Shattering and Compressing Networks for Betweenness Centrality*

Ahmet Erdem Sarıyüce[1,2], Erik Saule[1], Kamer Kaya[1], and Ümit V. Çatalyürek[1,3]

Depts. [1]Biomedical Informatics, [2]Computer Science and Engineering, [3]Electrical and Computer Engineering
The Ohio State University
Email:{*aerdem,esaule,kamer,umit*}*@bmi.osu.edu*

Abstract

The betweenness metric has always been intriguing and used in many analyses. Yet, it is one of the most computationally expensive kernels in graph mining. For that reason, making betweenness centrality computations faster is an important and well-studied problem. In this work, we propose the framework, **BADIOS**, which compresses a network and shatters it into pieces so that the centrality computation can be handled independently for each piece. Although **BADIOS** is designed and tuned for betweenness centrality, it can easily be adapted for other centrality metrics. Experimental results show that the proposed techniques can be a great arsenal to reduce the centrality computation time for various types and sizes of networks. In particular, it reduces the computation time of a 4.6 million edges graph from more than 5 days to less than 16 hours.

Keywords: Betweenness centrality; network analysis; graph mining

1 Introduction

Centrality metrics play an important role while detecting the central and influential nodes in various types of networks such as social networks [18], biological networks [15], power networks [12], covert networks [16] and decision/action networks [6]. The *betweenness* metric has always been an intriguing one and has been implemented in several tools which are widely used in practice for analyzing networks and graphs [19]. In short, the betweenness centrality (BC) score of a node is the sum of the fractions of the shortest paths between node pairs that pass through the node of interest [8]. Hence, it is a measure of the contribution/load/influence/effectiveness of a node while disseminating information through a network.

Although BC has been proved to be successful for network analysis, computing the centrality scores of all the nodes in a network is expensive. Brandes proposed an algorithm for computing BC with $\mathcal{O}(nm)$ and $\mathcal{O}(nm + n^2 \log n)$ time complexity and $\mathcal{O}(n + m)$ space complexity for unweighted and weighted networks, respectively, where n is the number of nodes in the network and m is the number of node-node interactions in the network [2]. Brandes' algorithm is currently the best algorithm for BC computations and it is unlikely that general algorithms with better asymptotic complexity can be designed [14]. However, it is not fast enough to handle Facebook's billion or Twitter's 200 million users.

In this work, we propose the **BADIOS** framework which uses a set of techniques (based on **B**ridges, **A**rticulation, **D**egree-1, and **I**dentical vertices, **O**rdering, and **S**ide vertices) for faster betweenness centrality computation. The framework shatters the network and reduces its size so that the BC scores of the nodes in two different pieces of network can be computed correctly and independently, and hence, in a more efficient manner. It also preorders the graph to improve cache utilization. The source code of **BADIOS**[1] and a technical report [22] are available.

For the sake of simplicity, we consider only standard, shortest-path vertex-betweenness centrality on undirected unweighted graph. However, our techniques can be used for other path-based centrality metrics such as *closeness*, or other BC variants, e.g., *edge* and *group betweenness* [3]. **BADIOS** also applies to weighted and/or directed networks. And all the techniques are compatible with previously proposed approximation and parallelization of the BC computation.

We apply **BADIOS** on a popular set of graphs with sizes ranging from 6K edges to 4.6M edges. We show an average speedup 2.8 on small graphs and 3.8 on large ones. In particular, for the largest graph we use, with 2.3M vertices and 4.6M edges, the computation time is reduced from more than 5 days to less than 16 hours.

*This work was partially supported by the U.S. Department of Energy SciDAC Grant DE-FC02-06ER2775 and NSF grants CNS-0643969, OCI-0904809 and OCI-0904802.

[1] http://bmi.osu.edu/hpc/software/badios/

The rest of the paper is organized as follows: In Section 2, an algorithmic background for BC is given. The shattering and compression techniques are explained in Section 3. Section 4 gives experimental results on various kinds of networks. We give the related work in Section 5 and conclude the paper with Section 6.

2 Background

Let $G = (V, E)$ be a network modeled as a simple graph with $n = |V|$ vertices and $m = |E|$ edges where each node is represented by a vertex in V, and a node-node interaction is represented by an edge in E. Let $\Gamma(v)$ be the set of vertices which are connected to v.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A *path* is a sequence of vertices such that there exists an edge between consecutive vertices. A path between two vertices s and t is denoted by $s \rightsquigarrow t$. Two vertices $u, v \in V$ are *connected* if there is a path from u to v. If all vertex pairs are connected we say that G is *connected*. If G is not connected, then it is *disconnected* and each maximal connected subgraph of G is a *connected component*, or a component, of G.

Given a graph $G = (V, E)$, an edge $e \in E$ is a *bridge* if $G - e$ has more connected components than G, where $G - e$ is obtained by removing e from E. Similarly, a vertex $v \in V$ is called an *articulation vertex* if $G - v$ has more connected components than G, where $G - v$ is obtained by removing v and its adjacent edges from V and E, respectively. G is *biconnected* if it is connected and it does not contain an articulation vertex. A maximal biconnected subgraph of G is a *biconnected component*: if G is biconnected it has only one biconnected component, which is G itself.

$G = (V, E)$ is a *clique* if and only if $\forall u, v \in V$, $\{u, v\} \in E$. The subgraph *induced by* a subset of vertices $V' \subseteq V$ is $G' = (V', E' = \{V' \times V'\} \cap E)$. A vertex $v \in V$ is a *side vertex* of G if and only if the subgraph of G induced by $\Gamma(v)$ is a clique. Two vertices u and v are *identical* if and only if **either** $\Gamma(u) = \Gamma(v)$ (type-I) **or** $\{u\} \cup \Gamma(u) = \{v\} \cup \Gamma(v)$ (type-II). A vertex v is a *degree-1* vertex if and only if $|\Gamma(v)| = 1$.

