Shattering and Compressing Networks for Centrality Analysis

Ahmet Erdem Sarıyüce[1,2], Erik Saule[1], Kamer Kaya[1], Ümit V. Çatalyürek[1,3]
[1] Dept. Biomedical Informatics, The Ohio State University
[2] Dept. Computer Science and Engineering, The Ohio State University
[3] Dept. Electrical and Computer Engineering, The Ohio State University
Email: {aerdem,esaule,kamer,umit}@bmi.osu.edu

September 27, 2012

(Previously submitted to ICDM on June 18, 2012)

Abstract

Who is more important in a network? Who controls the flow between the nodes or whose contribution is significant for connections? Centrality metrics play an important role while answering these questions. The betweenness metric is useful for network analysis and implemented in various tools. Since it is one of the most computationally expensive kernels in graph mining, several techniques have been proposed for fast computation of betweenness centrality. In this work, we propose and investigate techniques which compress a network and shatter it into pieces so that the rest of the computation can be handled independently for each piece. Although we designed and tuned the shattering process for betweenness, it can be adapted for other centrality metrics in a straightforward manner. Experimental results show that the proposed techniques can be a great arsenal to reduce the centrality computation time for various types of networks.

Keywords: Betweenness centrality; network analysis; graph mining; connected components

1 Introduction

Centrality metrics play an important role to successfully detect the central nodes in various types of networks such as social networks [1, 2], biological networks [3, 4], power networks [5], covert networks [6] and decision/action networks [7]. Among these metrics, *betweenness* has always been an intriguing one and it has been implemented in several tools which are widely used in practice for analyzing networks and graphs [8, 9]. In short, the betweenness centrality (BC) score of a node is the sum of the fractions of shortest paths between node pairs that pass through the node of interest [10]. Hence, it is a measure for the contribution/load/influence/effectiveness of a node while disseminating information through a network.

Although betweenness centrality has been proved to be successful for network analysis, computing betweenness centrality scores of all the nodes in a network is expensive. The first trivial algorithms for BC have $\Theta(n^3)$ and $\Theta(n^2)$ time and space complexity, respectively, where n is the number of nodes in the network. Considering the size of today's networks, these algorithms are not practical. Brandes proposed a faster algorithm which has $\mathcal{O}(nm)$ and $\mathcal{O}(nm + n^2 \log n)$ time complexity for unweighted and weighted networks, respectively, where m is the number of node-node interactions in the network [11]. Since the networks in real life are usually *sparse*, $m \approx kn$ for a small k, $\mathcal{O}(nm)$ is much better than $\mathcal{O}(n^3)$. Brandes' algorithm also has a better, $\mathcal{O}(n+m)$, space complexity and currently, it is the best algorithm for BC computations. Yet, it

is not fast enough to handle almost 1 billion users of Facebook or 150 million users of Twitter. Several techniques have been proposed to alleviate the complexity of BC computation for large networks. A set of works propose using estimated values instead of exact BC scores [12, 13], and others parallelize BC computations on distributed memory architectures [14], multicore CPUs [3, 15, 16], and GPUs [17, 18, 19].

In this work, we propose a set of techniques which compress a network and break it into pieces such that the BC scores of two nodes in two different pieces can be computed independently, and hence, in a more efficient manner. Although we designed and tuned these techniques for standard, shortest-path vertex-betweenness centrality, they can be modified for other path-based centrality metrics such as *closeness* or other BC variants such as *edge betweenness* and *group betweenness* [20]. Similarly, although we are interested in unweighted undirected networks in this paper, our shattering techniques are valid also for weighted directed networks. Experimental results show that proposed techniques are very effective and they can be a great arsenal to reduce the computation in practice.

The rest of the paper is organized as follows: In Section 2, an algorithmic background for betweenness centrality is given. The proposed shattering and compression techniques are explained in Section 3. Section 4 gives experimental results on various kinds of networks, and Section 5 concludes the paper.

2 Background

Let $G = (V, E)$ be a network modeled as a graph with n vertices and m edges where each node in the network is represented by a vertex in V, and an interaction between two nodes is represented by an edge in E. We assume that $\{v, v\} \notin E$ for any $v \in V$, i.e., G is *loop* free. 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 vertex sequence 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 and v in V are *connected* if there is a path from u to v. If u and v are connected for all $u, v \in V$ we say 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 edges from V and E, respectively. If G is connected and it does not contain an articulation vertex we say G is *biconnected*. A maximal biconnected subgraph of G is a *biconnected component*. Hence, 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 $\Gamma(u) = \Gamma(v)$. v is a *degree-1* vertex if and only if $|\Gamma(v)| = 1$.

