A Divide-and-Conquer Algorithm for Betweenness Centrality [*]

Dóra Erdős [†] Vatche Ishakian[‡] Azer Bestavros [†] Evimaria Terzi [†]

January 26, 2015

Abstract

Given a set of target nodes S in a graph G we define the betweenness centrality of a node v with respect to S as the fraction of shortest paths among nodes in S that contain v. For this setting we describe `Brandes++`, a divide-and-conquer algorithm that can efficiently compute the exact values of betweenness scores. `Brandes++` uses `Brandes`– the most widely-used algorithm for betweenness computation – as its subroutine. It achieves the notable faster running times by applying `Brandes` on significantly smaller networks than the input graph, and many of its computations can be done in parallel. The degree of speedup achieved by `Brandes++` depends on the community structure of the input network as well as the size of S. Our experiments with real-life networks reveal `Brandes++` achieves an average of 10-fold speedup over `Brandes`, while there are networks where this speedup is 75-fold. We have made our code public to benefit the research community.

1 Introduction

In 1977, Freeman [10] defined the betweenness centrality of a node v as the fraction of all pairwise shortest paths that go through v. Since then, this measure of centrality has been used in a wide range of applications including social, computer as well as biological networks.

A naïve algorithm can compute the betweenness centrality of a graph of n nodes in $O(n^3)$ time. This running time was first improved in 2001 by Brandes [4] who provided an algorithm that, for a graph of n nodes and m edges, does the same computation in $O(nm+n^2 \log n)$. The key behind this algorithm, which we call `Brandes` is that it reuses information on shortest path segments that are shared by many nodes.

Over the years, many algorithms have been proposed to improve the running and space complexity of `Brandes`. Although we discuss these algorithms in the next section, we point out here that most of them either provide approximate computations of betweenness via sampling [2, 6, 11, 23], or propose parallelization of the original computation [3, 18, 27, 9].

While we consider the general problem of betweenness centrality computation, we focus on a particular setting where a *target set* of nodes S (subset of the nodes in the graph) is given and the goal is to compute the betweenness centrality of a node v with respect to S; i.e., the fraction of shortest paths containing v connecting *any two nodes in* S. This setting arises in many applications where there is a set of prominent nodes in the network and only the paths to these nodes are considered valuable. Clearly, the original `Brandes` algorithm can be used for our setting and compute exactly the betweenness scores in time $O(|S|m + |S|n \log n)$.

The goal of our paper is to exploit the structure of the underlying graph and further improve this running time, while returning the exact values of betweenness scores. We achieve this goal by designing the `Brandes++` algorithm, which is a *divide-and-conquer* algorithm and works as follows: first it partitions the graph into subgraphs and runs some single-source shortest path computations on these subgraphs. Then it deploys a modified version of `Brandes` on a sketch of the original graph to compute the betweenness of *all* nodes in the graph. The key behind the speed-up of `Brandes++` over `Brandes` is that all computations are run over graphs that are significantly smaller than the original graph. Yet these speedups are only significant if the size of target nodes is comparatively smaller than the number of nodes in the input graph – otherwise `Brandes` and `Brandes++` are identical.

Our experiments with real-life networks suggest that there are networks for which `Brandes++` can yield a 75-fold improvement over `Brandes`. Our analysis reveals that this improvement depends largely on the structural characteristics of the network and mostly on its community structure.

Some other advantages of `Brandes++` are the fol-

[*]This research was supported in part by NSF awards PFI BIC #1430145, SaTC Frontier #1414119, CPS #1239021, CNS #1012798, III #1218437, CAREER #1253393, IIS #1320542 and gifts from Google and Microsoft.

[†]Boston University, Boston MA `[edori, best, evimaria]@cs.bu.edu`

[‡]IBM T. J. Watson Research Center, Cambridge MA `vishaki@us.ibm.com`

lowing: (*i*) Brandes++ can employ all existing speedups for Brandes. (*ii*) Many steps of our algorithm are easily paralellizable. (*iii*) Finally, we have made our code public to benefit the research community.

2 Overview of Related Work

Perhaps the most widely known algorithm for computing betweenness centrality is due to Ulrik Brandes [4], who also studied extension of his algorithm to groups of nodes in Brandes et al. [5]. The Brandes algorithm has motivated a lot of subsequent work that led to parallel versions of the algorithm [3, 18, 27, 9] as well as classical algorithms that approximate the betweenness centrality of nodes [2, 6, 11] or a very recent one [23]. The difference between approximation algorithms and Brandes++ is that in case of the former a subset of the graph (either pivots, shortest paths, etc. depending on the approach) is taken to *estimate* the centrality of all nodes in the graph. In contrary, Brandes++ computes the exact value for every node with respect to the target set S. Further, any parallelism that can be exploited by Brandes can also be exploited by Brandes++.