2.1 Betweenness Centrality: Given a connected graph G, let σ_{st} be the number of shortest paths from a source $s \in V$ to a target $t \in V$. Let $\sigma_{st}(v)$ be the number of such $s \rightsquigarrow t$ paths passing through a vertex $v \in V$, $v \neq s, t$. Let the *pair dependency* of v to s, t pair be the fraction $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. The betweenness centrality of v is defined by

$$\texttt{bc}[v] = \sum_{s \neq v \neq t \in V} \delta_{st}(v). \tag{2.1}$$

Since there are $\mathcal{O}(n^2)$ pairs in V, one needs $\mathcal{O}(n^3)$ operations to compute $\texttt{bc}[v]$ for all $v \in V$ by using (2.1). Brandes reduced this complexity and proposed an $\mathcal{O}(mn)$ algorithm for unweighted networks [2]. The algorithm is based on the accumulation of pair dependencies over target vertices. After accumulation, the dependency of v to $s \in V$ is

$$\delta_s(v) = \sum_{t \in V} \delta_{st}(v). \tag{2.2}$$

Let $\texttt{P}_s(u)$ be the set of u's predecessors on the shortest paths from s to all vertices in V. That is,

$$\texttt{P}_s(u) = \{v \in V : \{u, v\} \in E, \texttt{d}_s(u) = \texttt{d}_s(v) + 1\}$$

where $\texttt{d}_s(u)$ and $\texttt{d}_s(v)$ are the shortest distances from s to u and v, respectively. $\texttt{P}_s$ defines the *shortest paths graph* rooted in s. Brandes observed that the accumulated dependency values can be computed recursively:

$$\delta_s(v) = \sum_{u : v \in \texttt{P}_s(u)} \frac{\sigma_{sv}}{\sigma_{su}} \times (1 + \delta_s(u)). \tag{2.3}$$

To compute $\delta_s(v)$ for all $v \in V \setminus \{s\}$, Brandes' algorithm uses a two-phase approach (Algorithm 1). First, a breadth first search (BFS) is initiated from s to compute σ_{sv} and $\texttt{P}_s(v)$ for each v. Then, in a *back propagation* phase, $\delta_s(v)$ is computed for all $v \in V$ in a bottom-up manner by using (2.3). Each phase considers all the edges at most once, taking $\mathcal{O}(m)$ time. The phases are repeated for each source vertex. The overall complexity is $\mathcal{O}(mn)$.

3 Shattering and Compressing Networks

BADIOS uses bridges and articulation vertices for shattering graphs. These structures are important since for many vertex pairs s, t, all $s \rightsquigarrow t$ (shortest) paths are passing through them. It also uses three *compression* techniques, based on removing degree-1, side, and identical vertices from the graph. These vertices have special properties: The BC score of each degree-1 and side vertex is 0, since they cannot be on a shortest path unless they are one of the endpoints. And when u and v are identical, $\texttt{bc}[u]$ and $\texttt{bc}[v]$ are equal. A toy graph and a basic shattering/compression process via **BADIOS** is given in Figure 1.

Exploiting the existence of above mentioned structures on BC computations can be crucial. For example, all non-leaf vertices in a binary tree $T = (V, E)$ are articulation vertices. When Brandes' algorithm is used, the complexity of BC computation is $\mathcal{O}(n^2)$. One can do much better: Since there is exactly one path between each vertex pair in V, for $v \in V$, $\texttt{bc}[v]$ is equal to the number of pairs communicating via v, i.e.,

Data: $G = (V, E)$
$\texttt{bc}[v] \leftarrow 0, \forall v \in V$
for each $s \in V$ **do**
  $S \leftarrow$ empty stack, $Q \leftarrow$ empty queue
  $\texttt{P}[v] \leftarrow$ empty list, $\sigma[v] \leftarrow 0, \texttt{d}[v] \leftarrow -1, \forall v \in V$
  Q.push(s), $\sigma[s] \leftarrow 1$, $\texttt{d}[s] \leftarrow 0$
  ▷`Phase 1: BFS from` s
  while Q *is not empty* **do**
    $v \leftarrow Q$.pop(), S.push(v)
    for all $w \in \Gamma(v)$ **do**
      if $\texttt{d}[w] < 0$ **then**
        Q.push(w)
        $\texttt{d}[w] \leftarrow \texttt{d}[v] + 1$
      if $\texttt{d}[w] = \texttt{d}[v] + 1$ **then**
1        $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
        $\texttt{P}[w]$.push(v)
  ▷`Phase 2: Back propagation`
  $\delta[v] \leftarrow \frac{1}{\sigma[v]}, \forall v \in V$
  while S *is not empty* **do**
    $w \leftarrow S$.pop()
    for $v \in P[w]$ **do**
2      $\delta[v] \leftarrow \delta[v] + \delta[w]$
    if $w \neq s$ **then**
3      $\texttt{bc}[w] \leftarrow \texttt{bc}[w] + (\delta[w] \times \sigma[w] - 1)$
return bc

Algorithm 1: Bc-Org

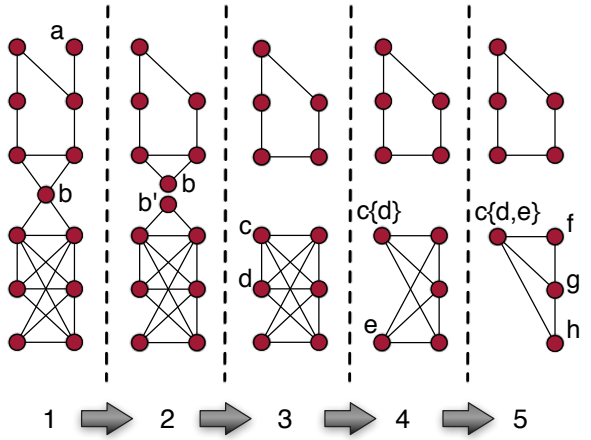


Figure 1: **(1) a is a degree-1 vertex and b is an articulation vertex. The framework removes a and create a copy b' to represent b in the bottom component. (2) There is no degree-1, articulation, or identical vertex, or a bridge. Vertices b and b' are now side vertices and they are removed. (3) Vertex c and d are now type-II identical vertices: d is removed, and c is kept. (4) Vertex c and e are now type-I identical vertices: e is removed, and c is kept. (5) Vertices c and g are type-II identical vertices and f and h are now type-I. The last reductions are not shown but the bottom component is compressed to a singleton vertex. The 5-cycle above cannot be reduced.**

$\texttt{bc}[v] = 2\times((l_v r_v) + (n - l_v - r_v - 1)(l_v + r_v))$ where l_v and r_v are the number of vertices in the left and right subtrees of v, respectively. This approach takes only $\mathcal{O}(n)$ time. A similar argument can be given for cliques since every vertex is a side vertex and has a 0 BC score.

