Graph Manipulations for Fast Centrality Computation

AHMET ERDEM SARIYÜCE, Sandia National Laboratories
KAMER KAYA, Sabancı University
ERIK SAULE, University of North Carolina at Charlotte
ÜMİT V. ÇATALYÜREK, Georgia Institute of Technology

The betweenness and closeness metrics are widely used metrics in many network analysis applications. Yet, they are expensive to compute. For that reason, making the betweenness and closeness centrality computations faster is an important and well-studied problem. In this work, we propose the framework *BADIOS* that manipulates the graph by compressing it and splitting into pieces so that the centrality computation can be handled independently for each piece. 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 betweenness centrality computation time of a 4.6 million edges graph from more than 5 days to less than 16 hours. For the same graph, the closeness computation time is decreased from more than 3 days to 6 hours (12.7x speedup).

CCS Concepts:• **Mathematics of computing → Paths and connectivity problems**; • **Information systems** → *Data mining;*

Additional Key Words and Phrases: Betweenness centrality, closeness centrality, shortest path

ACM Reference Format:
Ahmet Erdem Sarıyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. 2017. Graph manipulations for fast centrality computation. ACM Trans. Knowl. Discov. Data 11, 3, Article 26 (March 2017), 25 pages.
DOI: http://dx.doi.org/10.1145/3022668

1. INTRODUCTION

Centrality metrics are crucial for detecting the central and influential vertices in various types of networks, such as social networks [Lou et al. 2010], biological networks [Koschützki and Schreiber 2008], power networks [Jin et al. 2010], covert networks [Krebs 2002], and decision/action networks [Şimşek and Barto 2008]. The *betweenness* and *closeness* are two intriguing metrics and have been implemented in several tools that are widely used in practice for analyzing networks [Lugowski et al. 2012]. The betweenness centrality (BC) score of a vertex is the sum of the ratios of the shortest paths between vertex pairs that pass through the vertex of interest to the total number of shortest paths between them [Freeman 1977]. The closeness centrality

This work is partially supported by TÜBİTAK grant 115C018.

Authors' addresses: A. E. Sarıyüce, Data Sciences & Cyber Analytics Department, Sandia National Laboratories, 7011 East Avenue, Livermore, CA 94551, USA; email: asariyu@sandia.gov; K. Kaya, Faculty of Engineering and Natural Sciences, Computer Science & Engineering, Sabancı University, Orhanlı/Tuzla 34956, Istanbul, Turkey; email: kaya@sabanciuniv.edu; E. Saule, Computer Science Department, University of North Carolina at Charlotte, Woodward Hall 210D, 9201 University City Blvd, Charlotte, NC 28262, USA; email: esaule@uncc.edu; Ü. V. Çatalyürek, School of Computational Science and Engineering, Georgia Institute of Technology, Klaus Advanced Computing Building, 266 Ferst Drive, Atlanta, GA 30332, USA; email: umit@gatech.edu.

DOI: http://dx.doi.org/10.1145/3022668

(CC) score of a vertex is the inverse of the sum of shortest distances from the vertex of interest to all other vertices. Hence, contribution, load, influence, or effectiveness of a vertex, while disseminating information through a network, is determined with betweenness and/or closeness metrics. Although BC and CC have been proved to be successful for network analysis, computing the centrality scores of all the vertices 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 and $\mathcal{O}(n + m)$ space complexity for unweighted and weighted networks, respectively, where n is the number of vertices and m is the number of vertex–vertex interactions in the network [Brandes 2001]. Brandes' algorithm is currently the best algorithm for BC computations, and it is unlikely that general algorithms with better asymptotic complexity can be designed [Kintali 2008]. However, it is not fast enough to handle Facebook's billion or Twitter's 200 million users. Computing CC has a similar cost.

We propose the *BADIOS* framework that 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 BC and CC computation. The framework splits the network and reduces its size so that the BC and CC scores of the vertices 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.

For the sake of simplicity, we consider only standard, shortest-path vertex-BC and vertex-CC on undirected unweighted graphs. However, our techniques can be used for other path-based centrality metrics, or other BC variants, e.g., *edge* and *group betweenness* [Brandes 2008]. *BADIOS* can also be applied to weighted and/or directed networks. Furthermore, it is compatible with the existing approximation and parallelization techniques of the BC and CC computation.

BC computation of *BADIOS* is previously published in our earlier work [Sarıyüce et al. 2013b]. In this article, we extend our earlier work by applying *BADIOS* framework on CC computation. We apply *BADIOS* on a popular set of graphs with sizes ranging from 6K edges to 4.6M edges. For BC, we show an average speedup of 2.8 on small graphs and of 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. For CC, the average speedup is 2.4 and 3.6 on small and large networks.