Despite the huge literature on the topic, there has been only little work on finding an improved centralized algorithm for computing betweenness centrality. To the best of our knowledge, only recently Puzis et al. [21] and Sariyüce et al. [25] focus on that. In the former, the authors suggest two heuristics to speedup the computations. These heuristics can be applied independent of each other. The first one, contracts *structurally-equivalent* nodes (nodes that have identical neighborhoods) into one "supernode". The second heuristic relies on finding the biconnected components of the graph and contracting them into a new type of "supernodes". These latter supernodes are then connected in the graph's biconnected tree. The key observation is that if a shortest path has its endpoints in two different nodes of this tree then all shortest paths between them will traverse the same edges of the tree. Sariyüce et al. [25] rely on these two heuristics and some additional observations to further simplify the computations.

The similarity between our algorithm and the algorithms we described above is in their divide-and-conquer nature. One can see the biconnected components of the graph as the input partition that is provided to our algorithm. However, since our algorithm works with *any* input partition it is more general and thus more flexible. Indicatively, we give some examples of how our algorithm outperforms these two heuristics by comparing some of our experimental results to the results reported in [21] and [25]. In the former, we see that the biconnected component heuristic of Puzis et al. achieves a 3.5-times speedup on the WikiVote dataset. Our experiments with the same data show that Brandes++ provides a 78-factor speedup. For the DBLP dataset Puzis et al. achieve a speedup factor between $2-6$ – depending on the sample. We achieve a factor of 7.8. The best result on a social-network type graph in [25] is a factor of 7.9 speedup while we achieve factors 78 on WikiVote and 7.7 on the EU data.

3 Preliminaries

We start this section by defining betweenness centrality. Then we review some necessary previous results.

Notation: Let $G(V, E, W)$ be an undirected weighted graph with nodes V, edges E and non-negative edge weights W. We denote $|V| = n$ and $|E| = m$. Let $S \subseteq V$ be a subset of nodes. We call S the *target* nodes and assume $2 \leq |S| \leq n$. Let $u, v \in V$. The *distance* between u and v is the *length* of the (weighted) shortest path in G connecting them, we denote this by $d(u, v)$. We denote by $\sigma(u, v)$ the *number* of shortest paths between u and v. For $s, t \in S$ the value $\sigma(s, t|v)$ denotes the number of shortest paths connecting s and t that contain v. Observe, that σ is a symmetric function, thus $\sigma(s, t) = \sigma(t, s)$.

The *dependency* of s and t on v is the fraction of shortest paths connecting s and t that go through v, thus

$$\delta(s, t|v) = \frac{\sigma(s, t, |v)}{\sigma(s, t)}.$$

Given the above, the *betweenness centrality* $C(v)$ of node v can be defined as the sum of its dependencies.

$$C(v) = \sum_{s \neq t \in S} \delta(s, t|v). \tag{3.1}$$

Note, that in the original definition of betweenness (and using our notation) $S = V$. Observe that the definition in Eq. (3.1) covers this version of centrality as well. Throughout the paper we use the terms betweenness, centrality and betweenness centrality interchangeably.

A naïve algorithm for betweenness centrality: In order to compute the dependencies in Eq. (3.1) we need to compute $\sigma(s, t)$ and $\sigma(s, t|v)$ for every triple s, t and v. Observe that v is contained in a shortest path between s and t if and only if $d(s, t) = d(s, v) + d(v, t)$. If this equality holds, then any shortest path from s to t can be written as the concatenation of a shortest path connecting s and v and a shortest path from v to t. Hence, $\sigma(s, t|v) = \sigma(s, v) \cdot \sigma(v, t)$. If $P_v = \{u \in V | (u, v) \in E, d(s, v) = d(s, u) + w(u, v)\}$ is the set of parent nodes of v, then it is easy to see that

$$\sigma(s, v) = \sum_{u \in P_v} \sigma(s, u). \tag{3.2}$$

We can compute $\sigma(s,v)$ for a given target s and all possible nodes v by running a weighted single source shortest paths algorithm (such as the `Dijkstra` algorithm) with source s. While the search tree in `Dijkstra` is built $\sigma(s,v)$ is computed by formula (3.1). The running time of `Dijkstra` is $O(m + n \log n)$ per source using a Fibonacci-heap implementation (the fastest known implementation of `Dijkstra`). Finally, a naïve computation of the dependencies can be done as

$$\delta(s,t|v) = \frac{\sigma(s,v) \cdot \sigma(t,v)}{\sigma(s,t)}.$$

Even given if all $\sigma(s,t)$ values are given, this computations requires time equal to the number of dependencies, i.e., $O(|S|^2 \cdot n)$.

The `Brandes` algorithm: Let $\delta(s|v)$ define the dependency of a node v on a single target s as the sum of the dependencies containing s, thus

$$\delta(s|v) = \sum_{t \in S} \delta(s,t|v). \tag{3.3}$$

The key observation of `Brandes` is that for a fixed target s we can compute $\delta(s|v)$ by traversing the shortest-paths tree found by `Dijkstra` in the reversed order of distance to s using the formula:

$$\delta(s|u) = \sum_{v: u \in P_v} \frac{\sigma(s,u)}{\sigma(s,v)} (I_{v \in S} + \delta(s|v)). \tag{3.4}$$

Where $I_{v \in S}$ is an indicator that is 1 if $v \in S$ and zero otherwise. This is used to make sure that we only sum dependencies between pairs of target nodes. Using this trick, the dependencies can be computed in time $O(|S|m)$, yielding a total running time of $O(|S|m + |S|n \log n)$ for `Brandes`.

