A Divide-and-Conquer Algorithm for Betweenness Centrality

Dóra Erdös[1], Vatche Ishakian[2], Azer Bestavros[1], and Evimaria Terzi[1]

[1] Boston University, Boston, MA `[edori,best,evimaria]@cs.bu.edu`
[2] Raytheon BBN Technologies, Cambridge, MA `visahak@cs.bu.edu`

Abstract. The problem of efficiently computing the betweenness centrality of nodes has been researched extensively. To date, the best known exact and centralized algorithm for this task is an algorithm proposed in 2001 by Brandes [7]. The contribution of our paper is `Brandes++`, an algorithm for exact efficient computation of betweenness centrality. The crux of our algorithm is that we create a sketch of the graph, that we call the SKELETON, by replacing subgraphs with simpler graph structures. Depending on the underlying graph structure, using this SKELETON and by keeping appropriate summaries `Brandes++` we can achieve significantly low running times in our computations. Extensive experimental evaluation on real life datasets demonstrate the efficacy of our algorithm for different types of graphs. We release our code for benefit of the research community.

1 Introduction

The network-analysis literature is rich in measures defined to quantify the *centrality* of nodes in a network. The key assumption in that line of work is that central nodes are also "important" nodes of a network. Hence, identifying them can give valuable insights about the structure and the function of the underlying network. From the 1940s to today, there have been proposed a lot of centrality measures [2,5,6,8,14,17,19,20,32], which aim to assist in analyzing network data from (online) social and media networks, the Internet, the Web, transportation networks and many more.

The most widely accepted measure of node centrality is *betweenness centrality*. Betweenness centrality, was first proposed in the 1950s by Shimbel [32] who suggested that the centrality of a node can be quantified by the number of shortest paths that go through it. The formal definition of betweenness centrality can be attributed to Freeman [14] who, in the 1970s, defined the betweenness centrality of a node v as the fraction of all pairwise shortest paths that go through v. In many applications certain pairs of nodes are more important than others and instead of considering all pairwise shortest paths, we only focus on the pairwise shortest paths between a subset of nodes, called *target nodes*.

From the computational point of view, the betweenness centrality of all nodes of a network can be computed for all nodes in time $O(|S|n^2)$, where S is the set of target nodes and n the number of nodes of the network. For many years, this algorithm was the best-known algorithm, until 2001, when Brandes [7] proposed a faster algorithm for computing the betweenness centrality of all nodes. The novelty of his approach lies in a new method to summarize the shortest-paths information obtained after computing all-pairs shortest paths in a graph. The resulting algorithm, which we call `Brandes`, runs in time $O(|S|m + |S|n \cdot \log(n))$, where m is the number of edges of the network. Although the `Brandes` algorithm has the same worst-case time complexity as the previously-known algorithm, it can leverage the sparsity of real datasets and thus enable centrality computations in large real graphs. To the best of our knowledge, `Brandes` is the best centralized algorithm to date for computing the betweenness centrality of nodes *exactly*.

In this paper, we develop a new algorithm, which we call `Brandes++`, and which can compute the betweenness centralities of nodes efficiently in a "divide-and-conquer" fashion: the algorithm first assumes a partition of the graph into smaller subgraphs – called *supernodes*. These supernodes are connected via multiple edges forming what we call the SKELETON graph. `Brandes++` first applies Dijkstra's algorithm in each supernode. The results of the computations are then combined via an application of the `Brandes` algorithm on the SKELETON graph in order to compute the final centralities of the nodes. `Brandes++` represents supernodes with (weighted) cliques allowing for an exact computation of the centrality values of all the nodes of the original graph.

Clearly, the running time of `Brandes++` depends on the partition of the nodes into supernodes, and the resulting structure of the SKELETON graph. Different partitions have different effect on the running time of our algorithm. In our experimental evaluation we investigate this relationship by deploying different graph-partitioning algorithms. Our results indicate that there are datasets for which `Brandes++` – when combined with an off-the-shelf graph-partition methods – computes the centralities of all nodes 5–80 times faster than `Brandes`.