As shown in Figure 1, **BADIOS** applies a series of operations as a preprocessing phase: Let $G = G_0$ be the initial graph, and G_ℓ be the one after the ℓth shattering/compression operation. The $\ell + 1$th operation modifies a single connected component of G_ℓ and generates $G_{\ell+1}$. The preprocessing continues if $G_{\ell+1}$ is amenable to further modification. Otherwise, it terminates and the final BC computation phase of the framework begins.

3.1 Shattering Graphs: Let $G = (V, E)$ be the original graph. For simplicity, we assume that G is connected. To correctly compute the BC scores after shattering G, we assign a `reach` attribute to each vertex. Let v' be a vertex in C', a component in the shattered graph G': $\texttt{reach}[v']$ is the number of vertices in G which are only reachable from C' via v'. At the beginning, we set $\texttt{reach}[v] = 1$ for all $v \in V$.

3.1.1 Shattering with articulation vertices: Let u' be an articulation vertex in a component $C \subseteq G_\ell$ after the ℓth operation of the preprocessing phase. We first shatter C into k (connected) components C_i for $1 \leq i \leq k$ by removing u' from G_ℓ and adding a *local copy* u'_i of u' to each new component by connecting u'_i to the same vertices u was connected within C_i. The `reach` values for each local copy is set with

$$\texttt{reach}[u'_i] = \sum_{v' \in C \setminus C_i} \texttt{reach}[v'] \tag{3.4}$$

for $1 \leq i \leq k$. We will use $\mathbf{org}(v')$ to denote the mapping from V' to V, which maps a local copy $v' \in V'$ to the corresponding original copy in V.

At any time of the preprocessing phase, a vertex $s \in V$ has exactly one *representative* u' in each component C such that $\texttt{reach}[u']$ is incremented by one due to s. This vertex is denoted as $\mathbf{rep}(C, s)$. Note that each local copy is a representative of its original. Note also that, if $\mathbf{rep}(C, s) = u'$ and $\mathbf{rep}(C, t) = v'$ with $v' \neq u'$ then $\mathbf{org}(u')$ is on all $s \rightsquigarrow t$ paths in G.

Algorithm 2 computes the BC scores of the vertices in a shattered graph. Note that the only difference with Bc-Org are lines 1 and 3, and if $\texttt{reach}[v] = 1$ for all $v \in V$, then the algorithms are equivalent. Hence, the complexity of Bc-Reach is also $\mathcal{O}(mn)$ for a graph with n vertices and m edges.

Let $G = (V, E)$ be the initial graph, $|V| = n$, and $G' = (V', E')$ be the one shattered via preprocessing. Let bc and $\texttt{bc}'$ be the scores computed by Bc-Org(G) and Bc-Reach(G'), respectively. We will prove that

$$\texttt{bc}[v] = \sum_{v' \in V' | \mathbf{org}(v') = v} \texttt{bc}'[v'], \tag{3.5}$$

Data: $G' = (V', E')$ and `reach`
$\texttt{bc}'[v] \leftarrow 0, \forall v \in V'$
for each $s \in V'$ **do**
  $\cdots$ ▷same as Bc-Org
  while Q *is not empty* **do**
    $\cdots$ ▷same as Bc-Org
1 $\delta[v] \leftarrow \texttt{reach}[v] - 1, \forall v \in V'$
  while S *is not empty* **do**
    $w \leftarrow S$.pop()
    for $v \in \texttt{P}[w]$ **do**
2 $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \times (1 + \delta[w])$
    if $w \neq s$ **then**
3 $\texttt{bc}'[w] \leftarrow \texttt{bc}'[w] + (\texttt{reach}[s] \times \delta[w])$
return bc'

Algorithm 2: Bc-Reach

when the graph is shattered at articulation vertices. That is, $\texttt{bc}[v]$ is distributed to $\texttt{bc}'[v']$s where v' is a local copy of v. Let us start with two lemmas.

Lemma 3.1. *Let u, v, s be vertices of G such that all $s \rightsquigarrow v$ paths contain u. Then, $\delta_s(v) = \delta_u(v)$.*

Proof. For any target vertex t, if $\sigma_{st}(v)$ is positive then

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} = \frac{\sigma_{su}\sigma_{ut}(v)}{\sigma_{su}\sigma_{ut}} = \frac{\sigma_{ut}(v)}{\sigma_{ut}} = \delta_{ut}(v)$$

since all $s \rightsquigarrow t$ paths are passing through u. According to (2.2), $\delta_s(v) = \delta_u(v)$.■

Lemma 3.2. *For any vertex pair $s, t \in V$, there exists exactly one component C of G' which contains a copy of t and a representative of s as two distinct vertices.*

Proof. (*by induction*) Given $s, t \in V$, the statement is true for the initial (connected) graph G since it contains one copy of each vertex. Assume that it is also true after the ℓ-th shattering. Let C be this component. When C is further shattered via t's copy, all but one newly formed (sub)components contains a copy of t as the representative of s. For the remaining component C', $\mathbf{rep}(C', s) = \mathbf{rep}(C, s)$ which is not a copy of t.

For all components other than C, which contain a copy t' of t, the representative of s is t' by the inductive assumption. When such components are further shattered, the representative of s will be again a copy of t. Hence the statement is true for $G_{\ell+1}$, and by induction, also for G'.■

The local copies of an articulation vertex v, created while shattering, will take the role of v in their components. Once the `reach` value for each copy is set as in (3.4), line 1 of Bc-Reach handles the BC contributions from each new component (except the one containing the source), and line 3 of Bc-Reach fixes the contribution of vertices reachable only via the source s.