4 The SKELETON Graph

In this section, we introduce the SKELETON of a graph G. The purpose of the SKELETON is to get a simplified representation of G that still contains all information on shortest paths between target nodes in S.

Let $G(V,E,W)$ be a weighted undirected graph with nodes V, edges E and edge weights $W : E \to [0,\infty)$. We also assume that we are given a partition $\mathcal{P}$ of the nodes V into k parts: $\mathcal{P} = \{P_1, \ldots, P_k\}$ such that $\cup_{i=1}^k P_i = V$ and $P_i \cap P_j = \emptyset$ for every $i \neq j$.

The SKELETON of G is defined to be a graph $G_{\text{sk}}^{\mathcal{P}}(V_{\text{sk}}, E_{\text{sk}}, W_{\text{sk}})$; its nodes V_{sk} are a subset of V. For every edge $e \in E_{\text{sk}}$ the function W_{sk} represents a pair of weights called the *characteristic tuple* associated with e. All of V_{sk}, E_{sk} and W_{sk} depend on the partition $\mathcal{P}$. Whenever it is clear from the context which partition is used we drop $\mathcal{P}$ from the notation and use G_{sk} instead of $G_{\text{sk}}^{\mathcal{P}}$. We now proceed to explain in detail how V_{sk}, E_{sk} and W_{sk} are defined.

Supernodes: Given $\mathcal{P}$, we define G_i to be the subgraph of G that is spanned by the nodes in $P_i \subseteq V$, that is $G_i = G[P_i]$. We denote the nodes and edges of G_i by V_i and E_i respectively. We refer to the subgraphs G_i as *supernodes*. Since $\mathcal{P}$ is a partition, all nodes in V belong to one of the supernodes G_i.

Nodes in the SKELETON (V_{sk}): Within every supernode $G_i(V_i, E_i)$ there are some nodes $F_i \subseteq V_i$ of special significance. These are the nodes that have at least one edge connecting them to a node of another supernode G_j. We call F_i the *frontier* of G_i. In Figure 1(a) the supernode G_i consists of nodes and edges inside the large circle. The frontier of G_i is $F_i = \{1,2,3\}$. Observe that nodes a, b and c are also frontier nodes in their respective supernodes. The *nodes* V_{sk} of the SKELETON consist of the union of all frontier nodes i.e., $V_{\text{sk}} = \cup_{i=1}^k F_i$.

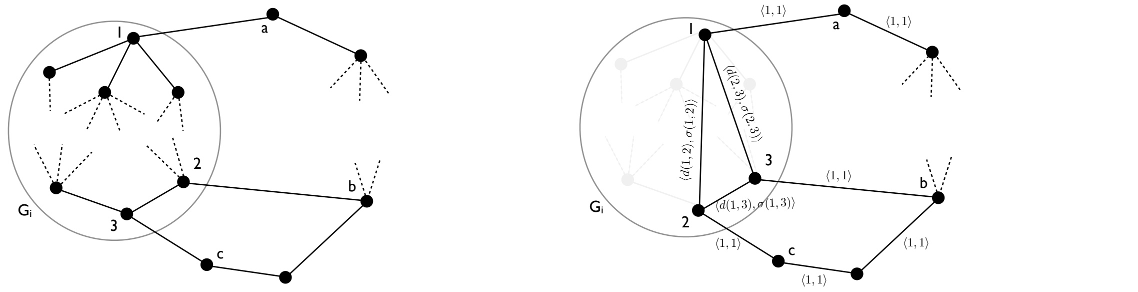
Edges in the SKELETON (E_{sk}): The edges in G_{sk} are defined with help of the frontiers in G. First, in order to see the significance of the frontier nodes, pick any two target nodes $s,t \in S$. Observe, that some of the shortest paths between s and t may pass through G_i. Any such path has to enter the supernode through one of the frontier nodes $f \in F_i$ and exit through another frontier $q \in F_i$. It is easy to check, whether there are any shortest paths through f and q; given $d(f,q)$, there is a shortest path between s and t passing through f and q if and only if

$$d(s,t) = d(s,f) + d(f,q) + d(q,t). \tag{4.5}$$

Also the number of paths passing through f and q is:

$$\sigma_G(s,t|f,q) = \sigma(s,f) \cdot \sigma(f,q) \cdot \sigma(q,t). \tag{4.6}$$

Recall that the nodes V_{sk} of the SKELETON are the union of all frontiers in the supernodes. The *edges* E_{sk} serve the purpose of representing the possible shortest paths between pairs of frontier nodes, and as a result, the paths between pairs of target nodes in G. The key observation to the definition of the SKELETON is, that we solely depend on the frontiers and do not need to list all possible (shortest) paths in G. We want to emphasize here that in order not to double count, we only consider the paths connecting f and q that do not contain any other frontier inside the path. Paths containing more than two frontiers will be considered as concatenations of shorter paths during computations on the entire SKELETON. The exact details will be clear once we define the edges and some weights assigned to the edges in the following paragraphs.



