2012 ASE/IEEE International Conference on Social Computing and 2012 ASE/IEEE International Conference on Privacy, Security, Risk and Trust

Heuristics for Speeding up Betweenness Centrality Computation

Rami Puzis*, Polina Zilberman†, Yuval Elovici‡, Shlomi Dolev§ and Ulrik Brandes¶
*University of Maryland Institute for Advanced Computer Studies (UMIACS),College Park,MD, USA. Email: puzis@umd.edu
†Telekom Innovation Laboratories at Ben-Gurion University of the Negev, Israel. Email: polinaz@bgu.ac.il
‡Information Systems Engineering Dept., Ben-Gurion University of the Negev, Israel. Email: elovici@bgu.ac.il
§Dept. of Computer Science, Ben-Gurion University of the Negev, Israel. Email: dolev@cs.bgu.ac.il
¶Department of Computer & Information Science University of Konstanz, Germany. Email: Ulrik.Brandes@uni-konstanz.de

***Abstract*—We propose and evaluate two complementary heuristics to speed up exact computation of the shortest-path betweenness centrality. Both heuristics are relatively simple adaptations of the standard algorithm for betweenness centrality. Consequently, they generalize the computation of edge betweenness and most other variants, and can be used to further speed up betweenness estimation algorithms, as well.**

In the first heuristic, structurally equivalent vertices are contracted based on the observation that they have the same centrality and also contribute equally to the centrality of others. In the second heuristic, we first apply a linear-time betweenness algorithm on the block-cutpoint tree and then compute the remaining contributions separately in each biconnected component. Experiments on a variety of large graphs illustrate the efficiency and complementarity of our heuristics.

I. Introduction

While various kinds of network representations are studied in a variety of disciplines, centrality indices capturing the structural importance of nodes and links are an essential tool across all substantive domains. Some centrality indices are trivial to compute, but others require more involved algorithms. In this paper, we are interested in centrality indices based on shortest paths, and shortest-path betweenness centrality in particular [1], [2]. Algorithms to compute these indices typically involve performing a large number of single-source shortest-paths computations, and therefore do not scale well to the large graphs encountered in, for instance, the analysis of data produced in socio-technical systems.

The algorithm most commonly used to compute shortest-path betweenness centrality requires $O(nm)$ time in unweighted graphs with n nodes and m links [3]. It has therefore been adapted to run faster on special hardware such as compute clusters and multiprocessor architectures [4]–[7]. Another line of attack are algorithms which compute the scores that are only approximately correct by extrapolating from single-source shortest-paths on fewer sources, or with a bounded distance [8]–[11].

We propose two speed-up heuristics which can be used with the original exact algorithm as well as previous speed-up techniques because they are based on a reduction of instance complexity. To the best of our knowledge, this is the first time that the exact sequential algorithm of [3] has been improved.

The remainder of this paper is structured as follows. In Section II we briefly review betweenness definitions and state of the art computation method.

The first heuristic described in Section III is based on size reduction by vertex contraction, while the second heuristic is based on size reduction by decomposition, as described in Section IV. Both can be combined as described in Section V. Empirical performance results for the individual heuristics and their combination are reported in Section VI, and we conclude in Section VII.

II. Background

Betweenness Centrality (BC) was introduced as a measure of control an individual may have over communication between all others in a social network [1], [2]. The algorithms presented in this paper make use of the augmented variants of the fast algorithm for BC computation [3]. In particular, in order to support the unification of structurally equivalent vertices, we define a BC variant for weighted graphs where vertices may have multiplicity weights. A similar variant of BC for multigraphs (i.e., graphs with multiplicity weights on edges) was defined in [12]. We also use the BC variant defined for networks with arbitrary communication patterns [13] both for unifying structurally equivalent vertices and for computing BC separately within each bi-connected component.

We start by providing a basic definition of BC and its previously presented variants used in this paper. Let $G = (V, E)$ denote an undirected unweighted graph where V is the set of vertices and E is the set of edges. The graph G can be augmented by weight functions with various semantics. Weights can be either unary (e.g. w_v where $v \in V$) or binary (e.g. $w_{u,v}$ where $u, v \in V$). Note that vertices provided as arguments for binary weights are not required to be neighbors. For example, let h_{st} represent the importance or the volume of communication between two vertices s and t. BC, with respect to arbitrary overlay networks [13], can be computed by using the h matrix, as explained in the following paragraphs.

We gratefully acknowledge financial support from US Army Research Office Grant W911NF0910206 and Israel Ministry of Defense Grant 4440366064.

Let σ_{st} denote the number of shortest paths between s and t. Let $\sigma_{st}(v)$ denote the number of shortest paths between s and t that pass through v. Assuming that v lies on the shortest path from s to t, $\sigma_{st}(v)$ is equal to the product $\sigma_{sv} \cdot \sigma_{vt}$. Let $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$ denote the pairwise dependency of s and t on the vertex v. $\delta_{st}(v)$ stands for the potential control that the vertex v may have over the communication between s and t.

Originally, betweenness of the vertex $v \in V$ was defined as the sum of dependencies of all pairs of vertices on v.

$$BC(v) = \sum_{s,t \in V, s,t \neq v} \delta_{st}(v)$$

Later definitions allow the inclusion of end vertices ($s = v$ or $t = v$) in the computation of betweenness. For example, in communication networks, it is reasonable to assume that both the source and the target of a message have control over the message content. Therefore, communication emanating from v or targeted at v should be accounted for in v's betweenness. Taking into account the communication importance (or volume) denoted by h, BC can be defined as a sum of products:

$$BC(v) = \sum_{s,t \in V} \delta_{st}(v) \cdot h_{st}. \tag{1}$$

In order to compute the BC for all vertices on graph G in $O(nm)$ a source dependency $\delta_s(v)$ should be used [3]. $\delta_s(v)$ stands for the potential control that v may have over all communication emanating from s:

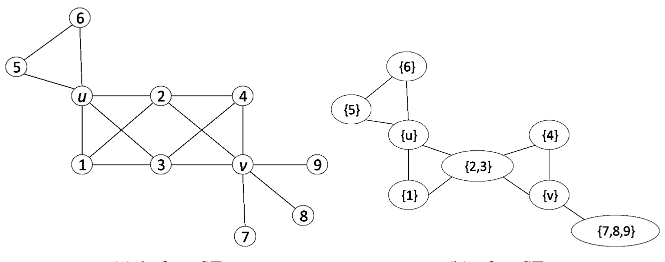
$$\delta_s(v) = \sum_{t \in V} \delta_{st}(v) \cdot h_{st} \tag{2}$$

Since $BC(v) = \sum_{s \in V} \delta_s(v)$, it is possible to compute the BC using combinatorial path counting, as described in [3], [13]. Let d_{st} denote the distance between the vertices s and t. Let $P_s(u) = \{v | (v,u) \in E \wedge d_{su} = d_{sv} + 1\}$ denote the set of predecessors of u on the way from s:

$$\delta_s(v) = h_{sv} + \sum_{u : v \in P_s(u)} \frac{\sigma_{sv}}{\sigma_{su}} \cdot \delta_s(u) \tag{3}$$

Source dependency of s on all vertices in the graph can be computed in $O(m)$ time by a single BFS traversal followed by accumulation of δ_s performed in the reverse distance order from s. Most state of the art algorithms for BC computation are based on the accumulation of source dependencies during such $O(m)$ time traversals from each source vertex in the graph, resulting in algorithms for BC computation whose time complexity is $O(nm)$ [3], [12]–[14].