Theorem 3.1. *Eq. 3.5 is correct after shattering G with articulation vertices.*

Proof. Let C be a component of G', s', v' be two vertices in C, and s, v be their original vertices in V, respectively. Note that $\texttt{reach}[v'] - 1$ is the number of vertices $t \neq v$ such that t does not have a copy in C and v lies on all $s \rightsquigarrow t$ paths in G. For all such vertices, $\delta_{st}(v) = 1$, and the total dependency of v' to all such t is $\texttt{reach}[v'] - 1$. When the BFS is started from s', line 1 of Bc-Reach initiates $\delta[v']$ with this value and computes the final $\delta[v'] = \delta_{s'}(v')$. This is the same dependency $\delta_s(v)$ computed by Bc-Org.

Let C be a component of G', u' and v' be two vertices in C, and $u = \mathbf{org}(u')$, $v = \mathbf{org}(v')$. According to the above paragraph, $\delta_u(v) = \delta_{u'}(v')$ where $\delta_u(v)$ and $\delta_{u'}(v')$ are the dependencies computed by Bc-Org and Bc-Reach, respectively. Let $s \in V$ be a vertex, s.t. $\mathbf{rep}(C, s) = u'$. According to Lemma 3.1, $\delta_s(v) = \delta_u(v) = \delta_{u'}(v')$. Since there are $\texttt{reach}[u']$ vertices represented by u' in C, the contribution of the BFS from u' to the BC score of v' is $\texttt{reach}[u'] \times \delta_{u'}(v')$ as shown in line 3 of Bc-Reach. Furthermore, according to Lemma 3.2, $\delta_{s'}(v')$ will be added to exactly one copy v' of v. Hence, (3.5) is correct.■

3.1.2 Shattering with bridges: Although the existence of a bridge implies the existence of two articulation vertices, handling bridges are easier and only requires the removal of the bridge. We embed this operation to **BADIOS** as follows: Let G_ℓ be the shattered graph obtained after ℓ operations, and let $\{u', v'\}$ be a bridge in a component C of G_ℓ. Hence, u' and v' are both articulation vertices. Let $u = \mathbf{org}(u')$ and $v = \mathbf{org}(v')$. A bridge removal operation is similar to a shattering via an articulation vertex, however, no new copies of u or v are created. Instead, we let u' and v' act as a copy of v and u in the newly created components.

Let C_u and C_v be the components formed after removing edge $\{u', v'\}$ which contain u' and v', respectively. Similar to (3.4), we add $\sum_{w \in C_v} \texttt{reach}[w]$ and $\sum_{w \in C_u} \texttt{reach}[w]$ to $\texttt{reach}[u']$ and $\texttt{reach}[v']$, respectively, to make u' (v') the representative of all the vertices in C_v (C_u).

After a bridge removal, updating the `reach` values is not sufficient to make Lemma 3.2 correct. No component contains distinct representative of u (v) and copy of v (u) anymore. Hence, $\delta_v(u)$ and $\delta_u(v)$ will not be added to any copy of u and v, respectively, by Bc-Reach. But we can compute the difference and add

$$\delta_{v'}(u') = \left(\left(\sum_{w \in C_u} \texttt{reach}[w] \right) - 1 \right) \times \sum_{w \in C_v} \texttt{reach}[w],$$

to $\texttt{bc}'[u']$ and $\delta_{u'}(v')$ to $\texttt{bc}'[v']$, where $\delta_{u'}(v')$ is computed by interchanging u and v in the right side of the above equation. Note that Lemma 3.2 is correct for all other vertex pairs.

COROLLARY 1. *Eq. 3.5 is correct after shattering G with articulation vertices and bridges.*

3.2 Compressing Graphs: BADIOS's compression techniques aim to reduce the number of vertices and edges in the graph.

3.2.1 Compression with degree-1 vertices: Let G_ℓ be the graph after ℓ operations, and let $u' \in C$ be a degree-1 vertex in a component C of G_ℓ which is only connected to v'. Removing a degree-1 vertex from a graph is the same as removing the bridge $\{u', v'\}$ from G_ℓ. But this also reduces the number of vertices. Hence, we handle this case separately and set $G_{\ell+1} = G_\ell - u'$. The updates are the same as the ones for bridge removal. That is, we add $\texttt{reach}[u']$ to $\texttt{reach}[v']$ and increase $\texttt{bc}'[u']$ and $\texttt{bc}'[v']$, respectively, with

$$\delta_{v'}(u') = (\texttt{reach}[u'] - 1) \times \sum_{w \in C \setminus \{u'\}} \texttt{reach}[w],$$

$$\delta_{u'}(v') = \left(\left(\sum_{w \in C \setminus \{u'\}} \texttt{reach}[w] \right) - 1 \right) \times \texttt{reach}[u'].$$

COROLLARY 2. *Eq. 3.5 is correct after shattering G with articulation vertices and bridges, and compressing it with degree-1 vertices.*

3.2.2 Compression with identical vertices: If some vertices in G are identical their BC scores are the same. Hence, it is possible to combine these vertices and avoid extra computation. We use 2 types of identical vertices: Vertices u and v are type-I (or type-II) identical if and only if $\Gamma(u) = \Gamma(v)$ (or $\Gamma(u) \cup \{u\} = \Gamma(v) \cup \{v\}$). For the identical-vertex-based compression, we assign an $\texttt{ident}$ attribute to each vertex where $\texttt{ident}(v')$ denotes the number of vertices in G that are identical to v' in G'. Initially, $\texttt{ident}[v']$ is set to 1 for all $v \in V$.