(a) original graph (b) SKELETON

Figure 1: Graph $G(V, E)$ (Figure 1(a)) is given as input to `Brandes++`. The nodes and edges inside the circle correspond to supernode G_i. The set of frontier nodes in G_i is $F_i = \{1, 2, 3\}$. Supernode G_i is replaced by a clique on nodes $\{1, 2, 3\}$ with characteristic tuple $\langle d_{jk}, \sigma_{jk}\rangle$ on edge (j, k) in the SKELETON (Figure 1(b)).

E_{sk} consists of two types of edges; first, the edges that connect frontiers in different supernodes (such as edges $(1, a)$, $(2, b)$ and $(3, c)$ in Figure 1(a)). We denote these edges by R. Observe that these edges are also in the original graph G, namely $R = E \setminus \{\cup_{i=1}^{k} E_i\}$. The second type are edges between all pairs of frontier nodes $f, q \in F_i$ within each supernode. To be exact, we add the edges X_i of the clique $C_i = (F_i, X_i)$ to the SKELETON. Hence, the edges of the SKELETON can be defined as the union of R and the cliques defined by the supernodes, i.e., $E_{\text{sk}} = R \cup \{\cup_{i=1}^{k} X_i\}$.

Characteristic tuples in the SKELETON (W_{sk}): We assign a *characteristic tuple* $W_{\text{sk}}(e) = \langle \delta(e), \sigma(e)\rangle$, consisting of a *weight* and a *multiplicity*, to every edge $e \in E_{\text{sk}}$. For edge $e(u, v)$ the weight represents the length of the shortest path between u and v in the original graph; the multiplicity encodes the number of different shortest paths between these two nodes. That is, if $e \in R$, then $W_{\text{sk}}(e) = \langle w(e), 1\rangle$, where $w(e)$ is the weight of e in G. If $e = (f, q)$ is in X_i for some i, then $f, q \in F_i$ are frontiers in G_i. In this case $W_{\text{sk}}(e) = \langle d(f, q), \sigma(f, q)\rangle$. The values $d(f, q)$ and $\sigma(u, v)$ are used in Equations (4.5) and (4.6). While these equations allow to compute the distance $d(s, t)$ and multiplicity $\sigma(s, t|f, q)$ between target nodes s and t, both values are independent of the target nodes themselves. In fact, $d(f, q)$ and $\sigma(f, q)$ only depend and are characteristic of their supernode G_i.

We compute $d(f, q)$ and $\sigma(f, q)$ by applying `Dijkstra` – as described in section 3 – in G_i using the set of frontiers F_i as sources. We want to emphasize here, that the characteristic tuple only represent the shortest paths between f and q that are entirely within the supernode G_i and do not contain any other frontier node in F_i. This precaution is needed to avoid double counting paths between f and q that leave G_i and then come back later. To ensure this, we apply a very simple modification to the `Dijkstra` algorithm; in equation (3.2) we only sum over the set of parents P_v^- of a node that are *not* frontiers themselves, thus

$$(4.7) \quad P_v^- = \{u \in V_i \setminus F_i \mid (u, v) \in E_i, d(s, v) = d(s, u) + w(u, v)\}.$$

We refer to this modified version of `Dijkstra`'s algorithm that is run on the supernodes as `Dijkstra_SK`. For recursion (4.6) we also set $\sigma(f, f) = 1$.

The SKELETON: Combining all the above, the SKELETON of a graph G is defined by the supernodes generated by the partition $\mathcal{P}$ and can be described formally as

$$G_{\text{sk}}^{\mathcal{P}} = (V_{\text{sk}}, E_{\text{sk}}, W_{\text{sk}}) = (\cup_{i=1}^{k} F_i, R \cup \{\cup_{i=1}^{k} X_i\}, W_{\text{sk}})$$

Figure 1(b) shows the SKELETON of the graph from Figure 1(a). The nodes in G_{sk} are the frontiers of G and the edges are the dark edges in this picture. Edges in R are for example $(1, a)$, $(2, b)$ and $(3, c)$ while edges in X_i are $(1, 2)$, $(1, 3)$ and $(2, 3)$.

Properties of the SKELETON: We conclude this section by comparing the number of nodes and edges of the input graph $G = (V, E, W)$ and its skeleton $G_{\text{sk}} = (V_{\text{sk}}, E_{\text{sk}}, W_{\text{sk}})$. This comparison will facilitate the computation of the running time of the different algorithms in the next section.

Note that G_{sk} has less nodes than G: the latter has $|V|$ nodes, while the former has only $|V_{\text{sk}}| = \sum_{i=1}^{k} |F_i|$. Since not all nodes in G_{sk} are frontier nodes, then $|V| \le |V_{\text{sk}}|$. For the edges, the original graph has $|E|$ edges, while its skeleton has $|E_{\text{sk}}| = |E| - \sum_{i=1}^{k} |E_i| + \sum_{i=1}^{k} \binom{|F_i|}{k}$. The relative size of $|E|$ and $|E_{\text{sk}}|$ depends on the partition $\mathcal{P}$ and the number of frontier nodes and edges between them it generates.

5 The `Brandes++` Algorithm

In this section we describe `Brandes++`, which leverages the speedup that can be gained by using the SKELETON of a graph. At a high level `Brandes++` consists of three main steps, first the SKELETON is created, then a multipiclity-weighted version of `Brandes`'s algorithm is run on the SKELETON. In the final step the centrality of all other nodes in G is computed.