In principle, our algorithm operates independently on each subgraph, enabling parallelization of the computations. Thus, `Brandes++` can be used for very large datasets – even for datasets that cannot fit in the main memory.

2 Overview of Related Work

From the formal definition of betweenness centrality by Freeman [14] almost thirty years had to pass for the first algorithm to improve on the running time $O(|S|n^2)$ of the naive implementation. This algorithm is due to Ulrik Brandes [7], who also studied extension of his algorithm to groups of nodes in Brandes *et al.* [8]. The `Brandes` algorithm has motivated a lot of subsequent work that led to parallel versions of the algorithm [4,25,33,13] as well as algorithms that *approximate* the betweenness centrality of nodes [3,9,15].

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.* [28] and Catalyürek *et al.* [10] 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”. This can be done, since structurally-equivalent nodes have identical betweenness centrality scores (with respect to nodes outside their 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. Then, a modified version of `Brandes`, that takes multiplicities into consideration, is used to compute global betweenness scores with help of the local ones. Catalyürek *et al.*[10] rely on these two heuristics and additional simplifying 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. For example, when having as input a 2-connected graph that consists of densely connected components, the above methods will not partition the graph, and thus provide no significant running-time improvement to `Brandes`. On the other hand, our algorithm will exploit the underlying graph structure towards computational savings. 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 [28] and [10]. 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 [10] 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 node set V of cardinality n, set of edges E of cardinality m and non-negative edge weights W. Let $S \subseteq V$ be a subset of nodes in V that we call *target* nodes; we assume that $2 \leq |S| \leq n$. The *distance* between two nodes 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 between s and t that contain node v. We say that v *covers* these shortest paths. Observe that σ is a symmetric function, thus $\sigma(s, t) = \sigma(t, s)$.

We say that the *dependency* of s and t on v is the fraction of shortest paths between 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 a node v can be defined as the sum of its dependencies.

$$C(v) = \sum_{s \neq t \in S} \delta(s,t|v). \tag{1}$$

Note that throughout the paper we use the terms betweenness, centrality and betweenness centrality interchangeably.

A simple algorithm for betweenness centrality: In order to compute the dependencies that appear in the above equation, we need to compute $\sigma(s,t)$ and $\sigma(s,t|v)$ for every pair of target $s,t \in S$ and node $v \in V$. First, 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 containing v is a concatenation of a shortest path between 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

$$\sigma(s,v) = \sum_{u \in P_v} \sigma(s,u). \tag{2}$$

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 `Dijkstra`'s algorithm) starting from s. During the execution of `Dijkstra`'s algorithm we keep track of the set of parents P_v of every node v. Once the algorithm discovers the true distance $d(s,v)$ we can compute $\sigma(s,v)$ with help of Equation (2). The running time of `Dijkstra` is $O(m+n\log n)$ per source using a Fibonacci-heap implementation (the fastest known implementation of `Dijkstra`).

After running `Dijkstra`'s algorithm from every node in S, we can compute the dependencies as

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

Since the number of dependencies is $O(|S|^2 \cdot n) = O(n^3)$, even given all $\sigma(s,t)$ values, the naive computation of the node centralities is cubic in the number of nodes.

The `Brandes` algorithm: One of the greatest advancements in computing betweenness centrality is the algorithm developed by Brandes [7] in 2001 that improves on this second part of the above 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

$$\delta(s|v) = \sum_{t \in S} \delta(s,t|v). \tag{3}$$

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

$$\delta(s|u) = \sum_{v:u\in P_v} \frac{\sigma(s,u)}{\sigma(s,v)}(I_{v\in S} + \delta(s|v)). \tag{4}$$

Where $I_{v\in S}$ is an indicator that is 1 if v is a target node 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 $O(|S|m)$. Since most real life graphs where the betweenness centrality plays a role are sparse (i.e., the number of edges m is small) this observation leads to considerable improvements in the running time in practice.

Combining the two steps (computing the single source shortest paths for every target s first and then computing the dependencies via Equation (3)) results in the `Brandes` algorithm with a total ruining time of $O(|S|m+|S|n\log n)$.