2.1 Betweenness Centrality

The betweenness metric is first defined by Freeman in Sociology to quantify a person's importance on other people's communication in a social network [10]. Given a 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 as

$$\mathtt{bc}[v] = \sum_{s \neq v \neq t \in V} \delta_{st}(v). \tag{1}$$

Since there are $\mathcal{O}(n^2)$ pairs in V, one needs $\mathcal{O}(n^3)$ operations to compute $\mathtt{bc}[v]$ for all $v \in V$ by using (1). Brandes reduced this complexity and proposed an $\mathcal{O}(mn)$ algorithm for unweighted networks [11]. The algorithm is based on the accumulation of pair dependencies over target vertices. After accumulation, the dependency of v to $s \in V$ is

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

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

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

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

$$\delta_s(v) = \sum_{u : v \in \mathtt{P}_s(u)} \frac{\sigma_{sv}}{\sigma_{su}} \left(1 + \delta_s(u)\right). \tag{3}$$

To compute $\delta_s(v)$ for all $v \in V \setminus \{s\}$, Brandes' algorithm uses a two-phase approach. First, to compute σ_{sv} and $\mathtt{P}_s(v)$ for each v, a breadth first search (BFS) is initiated from s. Then in a *back propagation* phase, $\delta_s(v)$ is computed for all $v \in V$ in a bottom-up manner by using (3). Each phase takes a linear time, and hence this process takes $\mathcal{O}(m)$ time. Since there are n source vertices and the phases are repeated for each source vertex, the total complexity of the algorithm is $\mathcal{O}(mn)$. The pseudo-code of Brandes' betweenness centrality algorithm is given in Algorithm 1.

3 Shattering and Compressing Networks

3.1 Principle

Let us start with a simple example: Let $G = (V, E)$ be a binary tree with n vertices hence $m = n - 1$. If Brandes' algorithm is used the complexity of computing the BC scores is $\mathcal{O}(n^2)$. However, by using a structural property of G, one can do much better: there is exactly one path between each vertex pair in V. Hence for a vertex $v \in V$, $\mathtt{bc}[v]$ is the number of (ordered) pairs communicating via v, i.e.,

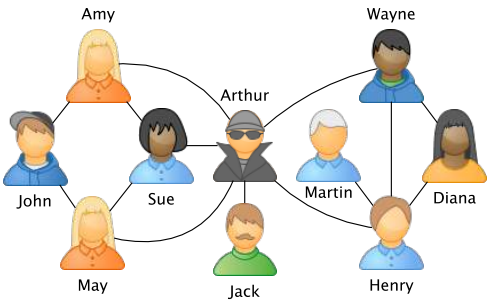
$$\mathtt{bc}[v] = 2 \times ((l_v r_v) + (n - l_v - r_v - 1)(l_v + r_v))$$

where l_v and r_v are the number of vertices in the left and the right subtrees of v, respectively. Since l_v and r_v can be computed in linear time for all $v \in V$, this approach, which can be easily extended to an arbitrary tree, takes only $\mathcal{O}(n)$ time.

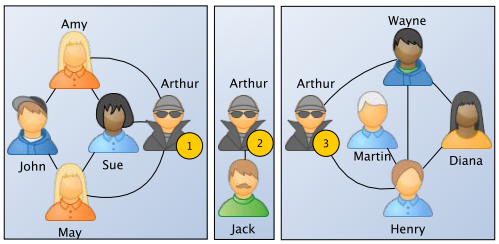
As mentioned in Section 1, computing BC scores is an expensive task. However, as the above example shows, some structural properties of the networks can be effectively used to reduce the complexity. Unfortunately, an n-fold improvement on the execution time is usually not possible since real-life networks rarely have a tree-like from. However, as we will show, it is still possible to reduce the execution time by using a set of special vertices and edges.

Algorithm 1: BC-ORG

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, $\forall v \in V$
  $\sigma[v] \leftarrow 0, \forall v \in V$
  $\texttt{d}[v] \leftarrow -1, \forall v \in V$
  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**
        $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
        $\texttt{P}[w]$.push(v)
  ▷`Phase 2: Back propagation`
  $\delta[v] \leftarrow 0, \forall v \in V$
  while S *is not empty* **do**
    $w \leftarrow S$.pop()
    for $v \in P[w]$ **do**
      $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
    if $w \neq s$ **then**
      $\texttt{bc}[w] \leftarrow \texttt{bc}[w] + \delta[w]$
return bc



(a) A toy social network with various types of vertices: Arthur is an articulation vertex, Diana is a side vertex, Jack and Martin are degree-1 vertices, and Amy and May are identical vertices.



(b) The network shattered at Arthur to three components.

Figure 1: A toy social network and its shattered form due to an articulation vertex.

Consider the toy graph G of a social network given in Figure 1.(a). Arthur is an articulation vertex in G and he is responsible from all inter-communications among three (biconnected) components as shown in Figure 1.(b). Let s and t be two vertices which lie in different components. For all such s, t pairs, the pair dependency of Arthur is 1. Since shattering the graph at Arthur removes all $s \rightsquigarrow t$ paths, one needs to keep some information to correctly update the BC scores of the vertices inside each component, and this can be achieved creating local copies of Arthur in each component.

In addition to shattering a graph G into pieces, we investigated three compression techniques using degree-1 vertices, side vertices, and identical vertices. These vertices have special properties: All degree-1 and side vertices always have a zero BC score since they cannot be on a shortest path unless they are one of the endpoints. Furthermore, $\texttt{bc}[u]$ is equal to $\texttt{bc}[v]$ for two identical vertices u and v. By using these observations, we will formally analyze the proposed shattering and compression techniques and provide formulas to compute the BC scores correctly.