The rest of the article is organized as follows: In Section 2, an algorithmic background for CC and BC computation are given. The splitting and compression techniques for CC and BC are explained in Sections 4 and 5, respectively. Section 6 gives experimental results on various kinds of networks. We give the related work in Section 7 and summarize the article in Section 8.

2. NOTATION AND BACKGROUND

Let $G = (V, E)$ be a network modeled as an undirected graph with $n = |V|$ vertices and $m = |E|$ edges where each entity is represented by a vertex in V, and an interaction is represented by an edge in E. Let $\Gamma(v)$ be the set of vertices that are interacting with 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 this is the case, $\texttt{dst}_G(u, v) = \texttt{dst}_G(v, u)$ shows the length of the shortest $u \rightsquigarrow v$ path in G. Otherwise, $\texttt{dst}_G(u, v) = \texttt{dst}_G(v, u) = \infty$. 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 number of 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. The graph 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. Closeness Centrality

Given a graph G, the CC of u is be defined as

$$\texttt{far}[u] = \sum_{\substack{v \in V \\ \texttt{dst}_G(u,v) \neq \infty}} \texttt{dst}_G(u, v),$$

$$\texttt{cc}[u] = \frac{1}{\texttt{far}[u]}.$$

If u cannot reach any vertex in the graph, we take by convention $\texttt{cc}[u] = 0$.

For a sparse unweighted graph $G = (V, E)$, the complexity of CC computation is $\mathcal{O}(n(m+n))$ [Brandes 2001]. The pseudocode is given in Algorithm 1. For each vertex $s \in V$, the algorithm initiates a breadth-first search (BFS) from s, computes the distances to the other vertices, and accumulates to $\texttt{cc}[s]$. Since a BFS takes $\mathcal{O}(m + n)$ time, and n BFSs are required in total, the complexity follows.

ALGORITHM 1: Cc-Org: Closeness Centrality Computation Kernel

```
Data: $G = (V, E)$
Output: $\texttt{cc}[.]$
for each $s \in V$ do
    $Q \leftarrow$ empty queue
    $Q$.push($s$)
    $\texttt{dst}[s] \leftarrow 0$
    $far \leftarrow 0$
    $\texttt{cc}[s] \leftarrow 0$
    $\texttt{dst}[v] \leftarrow \infty, \forall v \in V \setminus \{s\}$
    while $Q$ is not empty do
        $v \leftarrow Q$.pop()
        for all $w \in \Gamma_G(v)$ do
            if $\texttt{dst}[w] = \infty$ then
                $Q$.push($w$)
                $\texttt{dst}[w] \leftarrow \texttt{dst}[v] + 1$
                $far \leftarrow far + \texttt{dst}[w]$
            end
        end
    end
    $\texttt{cc}[s] \leftarrow \frac{1}{far}$
end
return $\texttt{cc}[.]$
```

2.2. 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 BC of v is defined by

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

ALGORITHM 2: Bc-Org: Betweenness Centrality Computation Kernel

```
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, $\sigma[v] \leftarrow 0$
    $\texttt{dst}[v] \leftarrow \infty, \forall v \in V \setminus \{s\}$
    $Q$.push($s$), $\sigma[s] \leftarrow 1$, $\texttt{dst}[s] \leftarrow 0$
    ▷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 $\texttt{dst}[w] = \infty$ then
                $Q$.push($w$)
                $\texttt{dst}[w] \leftarrow \texttt{dst}[v] + 1$
            end
            if $\texttt{dst}[w] = \texttt{dst}[v] + 1$ then
1               $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
                $\texttt{P}[w]$.push($v$)
            end
        end
    end
    ▷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           $\delta[v] \leftarrow \delta[v] + \delta[w]$
        end
        if $w \neq s$ then
3           $\texttt{bc}[w] \leftarrow \texttt{bc}[w] + (\delta[w] \times \sigma[w] - 1)$
        end
    end
end
return bc
```

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 (1). Brandes reduced this complexity and proposed an $\mathcal{O}(mn)$ algorithm for unweighted networks [Brandes 2001]. 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 $\texttt{P}_s(u)$ 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 $\mathtt{d}_s(u)$ 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:

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

To compute $\delta_s(v)$ for all $v \in V \setminus \{s\}$, Brandes' algorithm uses a two-phase approach (Algorithm 2). First, a BFS is initiated from s to compute σ_{sv} and $\mathtt{P}_s(v)$ 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 (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. THE *BADIOS* FRAMEWORK

As mentioned in Section 1, closeness- and betweenness-based graph analysis can be an expensive task. The size of the graph, in particular the size of the largest component in the graph, is the main parameter that affects the practical computation time of many distance-related graph metrics. Hence, compression techniques that can reduce the number of vertices/edges in a graph are promising to make them faster. Furthermore, splitting graphs into multiple connected components, and hence reducing the largest component size, can also help in practice.