The pseudocode of `Brandes++` is given in Alg. 1. The input to this algorithm is the weighted undirected graph $G = (V, E, W)$, the set of targets S and partition $\mathcal{P}$. The algorithm outputs the exact values of betweenness centrality for every node in V. Next we explain the details of each step.

Algorithm 1 `Brandes++` to compute the exact betweenness centrality of all nodes.

```
   Input: graph $G(V, E, W)$, targets $S$, partition $\mathcal{P} = \{P_1, \ldots, P_k\}$.
1: $G_{\text{sk}}(V_{\text{sk}}, E_{\text{sk}}, \langle ., . \rangle) = $ Build_SK($G, \mathcal{P}$)
2: $\{C(G_1), \ldots, C(G_k)\} = $ Brandes_SK($G_{\text{sk}}$)
3: $\{C(v) | v \in V\} = $ Centrality($\{C(G_1), \ldots, C(G_k)\}$)
   return: $C(v)$ for every $v \in V$
```

Step 1: The `Build_SK` algorithm: `Build_SK` (Alg. 2) takes as input G and the partition $\mathcal{P}$ and outputs the SKELETON $G_{\text{sk}}(V_{\text{sk}}, E_{\text{sk}}, W_{\text{sk}})$. First it decides the set of frontiers F_i in the supernodes (line 1). Then the characteristic tuples W_{sk} are computed in every supernode by way of `Dijkstra_SK` (line 3). Characteristic tuples on edges $e \in R$ are $\langle 1, 1 \rangle$ by definition.

Algorithm 2 `Build_SK` algorithm to create the SKELETON of G.

```
   Input: graph $G(V, E, W)$, targets $S$, partition $\mathcal{P}$.
1: Find frontiers $\{F_1, F_2, \ldots, F_k\}$
2: for $i = 1$ to $k$ do
3:     $\{\langle d(f, q), \sigma(f, q) \rangle \mid$ for all $f, q, \in F_i\} = $ Dijkstra_SK($F_i$)
4: end for
   return: SKELETON $G_{\text{sk}}(V_{\text{sk}}, E_{\text{sk}}, W_{\text{sk}})$
```

Running time: The frontier sets F_i of each supernode can be found in $O(|E|)$ time as it requires to check for every node whether they have a neighbor in another supernode. `Dijkstra_SK` has running time identical to the traditional `Dijkstra` algorithm, that is $O(|F_i|(|E_i| + |V_i| \log |V_i|))$.

Target nodes in the SKELETON*:* Note that since we need to know the shortest paths for every target node $s \in S$, we treat the nodes in S specially. More specifically, given the input partition $\mathcal{P}$, we remove all targets from their respective parts and add them as singletons. Thus, we use the partition $\mathcal{P}' = \{P_1 \setminus S, P_2 \setminus S, \ldots, P_k \setminus S, \cup_{s \in S}\{s\}\}$. Assuming that the number of target nodes is relatively small compared to the total number of nodes in the network, this does not have a significant effect on the running time of our algorithm.

Observe that the characteristic tuples of different supernodes are independent of each other allowing for a parallel execution of `Dijkstra_SK`.

Step 2: The `Brandes_SK` algorithm: The output of `Brandes_SK` are the exact betweenness centrality values for all nodes in G_{sk}, that is all frontiers in G.

Remember from Section 3 that for every target node $s \in S$ `Brandes` consists of two main steps; (1) running a single-source shortest paths algorithm from s to compute the distances $d(s, v)$ and number of shortest paths $\sigma(s, v)$. (2) traversing the BFS tree of `Dijkstra` in reverse order of discovery to compute the dependencies $\delta(s|v)$ based on Equation (3.4). The only difference between `Brandes_SK` and `Brandes` is that we take the distances and multiplicities on the edges of the SKELETON into consideration. This means that Equation (5.8) is used instead of (3.2).

$$\sigma(s, v) = \sum_{u \in P_v^{sk}} \sigma(s, u)\sigma(u, v). \tag{5.8}$$

Here $P_v^{sk} = \{u \in V_{\text{sk}} | (u, v) \in E_{\text{sk}}, d(s, v) = d(s, u) + d(u, v)\}$ is the set of parent nodes of v in G_{sk}. Observe that $\sigma(s, v)$ in Equation (5.8) is the multiplicity of shortest paths between s and v both in G_{sk} as well as in G. That is why we do not use subscripts (such as $\sigma_{sk}(s, v)$) in the above formula.