In this paper we aim at reducing the time required for BC computation by: (1) reducing the number of vertices and edges in the graph, and (2) partitioning the graph into several edge disjoint components and executing the BC algorithms separately within each component. The BC computation results for each component are combined in order to derive the BC values for all the vertices in the graph.



(a) before SE (b) after SE

Fig. 1. Example network before and after unifying structurally equivalent vertices.

III. Unifying structurally equivalent vertices

In this section we describe how to reduce the number of vertices and the number of edges in the graph by unifying structurally equivalent vertices in order to speed up the computation of BC (the speed up is shown in Section VI). Vertices can be considered as structurally equivalent in unweighted graphs if they have the same set of neighbors [15], [16], as in the network in Fig. 1. An algorithm for finding structurally equivalent vertices can be found in [15].

With respect to topology, these vertices play the exact same role in the network. In particular, structurally equivalent vertices have the same centrality indexes e.g., degree, closeness, and betweenness. Moreover, their contribution to the betweenness of other vertices is the same, both as sources and targets. It is therefore a waste of computational resources to account for all links connecting each one of the structurally equivalent vertices during BC computation.

Next, we define the reduction of the BC computation problem for G to the BC computation problem for the quotient graph with equivalent vertices contracted. We will demonstrate how all sets of structurally equivalent vertices, namely structural equivalence classes, can be contracted in linear time without compromising the computation of BC. The time required to compute BC in the derived graph depends on the number of equivalence classes rather than on the number of vertices in the original graph. We show that the BC value of an equivalence class divided by the size of the class is equal to the BC values of its members computed on the original graph.

Let $G = (V, E)$ be a graph where V is the set of vertices and E is the set of edges. Let $C = C_1, C_2, \ldots, C_k, (C_i \subseteq V)$ be the set of structural equivalence classes. We define the equivalence class network as $G/C = (C, E/C)$ where C, the set of equivalence classes, is the set of vertices and E/C is the set of links between equivalence classes. Two equivalence classes C_i and C_j are connected if and only if all vertices in class C_i are connected to all vertices in class C_j in the original network. Note that since C_i and C_j contain structurally equivalent vertices (vertices that share the same neighbors), we can define the set of links E/C as follows:

$$E/C = \{(C_i, C_j) \in C^2 | \exists v \in C_i \exists u \in C_j, (v,u) \in E\}$$

The vertices in G/C are weighted. The weight of each vertex $C_i \in C$ is defined as the cardinality of the respective equivalence class. It should be noted that by encapsulating a set of vertices within a single vertex in G/C (e.g. C_i) we lose the paths connecting these vertices with each other (see Fig. 2). In order to proceed with the betweenness computation we first show how to compensate for the loss of this information. We then, illustrate how to account for all other paths (that start and end within different equivalence classes) by traversing the network G/C only. Let $C(v)$ stand for the equivalence class of vertex $v \in V$. As the result of both steps, the BC values of $C(v)$ computed for the graph G/C will be exactly $|C(v)|$ times the BC value of v ($v \in C(v)$) in the original graph G.

A. Paths between structurally equivalent vertices

Let h/C be the communication importance matrix for the network G/C. According to Equation 1 and the definition of $\delta_{st}(v)$, values on the main diagonal of the communication importance matrix ($h_{vv}, v \in V$) are simply added to the betweenness of the respective vertex. In most applications h_{vv} is typically equal to zero. We will use $h/C_{C(v)C(v)}$ to store the contribution of paths that start and end within the same equivalence class. Overloading of $h/C_{C(v)C(v)}$ with this additional information allows us to avoid significant changes in the state-of-the-art BC computation executed on G/C.

The distance between every two nodes $u, v \in C(v)$ is exactly two hops (see Proposition 3.1). In other words, the changes we need to make in order to encapsulate the paths that start and end at vertices within the same equivalence class are local; constrained to the structurally equivalent vertices and their neighbors. Assuming the default all-to-all communication pattern in the original graph, the number of communication flows within an equivalence class $C(v)$ is $|C(v)|\cdot(|C(v)|-1)$, as each vertex equally communicates with all others in the same class. Note that both communications that start at v, as well as those that end at v, contribute to v's betweenness. The contribution of all structurally equivalent peers to the BC of v is $2 \cdot (|C(v)| - 1)$. Therefore, the total contribution of structurally equivalent vertices in $(C(v))$ to their own BC values is $2 \cdot (|C(v)| - 1) \cdot |C(v)|$.

Next, we take into account the contribution of paths between structurally equivalent vertices on the BC values of their neighbors. Let $NN(v)$ be the number of v's neighbors in the original network G. $NN(v)$ can also be computed using the equivalence class network G/C, as following:

$$NN(v) = \sum_{(C(w),C(v)) \in E/C} |C(w)|.$$

For each two structurally equivalent vertices $u, v \in C(v)$, the number of shortest paths between them is equal to the number of neighbors $\sigma_{uv} = NN(v)$. Since the communication between u and v contributes $1/\sigma_{uv}$ to the BC of each one of their neighbors, the total contribution of all communications between vertices within an equivalence class $C(v)$ to their common neighbor w is $|C(v)| \cdot (|C(v)| - 1)/NN(v)$. We further sum these contributions across all vertices in $C(w)$ to get the contribution of all paths starting and ending within $C(v)$ on the total betweenness of vertices in $C(w)$: $|C(w)|\cdot|C(v)|\cdot(|C(v)|-1)/NN(v)$. Finally, recall that $C(w)$ may have many neighbor equivalence classes that contribute to the betweenness of its members. Adding the contribution of paths that begin and end within $C(w)$ to the summation over all $C(w)$'s neighbor equivalence classes results in:

$$\begin{aligned} h/C_{C(w)C(w)} &= 2 \cdot (|C(w)| - 1) \cdot |C(w)| + \\ &+ \sum_{(C(v),C(w)) \in E/C} \frac{2|C(w)|\binom{|C(v)|}{2}}{NN(v)} \end{aligned} \tag{4}$$

As illustrated in Equation 4, $h/C_{C(w)C(w)}$ is composed of two addends. The first addend accounts for paths that begin and end within $C(w)$. The second addend accounts for paths that begin and end within each one of the neighbor equivalence classes of $C(w)$.