We apply our techniques in a preprocessing phase as follows: Let $G = G_0$ be the initial graph, and G_ℓ be the graph after the ℓth shattering/compression operation. Without loss of generality, we assume that the initial graph G is connected. The $\ell + 1$th operation modifies a single connected component of G_ℓ and generates $G_{\ell+1}$. The preprocessing phase then checks if $G_{\ell+1}$ is amenable to further modification, and if this is the case, it continues. Otherwise, it terminates and the final BC computation begins.

3.2 Shattering Graphs

To correctly compute the BC scores after shattering a graph, we assign a `reach` attribute to each vertex. Let $G = (V, E)$. Let v' be a vertex in the shattered graph G' and C' be its component. Then $\texttt{reach}[v']$ is the number of vertices of G which are represented by v' in C'. For instance in Figure 1.(b), $\texttt{reach}[Arthur_3]$ is 6 since Amy, John, May, Sue, Jack, and Arthur have the same shortest path graphs in the right component. At the beginning, we set $\texttt{reach}[v] = 1$ for all $v \in V$.

3.2.1 Shattering with articulation vertices

Let u' be an articulation vertex detected in a connected 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 component by connecting it to the same vertices u was connected. The `reach` values for each local copy is set as

$$\texttt{reach}[u'_i] = \sum_{v' \in C \setminus C_i} \texttt{reach}[v'] \tag{4}$$

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.

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

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

Algorithm 2: BC-REACH

```
Data: G' = (V', E') and reach
bc'[v] ← 0, ∀v ∈ V'
for each s ∈ V' do
    ··· ▷same as Bc-Org
    while Q is not empty do
        ··· ▷same as Bc-Org
1   δ[v] ← reach[v] − 1, ∀v ∈ V'
    while S is not empty do
        w ← S.pop()
        for v ∈ P[w] do
2           δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w])
        if w ≠ s then
3           bc'[w] ← bc'[w] + (reach[s] * δ[w])
return bc'
```

Let $G = (V, E)$ be the initial graph with n vertices and $G' = (V', E')$ be the shattered graph after preprocessing. Let $\texttt{bc}$ and $\texttt{bc}'$ be the BC scores computed by BC-ORG(G) and BC-REACH(G'), respectively. We will prove that

$$\texttt{bc}[v] = \sum_{v' \in V' | \mathbf{org}(v') = v} \texttt{bc}'[v'], \tag{5}$$

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

Lemma 1 *Let u, v, s be vertices such that all $s \rightsquigarrow v$ paths contain u. Then,*

$$\delta_s(v) = \delta_u(v).$$

Pf. 1 *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), $\delta_s(v) = \delta_u(v)$.

Lemma 2 *For any vertex pair $s, t \in V$, there exists exactly one component C of G' which contains a copy of t which is not the same vertex as the representative of s in C.*

Pf. 2 *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 ℓth shattering and 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'.

Theorem 1 *Eq. 5 is correct after shattering G with articulation vertices.*

Pf. 3 *Let C be a component of G', s', v' be two vertices in C, and s, v be the corresponding 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 exactly 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 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 2, $\delta_{s'}(v')$ will be added to exactly one copy v' of v. Hence,* (5) *is correct.*

3.2.2 Shattering with bridges

Although the existence of a bridge implies the existence of an articulation vertex, handling bridges are easier and only requires the removal of the bridge. We embed this operation to the preprocessing phase 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.

Let C_u and C_v be the components formed after removing edge $\{u', v'\}$ which contain u' and v', respectively. Similar to (4), we add

$$\sum_{w \in C_v} \texttt{reach}[w] \quad \text{and} \quad \sum_{w \in C_u} \texttt{reach}[w]$$

to $\texttt{reach}[u']$ and $\texttt{reach}[v']$, respectively, to make u' (v') as the representative of all vertices in C_u (C_v).

After removing the bridge and updating the `reach` array, Lemma 2 is not true: there cannot be a component which contain a representative of u (v) and a 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. To alleviate this, we add

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

to $\texttt{bc}'[u']$ and $\texttt{bc}'[v']$, respectively. Note that Lemma 2 is true for all other vertex pairs.

Corollary 1 *Eq. 5 is correct after shattering G with articulation vertices and bridges.*

3.3 Compressing Graphs

Although, the compression techniques do not reduce the number of connected components, they reduce the number of vertices and edges in a graph. Since the complexity of Brandes' algorithm is $\mathcal{O}(mn)$, a reduction on m and/or n will help to reduce the execution time of the algorithm.

3.3.1 Compression with degree-1 vertices

Let G_ℓ be the graph after ℓ shattering 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 time, we reduce the number of vertices and the graph is compressed. Hence, we handle this case separately and set $G_{\ell+1} = G_\ell - u'$. The updates are the same with the bridge removal. That is, we add $\texttt{reach}[u']$ to $\texttt{reach}[v']$ and increase $\texttt{bc}'[u']$ and $\texttt{bc}'[v']$, respectively, with

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

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

Corollary 2 *Eq. 5 is correct after shattering G with articulation vertices and bridges, and compressing it with degree-1 vertices.*

3.3.2 Compression with side vertices

Let G_ℓ be the graph after ℓ shattering and compression operations, and let u' be a side vertex in a component C of G_ℓ. Since $\Gamma(u')$ is a clique, there is no shortest path passing through u'. That is, 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 alleviate this, we initiate a BFS from u' as given in Algorithm 2, which is similar to the ones in Bc-Reach. The only difference between Bfs-Side and a BFS of Bc-Reach is an additional line 2.