In the second step the dependencies of nodes in G_{sk} are computed by applying Equation (5.9) – which is the counterpart of Eq. (3.4) that takes multiplicities into account – to the reverse order traversal of the BFS tree.

$$\delta(s|u) = \sum_{v : u \in P_v} m(u, v) \cdot \frac{\sigma(s, u)}{\sigma(s, v)} (I_{v \in S} + \delta(s|v)) \tag{5.9}$$

Running time: `Brandes` and `Brandes_SK` have the same computational complexity but are applied to different graphs (G and G_{sk} respectively). Hence we get that `Brandes_SK` on the skeleton runs in $O(|S|E_{\text{sk}} + |S|V_{\text{sk}} \log V_{\text{sk}})$ time. If we express the same running time in terms of the frontier nodes we get

$$O\left(|S|(|R| + \sum_{i=1}^{k} \binom{|F_i|}{2}) + |S| \left(\sum_{i=1}^{k} |F_i|\right) \log \left(\sum_{i=1}^{k} |F_i|\right)\right).$$

Step 3: The `Centrality` algorithm: In the last step of `Brandes++`, the centrality values of all remaining nodes $v \in V_i \setminus F_i$ in G are computed. Let us focus on supernode G_i; for any node $v \in V_i \setminus F_i$ and $s \in S$ there exists a frontier $f \in F_i$ such that there exists a shortest path from s to f containing v. Using Equation (5.9), we can compute the dependency $\delta(s|v)$ as follows:

$$\delta(s|v) = \sum_{f \in F_i} \frac{\sigma(s,v)}{\sigma(s,f)} \sigma(v,f) \left(I_{v \in S} + \delta(s|f)\right). \tag{5.10}$$

Then, the centrality of v is $C(v) = \sum_{s \in S} \delta(s|v)$.

To determine whether v is contained in a path from s to f we need to remember the information $d(f,v)$ for v and every frontier $f \in F_i$. This value is actually computed during the Build_SK phase of Brandes++. Hence with additional use of space but without increasing the running time of the algorithm we can make use of it. At the same time with $d(f,v)$ the multiplicity $\sigma(f,v)$ is also computed.

Space complexity: The Centrality algorithm takes two values – $d(f,v)$ and $\sigma(f,v)$ – for every pair $v \in V_i \setminus F_i$ and $f \in F_i$. This results in storing a total of $\sum_{i=1}^{k} (|F_i||V_i \setminus F_i|)$ values for the SKELETON.

Running time: Since we do not need to allocate any additional time for computing $d(f,v)$ and $\sigma(f,v)$ computing Equation (5.10) takes $O(|F_i|)$ time for every $v \in V_i \setminus F_i$. Hence, summing over all supernodes we get that the running time of Centrality is $O\left(\sum_{i=1}^{k} |F_i||V_i \setminus F_i|\right)$.

Running time of Brandes++: The total time that Brandes++ takes is the combination of time required for steps 1,2 and 3. The asymptotic running time is a function of the number of nodes and edges in each supernode, the number of frontier nodes per supernode and the size of the SKELETON. To give some intuition, assume that all supernodes have approximately $\frac{n}{k}$ nodes with at most half of the nodes being frontiers in each supernode. Further, assume that $R \leq \frac{m}{2}$. Substituting these values into steps 1–3, we get that for a partition of size k Brandes++ is order of k-times faster than Brandes. While these assumptions are not necessarily true, they give some insight on how Brandes++ works. For $k = 1$ (thus when there is no partition) the running times of Brandes++ and Brandes are identical while for larger values of k the computational speedup is much more significant.

6 Experiments

Experimental setup: For all our experiments we follow the same methodology; given the partition $\mathcal{P}$, we run Steps 1–3 of Brandes++ (Alg. 1) using $\mathcal{P}$ as input. Then, we report the running time of Brandes++ using this partition. The local computations on the supernodes G_i (lines 1 and 3 of Alg. 2) can be done in parallel across the G_i's. Hence, the running time we report is the sum of: (i) the running time of Build_SK on the largest supernode G_i (*ii*) running Brandes_SK on the SKELETON and (*iii*) computing the centrality of *all* remaining nodes in G.

Implementation: In all our experiments we compare the running times of Brandes++ to Brandes [4] on weighted undirected graphs. While there are several high-quality implementations of Brandes available, we use here our own implementation of Brandes and, of course, Brandes++. All the results reported here correspond to our Python implementations of both algorithms. The reason for this is, that we want to ensure a fair comparison between the algorithms, where the algorithmic aspects of the running times are compared as opposed to differences due to more efficient memory handling, properties of the used language, etc. As the Brandes_SK algorithm run on the SKELETON is almost identical to Brandes (see Section 5), in our implementation we use the exact same codes for Brandes as Brandes_SK, except for appropriate changes that take into account edge multiplicities. We also make our code available[1].

Hardware: All experiments were conducted on a machine with Intel X5650 2.67GHz CPU and 12GB of memory.

Datasets: We use the following datasets:

WikiVote *dataset [16]:* The nodes in this graph correspond to users and the edges to users' votes in the election to being promoted to certain levels of Wikipedia adminship. We use the graph as undirected, assuming that edges simply refer to the user's knowing each other. The resulting graph has 7066 nodes and 103K edges.

AS *dataset: [12]* The AS graph corresponds to a communication network of who-talks-to whom from BGP logs. We used the directed Cyclops AS graph from Dec. 2010 [12]. The nodes represent Autonomous Systems (AS), while the edges represent the existence of communication relationship between two ASes and, as before, we assume the connections being undirected. The graph contains $37K$ nodes and 132K edges, and has a power law degree distribution.

EU *dataset [17]:* This graph represents email data from a European research institute. Nodes of the graph correspond to the senders and recipients of emails, and the edges to the emails themselves. Two nodes in the graph are connected with an undirected edge if they have ever exchanged an email. The graph has 265K nodes and 365K edges.