B. Paths between different equivalence classes

After encapsulating all shortest paths that were lost due to collapse of equivalence classes in $h/C_{C(w)C(w)}$, we show that other paths (that begin and end within different equivalence classes) can be accounted for by traversing the network G/C only. In the following paragraphs we will define inter-class communication importance ($h/C_{C(v)C(u)}$), show that distances in G/C and G are consistent, and define path counts in networks with multiplicity weights on vertices.

Let $C(u), C(v) \subseteq V$ be different nodes in G/C. Since the subsets $C(u)$ and $C(v)$ are different structural equivalence classes, they are mutually exclusive. We define $h/C_{C(u)C(v)}$, where $C(u) \neq C(v)$, as the total communication from vertices in $C(u)$ to vertices in $C(v)$, as shown in Equation 5:

$$h/C_{C(u)C(v)} = |C(u)| \cdot |C(v)|. \tag{5}$$

Let $s \neq t \in V$ be two vertices. Let $C(s), C(t) \subseteq V$ be equivalence classes of s and of t, respectively. For any two structurally non-equivalent vertices $s, t \in V$ the distance between s and t in G is equal to the distance between the respective equivalence classes in G/C.

Proposition 3.1: For all $s \neq t \in V$ it holds that:

- $dist(st) = 2$ if $C(s) = C(t)$ and
- $dist(st) = dist(C(s), C(t))$ otherwise.

Corollary 3.1.1: Based on Proposition 3.1, we conclude that structurally equivalent vertices cannot share a shortest path that have three or more edges.

Proposition 3.1 allows denoting the distance between structurally non-equivalent vertices $u, v \in V$ as $dist(uv)$ or $dist(C(u)C(v))$, interchangeably.

For any two different equivalence classes $C(s) \neq C(t)$ let $\sigma_{C(s)C(t)}$ be the total number of shortest paths between all

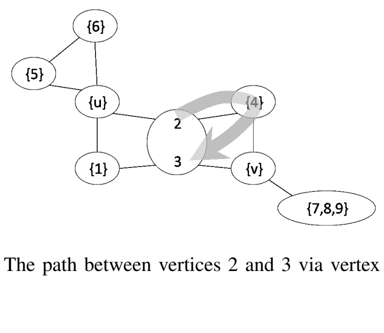


Fig. 2. The path between vertices 2 and 3 via vertex 4 is lost.

pairs of vertices in $C(s)$ and $C(t)$, respectively. Since there are $|C(s)|$ structurally equivalent sources and $|C(t)|$ structurally equivalent targets, we can represent $\sigma_{C(s)C(t)}$ as:

$$\sigma_{C(s)C(t)} = |C(s)| \cdot |C(t)| \cdot \sigma_{st}. \tag{6}$$

For example, if $C(s)$ and $C(t)$ are neighbors, then $\sigma_{C(s)C(t)} = |C(s)| \cdot |C(t)|$. $\sigma_{C(v)C(v)}$ is defined as unity. Note that $\sigma_{C(s)C(t)}(C(s)) = \sigma_{C(s)C(t)}$ since any route that begins at some vertex $s \in C(s)$ cannot pass through any other $v \in C(s) \wedge v \neq s$ (therefore, the start point s is the only intersection of the route with $C(s)$).

Let $\sigma_{st}(C(v))$ be the number of shortest paths between s and t passing through at least one vertex in $C(v)$. The same definition of path counting was used for group betweenness centrality calculations in [17], [18]. In contrast to general sets of vertices it is much easier to count the number of shortest paths passing through a structural equivalence class. Each equivalence class multiplies the number of shortest paths passing through it by its cardinality:

$$\sigma_{st}(C(v)) = |C(v)| \cdot \sigma_{st}(v). \tag{7}$$

The intuition and calculation of the number of shortest paths for multi-vertices (i.e., structural equivalence classes in G/C) is similar to the case of multigraphs where multiple edges are allowed between vertices [12]. When a path in G/C contains several structural equivalence classes every combination of vertices from each one of the classes yields a unique path in G. Therefore, the total number of paths in G obtained from a single path in G/C is the product of the cardinalities of equivalence classes along the path. Let $\sigma_{C(s)C(t)}(C(v))$ be the number of shortest paths between all vertices in $C(s)$ and all vertices in $C(t)$ passing through at least one vertex in $C(v)$. $\sigma_{C(s)C(t)}(C(v))$ can be computed from $\sigma_{st}(v)$ using Equations 6 and 7:

$$\sigma_{C(s)C(t)}(C(v)) = |C(s)| \cdot |C(t)| \cdot |C(v)| \cdot \sigma_{st}(v). \tag{8}$$

Next we will use Equations 6 and 8 to represent $\sigma_{C(s)C(t)}(C(v))$ based on path counting in G/C only. For any $s \neq v \neq t$, such that $dist(st) = dist(sv) + dist(vt)$, $\sigma_{C(s)C(t)}(C(v))$ is computed as follows:

$$\sigma_{C(s)C(t)}(C(v)) = \frac{\sigma_{C(s)C(v)} \cdot \sigma_{C(v)C(t)}}{|C(v)|} \tag{9}$$

Note that $\sigma_{C(s)C(t)}(C(v))$, as computed by Equation 9, is a generalization of $\sigma_{st}(v)$ for weighted graphs where multiplicity weights on vertices are allowed.

Let $\delta_{C(s)C(t)}(C(v)) = \frac{\sigma_{C(s)C(t)}(C(v))}{\sigma_{C(s)C(t)}}$ be the total fraction of shortest paths from any vertex in $C(s)$ to any vertex in $C(t)$ passing through $C(v)$. We will denote the BC of an equivalence class $C(v)$ in the network G/C as $BC_{G/C}(C(v))$:

$$BC_{G/C}(C(v)) = \sum_{C(s),C(t) \in C} \delta_{C(s)C(t)}(C(v)) \cdot h/C_{C(s)C(t)} \tag{10}$$

Let $\delta_{C(s)}(C(v)) = \sum_{C(t) \in C} \delta_{C(s)C(t)}(C(v))$ be the total potential control that vertices in $C(v)$ may have over communication originating at all vertices in $C(s)$. $BC_{G/C}$ can be expressed as a sum of $\delta_{C(s)}(C(v))$ similar to the original BC. Let $P/C_{C(s)}(C(u))$ be a set of predecessors of $C(u)$ on the way from $C(s)$ in graph G/C. It is possible to express $\delta_{C(s)}(C(v))$ using a recursive formula:

$$\begin{aligned} \delta_{C(s)}(C(v)) &= h/C_{C(s)C(v)} + \\ &+ \sum_{C(u): C(v) \in P/C_{C(s)}(C(u))} \frac{\sigma_{C(s)C(v)}}{\sigma_{C(s)C(u)}} \cdot \delta_{C(s)}(C(u)) \cdot |C(u)| \end{aligned} \tag{11}$$

Comparing Equations 3 and 11, we can see that the only change required in a BC computation algorithm is multiplication by cardinality (i.e., multiplicity) of vertices during the aggregation stage. This is a generalization of the existing algorithms since the cardinality of vertices in regular networks can be defined as unity.