Algorithm 3: Bfs-Side

```
   Data: G_ℓ = (V_ℓ, E_ℓ), a side vertex s, reach, and bc'
   ··· ▷same as the BFS init.  in Bc-Reach
   while Q is not empty do
   |  ··· ▷same as BFS in Bc-Reach
   δ[v] ← reach[v] − 1, ∀v ∈ V_ℓ
   while S is not empty do
   |  w ← S.pop()
   |  for v ∈ P[w] do
   |  |  δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w])
   |  if w ≠ s then
1  |  |  bc'[w] ← bc'[w] + (reach[s] * δ[w])+
2  |  |           (reach[s] * (δ[w] − (reach[w] − 1))
   return bc'
```

Removing u' affects three types of dependencies:

1. Let $s \in V$ be a vertex s.t. $\textbf{rep}(C, s) = u'$ and let v' be a vertex in C where $v = \textbf{org}(v')$. Due to Lemma 2, when we remove u' from C, $\delta_s(v) = \delta_{u'}(v')$ cannot be added anymore to any copy of v. Line 1 of Bfs-Side solves this problem and adds the necessary values to $\texttt{bc}'(v')$.

2. Let $s \in V$ be a vertex s.t. $\textbf{rep}(C, s) = v' \neq u'$. If we remove u' from C, due to Lemma 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

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

to $\texttt{bc}'[u']$ after removing u' from C.

3. Let v', w' be two vertices in C different than u', and v, w be the corresponding 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 2 of Bfs-Side takes into account all these dependencies.

Corollary 3 *Eq. 5 is correct after shattering G with articulation vertices and bridges, and compressing it with degree-1 and side vertices.*

3.3.3 Compression with identical vertices

When two vertices in G are identical, all of their pair dependencies, source dependencies, and BC scores are the same. Hence, it is possible to combine these vertices and avoid extra computation. We distinguish 2 different types of identical vertices. Vertices u and v are type-I identical if and only if $\Gamma(u) = \Gamma(v)$. Vertices u and v are type-II identical if and only if $\Gamma(u) \cup \{u\} = \Gamma(v) \cup \{v\}$.

To handle this, we assign `ident` attribute to each vertex. $\texttt{ident}(v')$ denotes the number of vertices in G that are identical to v' in G'. Initially, $\texttt{ident}[v']$ is set to 1 for all $v \in V$.

Let $\mathcal{I} \subset V$ be a set of identical vertices. We remove all vertices $u' \in \mathcal{I}$ from G except one of them. Let v' be this remaining vertex. We increase $\texttt{ident}[v']$ by $|\mathcal{I}| - 1$, and keep a list of $\mathcal{I} \backslash \{v'\}$'s associated with v'.

When constructing the BFS graph, the number of paths $\sigma[w]$ is updated incorrectly for an edge $\{v, w\}$ when v is not the source. The edge leads to $\texttt{ident}[v]$ paths: $\sigma[w] \leftarrow \sigma[w] + (\sigma[v] * \texttt{ident}[v])$ if $v \neq s$.

The propagation of the dependencies $\texttt{ident}[w]$ along the edge $\{v, w\}$ should be accounted multiple times as in $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \texttt{ident}[w](\delta[w] + 1)$.

Finally, for a given source s, there are $\texttt{ident}[s]$ similar shortest path graphs, and the accumulation of the BC value is $\texttt{bc}'[w] \leftarrow \texttt{bc}'[w] + \texttt{ident}[s]\delta[w]$.

The only path that are ignored in this computation of BC are the paths between $u \in \mathcal{I}$ and $v \in \mathcal{I}$. If $\mathcal{I}$ is a type-II identical set, then this path are direct and the computation of BC is correct. However, if $\mathcal{I}$ is a type-I identical set, these paths have some impact. Fortunately, it only impacts the direct neighboor of $\mathcal{I}$. There are exactly $|\mathcal{I}| \times (|\mathcal{I}| - 1)$ paths whose impact is equally distributed among the neighbors of $\mathcal{I}$.

The technique presented in this section has been presented without taking `reach` into account. Both techniques can be applied simultaneously but the details are not presented here due to space limitation.

Corollary 4 *Eq. 5 is correct after shattering G with articulation vertices and bridges, and compressing it with degree-1, side, and identical vertices.*

3.4 Implementation Details

There exist linear time algorithms for detecting articulation vertices and bridges [21, 22]. In our implementation of the preprocessing phase, after detecting all articulation vertices with [22], the graph is decomposed into its biconnected components at once. Note that the final decomposition is the same when the graph is iteratively shattered one articulation point at a time as described above. But decomposing the graph into its biconnected components is much faster. A similar approach works for bridges and removes all of them at once. Since the detection algorithms are linear time, each cumulative shattering operation takes $\mathcal{O}(m+n)$ time.