The compression works as follows: Let $G_\ell = (V_\ell, E_\ell)$ be the graph after ℓ operations, and let $\mathcal{I} \subset V_\ell$ be a set of identical vertices. To obtain $G_{\ell+1}$, we remove all $u' \in \mathcal{I}$ from G_ℓ except one, which acts as a proxy for the others. Let $v' \in V_{\ell+1}$ be the proxy vertex. We increase $\texttt{ident}[v']$ by $\sum_{v'' \in \mathcal{I}, v'' \neq v'} \texttt{ident}[v'']$, and associate a list $\mathcal{I} \setminus \{v'\}$ with v'. The integration of the identical-vertex compression is realized in three modifications on Algorithm 1: During the first phase, line 1 is changed to $\sigma[w] \leftarrow \sigma[w] + \sigma[v] \times \texttt{ident}[v]$, since v can be a proxy for some vertices other than itself. Similarly, w can be a proxy, and line 2 is modified as $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \times (\delta[w] + 1) \times \texttt{ident}[w]$ to correctly simulate w's identical vertices. Finally, the source s can be a proxy, and the current BFS phase can be a representative for $\texttt{ident}[s]$ phases. To handle that, the BC updates at line 3 are changed to $\texttt{bc}'[w] \leftarrow \texttt{bc}'[w] + \texttt{ident}[s] \times \delta[w]$. The BC scores of all the vertices in $\mathcal{I}$ are equal.

The only paths ignored via these modifications are the paths between $u \in \mathcal{I}$ and $v \in \mathcal{I}$. If $\mathcal{I}$ is type-II the $u \rightsquigarrow v$ path contains a single edge and has no effect on dependency (and BC) values. However, if $\mathcal{I}$ is type-I, such paths have some impact. Fortunately, it only impacts the immediate neighbors' BC scores of $\mathcal{I}$. Since there are exactly $\sum_{u \in \mathcal{I}} (\texttt{ident}[u] (\sum_{v \in \mathcal{I}, u \neq v} \texttt{ident}[v]))$ such paths, this amount is equally distributed among the immediate neighbors of $\mathcal{I}$.

The technique presented in this section has been presented without taking the $\texttt{reach}$ attribute into account. Both attributes can be maintained simultaneously. The details are not presented here due to space limitation. The main challenge is to keep track of the BC of each identical vertex since they can differ if the reach value of the identical vertices are not equal to 1.

COROLLARY 3. *Eq. 3.5 is correct after shattering G with articulation vertices and bridges, and compressing it with degree-1, and identical vertices.*

3.2.3 Compression with side vertices: Let G_ℓ be the graph after ℓ operations, and let u' be a side vertex in a component C of G_ℓ. Since $\Gamma(u')$ is a clique, no shortest path is passing through u', i.e., u' is always on the sideways. Hence, we can remove u' from G_ℓ by only compensating the effect of the shortest $s' \rightsquigarrow t'$ paths where u' is either s' or t'. To do this, we initiate a BFS from u' similar to the one in BC-REACH. As Algorithm 3 shows, the only differences are two additional lines 1 and 2.

Data: $G_\ell = (V_\ell, E_\ell)$, a side vertex s, $\texttt{reach}$, and $\texttt{bc}'$
$\cdots$ ▷same as BC-REACH
while Q *is not empty* **do**
  $\cdots$ ▷same as the BFS in BC-REACH
$\delta[v] \leftarrow \texttt{reach}[v] - 1, \forall v \in V_\ell$
while S *is not empty* **do**
  $w \leftarrow S$.pop()
  for $v \in \texttt{P}[w]$ **do**
    $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
  if $w \neq s$ **then**
    $\texttt{bc}'[w] \leftarrow \texttt{bc}'[w] + (\texttt{reach}[s] \times \delta[w]) +$
      $(\texttt{reach}[s] \times (\delta[w] - (\texttt{reach}[w] - 1))$
1
2 $\texttt{bc}'[s] \leftarrow \texttt{bc}'[s] + (\texttt{reach}[s] - 1) \times \delta[s]$
return $\texttt{bc}'$

Algorithm 3: BFS-SIDE

Let v', w' be two vertices in C different than u', and v, w be their original vertices. Although both vertices will keep existing in $C - u'$, since u' will be removed, $\delta_{v'}(w')$ will be $\texttt{reach}[u'] \times \delta_{v'u'}(w')$ less than it should be. For all such v', the aggregated dependency will be

$$\sum_{v' \in C, v' \neq w'} \delta_{v'u'}(w') = \delta_{u'}(w') - (\texttt{reach}[w'] - 1),$$

since none of the $\texttt{reach}[w'] - 1$ vertices represented by w' lies on a $v' \rightsquigarrow u'$ path and $\delta_{v'u'}(w') = \delta_{u'v'}(w')$. The same dependency appears for all vertices represented by u'. Line 1 of BFS-SIDE takes into account all these dependencies.

Let $s \in V$ be a vertex s.t. $\textbf{rep}(C, s) = v' \neq u'$. When we remove u' from C, due to Lemma 3.2, $\delta_s(u) = \delta_{v'}(u')$ will not be added to any copy of u. Since, u' is a side vertex, $\delta_{v'}(u') = \texttt{reach}[u'] - 1$. Since there are $\sum_{v' \in C - u'} \texttt{reach}[v']$ vertices which are represented by a vertex in $C - u'$, we add

$$(\texttt{reach}[u'] - 1) \times \sum_{v' \in C - u'} \texttt{reach}[v']$$

to $\texttt{bc}'[u']$ after removing u' from C. Line 2 of BFS-SIDE compensates this loss.

Removing a single side vertex has a little impact of the overall time since BFS-SIDE is almost as expensive as BC-ORG for a given source. The main interest of side vertices removal is to discover new special vertices in the graph, which are cheaper to remove.

COROLLARY 4. *Eq. 3.5 is correct after shattering G with articulation vertices and bridges, and compressing it with degree-1, identical, and side vertices.*

3.3 Combination of Techniques: BADIOS

BADIOS's preprocessing phase is a loop where an iteration tries to shatter/compress the graph by the techniques until no reduction is possible. Indeed, a single iteration does not cover all the reduction possibilities, since each technique can make the graph amenable to another one. This is one of the novel features of the framework. More specifically, a degree-1 removal can create new degree-1, identical, and side vertices. Or, a shattering can reveal new degree-1 and side vertices. Similarly, by removing an identical vertex, new identical, degree-1, articulation, and side vertices can appear. And lastly, new identical and degree-1 vertices can be discovered when a side vertex is removed from the graph. Hence, a loop is necessary to fully exploit the reduction. **BADIOS** first applies degree-1 removal since it is the cheapest to handle. Next, it shatters the graph by first removing the bridges, and then articulation vertices. The order is important for efficiency because a bridge removal is cheaper than an articulation point removal. We then remove the identical vertices in the graph in the order of type-II and type-I. Notice that type-II removals can reveal new type-I identical vertices but the reverse is not possible. The framework iteratively uses these 4 techniques until it reaches a point where no reduction is possible. At that point, it removes the side vertices to discover new special vertices. The reason behind delaying the side-vertex removal is that it is expensive, i.e., an extra two-phase BFS is needed. Hence, **BADIOS** does not use side vertices until it really needs them.