BC of $C(v)$ computed on G/C w.r.t. h/C according to Equation 10 is exactly $|C(v)|$ times larger than BC of v computed on G w.r.t. the all-to-all communication pattern (Equation 1). In Section VI we will evaluate the effectiveness of the proposed SE based network size reduction technique in terms of BC computation time.

Theorem 3.2 (BC of structural equivalence classes):

$$BC(v) = \frac{BC_{G/C}(C(v))}{|C(v)|}$$

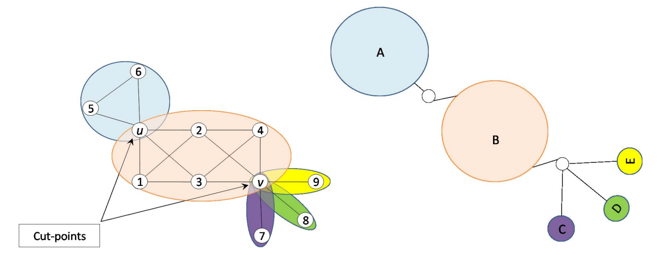
IV. Partitioning to bi-connected components

This section describes another heuristic for speeding up the betweenness centrality computation. The following heuristic uses the fact that computing BC on trees is easily done in linear time. The basic idea of this method is that we can count the number of vertices separated from each other by removing a node from the tree. It is trivial for leaves; if end-points are considered during BC computation, then their BC equals $n-1$, otherwise, it is zero. Next, the nodes on the second (from the bottom) level of the tree separate leaves from each other and from the rest of the tree. Again, this enables an easy computation of their BC. In general, removing a node (in lets say, a rooted tree) separates all its sub-trees from each other and from the rest of the tree.

We exploited the above insight to speed up betweenness centrality computations on large graphs by integrating the BC computation with the algorithm for graph partitioning to bi-connected components and the construction of the bi-connected components tree. Then, after a linear time analysis of the tree we compute the BC of vertices within each component without traversing the whole network. The speedup on large sparse graphs (that consist of many bi-connected components) is achieved because the computationally expensive (quadratic to cubic time) BC calculations are performed many times on small subgraphs rather than once on a large network. Section VI demonstrates the achieved speedup.

A. Bi-connectivity and the bi-connected components tree

Let $G(V, E)$ be a connected graph. Let $v \in V$ be a vertex in G. If removal of v disconnects G, then v is said to be a cut-point. A graph is said to be bi-connected if it is a connected



(a) BCC partition (b) BCC tree

Fig. 3. Bi-connected components and the block-cut-point-tree of the example network.

graph with no cut-points (i.e., at least two vertices should be removed to disconnect the graph). We assume that a graph consisting of a single edge and its two ends is bi-connected. A set of vertices $B \subseteq V$ is said to be a bi-connected component of G if the following constraints are satisfied:

(a) the subgraph induced by B is bi-connected and
(b) B can not be expanded to $B' \supset B$ such that B' is a bi-connected subgraph.

We refer the reader to [19] for additional information on bi-connectivity.

A Proxy Graph algorithm [20], for example, or its eager counterpart [21], can be used to partition a given graph into bi-connected components in linear time. Hereafter we will refer to bi-connected components as "components". Let $B_1, \ldots, B_k$ ($B_i \subseteq V$) be the set of components produced by the Proxy Graph algorithm. Two components B_i and B_j ($i \neq j$) are adjacent if they have one common vertex. Note that two components can not have more than one common vertex since in such case their unification will produce a bigger bi-connected sub-graph in contrast to the constraint (b) above. Let v be the vertex common to components B_i and B_j ($B_i \cap B_j = \{v\}$). v is a cut-point since its removal will separate B_i and B_j and disconnect the graph. Fig. 3 is an example for a network partitioned to bi-connected components.

Next, we define $T = (V(T), E(T), w)$; a weighted block-cutpoint tree [19] whose vertices are either bi-connected components or cut-points:

$$V(T) = \{B_1, \ldots, B_k\} \cup \{v | \exists_{i \neq j}, v \in B_i \wedge v \in B_j\}.$$

A cut-point v and a component B are connected in the tree if and only if $v \in B$.

$$E(T) = \{(B, v) | v \in V(T) \wedge B \in V(T) \wedge v \in B\}$$

Let B be a component and $v \in B$ be a cut-point. Since v is a cut-point, its removal disconnects G into several connected components. Let $B^+(v)$ ("cut with" B) be the *connected component* in G which includes B after removal of v. Or, in other words, $B^+(v)$ includes the vertices in G that are connected to v only via B and therefore maintain their connection with vertices in B, even after the removal of v (including those contained by B and not including v). We define the weight of a link $(v, B) \in E(T)$ to be the size of $B^+(v)$. We denote this weight by $D_B^{v(}$. Let $V - B^+(v) - \{v\}$ ("cut without" B) be

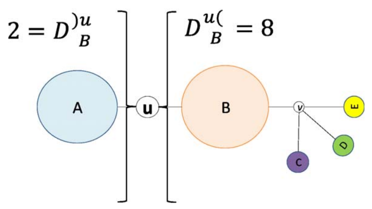


Fig. 4. The total number of vertices u cuts from component B $\left(D_B^{)u}\right)$ and the total number of vertices cut out from u including B $\left(D_B^{u(}\right)$.

```
Algorithm 1: Compute Component Tree Weights
Input: $V(T)$ collection of bi-connected components and
       cut-points from a graph $G(V, E)$;
       $E(T) \subseteq (V(T))^2$ collection of unweighted
       undirected links.
Output: Weights $D_B^{v(}$ for all links $(B, v) \in E(T)$
Data: A queue $Q$
▼ initialization
    for each link $(B, v) \in E(T)$ do $D_B^{v(} \leftarrow unknown$;
    for each component $B$ that has one cut-point $v$ do
    add the pair $(B, v)$ to $Q$;
while $Q$ is not empty do
    pair ← pull from $Q$;
    if pair is a component-vertex pair $(B, v)$ then
        $size \leftarrow |B| - 1$;
        for $n \in \{n | (B, n) \in E(T) \wedge D_B^{n(} \neq unknown\}$
        do
            $size \leftarrow size + |V| - D_B^{n(}$;
        $D_B^{v(} \leftarrow size$;
        ▼ update $Q$
            if there is one $B'$ such that $D_{B'}^{v(} = unknown$
            then add the pair $(v, B')$ to $Q$

    if pair is a vertex-component pair $(v, B)$ then
        $size \leftarrow 1$;
        for $N \in \{N | (N, v) \in E(T) \wedge D_N^{v(} \neq unknown\}$
        do $size \leftarrow size + D_N^{v(}$;
        $D_B^{v(} \leftarrow |V| - 1 - size$;
        ▼ update $Q$
            if there is one $v'$ such that $D_B^{v'(} = unknown$
            then add the pair $(B, v')$ to $Q$
```