4 The SKELETON Graph

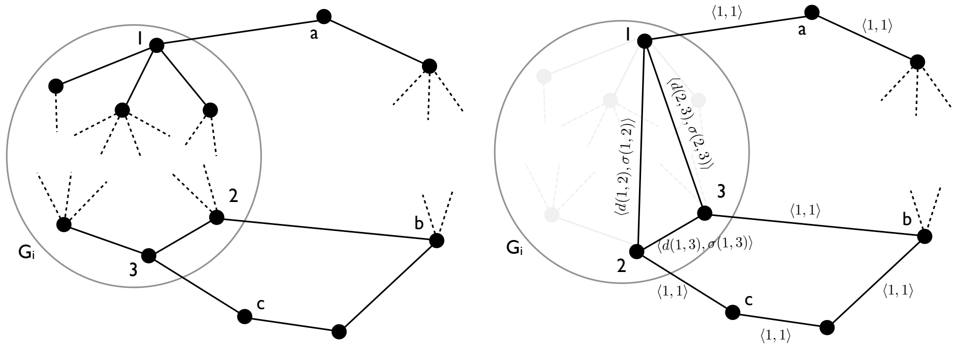
The key to computing betweenness centrality efficiently is to improve on the counting and handling of shortest paths in the graph. In this section we introduce the SKELETON of a graph G. The purpose of the SKELETON is to get a simplified representation of G that still contains all information on shortest 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 $i \neq j$. We note that $\mathcal{P}$ can be any partition of the vertices in G, in Section 6 we discuss what the best partition for `Brandes++` is on certain graphs.

The SKELETON of G is defined to be a graph $G_{\text{sk}}^{\mathcal{P}}(V_{\text{sk}}, E_{\text{sk}}, W_{\text{sk}})$. The nodes V_{sk} are a subset of V. Both the set of nodes V_{sk} and edges E_{sk} in the SKELETON 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 $G_{\text{sk}}^{\mathcal{P}}$. For every edge $e \in E_{\text{sk}}$ the function W_{sk} represents a pair of weights called the *characteristic tuple* associated with e. 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



(a) original graph (b) SKELETON

Fig. 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)).

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 in G, that is $V_{\text{sk}} = \cup_{i=1}^{k} F_i$.

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

$$d(s,t) = d(s,f) + d(f,q) + d(q,t). \tag{5}$$

It is also easy to compute the number of paths passing through f and q:

$$\sigma_G(s,t|f,q) = \sigma(s,f) \cdot \sigma(f,q) \cdot \sigma(q,t). \tag{6}$$

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. The exact details will be clear once we define the edges and some weights assigned to the edges in the following paragraphs.

E_{sk} consists of two types of edges; the first type are 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. Remember that a clique is a complete graph defined on a set of nodes. Thus, in this case C_i is the complete graph on F_i. 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 (5) and (6). Observe, that while the equations serve 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, hence the name.

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 (2) we only sum over the set of parents P_v^- of a node that are *not* frontiers themselves, thus

$$P_v^- = \{u \in V_i \setminus F_i \mid (u, v) \in E_i, d(s, v) = d(s, u) + w(u, v)\}. \qquad (7)$$

We refer to this modified version of `Dijkstra`'s algorithm that is run on the supernodes as `Dijkstra_SK`.

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

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

A schematic overview of the nodes and edges in the SKELETON is given in Figure 1(b). This Figure 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.

In terms of the number of nodes, G_{sk} has in fact 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 $|V| \leq |V_{\text{sk}}|$. In terms of the number of 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

We propose `Brandes++` for computing betweenness centrality in an efficient way. The algorithm 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. This step already results in the exact betweenness centrality values for all frontier nodes. In the final step the centrality of all other nodes in G is computed.

The pseudocode of `Brandes++` is given in Algorithm 1. The input 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.