For compression techniques, detecting recursively all degree-1 vertices takes $\mathcal{O}(n)$ time. Detecting identical vertices is expected to take a linear time provided a good hash function to compute the hash of the neighborhood of each vertex. In our implementation, 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 side vertices of degree k, we use a simple algorithm which for each vertex v of degree k, verifies if the graph induced by $\Gamma(v)$ is a clique. In practice, our implementation does not search for cliques of more than 4 vertices since our preliminary experiments show that searching these cliques is expensive. 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.

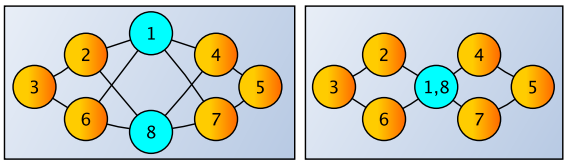


Figure 2: Combining identical vertices can create an articulation point: Vertices 1 and 8 are identical vertices with neighbors $\{2, 4, 6, 7\}$. When one of the identical vertices is removed, the remaining one is an articulation point.

The preprocessing phase is implemented as a loop where a single iteration consecutively tries to shatter/compress the graph by using the above mentioned five operations. The loop continues as long as improvement are made. Indeed, a single iteration of this loop may not be sufficient since each operation can make the graph amenable to another one. For example, in our toy graph given in Figure 1.(a), removing the degree-1 vertex Martin makes Wayne and Henry identical. Furthermore, when Diana is also removed as a side vertex, Henry and Wayne both become side vertices. Or as Figure 2 shows, removing identical vertices can form an articulation vertex .

4 Experimental Results

We implemented the original and modified BC algorithms, and the proposed optimization techniques in `C++`. The code is compiled with `icc v12.0` and optimization flags `-O2 -DNDEBUG`. The graph is kept in memory in the compressed row storage (CRS) format using 32-bit data types. The experiments are run on a node with two Intel Xeon E5520 CPU clocked at 2.27GHz and equipped with 48GB of main memory. Despite the machine is equipped with 8 cores, all the experiments are run sequentially.

Graph				Time	
application	name	#vertices	#edges	org.	best
Category *social*					
Social	CondMat	16,726	47,594	21.1	9.1
	CondMat03	27,519	116,181	102.0	52.3
	hep-th	8,361	15,751	3.2	1.6
	CondMat05	40,421	175,691	209.0	107.0
	PGPgiant	10,680	24,316	10.7	3.7
	astro-ph	16,706	121,251	40.3	22.2
Category *structural*					
Auto	bcsstk29	13,992	302,748	68.3	26.4
	bcsstk30	28,924	1,007,284	399.0	41.4
	bcsstk31	35,588	572,914	363.0	106.0
	bcsstk32	44,609	985,046	737.0	77.3
	bcsstk33	8,738	291,583	37.0	11.1
Category *geographical*					
Redistricting	ak2010	45,292	108,549	178.0	114.0
	ct2010	67,578	168,176	514.0	369.0
	de2010	24,115	58,028	61.4	40.6
	hi2010	25,016	62,063	18.4	12.9
Road	luxembourg	114,599	119,666	632.0	390.0
Category *misc*					
Router	as-22july06	22,963	48,436	39.9	15.5
Power	power	4,941	6,594	1.3	0.7
Biology	ProtInt	9,673	37,081	11.2	8.1
Semi-	add32	4,960	9,462	1.4	0.3
Conductor	memplus	17,758	54,196	17.6	11.2
			Geomean	47.4	19.6

Table 1: Properties of 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 by a combination of the proposed heuristics.

For the experiments, we used 21 real-life networks from the dataset of DIMACS Graph Partitioning and Graph Clustering Challenge. The graphs and their properties are summarized in Table 1. They are classified into four categories. The first one, *social*, contains 6 social networks. The second one, *structural*, contains 5 structural engineering graphs. The third one, *geographical*, contains 4 redistricting graphs and one road graph. The last one, *misc*, contains graphs from various applications such as autonomous systems, protein-protein interaction, and power grids.

4.1 Ordering sparse networks

As most of the graph-based kernels in data mining, the order of the vertices and edges accessed by Brandes' algorithm is important. In today's hardware, cache is one of the fastest and one of the most scarce resources. When the graphs are big, they do not fit in the cache, and the number of cache misses along with the number of memory accesses increases.

If two vertices in a graph are close, 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 will probably decrease. Following this reasoning, we initiated a BFS from a random vertex in G and use 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 [23].

For each graph in our set, Figure 3 shows the time taken by both the BFS ordering and Bc-Org relative to the original Bc-Org execution time with the natural vertex ordering. For

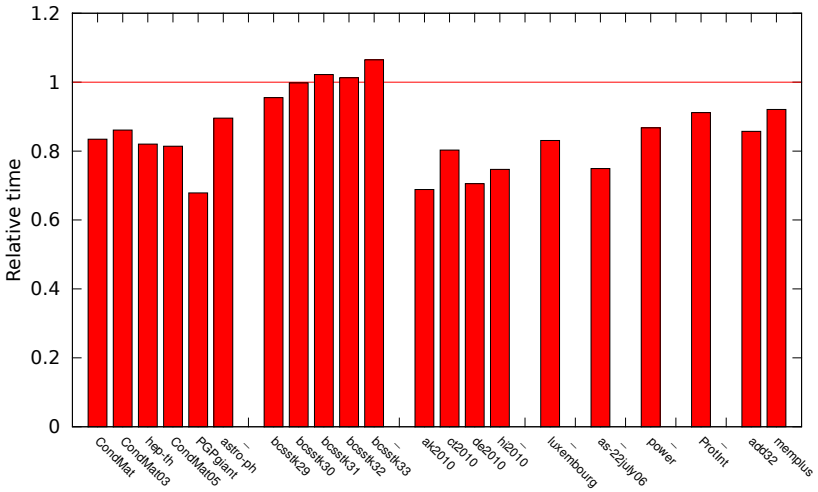


Figure 3: Relative performance of BFS ordering with respect to original time with natural ordering.