the vertices that are separated from $B^+(v)$ by removing the cut-point v (not including v itself). Similarly, we define $D_B^{)v}$ to be the size of $V - B^+(v) - \{v\}$. For illustration see Fig. 4. For every component B and a cut-point $v \in B$ holds:

$$D_B^{)v} + 1 + D_B^{v(} = |V| \tag{12}$$

Next, we describe how to compute all $D_B^{v(}$ weights. The weights of edges in T can be computed starting from the leaves. A leaf in T is a component B that has only one cut-point v. In this case, $D_B^{v(}$ is equal to the number of vertices in the component not including v. After computing the weights of all links that connect leaves, we can proceed to the next level of nodes in T (a level of cut-points).

Let v be a cut-point such that the weights of all but one of its' links in T are already computed. Let B be a component such that $D_B^{v(}$ is not yet computed. $D_B^{v(}$ should be the number of vertices that are connected to v via B. The weights of edges that connect v with other components are already computed. Let $B' \neq B$ be a component such that $D_{B'}^{v(}$ is known. $D_{B'}^{v(}$ is the number of vertices that are connected to v via B'. Since G is a connected graph, any vertex in G is connected to v through one of the components that include v. Therefore, the number of vertices in the graph is equal to the sum of the weights of all links adjacent to v plus one (to account for v):

$$|V| = 1 + \sum_{B':v\in B'} D_{B'}^{v(} \tag{13}$$

By extracting $D_B^{v(}$ from the sum above we get:

$$D_B^{v(} = |V| - 1 - \sum_{B':B'\neq B \wedge v\in B'} D_{B'}^{v(}. \tag{14}$$

Let B be a component such that weights of all but one of its links in T are already computed. Let v be a cut point such that $D_B^{v(}$ is not yet computed. $D_B^{v(}$ should be the number of vertices that are connected to v via B. Weights of edges that connect B with other cut-points are already computed. Let $u \neq v$ be a cut-point such that $D_B^{u(}$ is known. $D_B^{u(}$ is the number of vertices that are connected to u via B. Hence, $|V| - D_B^{u(}$ is equal to the number of vertices that are connected to some vertex in B via u (including u). These vertices will maintain their connection with B even after removal of v. The sum of $|V| - D_B^{u(}$ for all $u \neq v$ such that $(B, u) \in E(T)$ plus the size of $|B|$ minus one (to subtract v) is the total number of vertices that will remain connected to B if v is removed. *Minus one* is in order to be consistent with the definition of $D_B^{v(}$ and not to account for v:

$$D_B^{v(} = |B| - 1 + \sum_{u\neq v|(B,u)\in E(T)} \left(|V| - D_B^{u(}\right) \tag{15}$$

This completes the description of the computation of all the weights in the bi-connected components tree which will be used in the next section to speedup the computation of betweenness. See Algorithm 1 for a pseudo code of the weights computation.

B. Betweenness computation

Let $G(V, E)$ be a sparse unweighted, undirected network. In this section we will show how to compute the betweenness for each (bi-connected) component of G separately using the weighted components tree described in Section IV-A. This computation will result in values will be equal to betweenness computed on the whole graph. To do this we use a variant of the shortest path betweenness centrality that weights shortest paths proportionally to the intensity of communication between the source vertex and the destination vertex [13]. Let h_{st} represent the communication intensity between s and t. Betweenness centrality of a vertex v with respect to h is defined by Equation 1.

Let $B_1, \ldots, B_k \subseteq V$ be the bi-connected components of G. Let $D_{B_i}^{)v}$ be the number of vertices that are separated from $B_i^+(v)$ by removing the cut-point v ($D_{B_i}^{)v}$ does not include v). Next we define a traffic matrix h^i for each component B_i ($1 \leq i \leq k$). Each element h_{xy}^i in the traffic matrix represents the communication intensity between the vertices $x, y \in B_i$. Traffic can flow in or out of the component through cut-points only. If either x or y is a cut-point, then h_{xy}^i also includes the communication that enters or leaves C_i through the cut-point.

We assume that all vertices in G equally communicate with each other and that no vertex communicates with itself:

$$h_{xy} = \begin{cases} 0 & x = y \\ 1 & x \neq y \end{cases}$$

Let both x and y be non-cut-point vertices in B_i. We set h_{xy}^i to 1 if $x \neq y$ and to 0 otherwise. If x is a cut-point, then $h_{xx}^i = h_{xx} = 0$. If x is a regular vertex and y is a cut-point, then y acts as a proxy for x. In this case h_{xy}^i represents the intensity of communication emanating from x to y and to other vertices in the network through y:

$$h_{xy}^i = D_{B_i}^{)y} + 1 \tag{16}$$

Since we assume that the traffic matrix is symmetric $h_{xy}^i = h_{yx}^i$. If x and y are two different cut-points, then h_{xy}^i is the intensity of communication that enters B_i through x and leaves B_i through y.

$$h_{xy}^i = h_{yx}^i = \left(D_{B_i}^{)x} + 1\right) \cdot \left(D_{B_i}^{)y} + 1\right) \tag{17}$$

Finally, for each component B_i we compute the betweenness values of all its vertices with respect to h^i:

$$BC(v)_{v\in B_i} = \sum_{s,t\in B_i} \delta_{st}(v) \cdot h_{st}^i \tag{18}$$

Lemma 4.1 (Local betweenness computation): For each non-cut-point $v \in V$ and bi-connected component $B \ni v$, the betweenness value of v computed "locally" on B is equal to the betweenness value of v computed in the original graph G:

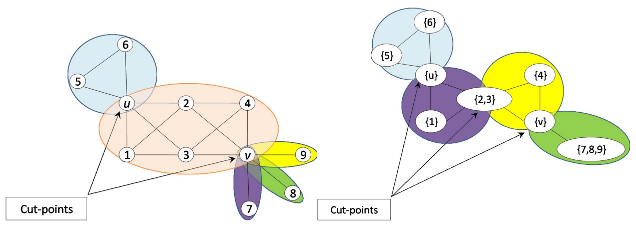
$$BC(v) = BC(v)_{v\in B}$$

Let v be a cut-point. We have computed the betweenness values of v within each component it belongs to. Each time the computation was done from the perspective of the single component. Let x and y be two vertices that would be in different connected subgraphs if v was removed from G. In such a case, $\delta_{xy}(v)$ always equals one since all paths between x and y pass through v. Therefore, the contribution of such vertices to the betweenness of v can be easily computed. Let inter-component communication (denoted as $BC^{inter}(v)$) be the total contribution of all pairs of vertices x, y cut from each other (completely separated) by v to the betweenness of v:

$$BC^{inter}(v) = \sum_{B\ni v} D_B^{v(} \cdot D_B^{)v} \tag{19}$$

Algorithm 2: Computing betweenness

```
   Input: unweighted undirected graph $G(V,E)$
1  $\{B_1,\ldots,B_k\} \leftarrow$ bi-connected components of $G$;
   // create undirected weighted
      components tree $T(V(T),E(T),w)$
2  $V(T) \leftarrow \{B_1,\ldots,B_k\} \cup \{v | \exists_{i\neq j} v \in B_i \wedge v \in B_j\}$;
3  $E(T) \leftarrow \{(v,B) | v \in V(T) \wedge B \in V(T) \wedge v \in B\}$;
4  ComputeComponentTreeWeights($V(T)$,$E(T)$);
5  for cutoff $v \in V$ do
6      $BC(v) -= |B_i : (v,B_i) \in E(T)| - 1 + \sum_{B_i:(v,B_i)\in E(T)} D^{v(}_{B_i} \cdot D^{)v}_{B_i}$;
7  for component $B \in V$ do
8      $h^B \leftarrow ComputeTrafficMatrix$;
       $BC^B \leftarrow ComputeBetweenness(B, h^B)$; for
       $x \in B$ do
9          $BC(x) += BC^B(x)$;
10
11
```



(a) BCC partition before SE (b) BCC partition after SE

Fig. 5. Before and after unifying structurally equivalent vertices.

Lemma 4.2 (Inter-component betweenness computation): For each cut-point $v \in V$, the sum of the betweenness values of v computed "locally" on all $B \ni v$ minus the inter-component betweenness of v is equal to the betweenness of v in the original graph G:

$$BC(v) = \sum_{B \ni v} BC(v)_{v\in B} - BC^{inter}(v)$$

V. Combining speedup heuristics

In this section we demonstrate how the heuristics for speeding up the BC computation can be combined in order to produce an even more efficient algorithm. Although the marginal contribution of an additional heuristic is lower than its speedup, as will be described in Section VI, the overall effect is significant. A graph with many structurally equivalent vertices forms a graph with few cut points, and thus few bi-connected components (see Fig. 5). Merging the structurally equivalent vertices can potentially produce more cut-points. Thus, combining BCC and the SE speedup heuristics may further speed up the computation of BC.

The first step in achieving the speedup heuristics is reducing the network size by contracting the structural equivalence classes into single vertices with multiplicity weights, as described in Section III. Next, the resulting network is partitioned into a set of bi-connected components. In contrast to Section IV-A where the vertices have no multiplicity, here each vertex v represents an equivalence class and the multiplicity of v, which can be denoted as $|v|$, equals the size of the respective equivalence class.

To maintain the compatibility of the BC computation with unified graphs, equations 12 – 17 must support the multiplicity of vertices. The multiplicity of the vertices must be accounted for during all computations. When considering a cutoff vertex v, we consider it not as one vertex but as a multitude of $|v|$ vertices; when computing the size of a component B, we do not consider its cardinality ($|B|$) but rather $\sum_{v\in B}|v|$. Hence the updated size of the group $B^+(v)$ is:

$$D^{v(}_{B} = \sum_{x\in B^+(v)} |x| \tag{20}$$

Similarly, the size of $V - B^+(v) - v$ is:

$$D^{)v}_{B} = \sum_{x\in V-B^+(v)-v} |x| \tag{21}$$

Equation 13 is replaced by the following equation, which considers the multiplicity of v:

$$|V| = |v| + \sum_{B':v\in B'} D^{v(}_{B'} \tag{22}$$

Obviously these updates lead to modifications in the computation of the component-tree weights (see Algorithm 3).

We next examine the computation of the matrix h^i which represents the communication intensity between every two vertices x and y in a component B_i where x and y represent two different groups of structurally equivalent vertices. For vertex x (both cutoff and non-cutoff) $h^i_{xx} = h/C_{xx}$, where h/C_{xx} is defined by Equation 4. Assume x and y are both non-cutoffs, then $h^i_{xy} = h/C_{xy}$, as is defined by Equation 5. Having both x and y cutoffs, the following definition of h^i_{xy} replaces Equation 17:

$$h^i_{xy} = h^i_{yx} = \left(D^{)x}_{B_i} + |x|\right) \cdot \left(D^{)y}_{B_i} + |y|\right) \tag{23}$$

If y is a cutoff and x is not, Equation 16 is replaced by:

$$h^i_{xy} = D^{)y}_{B_i} + |y| \tag{24}$$

Algorithm 4 presents the steps to combine both the network size reduction (i.e., SE) and network partitioning (i.e., BCC) speedup heuristics.

VI. Evaluation

In this section we compare the performance of state-of-the-art algorithm for computing betweenness (BC) [3] and the speedup heuristics presented in this paper, namely: unification of structurally equivalent vertices (SEBC); partitioning to bi-connected components (BCCBC); and combination thereof (SEBCCBC). All evaluated algorithms were implemented in Java and executed under Windows Server 2008 on a machine with Intel Xeon, E5410, 2.33GHz CPU and 64GB RAM. No parallelization was utilized during the execution and the

Algorithm 3: Compute Component Tree Weights

```
Input: $V(T)$ collection of bi-connected components and
       cut-points from a graph $G(V, E)$;
       $E(T) \subseteq (V(T))^2$ collection of unweighted
       undirected links.
Output: Weights $D_B^{v(}$ for all links $(B, v) \in E(T)$
Data: A queue $Q$
▶ initialization
while $Q$ is not empty do
    pair ← pull from $Q$;
    if pair is a component-vertex pair $(B, v)$ then
        $size \leftarrow |B| - |v|$;
        for $n \in \{n|(B, n) \in E(T) \wedge D_B^{n(} \neq unknown\}$
        do
            $size \leftarrow size + |V| - D_B^{n(}$;
        $D_B^{v(} \leftarrow size$;
        ▶ update $Q$
    if pair is a vertex-component pair $(v, B)$ then
        $size \leftarrow |v|$;
        for $N \in \{N|(N, v) \in E(T) \wedge D_N^{v(} \neq unknown\}$
        do
            $size \leftarrow size + D_N^{v(}$;
        $D_B^{v(} \leftarrow |V| - |v| - size$;
        ▶ update $Q$
```

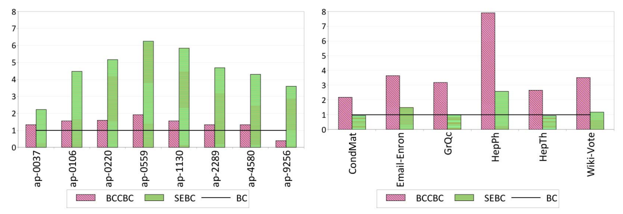
Algorithm 4: SE+(BCC+BC(TM))

```
Unify structurally equivalent vertices in the graph and
modify the traffic matrix accordingly;
Partition the network into bi-connected components ;
for each component $B$ do
    Compute traffic matrix for each component;
    Compute betweenness of all vertices in $B$ with
    respect to the modified traffic matrix;
```

highest reported memory consumption was lower than 10GB. We also note that the same implementation of the state-of-the-art BC computation was used as the baseline and within the different heuristics.