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: $G_{\text{sk}}(V_{\text{sk}}, E_{\text{sk}}, \langle .,. \rangle) = \texttt{Build\_SK}(G, \mathcal{P})$
2: $\{C(G_1), \ldots, C(G_k)\} = \texttt{Brandes\_SK}(G_{\text{sk}})$
3: $\{C(v) | v \in V\} = \texttt{Centrality}(\{C(G_1), \ldots, C(G_k)\})$
   return: $C(v)$ for every $v \in V$
```

We now proceed to discuss each step of `Brandes++` in detail.

Step 1: the `Build_SK` algorithm The first step of computing betweenness centrality in G is to build its SKELETON G_{sk}. The algorithm `Build_SK` (Algorithm 2) takes as input G and the partition $\mathcal{P}$. As a first step it decides the set of frontiers F_i in every supernode (line 1). Then `Dijkstra_SK` is run in every supernode G_i with sources F_i to compute the characteristic tuples assigned to the edges of the clique C_i (line 3). Finally, it outputs the graph $G_{\text{sk}}(V_{\text{sk}}, E_{\text{sk}}, W_{\text{sk}})$ with the appropriate characteristic tuples assigned to each edge. The pseudocode of `Build_SK` is given in Algorithm 2.

Target nodes in the SKELETON*:* As we have mentioned earlier in section 4, we need to be careful with the handling of target nodes S in the SKELETON. Specifically, there is no difficulty if a target $s \in S$ is a frontier node, and hence also a node in G_{sk}. However, `Brandes++` is not equipped to take shortest paths starting from s into consideration if $s \in V_i \setminus F_i$ for some supernode G_i.

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:     $\{\langle d(f, q), \sigma(f, q)\rangle \mid \text{for all } f, q, \in F_i\} =$ Dijkstra_SK($F_i$)
4: end for
   return: SKELETON $G_{\text{sk}}(V_{\text{sk}}, E_{\text{sk}}, W_{\text{sk}})$
```

To avoid this problem, we modify the given input partition $\mathcal{P}$, by removing all targets from their respective parts and introducing them as a singleton part each. Thus, we use the partition $\mathcal{P}' = \{P_1 \setminus S, P_2 \setminus S, \ldots, P_k \setminus S, \cup_{s \in S}\{s\}\}$.

How does this affect the running time of our algorithm? Worst case, every node in G is a target, thus $V = S$. In this case the above operation would result in a SKELETON G_{sk} that is identical to G, and hence no speedup of the traditional `Brandes` algorithm is achieved. However, we believe that in many real life applications the true number of targets is much smaller – in our experiments we assume in some cases that it is about 1% of the original graph. In this case `Brandes++` will yield a significant improvement over `Brandes` as we show in Section 6.

Running time: The frontier set of a supernode G_i can be found by checking the neighbors for each node $v \in V_i$; if one of its neighbors is not in P_i, then v is a frontier and hence in F_i. Worst case, we need to check all neighbors of a node, which takes takes $O(E)$ time.

The running time of `Dijkstra_SK` for a single source f in supernode G_i takes $O(|E_i| + |V_i| \log |V_i|)$, hence the total time it takes is $|F_i|$-times this.

It is important to note, that both finding the frontiers of each supernode in line 1 and then running `Dijkstra_SK` in line 3 is independent of the other supernodes, hence these steps can be executed in parallel on the supernodes. Hence, when we evaluate our algorithms in the experiments section 6 we only report the slowest running time of `Dijkstra_SK` on the supernodes.

Step 2: The `Brandes_SK` algorithm: In the second step of `Brandes++` a version of `Brandes`' algorithm is run that is adjusted to the SKELETON. Remember from Section 3 that for every target node $s \in S$ `Brandes` consists of two main steps; (1) computing distances $d(s, v)$ and number of shortest paths $\sigma(s, v)$ from s to all other nodes v using `Dijkstra`'s algorithm. (2) traversing the BFS tree of `Dijkstra` in reverse order of discovery to compute the dependencies based on the formula given in Equation (4).