3.4 Implementation Details:

Linear time algorithms for articulation vertex and bridge detection exist [24, 10]. **BADIOS** uses the algorithm in [10] to detect the articulation vertices and it decomposes the graph into its biconnected components at once. Although the final decomposition is the same when the graph is iteratively shattered one vertex at a time, a biconnected component decomposition is faster. A similar approach has also been employed for the bridges and they are removed at once.

For compression, detecting all degree-1 vertices in a single iteration takes $\mathcal{O}(m + n)$ time. Detecting identical vertices is also expected to be a linear-time process if for all $v \in V_\ell$, the hash of $\Gamma(v)$ is computed via a collision resistant function. In **BADIOS**, for all $v \in V_\ell$, we use $hash(v) = \sum_{u \in \Gamma(v)} u$. Upon collision of hash values, the neighborhood of the two vertices are explicitly compared.

To detect a side vertex v of degree k, we use a simple algorithm which verifies if the graph induced by $\Gamma(v)$ is a k-clique. **BADIOS** only searches for cliques of less than 5 vertices, since preliminary experiments showed that searching larger cliques is expensive to be practical. Similar to shattering, after detecting all vertices from a certain type, we apply a cumulative compression operation to remove all the detected vertices at once.

4 Experimental Results

We implemented **BADIOS** in `C++`. The code is compiled with `gcc v4.4.4` and optimization flags `-O2 -DNDEBUG`. The graph is kept in memory in the compressed row storage (CRS) format. The experiments are run on a computer with two Intel Xeon E5520 CPU clocked at 2.27GHz and equipped with 48GB of main memory. All the experiments are run sequentially.

For the experiments, we used 19 networks from the UFL Sparse Matrix Collection (`http://www.cise.ufl.edu/research/sparse/matrices/`). Their properties are summarized in Table 1. They are from different application areas, such as grid (*power*), router (*as-22july06*, *p2p-Gnutella31*), social (*hep-th*, *PGPgiantcompo*, *astro-ph*, *cond-mat-2005*, *soc-*

Graph			Time (in sec.)		
name	$\|V\|$	$\|E\|$	org.	best	Sp.
Power	4.9K	6.5K	1.47	0.60	2.4
Add32	4.9K	9.4K	1.50	0.19	7.6
HepTh	8.3K	15.7K	3.48	1.49	2.3
PGPgiant	10.6K	24.3K	10.99	1.55	7.0
ProtInt	9.6K	37.0K	11.76	7.33	1.6
AS0706	22.9K	48.4K	43.72	8.78	4.9
MemPlus	17.7K	54.1K	19.13	9.28	2.0
Luxemb.	114.5K	119.6K	771.47	444.98	1.7
AstroPh	16.7K	121.2K	40.56	19.41	2.0
Gnu31	62.5K	147.8K	422.09	188.14	2.2
CondM05	40.4K	175.6K	217.41	97.67	2.2
			geometric mean		**2.8**
Epinions	131K	711K	2,193	839	2.6
Gowalla	196K	950K	5,926	3,692	1.6
bcsstk32	44.6K	985K	687	41	16.5
NotreDame	325K	1,090K	7,365	965	7.6
RoadPA	1,088K	1,541K	116,412	71,792	1.6
Amazon0601	403K	2,443K	42,656	36,736	1.1
Google	875K	4,322K	153,274	27,581	5.5
WikiTalk	2,394K	4,659K	452,443	56,778	7.9
			geometric mean		**3.8**

Table 1: **The graphs used in the experiments. Column *org.* shows the original time of Bc-Org without any modification. And *best* is the minimum execution time achievable via BADIOS. The names of the matrices are kept short where the full names can be found in the text.**

sign-epinions, *loc-gowalla*, *amazon0601*, *wiki-Talk*), protein-interaction (*protein-interaction_1*), circuit simulation (*add32*, *memplus*), road (*luxemburg.osm*, *roadNet-PA*), auto (*bcsstk32*), and web networks (*web-NotreDame*, *web-Google*). We symmetrized the directed graphs.

4.1 Graph ordering: As most of the graph-based kernels in data mining, the order of the vertices and edges accessed by Brandes' algorithm is important due to cache locality. If two vertices in a graph are close to each other, a BFS will access them almost at the same time. Hence, if we put close vertices in G to close locations in memory, the number of cache misses are expected to decrease. For this reason, the framework initiates a BFS from a random vertex in G and uses the queue order of the vertices as their ordering in G. Further benefits of BFS ordering on the execution time of a graph-based kernel are explained in [5]. There are also some other graph ordering works in the literature [7, 13].

For each graph in our set, the first and second bars in Figure 2 show the time of Bc-Org with the natural and BFS vertex ordering, respectively. For 18/19 graphs, the BFS ordering improved the performance. It reduced the time by 14% on average and by 43% for *web-Google*. Hence a BFS ordering of the graph is usually preferable to the natural ordering of a real-life network for long graph mining kernels such as BC.