The evaluation was performed on a set of social and computer networks, summarized in Table I. There are four sets of computer and communication networks: Autonomous Systems (AS) topology of the Internet (as-rel) from CAIDA.ORG for the period from January 2004 till December 2009 [22]; AS peering information inferred from Oregon route-views (oregon1); AS peering information from Oregon route-views, Looking glass data, and Routing registry (oregon2) [23]; and peer-to-pear network of Gnutella hosts (p2p-Gnutella) [24].

We have extracted a set of author-to-paper networks from DBLP records (dblp-ap). The size of the networks range from 37 authors, 91 papers, and 275 links to 9256 authors, 42956 papers, and 110731 links. The networks were built in a breadth first fashion while extracting only those papers that connect at least two scientists in the network.



(a) DBLP (Author-to-Paper) (b) Social and Co-Authorship

Fig. 6. Speedup factor comparison for various networks.

Four Arxiv co-authorship networks were evaluated as well: Condensed Matter (CondMat), General Relativity (GrQc), High Energy Physics (HepPh), and High Energy Physics Theory (HepTh) [25]. A network of Wikipedia voters (Wiki-Vote) [26] and an email network (Email-Enron) [27] conclude the collection of evaluated networks. All networks except as-rel and delp-ap were obtained through the Stanford Network Analysis Project[1] (SNAP).

TABLE I
NETWORK DATA SETS

Label	Vertices	Links	Instances
as-rel	16K-33K	33K-75K	186
oregon1	10K-11K	22K-23K	9
oregon2	11K	30K-33K	9
p2p-Gnutella	6K-63K	20K-146K	9
dblp-ap	128-52K	275-110K	8
GrQc	5K	14K	1
CondMat	23K	93K	1
HepTh	10K	26K	1
HepPh	12K	119K	1
Wiki-Vote	7K	101K	1
Email-Enron	37K	184K	1

Fig. 6(a) presents the evaluation results on author-to-paper networks extracted from DBLP records. BC computation without speedup took approximately 1.6 hours on the network containing 9K authors (52K vertices). We can see that the SEBC computation results in a significant speedup on all the evaluated instances of the DBLP network. The primary reason for the efficiency of this heuristic is the large number of papers published by the same sets of authors. As evidence, the number of structural equivalence classes is, on average, almost two times (1.86) lower than the number of vertices in the network.

In contrast to SEBC, which performs efficiently on the author-to-paper network, the BCCBC heuristic achieves a speedup factor of at most 1.92. In one case (ap-9256) the overhead of using BCCBC is even higher than the speedup achieved by partitioning. The primary reason for the deficiency of BCCBC in the author-to-paper networks is the existence of a huge bi-connected component which, on average, contains as much as 90% of the vertices and 93% of the links.

Though the SEBC seems to be significantly better than the BCCBC in the author-to-paper networks, this is not the case in author-to-author and other social networks. As depicted in Fig.

[1]http://snap.stanford.edu/index.html

6(b), partitioning such networks to bi-connected components can speedup the BC computation up to 7.9 times, however unifying structurally equivalent vertices is by far less efficient. For CondMat and HepTh networks the overhead of using the SEBC is higher than the speedup gained by the graph size reduction, resulting in a slightly longer running time than the state-of-the-art algorithm. In social and co-authorship networks individuals are not expected to have the same set of peers, however, they might form loosely connected communities. Therefore, it is preferred to compute betweenness using the BCCBC algorithm in these networks.

Fig. 7 presents the evaluation results on the computer and communication networks as-rel, oregon1, oregon2, and p2p-Gnutella. The running time of BC computation using the BCCBC speedup heuristic is 2-3 times lower than of the state of the art BC computation and is more than four times lower for some AS relationship networks. In contrast to the BCCBC, the SEBC speedup heuristic has a different efficiency for the AS level topology of the Internet and for the Gnutella peer-to-peer network. We expect the SEBC heuristic to perform well for communication networks where redundancy and failover are important design principles. While in as-rel, oregon1, and oregon2, networks SEBC is as efficient as BCCBC, in p2p-Gnutella, the speedup factor of SEBC drops to an average of 1.14. We attribute this significant difference between effectiveness of SEBC and BCCBC to irregularity of ties in p2p-Gnutella. Yet a 14% speedup rate results in an average of 40 minutes of running time reduction per network instance.

Additionally, we observe that in as-rel, oregon1, and oregon2 networks the combination of SE and BCC speedup heuristics, denoted as SEBCCBC, results in yet another significant boost to the speed of the betweenness computation. Combined heuristics achieve a speedup factor of up to 7.3 on some instances of the AS relationships networks. Overall, for the different AS networks, the betweenness computation speedup produced by combining the BCCBC and the SEBC heuristics is almost twice as high as the speedup produced by each of them individually.

For networks where either one of the speedup heuristic does not perform well, the speedup of their combination is lower than the speedup of the better of the heuristics. For example, in Fig. 7 we can see that for p2p-Gnutella, the SEBCCBC heuristic performs worse than the BCCBC heuristic alone. Similar results were obtained for social and co-authorship networks, as well.

Lastly, though not reported in this paper, we have evaluated the speedup heuristics on a set of randomly generated artificial networks. All speedup heuristics perform well on sparse preferential attachment [28] and random networks [29] with an average degree close to 2.2. In fact, on such networks, the speedup factor grows with the size of the network suggesting that utilization of the speedup heuristics reduces the running time complexity of the betweenness computation.

Space Complexity. The space complexity of the SE and BCC heuristics are the same as the space complexity of the original algorithm for BC computation — linear in the size of the graph. However, both heuristics impose different space overheads during the BC computation. The main data structures used by SE heuristic are the unified graph and the unified traffic matrix. There is no need to store the unified traffic matrix, since every cell in the matrix is accessed only once and can be computed on the fly. Therefore, unified traffic matrix does not impose any space overhead. During the construction of the unified graph both the original and the unified graphs should be maintained, which may double the space consumption. However, the original graph can be discarded once the unified graph is created. Due to graph size reduction the number of nodes and links in the unified graph is smaller than in the original one.

BCC heuristic requires storing the bi-connected components, the block-cutpoint tree and the traffic matrices of each of the bi-connected components in order to compute BC. Since the number of vertices in the original graph (n) is an upper bound for the number of bi-connected components, the worst case space complexity of the block-cutpoint tree is $O(n)$. The space overhead imposed by traffic matrices used for BC computation within each bi-connected component can be avoided without compromising the time complexity of the BC computation. When computing the local betweenness values, every cell in the bi-connected component's traffic matrix is accessed only once, hence it can be computed on the fly based on the block-cutpoint tree link weights.