`Brandes_SK` consists of the exact same steps as `Brandes` except that the multiplicities in the characteristic tuples on the edges of G_{sk} are also taken into consideration. This means that for the first step Equation (2) in `Dijkstra` is replaced by Equation (8), which takes multiplicities into consideration

$$\sigma(s, v) = \sum_{u \in P_v^{sk}} \sigma(s, u)\sigma(u, v). \tag{8}$$

where $P_v^{sk} = \{u \in V_{\text{sk}} | (u,v) \in E_{\text{sk}}, d(s,v) = d(s,u) + d(u,v)\}$ is the set of parent nodes of v in G_{sk}. Observe that $\sigma(s,v)$ computed in Equation (8) is not only the multiplicity of shortest paths between s and v in G_{sk} but also the number of shortest paths between s and v in the original graph G. The same holds for any edge $(u,v) \in E_{\text{sk}}$; the multiplicity in its characteristic tuple corresponds to the number of shortest paths between nodes u and v 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, again, by the reverse order traversal of the BFS tree and exchanging Equation (4) for Equation (9). In Proposition 1 we show that this yields the desired result.

Proposition 1. *Let $G(V,E,M)$ be an undirected graph with weighted or unweighted edges, such that every edge $e(u,v) \in E$ has a multiplicity $m(u,v) \in M$. Let $s \in S$ be a target node. Further let $P_v = \{u \in V | (u,v) \in E, d(s,v) = d(s,u) + d(u,v)\}$ denote the set of parents of a node $v \in V$. then the following equation holds*

$$\delta(s|u) = \sum_{v:u \in P_v} m(u,v) \cdot \frac{\sigma(s,u)}{\sigma(s,v)} (I_{v \in S} + \delta(s|v)) \tag{9}$$

Proof. Equation (4) used in the original `Brandes` algorithm deals with a source node s, a node v and its parent u. That formula computes a fraction of shortest paths from s to v that contain u over all shortest paths between s and u. The only difference between Equation (4) and Equation (9) in this proposition is that the multiplicity of the edge $e(u,v)$ is included as a multiplying factor when the same fraction is computed.

Running time: Observe, that the running of `Brandes` and `Brandes_SK` only differ in the formulas used, but are the same in terms of which order nodes and edges are visited during the execution. Hence the running times of the two algorithms – that both depend only on the number of edges and not the mulitplicities – are the same.

Using the formula of the running time of `Brandes` for G_{sk} 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

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

Step 3: The `Centrality` algorithm: In the last step of `Brandes++`, the centrality values of all the nodes are computed. As the centrality of frontier nodes $f \in F_i$ are given, we only need to compute the betweenness centrality of nodes $v \in V_i \setminus F_i$. Let us focus on supernode G_i and an arbitrary node $v \in V_i \setminus F_i$ inside. Let $s \in S$ and let $f \in F_i$ be such a frontier node, that there exists a shortest path from s to f going through v. To determine, whether f is such a frontier node, we need to keep the information $d(f,v)$ for v and every frontier 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. using these two values, it follows from Proposition 1 that

$$\delta(s|v) = \sum_{f \in F_i} \frac{\sigma(s,v)}{\sigma(s,f)} \sigma(v,f) \left(I_{v \in S} + \delta(s|f) \right). \tag{10}$$

As a result, we simply compute the centrality $C(v) = \sum_{s \in S} \delta(s|v)$.

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: The computation of equation (10) takes $O(|F_i|)$ time for every $v \in V_i \setminus F_i$. Hence, computing the centrality of all nodes in G takes time $O(\sum_{i=1}^{k} |V_i||V_i \setminus F_i|)$.

Effect of the input partition So far, we have described `Brandes++` with respect to an input partition $\mathcal{P}$ of the original graph. The preceding discussion on the running time of `Brandes++` as well as its space requirements reveal that the structure of $\mathcal{P}$ has an effect both in the time and in the space complexity of `Brandes++`. Below we outline a set of desirable properties for the input partition and discuss how existing graph-clustering algorithms satisfy the desiderata.

Desiderata: At a high level the running time of `Brandes++` depends on the running time of its three steps depicted in the pseudocode of Algorithm 1. We discuss the types of partitions that would be ideal for improving the running time of each one of these steps.