18 of 21 matrices using a BFS ordering improved the performance. Overall, it reduced the time to approximately 80% of the original time on average. Hence compared with BFS ordering, the natural order of a real-life network has usually a detrimental effect on the execution time of BC.

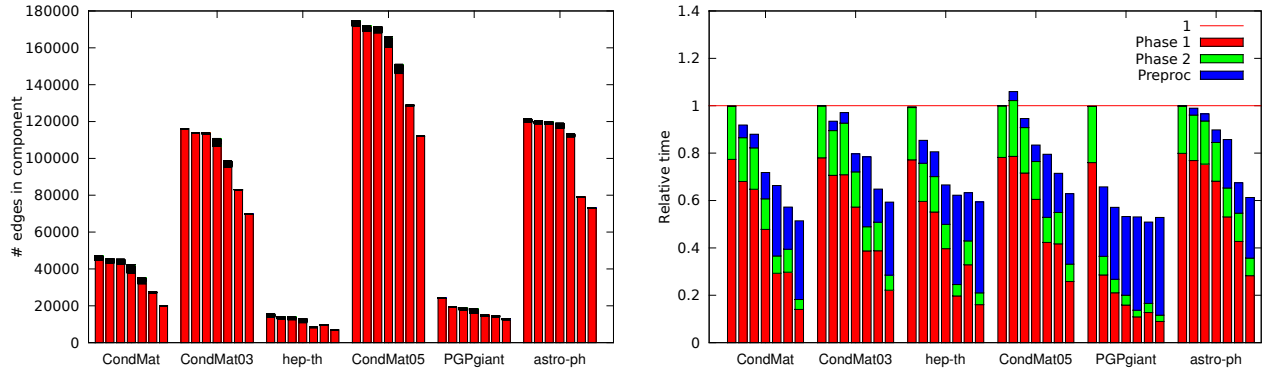
4.2 Shattering and compressing graphs

For each graph, we tested 7 different combinations of the improvements proposed in this paper: They are denoted with **o**, **od**, **odb**, **odba**, **odbas**, **odbai**, and **odbasi**, where **o** denotes the BFS **o**rdering, **d** denotes **d**egree-1 vertices, **b** denotes **b**ridge, **a** denotes **a**rticulation vertices, **s** denotes **s**ide vertices, and **i** denotes **i**dentical vertices. The ordering of the letters denotes the order of application of the respective improvements.

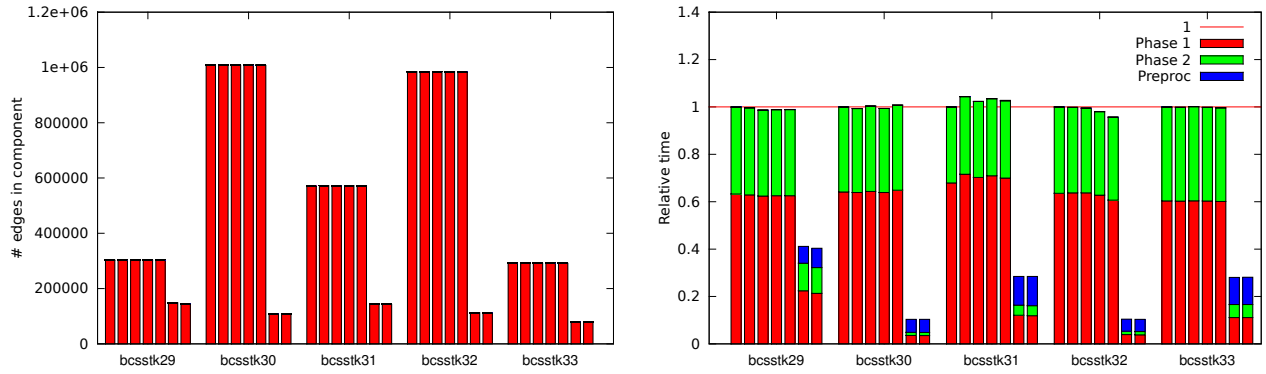
Given a graph G, we measure the time spent for preprocessing G by a combination to obtain G', computing the BC scores of the vertices in G', and using these scores computing the BC scores of the vertices in G. For each category, we have two kind of plots: the first plot shows the numbers of edges in each component of G'. Different components of G' are represented by different colors. The second plot shows the normalized execution times for all 7 combinations. The times for the second chart are normalized w.r.t. the first combination: the time spent by Bc-Org after a BFS ordering. For each graph in the category, each plot has 7 stacked bars representing a different combination in the order described above.