VII. Conclusions

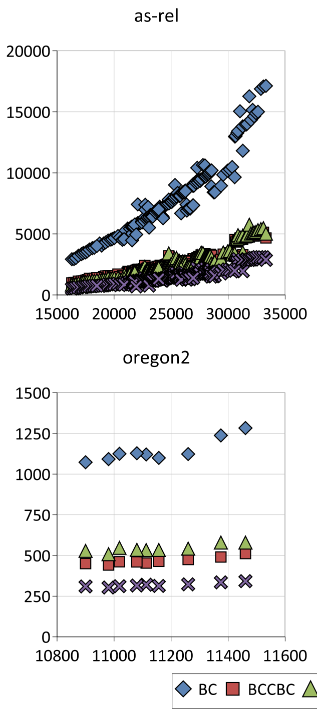
In this paper we presented heuristics for speeding up the exact sequential computation of betweenness centrality. The discussed heuristics are based on graph size reduction, graph partitioning, or combinations thereof. We report speedup factors of 7.9 on the co-authorship network extracted from Arxiv High Energy Physics and up to 7.3 (with average of 5.34) on CAIDA AS relationship networks. Exact Betweenness Centrality of all vertices can be computed at least twice as fast as using state-of-the-art sequential algorithm on all evaluated networks, without compromising space complexity.

Unifying structurally equivalent vertices appeared to be useless for social networks were one cannot expect two unconnected individuals to have exactly the same set of friends. However, two individuals may have almost similar sets of friends. Future work may trade BC computation accuracy by unifying vertices that are almost structurally equivalent.

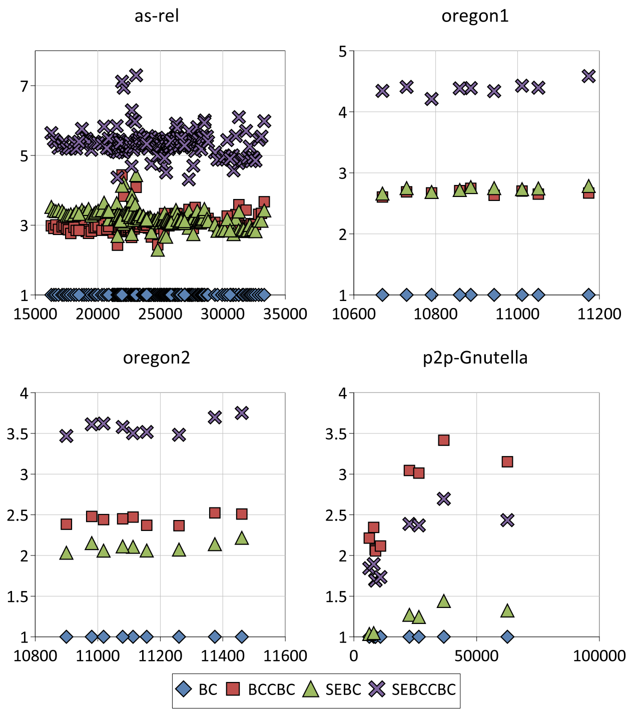
Since all presented computation methods are primarily based on the state-of-the-art algorithm, they are applicable to other derivations of this algorithm. For example, graph size reduction and graph partitioning can be used to further speedup betweenness approximation algorithms [9] or parallel algorithms of betweenness computation [4]. We also believe that network researchers will find the presented speedup heuristics useful for computing other centrality measures as well.

References

[1] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.



(a) time



(b) speedup

Fig. 7. Running time and speedup factor comparison. X-axis represents the number of vertices. Y-axis represents the running time in seconds (a) and the speedup factor (b).

[2] J. M. Anthonisse. The rush in a directed graph. Technical Report BN 9/71, Stichting Mathematisch Centrum, Amsterdam, 1971.

[3] U. Brandes. A faster algorithm for betweenness centrality. *Mathematical Sociology*, 25(2):163–177, 2001.

[4] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, 2009.

[5] G. Tan, D. Tu, and N. Sun. A parallel algorithm for computing betweenness centrality. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 340–347. IEEE, 2009.

[6] N. Edmonds, T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[7] Zhiao Shi and Bing Zhang. Fast network centrality analysis using GPUs. *BMC Bioinformatics*, 12(1):149, 2011.

[8] U. Brandes and C. Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos in Applied Sciences and Engineering*, 17(7):2303, 2007.

[9] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *Proceedings of the 5th international conference on Algorithms and models for the web-graph*, WAW'07, pages 124–137, San Diego, CA, USA, 2007.

[10] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2008.

[11] Jürgen Pfeffer and Kathleen M. Carley. k-Centralities: Local Approximations of Global Measures Based on Shortest Paths. In *WWW 2012 LSNA'12 Workshop*, Lyon, France, 2012.

[12] U. Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2):136–145, 2008.

[13] R. Puzis, M. D. Klippel, Y. Elovici, and S. Dolev. Optimization of nids placement for protection of intercommunicating critical infrastructures. In *EuroISI*, 2007.

[14] S. Dolev, Y. Elovici, and R. Puzis. Routing betweenness centrality. *Journal of the ACM*, 57(4):25:1–25:27, 2010.

[15] Structural equivalence: Meaning and definition, computation and application. *Social Networks*, 1(1):73 – 90, 1978.

[16] J. Lerner. *Network analysis: methodological foundations*, chapter Role Assignments. Springer LNCS 3418, 2005.

[17] M. G. Everett and S. P. Borgatti. The centrality of groups and classes. *Mathematical Sociology*, 23(3):181–201, 1999.

[18] R. Puzis, Y. Elovici, and S. Dolev. Fast algorithm for successive computation of group betweenness centrality. *Phys. Rev. E*, 76(5):056709, 2007.

[19] D. B. West. *Introduction to Graph Theory*. Prentice Hall, Upper Saddle River, NJ 07458, ii edition, 2001.

[20] M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley and Sons, 2002.

[21] H.N. Gabow. *Searching (Ch 10.1)*, pages 953–984. CRC Press, 2003.

[22] 2004-2009. The CAIDA AS Relationships Dataset. http://www.caida.org/data/active/as-relationships/.

[23] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2005.

[24] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[25] J. Kleinberg J. Leskovec and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (ACM TKDD)*, 1(1):Art. 2, 2007.

[26] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Signed networks in social media. In *CHI*, 2010.

[27] Y. Yang B. Klimmt. Introducing the enron corpus. In *CEAS conference*, 2004.

[28] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[29] P. Erdös and P. Rényi. On random graphs. *Publicationes Mathematicae*, 6:290–297, 1959.