The first step of `Brandes++` executes the `Dijkstra` algorithm inside every supernode G_i and it therefore requires $O(|F_i||E_i|)$ time per supernode. Since these computations can be done independently (and thus also in parallel) for every supernode, a *balanced partition* (i.e., a partition with supernodes of approximately the same size) would benefit the running time of this step. Moreover, the *number of frontier nodes per supernode should be small.*

The second step of `Brandes++` executes the `Brandes` algorithm using as input the SKELETON graph. To minimize this running time, the number of nodes and edges in the SKELETON should be small. Recall, that the number of nodes in the SKELETON is equal to the total number of frontier nodes. Hence, the first requirement imposed by Step 2 is that $\mathcal{P}$ should have a *small number of frontier nodes.* In the SKELETON both the size of frontier sets as well as their distribution among different parts affects the final number of the (across supernodes) edges. Hence, the second requirement imposed by Step 2 is to *minimize the number of edges between partitions.*

In `Brandes++` the running time of step 3 in Algorithm 1 is $O(|F_i||V_i|)$ per supernode G_i. Therefore, for this version of our algorithm it would be beneficial to create a partition with *small number of frontier nodes* as well as *small number of overall nodes* per supernode.

Overall, if our goal is to optimize the running time of `Brandes++` we would like to have balanced partitions, with small number of frontier nodes and small number of edges across supernodes. In fact, small number of frontier nodes per supernode also benefits the space complexity of our algorithm, as we discussed in the previous paragraph.

In addition to the above, one should recall that in the SKELETON subgraphs spanned by supernodes G_i are replaced by cliques. The gain (decrease in the size of the SKELETON) can come from this as the supernodes are formed by *high-density subgraphs* in the original graph. Therefore, supernodes with these property are also desirable for `Brandes++`.

Below we discuss the extent to which existing partition algorithm satisfy the above desiderata.

Partition algorithms: Unfortunately, there does not exist a graph-partitioning algorithm that satisfies exactly the requirements we discussed above. Neither is it clear how to formalize a graph-partitioning problem that takes into account simultaneously all the above requirements. However, there are a lot of graph-partitioning algorithms that satisfy a good subset of these requirements. We highlight these algorithms and discuss their applicability to our setting below.

Density-based partitioning: There is a wealth of graph – partitioning algorithms that aim to partition a graph into clusters (i.e., subgraphs) such that the density of edges within each cluster is higher than the density of edges that connect nodes in different clusters. Such algorithms aim mostly to identify communities of users in social networks. One of the most popular tools for achieving that is *modularity clustering* Intuitively, the modularity of a cluster is the number of edges with both their end points in the cluster minus the number of edges that the nodes in the cluster can potentially have (i.e., the sum of the degrees of the cluster's nodes). Given this, the modularity of a clustering is the summation of the modularities of its clusters and the goal of the corresponding algorithms is to find a clustering with high modularity[11,26,29,30]. Intuitively, the supernodes reported by these algorithms are relatively dense and only have few edges that span different supernodes. Unfortunately, our experimental evaluation demonstrates that partitioned obtained by optimizing this objective resulted in high running times for the `Brandes++` algorithm, the reason being that the density of the clusters itself is not adequate for small running times as the other desiderata need also be met.

Normalized-cut partitioning: Another group of graph – partitioning algorithms are those that optimize for the normalized cut objective. Intuitively, these algorithms [1,12,18,21,22,27,31] want to minimize the number of edges that go across clusters while at the same time they produce clusters that are both dense and relatively balanced (either with respect to the number of nodes or edges). Intuitively, the algorithms from this class satisfy many of the desirable properties we outlined above and are expected to lead to small running times of `Brandes++`. Indeed our experimental evaluation verifies this intuition.