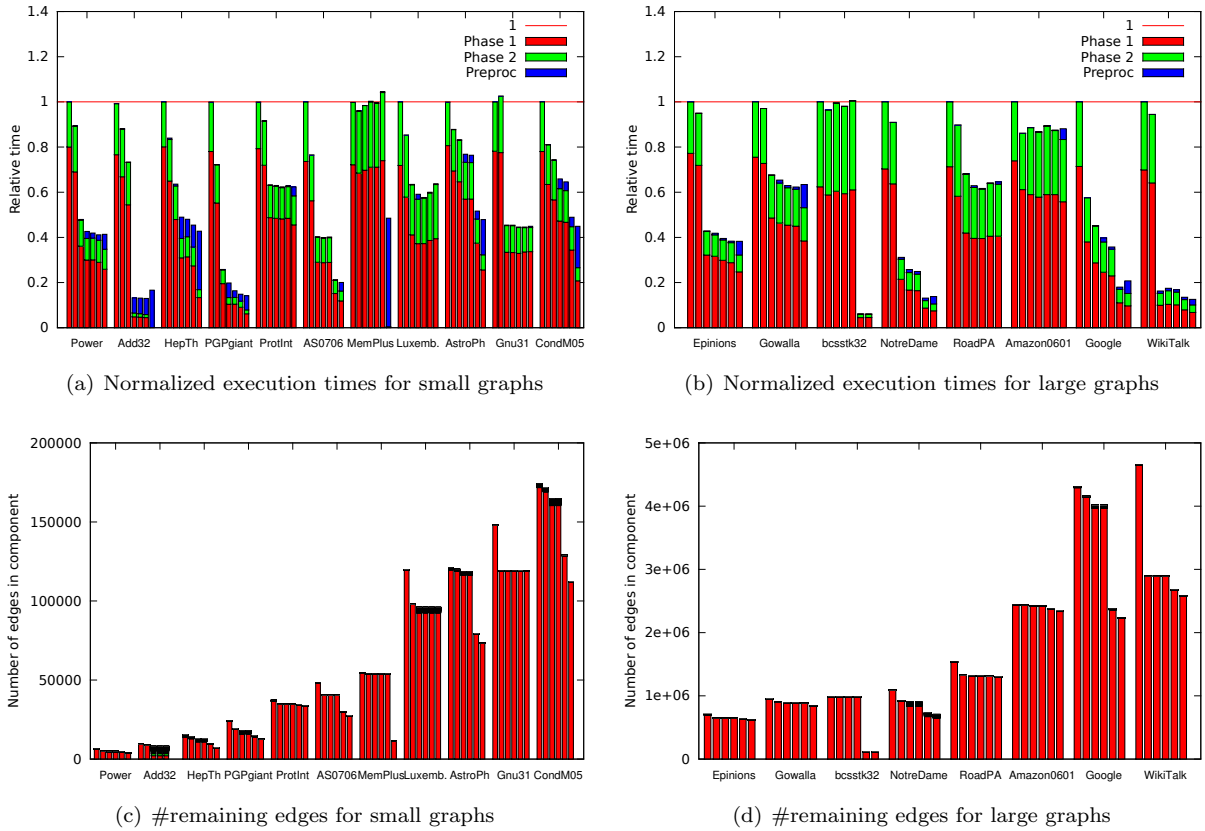
4.2 Shattering and compressing graphs: For each graph, we tested 7 different combinations of the improvements proposed in this paper: They are denoted with *o*, *do*, *dao*, *dbao*, *dbaio*, and *dbaiso*, where *'o'* is the BFS *o*rdering, *'d'* is *d*egree-1 vertices, *'b'* is *b*ridge, *'a'* is *a*rticulation vertices, *'i'* is *i*dentical vertices, and *'s'* is *s*ide vertices. The ordering of the letters denotes the order of techniques in the loop.

We measure the preprocessing time and BC computation time separately. Figures 2(a) and 2(b) presents the runtimes for each combination normalized w.r.t. Brandes' algorithm. For each graph, each figure has 7 stacked bars for the 7 combinations in the order described above. In Figures 2(c)–2(d), the number of edges remaining in the graph after the preprocessing phase are given for different combinations. In the figures, components are represented by different colors and 6 combinations are investigated for each graph (since ordering does not change the structure of the graph).

As Figure 2 shows, there is a direct correlation between the amount of edges in G' and the overall execution time. (Except for *soc-sign-epinions* and *loc-gowalla* where the improvement is correlated with a decrease in number of vertices, which are omitted for space constraint.) This proves that our rationale behind investigating shattering and compression techniques is valid. Yet, since red is almost always the dominating color, Figures 2(c)–2(d) show that real-life graphs do not contain *good* articulation vertices which allow shattering a graph into balanced sized components.

Table 1 shows the runtime of the base algorithm as well as the runtime of the combination that lead to the best improvement and the speedup obtained by that combination. Almost for all graphs, **BADIOS** provides a significant improvement. We observe up to 16.5 speedup on large graphs. For *wiki-Talk*, applying all techniques reduced the runtime from 5 days to 16 hours. Some of the techniques are shown to be very useful for some graphs. For example, the side-vertex removal enables a complete reduction for *memplus* and *add32*. On the other hand, for some of the graphs, e.g., *web-Google* and *web-NotreDame*, it increases the runtime by a small amount. As the figure shows, the identical-vertex removal technique is highly effective (see *bcsstk32*, *cond-mat-2005*, *as-22july06* or *astro-ph*). Also, as the results for *PGPgiantcompo* show, shattering via both bridges and articulation vertices is faster than shattering only via articulation vertices. Note that although both combinations result in the same graph, bridge removal is cheaper.

Although it is not that common, applying degree-1- and identical-vertex removal can degrade the performance by a small amount. When the number of vertices removed is small, their removal does not compensate the overhead induced by the `reach` and `ident` attributes



(a) Normalized execution times for small graphs

(b) Normalized execution times for large graphs

(c) #remaining edges for small graphs

(d) #remaining edges for large graphs

Figure 2: **The plots on the left and right show the results on graphs with less than and more than 500K edges, respectively. The top plots show the runtime of the variants: base, *o*, *do*, *dao*, *dbao*, *dbaio*, *dbaiso*. The times are normalized w.r.t. base and divided into three: preprocessing, the first phase and the second phase of the BC computation. The bottom plots show the number of edges in the largest 200 components after preprocessing.**