As Figure 4 shows, there is a direct correlation between the remaining edges in G' and the execution time. This proves that our rationale behind investigating shattering and compression techniques is valid. However, the figures on the left show that these graphs do not contain good articulation vertices and bridges which shatter a graph approximately half. Since, red is almost always the dominating color, we can argue that such vertices and edges do not exist in real life graphs.

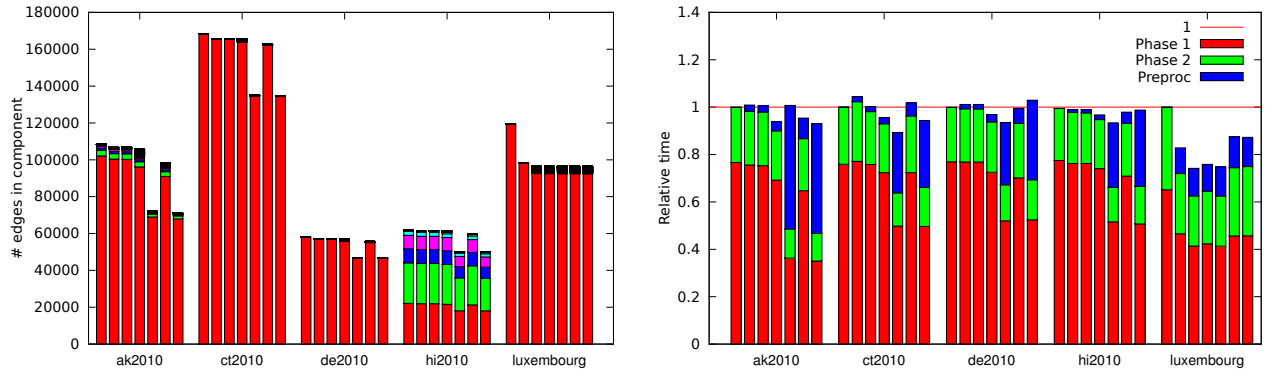
For *social* graphs, each added shattering and compression technique provides a significant improvement for almost all of the cases. That is, the best combination is **odbasi** for 5 out of 6 graphs, and the normalized execution time is continuously decreasing when a combination is enhanced with a new technique. According to the original and best execution times in Table 1, for *social* graphs, the techniques, including ordering, provide 53% improvement in total. For *structural* graphs, although the only working technique is identical vertices, the improvement is



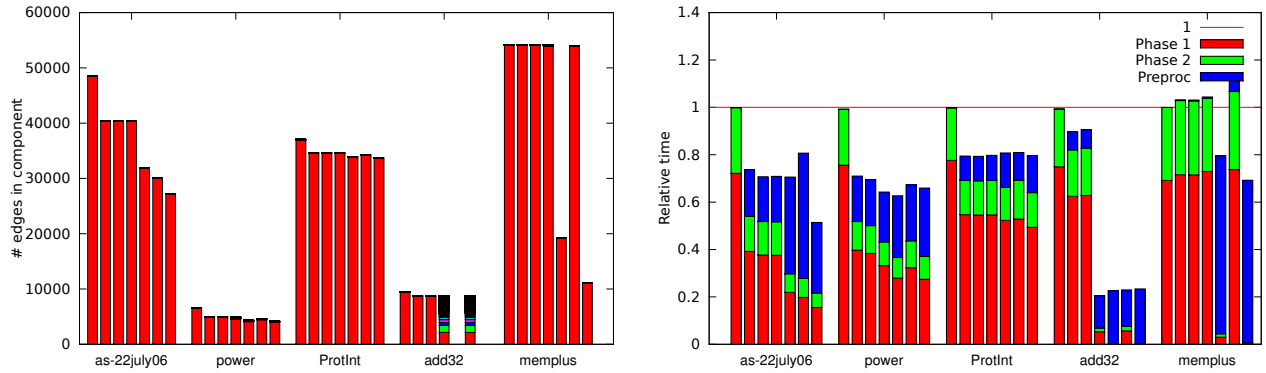
(a) Category *social*



(b) Category *structural*



(c) Category *geographical*



(d) Category *misc*

Figure 4: Left: The numbers of edges in the connected components of G' as stack bars. Each component is represented by a different color. Right: Normalized execution times of preprocessed BC computations with the combinations **o**, **od**, **odb**, **odba**, **odbas**, **odbai**, and **odbasi**, respectively, where the times are normalized w.r.t. **o** and divided to three stages: preprocessing time and the time spent in the first and second phases of the BFSs.

of 79% on the average. For the redistricting graphs in *geographical*, the techniques are not very useful. However, with the help of BFS ordering, we obtain 32% improvement on average. For the graph *luxembourg*, degree-1 and bridge removal techniques have the most significant impact. Since the graph is obtained from a road network, this is expected (roads have bridges). Hence, if the structure of the graph is known to some extent, the techniques can be specialized. For example, it is a well known fact that biological networks usually have a lot of degree-1 vertices but a few articulation vertex. And our results on the graph *ProtInt* confirms this fact since the only significant improvement is obtained with the combination **od**. In our experiments, the most interesting graph is *add32* since the combinations **odbas** and **odbasi** completely shatters it. Note that on the left, there is no bar since there is no remaining edge in G' and on the right, all the bar is blue which is the color of preprocessing. When all techniques are combined, we obtain a 59% improvement on average over all graphs.

