Betweenness Centrality in Static and Dynamic Graphs with Rademacher Averages. *ACM Transactions on Knowledge Discovery from Data*, 12(5):1–38, October 2018. ISSN 1556-4681, 1556-472X. doi: 10.1145/3208351.

[83] Agnieszka Rusinowska, Rudolf Berghammer, Harrie de Swart, and Michel Grabisch. Social networks: Prestige, centrality, and influence (Invited paper). In de Swart, editor, *RAMICS 2011*, Lecture Notes in Computer Science (LNCS) 6663, pages 22–39. Springer, 2011.

[84] Gert Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, December 1966. ISSN 0033-3123, 1860-0980. doi: 10.1007/BF02289527.

[85] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 76–85, Houston Texas USA, March 2013. ACM. ISBN 978-1-4503-2017-7. doi: 10.1145/2458523.2458531.

[86] Ahmet Erdem Sariyuce, Kamer Kaya, Erik Saule, and Umit V. Catalyurek. Incremental algorithms for closeness centrality. In *2013 IEEE International Conference*

on Big Data, pages 487–492, Silicon Valley, CA, October 2013. IEEE. ISBN 978-1-4799-1293-3. doi: 10.1109/BigData.2013.6691611.

[87] Ahmet Erdem Sariyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. Shattering and Compressing Networks for Betweenness Centrality. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 686–694. Society for Industrial and Applied Mathematics, May 2013. ISBN 978-1-61197-262-7 978-1-61197-283-2. doi: 10.1137/1.9781611972832.76.

[88] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Graph Manipulations for Fast Centrality Computation. *ACM Transactions on Knowledge Discovery from Data*, 11(3):1–25, August 2017. ISSN 1556-4681, 1556-472X. doi: 10.1145/3022668.

[89] Akrati Saxena and Sudarshan Iyengar. Centrality measures in complex networks: A survey. *CoRR*, abs/2011.07190, 2020.

[90] Akrati Saxena, Ralucca Gera, and S. R. S. Iyengar. Degree ranking using local information. *CoRR*, abs/1706.01205, 2017.

[91] Akrati Saxena, Ralucca Gera, and S. R. S. Iyengar. Fast Estimation of Closeness Centrality Ranking. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, pages 80–85, Sydney Australia, July 2017. ACM. ISBN 978-1-4503-4993-2. doi: 10.1145/3110025.3110064.

[92] Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, September 1983. ISSN 03788733. doi: 10.1016/0378-8733(83)90028-X.

[93] Puneet Sharma, Udayan Khurana, Ben Shneiderman, Max Scharrenbroich, and John Locke. Speeding Up Network Layout and Centrality Measures for Social Computing Goals. In John Salerno, Shanchieh Jay Yang, Dana Nau, and Sun-Ki Chai, editors, *Social Computing, Behavioral-Cultural Modeling and Prediction*, volume 6589, pages 244–251. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-19655-3 978-3-642-19656-0. doi: 10.1007/978-3-642-19656-0_35.

[94] Zhiao Shi and Bing Zhang. Fast network centrality analysis using GPUs. *BMC Bioinformatics*, 12(1):149, December 2011. ISSN 1471-2105. doi: 10.1186/1471-2105-12-149.

[95] Rishi Ranjan Singh. Centrality measures: A tool to identify key actors in social networks. *CoRR*, abs/2011.01627, 2020.

[96] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. NetworKit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, December 2016. ISSN 2050-1242, 2050-1250. doi: 10.1017/nws.2016.20.

[97] Paolo Suppa and Eugenio Zimeo. A Clustered Approach for Fast Computation of Betweenness Centrality in Social Networks. In *2015 IEEE International Congress on Big Data*, pages 47–54, New York City, NY, USA, June 2015. IEEE. ISBN 978-1-4673-7278-7. doi: 10.1109/BigDataCongress.2015.17.

[98] S. Trajanovski, J. Martin-Hernandez, W. Winterbach, and P. Van Mieghem. Robustness envelopes of networks. *Journal of Complex Networks*, 1(1):44–62, June 2013. ISSN 2051-1310, 2051-1329. doi: 10.1093/comnet/cnt004.

[99] Alexander Van Der Grinten and Henning Meyerhenke. Scaling Betweenness Approximation to Billions of Edges by MPI-based Adaptive Sampling. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 527–535, New Orleans, LA, USA, May 2020. IEEE. ISBN 978-1-72816-876-0. doi: 10.1109/IPDPS47924.2020.00061.

[100] Alexander van der Grinten, Eugenio Angriman, and Henning Meyerhenke. Parallel adaptive sampling with almost no synchronization. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*, volume 11725 of *Lecture Notes in Computer Science*, pages 434–447. Springer, 2019. doi: 10.1007/978-3-030-29400-7_31.

[101] Alexander Van Der Grinten, Eugenio Angriman, and Henning Meyerhenke. Scaling up network centrality computations – A brief overview. *it - Information Technology*, 62(3-4):189–204, May 2020. ISSN 2196-7032, 1611-2776. doi: 10.1515/itit-2019-0032.

[102] James H. Wilkinson. *The Algebraic Eigenvalue Problem*. Monographs on Numerical Analysis. Clarendon Press, Oxford, reprinted from corr. sheets of the 1. ed edition, 1978. ISBN 978-0-19-853403-7.

[103] Stefan Wuchty and Peter F. Stadler. Centers of complex networks. *Journal of Theoretical Biology*, 223(1):45–53, July 2003. ISSN 00225193. doi: 10.1016/S0022-5193(03)00071-7.

[104] Erjia Yan and Ying Ding. Applying centrality measures to impact analysis: A coauthorship network analysis. *Journal of the American Society for Information Science and Technology*, 60(10):2107–2118, October 2009. ISSN 15322882, 15322890. doi: 10.1002/asi.21128.

Acronyms

ABRA Approximating Betweenness with Rademacher Averages

APSP all-pairs shortest path

BFS breadth-first search

CPU central processing unit

GNN graph neural network

GPU graphics processing unit

KADABRA ADaptive Algorithm for Betweenness via Random Approximation

KONECT Koblenz Network Collection

MPI Message Passing Interface

MWSF maximum-weight spanning forest

MWST maximum-weight spanning tree

SNAP Stanford Network Analysis Platform

SSSP single-source shortest path