6 Experiments

Experimental setup: For all our experiments we follow the same methodology; given the partition $\mathcal{P}$, we run Steps 1–3 of the `Brandes++` algorithm (Alg. 1) using $\mathcal{P}$ as input. Then, we report the running time of `Brandes++` using this partition. As we mentioned in Section 5 finding the frontiers and running `Dijkstra_SK` (lines 1 and 3 of Alg. 2) are independent and can be run at the same time on all supernodes G_i. Hence, the running time that we report consists of the following components: (i) the running time of `Build_SK` on the largest supernode G_i (*ii*) running `Brandes_SK` on the SKELETON (i.e. Step 2 of Alg. 1) and (*iii*) computing the centrality of *all* remaining nodes in G (i.e., Step 3 of Alg. 1).

For all our datasets, unless we specify otherwise, we pick 200 nodes (uniformly at random) as sources.

Implementation: In all our experiments we compare the running times of `Brandes++` to Brandes' original algorithm [7] 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++`. So 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 we have pointed out in detail in Section 5, the `Brandes_SK` algorithm run on the SKELETON is almost identical to `Brandes`. Hence, 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 make our code available to use for research.[3]

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

Datasets: The following datasets are used for our experiments:

`WikiVote` *dataset [23]:* 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: [16]* 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 [16]. 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 [24]:* 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

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

with an undirected edge if they have ever exchanged an email. The graph has 265K nodes and 365K edges.

`DBLP` *dataset [34]:* 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.

Graph-partitioning: The speedup ratio of `Brandes++` over `Brandes` is determined by the size of the $\textsc{skeleton}(G_{\text{sk}})$ that is induced by the input graph partition. Recall that in G_{sk} every subgraph G_i is replaced by a clique X_i. Therefore, there is a tradeoff between nodes and edges removed from G_i and the size of the clique that is determined by the size of the frontier F_i.

In practice, graphs for which we expect to have good partitions are those that consist of a number of subgraphs that are only sparsely connected to each other. In our experiments we show that such graphs can be collaboration networks or road networks, where dense subgraphs are defined by communities or cities respectively. On the other hand, power-law graphs are not expected to have such structure since most nodes are connected to the high-degree nodes of the graph.

For our experiments we partition the input graph into subgraphs using well-established graph-partitioning algorithms. For that we use algorithms that either aim to find densely-connected subgraphs, which we call *modularity clustering* algorithms [11,26,29,30], or sparse cuts, which we call *normalized cut* algorithms [1,12,18,21,22,27,31]. We choose three very popular algorithms (described below), `Mod`, `Gl` and `Metis`, from these two groups.

`Mod`*:* The algorithm 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* [11] and its implementation is available at: `http://cs.unm.edu/~aaron/research/fastmodularity.htm`

`Gl`*:* This algorithm is a normalized-cut partitioning algorithm that was first introduced by Dhillon *et al.* [12]. An implementation of `Gl`, that uses a kernel k-means heuristic for producing a partition, is available at: `cs.utexas.edu/users/dml/Software/graclus.html`. `Gl` 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 `Gl`-k to denote the `Gl` clustering into at most k clusters.

`Metis`*:* This algorithm by Karypis *et al.* [22] 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 `Gl`, `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.

Before examining the effect of the different algorithms on `Brandes++`, we report the running times of the three clustering algorithms in Table 1. Note that for both `Gl` 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 among the values of k (between 2K–20K depending on the dataset) we consider. 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. The content of Table 1 is intended 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`	`Gl`	`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: Here, we evaluate the performance of `Brandes++` and explore the effect that the partitions produced by the different clustering algorithms have on its running time. As a reference point, we report in Table 1 the running times of the original `Brandes` algorithm on our datasets.

The properties of the partitions produced for all our real-life 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 `Gl` and `Metis` we experimented with several (about 10) values of k. For lack of space, we report only 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.

In each table N and M refer to the number of nodes and edges in the SKELETON using the given partition as an input to `Brandes++`. Remember, that in case of `Brandes++` the set of nodes in the skeleton is the union of the frontier nodes in each cluster. 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 tables 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++` given (in seconds) in the last row of the tables. Partitions produced by `Metis` are consistently faster than the same-sized partitions of `Gl`. We can also see that on `EU` and `DBLP` `Brandes++` with the `Mod` partition is faster than with `Metis`. This is not surprising, as both datasets are known for their distinctive community structure, which is what `Mod` optimizes for.