DBLP *dataset [28]:* The DBLP graph contains the co-authorship network in the computer science community. Nodes correspond to authors and edges capture co-authorships. There are 317K nodes and 1M edges.

[1]available at: http://cs-people.bu.edu/edori/code.html

For all our real datasets, we pick 200 nodes (uniformly at random) to form the target set S.

Graph-partitioning: The speedup ratio of `Brandes++` over `Brandes` is determined by the structure of the SKELETON(G_{sk}) that is induced by the input graph partition.

In practice, graphs that benefit most of `Brandes++` are those that have small k-cuts, such as those that have distinct community structure. On the other hand, graphs with large cuts, such as power-law graphs do not benefit that much from applying the partitioning of `Brandes++`.

For our experiments we partition the input graph into subgraphs using well-established graph-partitioning algorithms, which aim to either find densely-connected subgraphs or sparse cuts. Algorithms with the former objective fall under *modularity clustering* [7, 19, 22, 24] while the latter are *normalized cut* algorithms [1, 8, 13, 14, 15, 20, 26]. We choose the following three popular algorithms from these groups: `Mod`, `Gc` and `Metis`.

`Mod`*:* `Mod` is a hierarchical agglomerative algorithm that uses the modularity optimization function as a criterion for forming clusters. Due to the nature of the objective function, the algorithm decides the number of output clusters automatically and the number of clusters need not be provided as part of the input. `Mod` is described in Clauset *et al* [7] and its implementation is available at: `http://cs.unm.edu/~aaron/research/fastmodularity.htm`

`Gc`*:* `Gc` (`graclus`) is a normalized-cut partitioning algorithm that was first introduced by Dhillon *et al.* [8]. An implementation of `Gc`, that uses a kernel k-means heuristic for producing a partition, is available at: `cs.utexas.edu/users/dml/Software/graclus.html`. `Gc` takes as input k, an upper bound on the number of clusters of the output partition. For the rest of the discussion we will use `Gc`-k to denote the `Gc` clustering into at most k clusters.

`Metis`*:* This algorithm [15] is perhaps the most widely used normalized-cut partitioning algorithm. It does hierarchical graph bi-section with the objective to find a balanced partition that minimizes the total edge cut between the different parts of the partition. An implementation of the algorithm is available at: `glaros.dtc.umn.edu/gkhome/views/metis`. Similar to `Gc`, `Metis` takes an upper bound on the number of clusters k as part of the input. Again, we use the notation `Metis`-k to denote the `Metis` clustering into at most k clusters.

We report the running times of the three clustering algorithms in Table 1. Note that for both `Gc` and `Metis` their running times depend on the input number of clusters k – the larger the value of k the larger the running time. The table summarizes the largest running time for each dataset (see Table 2 for the value of k for each dataset). Note that the running times of the clustering algorithms cannot be compared to the running time of `Brandes++` for two reasons; these algorithms are implemented using a different (and more efficient) programming language than Python and are highly optimized for speed, while our implementation of `Brandes++` is not. We report Table 1 to compare the various clustering heuristics against each other.

Table 1: Running time (in seconds) of the clustering algorithms (reported for the largest number of clusters per algorithm) and of `Brandes` (last column).

	`Mod`	`Gc`	`Metis`	`Brandes`
`WikiVote`	13	1.29	1.9	5647
`AS`	6780	256.58	9.5	606
`EU`	1740	2088	12.2	14325
`DBLP`	3600	109.22	5.5	28057

Results: The properties of the partitions produced for our datasets by the different clustering algorithms, as well as the corresponding running times of `Brandes++` for each partition are shown in Table 2. In case of `Gc` and `Metis` we experimented with several (about 10) values of k. We report for three different values of k (one small, one medium and one large) for each dataset. The values were chosen in such a way, that the k-clustering with the best results in `Brandes++` is among those reported. As a reference point, we report in Table 1 the running times of the original `Brandes` algorithm on our datasets.

In Table 2 N and M refer to the number of nodes and edges in the SKELETON. Remember, that the set of nodes in the skeleton is the union of the frontier nodes in each supernode. Hence, N is equal to the total number of frontier nodes induced by $\mathcal{P}$. Across datasets we can see quite similar values, depending on the number of clusters used. `Mod` seems to yield the lowest values of N and M. The third and fourth rows in the table contain the number of clusters k (the size of the partition) for each algorithm and the total number of nodes (from the input graph) in the largest cluster of each partition.

The ultimate measure of performance is the running time of `Brandes++` in the last row of the table. We compare the running times of `Brandes++` to the running time of `Brandes` in Table 1 – last column. On the `WikiVote` data `Brandes` needs 5647 seconds while the corresponding time for `Brandes++` can be as small as 72 seconds! Note that the best running time for this dataset is achieved using the `Metis`-100 partition. Suggesting that the underlying "true" structure of the dataset consists of approximately 100 communities of

Table 2: Properties of the partitions (N: number of frontier nodes in SKELETON, M: number of edges in SKELETON, k: number of supernodes, LCS: number of original nodes in the largest cluster) produced by different clustering algorithms and running time of `Brandes++` on the different datasets.