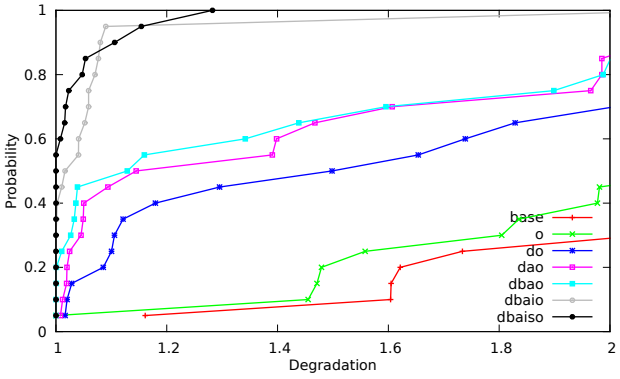


Figure 3: **Performance profile of the 7 combinations for BADIOS on all the selected graphs.**

in the algorithms. The only graph **BADIOS** does not perform well on is the co-purchasing network of Amazon website, *amazon0601*, where it brings less than 20% of improvement. This graph contains large cliques formed by the users purchasing the same item, and hence does not have enough number of special vertices.

The 7 combinations are compared with each other using a performance profile graph presented in Figure 3. A point (r, p) in the profile means that with p probability, the time of the corresponding combination on a graph G is at most r times worse than the best time obtained for that G. Hence, the closer to the y-axis is the better the combination is.

One can easily see that any parameter combination of **BADIOS** is better than the base algorithm. The combination with only graph ordering (o) has the worse performance profile of **BADIOS** and it is never optimal. According to the graph, most of the time, using all possible techniques is the best idea. This strategy is the optimal one with more than 60% probability. If only a little information is available *dbaiso* should be the default choice for **BADIOS**. However, given that preprocessing does not take too much time, one can run only the preprocessing first to get the amount of reduction obtained by each combination of parameters. Then, depending on that reduction, the best path can be selected. That way, the overhead induced of by the slight more expensive kernels can be avoided.

5 Related Work

Several techniques have been proposed to cope with large networks with limited success either by using approximate computations [4, 9], or by throwing hardware resources to the problem by parallelizing the computations on distributed memory architectures [17], multi-core CPUs [20], and GPUs [23, 11].

To the best of our knowledge, there are two concurrent works since our first release, noted in our technical report [22]. The first work introduces degree-1 vertex removal for BC [1]. In the second, Puzis et al. propose to remove articulation vertices and structurally equivalent vertices which correspond to our type-I identical vertices [21]. We did not compare our speedups with theirs for three reasons: the techniques they use form only a subset of the techniques we proposed in this work, they are not well integrated as we did in **BADIOS**, and even our base implementation is already 40–45 times faster than their fastest algorithm (see the results for *soc-sign-epinions* [1] and *p2p-Gnutella31 [21]*). We believe that an efficient implementation of a novel algorithm is mandatory to evaluate any improvement.

6 Conclusion

In this work, we proposed the **BADIOS** framework to reduce the execution time of betweenness centrality computations. It uses techniques that break graphs into pieces while keeping the information to recompute the pair and source dependencies which are the building blocks of BC scores. It also uses some compression techniques to reduce the number of vertices and edges. Combining these techniques provides great reductions in graph sizes and component numbers. An experimental evaluation with various networks shows that the proposed techniques are highly effective in practice and they can be a great arsenal to reduce the execution time for BC computation. For one of our social networks, we saved 4 days.

As a future work, we are planning to extend our techniques to other centrality measures such as closeness and group-betweenness. Some of our techniques can readily be extended for the weighted and directed graphs, but for some, a complete modification may be required. We will investigate these modifications. In addition, we are planning to adapt our techniques for parallel and/or approximate BC computations.

References

[1] M. Baglioni, F. Geraci, M. Pellegrini, and E. Lastres. Fast exact computation of betweenness centrality in social networks. In *ASONAM*, aug. 2012.

[2] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2), 2001.

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

[4] U. Brandes and C. Pich. Centrality estimation in large networks. *I. J. Bifurcation and Chaos*, 17(7), 2007.

[5] G. Cong and K. Makarychev. Optimizing large-scale graph analysis on a multi-threaded, multi-core platform. In *IPDPS'11*, 2011.

[6] Ö. Şimşek and A. G. Barto. Skill characterization based on betweenness. In *NIPS*, pp. 1497–1504, 2008.

[7] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. ACM national conference*, pp 157–172, 1969.

[8] L. Freeman. A set of measures of centrality based upon betweenness. *Sociometry*, 4:35–41, 1977.

[9] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, pp. 90–100, 2008.

[10] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[11] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart. Edge vs. node parallelism for graph centrality metrics. In W.-M. W. Hwu, editor, *GPU Computing Gems: Jade Edition*. pp. 15–28, 2011.

[12] S. Jin, Z. Huang, Y. Chen, D. Chavarria-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *IPDPS'10*, April 2010.

[13] U. Kang and C. Faloutsos. Beyond 'caveman communities': Hubs and spokes for graph compression and mining. In *ICDM*, 2011.

[14] S. Kintali. Betweenness centrality : Algorithms and lower bounds. *CoRR*, abs/0809.1906, 2008.

[15] D. Koschützki and F. Schreiber. Centrality analysis methods for biological networks and their application to gene regulatory networks. *Gene Regulation and Systems Biology*, 2, 2008.

[16] V. Krebs. Mapping networks of terrorist cells. *Connections*, 24, 2002.

[17] R. Lichtenwalter and N. V. Chawla. DisNet: A framework for distributed graph computation. In *ASONAM*, 2011.

[18] J.-K. Lou, S. d. Lin, K.-T. Chen, and C.-L. Lei. What can the temporal social behavior tell us? An estimation of vertex-betweenness using dynamic social information. In *ASONAM*, 2010.

[19] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *Proc. of SDM*, 2012.

[20] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDPS'09*, 2009.

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

[22] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Shattering and compressing networks for centrality analysis. (TR. arxiv 1209.6007), 2012.

[23] Z. Shi and B. Zhang. Fast network centrality analysis using GPUs. *BMC Bioinformatics*, 12:149, 2011.

[24] R. Tarjan. A note on finding the bridges of a graph. *Inf. Process. Lett.*, 2(6):160–161, 1974.