In order to see the speedup offered by `Brandes++`, one should compare the running time of `Brandes++` in Table 2 to the running time of `Brandes` in Table 1

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`	`Gl`-100	`Gl`-1K	`Gl`-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`	`Gl`-1K	`Gl`-10K	`Gl`-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`	`Gl`-1K	`Gl`-3K	`Gl`-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`	`Gl`-100	`Gl`-1K	`Gl`-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

– last column. For example, Brandes needs 5647 seconds for WikiVote while the corresponding time for Brandes++ can be as small as 72 seconds! Equally impressive are the results for the EU and the DBLP datasets. For those Brandes takes 14325 and 28057 seconds respectively, while the running time of Brandes++ can be 1872 and 5709 seconds respectively. This means that for most of these datasets we can get 5-fold to 8-fold improvement of the running time. 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.

7 Conclusions

In this paper, we proposed Brandes++, an algorithm that allows us to efficiently compute the betweenness centrality of nodes in large graphs. Brandes++ leverages a new data structure that we propose, which we call the SKELETON. Our experimental evaluation using datasets from a variety of applications demonstrated the significant computational gains we obtain by Brandes++, particularly when it is paired with effective algorithms for obtaining the initial partitioning of nodes that is the basis of forming the SKELETON.

References

1. R. Andersen. A local algorithm for finding dense subgraphs. *ACM Transactions on Algorithms*, 2010.
2. J. M. Anthonisse. *The rush in a directed graph.* SMC, 1971.
3. D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *WAW*, 2007.
4. D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *ICPP*, 2006.
5. A. Bavelas. A mathematical model for group structure. *Human Organizations*, 1948.
6. S. P. Borgatti. Centrality and network flow. *Social Networks*, Jan. 2005.
7. U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 2001.
8. U. Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 2008.
9. U. Brandes and C. Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 2007.
10. Ü. V. Çatalyürek, K. Kaya, A. E. Sariyüce, and E. Saule. Shattering and compressing networks for betweenness centrality. In *SDM*, 2013.

11. A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E*, 2004.
12. I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors: A multilevel approach. *IEEE Trans. Pattern Anal. Mach. Intell*, 2007.
13. N. Edmonds, T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *HiPC*, 2010.
14. L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 1977.
15. R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, 2008.
16. P. Gill, M. Schapira, and S. Goldberg. Let the market drive deployment: a strategy for transitioning to bgp security. In *SIGCOMM*, 2011.
17. K.-I. Goh, B. Kahng, and D. Kim. Universal behavior of load distribution in scale-free networks. *Phys. Rev. Lett.*, Dec 2001.
18. B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Supercomputing, 1995.
19. V. Ishakian, D. Erdös, E. Terzi, and A. Bestavros. Framework for the evaluation and management of network centrality. In *SDM*, 2012.
20. U. Kang, S. Papadimitriou, J. Sun, and H. Tong. Centralities in large networks: Algorithms and observations. In *SDM*, 2011.
21. R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 2004.
22. G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 1998.
23. J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting positive and negative links in online social networks. In *WWW*, 2010.
24. J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 2007.
25. 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.
26. M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review*, E 69, 2004.
27. A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS*, 2001.
28. R. Puzis, P. Zilberman, Y. Elovici, S. Dolev, and U. Brandes. Heuristics for speeding up betweenness centrality computation. In *SocialCom/PASSAT*, 2012.
29. 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.
30. R. Rotta and A. Noack. Multilevel local search algorithms for modularity clustering. *J. Exp. Algorithmics*, 2011.
31. S. E. Schaeffer. Survey: Graph clustering. *Comput. Sci. Rev.*, 2007.
32. A. Shimbel. Structural parameters of communication networks. *Bulletin of Mathematical Biology*, 1953.
33. G. Tan, D. Tu, and N. Sun. A parallel algorithm for computing betweenness centrality. In *ICPP*, 2009.
34. J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *ICDM*, 2012.