Please note that the implementation uses 4 different kernels depending on whether `reach` and `ident` are used. Each new attribute brings an increase in runtime which can be seen on *CondMat03* when going from **o** to **od** and on *luxembourg* when going from **odba** to **odbai**.

The combinations are compared with each other using a performance profile graph presented in Figure 5. 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.

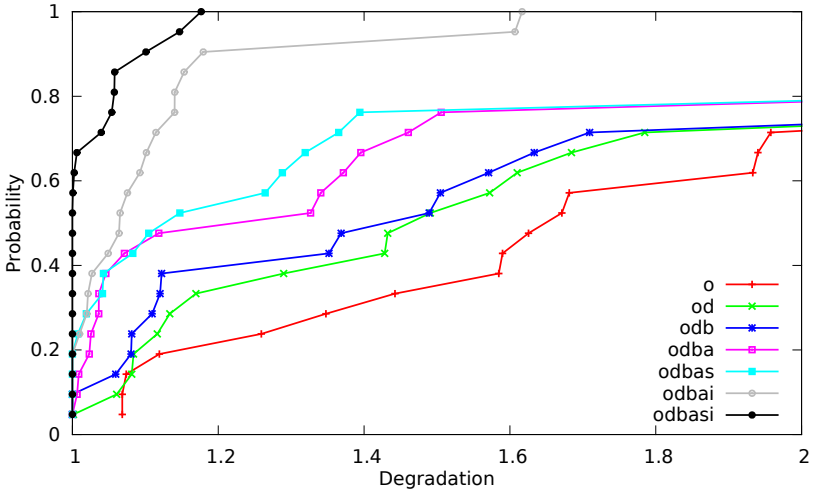


Figure 5: Performance profile of various combination of optimization on all the selected graphs.

Not using graph shattering techniques (**o**) has the worse performance profile. It is never optimal. According to the graph, using all possible techniques is the best idea. This strategy is the optimal one with more than 60% probability. Clearly, one always wants to use graph shattering techniques. If little information is available **odbasi** should be the default choice. However, if one believes that identical vertices will barely appear in the graph, then **odbas** might lead to better performances.

5 Conclusion

Betweenness is a very popular centrality metric in practice and proved to be successful in many fields such as graph mining. But, computing BC scores of the vertices in a graph is a time consuming task. In this work, we investigate shattering and compression of networks to reduce

the execution time of BC computations.

The shattering techniques break graphs into smaller components while keeping the information to recompute the pair and source dependencies which are the building blocks of BC scores. On the other hand, the compression techniques do not change the number of components but reduces the number of vertices and/or edges. 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 while computing BC scores.

We also noticed that the natural order of a real-life network has usually a detrimental effect on the execution time of BC. In our experiments, even with a simple and cheap BFS ordering, we managed to obtain 20% improvement on average. Unfortunately, we are aware of several works, which do not even consider a simple ordering while tackling a graph-based computation. So one rule of thumb: "Order your graphs".

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 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] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, C. Corley, R. M. Farber, and W. N. Reynolds, "Massive social network analysis: Mining twitter for social good," in *ICPP*, 2010, pp. 583–593.

[2] J.-K. Lou, S. de 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.

[3] D. A. Bader and K. Madduri, "A graph-theoretic analysis of the human protein-interaction network using multicore parallel algorithms," *Parallel Comput.*, vol. 34, pp. 627–639, Nov 2008.

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

[5] 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, pp. 1 –7.

[6] V. Krebs, "Mapping networks of terrorist cells," *Connections*, vol. 24, 2002.

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

[8] 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 *SIAM Conference on Data Mining (SDM)*, 2012.

[9] D. A. Bader and K. Madduri, "SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks," in *IPDPS'08*, 2008.

[10] L. Freeman, "A set of measures of centrality based upon betweenness," *Sociometry*, vol. 4, pp. 35–41, 1977.

[11] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[12] U. Brandes and C. Pich, "Centrality estimation in large networks," *I. J. Bifurcation and Chaos*, vol. 17, no. 7, pp. 2303–2318, 2007.

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

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

[15] D. A. Bader and K. Madduri, "Parallel algorithms for evaluating centrality indices in real-world networks," in *ICPP*, 2006, pp. 539–550.

[16] 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.

[17] Z. Shi and B. Zhang, "Fast network centrality analysis using GPUs," *BMC Bioinformatics*, vol. 12, p. 149, 2011.

[18] P. Pande and D. Bader, "Computing betweenness centrality for small world networks on a GPU," Poster at 15th High Performance Embedded Computing Conference, Lexington, Massachusetts, 2011.

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

[20] U. Brandes, "On variants of shortest-path betweenness centrality and their generic computation," *Social Networks*, vol. 30, no. 2, pp. 136–145, 2008.

[21] R. E. Tarjan, "A note on finding the bridges of a graph," *Inf. Process. Lett.*, vol. 2, no. 6, pp. 160–161, 1974.

[22] J. Hopcroft and R. Tarjan, "Algorithm 447: efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973.

[23] G. Cong and K. Makarychev, "Optimizing large-scale graph analysis on a multi-threaded, multi-core platform," in *IPDPS'11*, 2011, pp. 688–697.