`WikiVote` dataset							
	`Mod`	`Gc`-100	`Gc`-1K	`Gc`-2K	`Metis`-100	`Metis`-1K	`Metis`-2K
N	3833	5860	6365	6432	4920	5854	6181
M	26147	89318	96111	96743	91545	92091	97270
k	29	100	1000	2000	98	989	1927
LCS	3059	172	12	10	258	496	50
`Brandes++` running time in seconds							
	209.43	75.27	90.23	96.65	71.59	73	77.71

`AS` dataset							
	`Mod`	`Gc`-1K	`Gc`-10K	`Gc`-15K	`Metis`-1K	`Metis`-10K	`Metis`-15K
N	14104	31139	34008	34418	25022	32572	35244
M	28815	111584	120626	121833	93989	117808	125556
k	156	1000	10000	15000	991	9966	14484
LCS	8910	732	10	10	1304	433	18
`Brandes++` running time in seconds							
	1666	430.97	447.97	486.91	417	420.93	458.71

`EU` dataset							
	`Mod`	`Gc`-1K	`Gc`-3K	`Gc`-5K	`Metis`-1K	`Metis`-3K	`Metis`-5K
N	19332	143636	208416	215397	42333	54249	50171
M	45296	231089	208416	215397	117147	132573	129071
k	45296	231089	319006	327263	995	2996	4996
LCS	53224	7634	7633	7633	8917	7407	7271
`Brandes++` running time in seconds							
	188.79	5670	7816.7	8291.3	3601	2101	1872

`DBLP` dataset							
	`Mod`	`Gc`-100	`Gc`-1K	`Gc`-5K	`Metis`-100	`Metis`-1K	`Metis`-5K
N	102349	98281	130643	141955	104809	119417	132661
M	146584	164989	267350	310472	203834	257383	318969
k	3203	100	1000	5000	100	1000	4999
LCS	55897	116252	26666	21368	3270	344	93
`Brandes++` running time in seconds							
	3600	95982	16405	10335	13574	5805	5709

Wikipedia users. High speedup ratios are also achieved on EU and DBLP. For those Brandes takes 14325 and 28057 seconds respectively, while the running time of Brandes++ can be 189 and 3600 seconds respectively. This is an 8-fold speedup on DBLP and 75-fold on EU.

If we compare the running time of Brandes++ applied to the different partitions, we see that the algorithm with input by Metis is consistently faster than the same-sized partitions of Gc. Further, on EU and DBLP Brandes++ is the fastest with the Mod partition. Note the size of G_{sk} for each of these datasets. In case of EU Mod yields a skeleton where N is only 7% of the original number of nodes and M is 12% of the edges. The corresponding rations on DBLP are 32% and 14%. This is not surprising, as both datasets are known for their distinctive community structure, which is what Mod optimizes for. For AS, Brandes++ exhibits again smaller running time than Brandes, yet the improvement is not as impressive. Our conjecture is that this dataset does not have an inherent clustering structure and therefore Brandes++ cannot benefit from the partitioning of the data.

Note that the running times we report here refer only to the execution time of Brandes++ and do not include the actual time required for doing the clustering – the running times for clustering are reported in in Table 1. However, since the preprocessing has to be done only once and the space increase is only a constant factor, Brandes++ is clearly of huge benefit.

References

[1] R. Andersen. A local algorithm for finding dense subgraphs. *ACM Transactions on Algorithms*, 2010.

[2] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *WAW*, 2007.

[3] D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *ICPP*, 2006.

[4] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 2001.

[5] U. Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 2008.

[6] U. Brandes and C. Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 2007.

[7] A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E*, 2004.

[8] I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors: A multilevel approach. *IEEE Trans. Pattern Anal. Mach. Intell*, 2007.

[9] N. Edmonds, T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *HiPC*, 2010.

[10] L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 1977.

[11] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, 2008.

[12] P. Gill, M. Schapira, and S. Goldberg. Let the market drive deployment: a strategy for transitioning to bgp security. In *SIGCOMM*, 2011.

[13] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Supercomputing, 1995.

[14] R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 2004.

[15] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 1998.

[16] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting positive and negative links in online social networks. In *WWW*, 2010.

[17] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 2007.

[18] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarría-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDPS*, 2009.

[19] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review*, E 69, 2004.

[20] A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS*, 2001.

[21] R. Puzis, P. Zilberman, Y. Elovici, S. Dolev, and U. Brandes. Heuristics for speeding up betweenness centrality computation. In *SocialCom/PASSAT*, 2012.

[22] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences*, 2004.

[23] M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. WSDM '14, pages 413–422, 2014.

[24] R. Rotta and A. Noack. Multilevel local search algorithms for modularity clustering. *J. Exp. Algorithmics*, 2011.

[25] A. E. Sariyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Shattering and compressing networks for betweenness centrality. In *SDM*, 2013.

[26] S. E. Schaeffer. Survey: Graph clustering. *Comput. Sci. Rev.*, 2007.

[27] G. Tan, D. Tu, and N. Sun. A parallel algorithm for computing betweenness centrality. In *ICPP*, 2009.

[28] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *ICDM*, 2012.