BADIOS uses bridges and articulation vertices for splitting 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: No shortest path is passing through a side vertex unless the side vertex is one of the endpoints, all the shortest paths from/to a degree-1 vertex is passing through the same vertex, and for two vertices u and v with identical neighborhoods, $\mathtt{bc}[u]$ and $\mathtt{bc}[v]$ ($\mathtt{cc}[u]$ and $\mathtt{cc}[v]$) are equal. A toy graph and a high-level description of the splitting/compression process via *BADIOS* is given in Figure 1.

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 splitting/compression operation. The $\ell + 1$th operation modifies each 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 CC (or BC) computation begins.

Exploiting the existence of above-mentioned structures on CC and 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 $v \in V$, $\mathtt{bc}[v]$ is equal to the number of 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 right subtrees of v, respectively. This approach takes only $\mathcal{O}(n)$ time. These equations can also be modified for CC computations and a linear-time CC algorithm can also be obtained for trees.

A novel feature of *BADIOS* is fully exploiting the above-mentioned structures by employing an iterative preprocessing phase. Specifically, a degree-1 removal can create new degree-1, identical, and side vertices. Or, a splitting can reveal new degree-1 and side vertices. Similarly, by removing an identical vertex, new identical, degree-1, articulation, and side vertices can appear. At last, new identical vertices can be discovered when a side vertex is removed from the graph. To fully reduce the graph

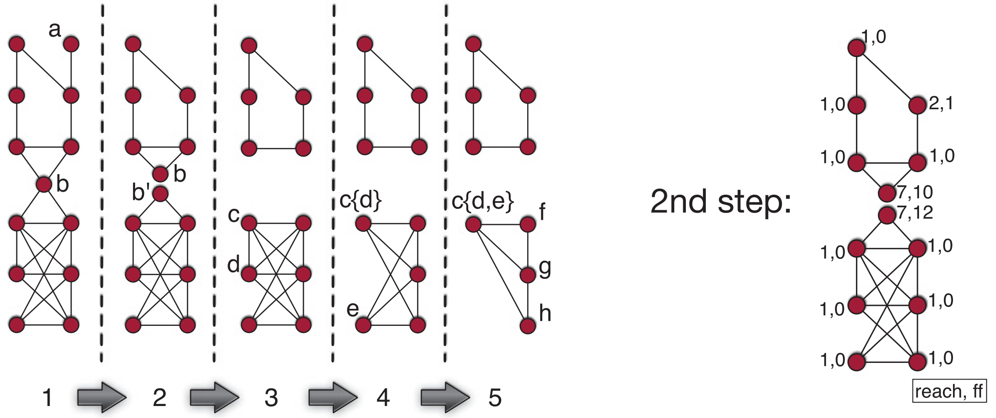


Fig. 1. (1) a is a degree-1 vertex and b is an articulation vertex. The framework removes a and also creates a clone b' to represent b in the bottom component. (2) There is no degree-1, articulation, or identical vertex, or a bridge. However, vertices b and b' are now side vertices and they are removed. (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 identical vertices. The last reductions are not shown but the bottom component is compressed to a singleton vertex. The five cycle above cannot be reduced. Rightmost figure shows the situation of `reach` and `ff` values in the second stage of manipulation. Values are shown next to each vertex.

by using the newly formed structures, the framework uses a loop where each iteration performs a set of manipulations on the graph.

4. *BADIOS* FOR CLOSENESS CENTRALITY

Based on the combinatorial structures mentioned above, we describe a set of *closeness-preserving* graph manipulation techniques to make a graph smaller and disconnected while preserving the information required to compute the distance-based metrics by using some auxiliary arrays. The proposed techniques will especially be useful on expensive distance-based graph kernels such as CC that will be our main application while describing the proposed approach.

In the preprocessing phase, *BADIOS* compresses the graph G, splits it into multiple connected components and obtains another graph $G' = (V', E')$ with several graph manipulations. Let u be a vertex in V' and C' be the connected component of G' containing u. Let $\mathcal{R}_u$ be the set of vertices $v \in (V \setminus C') \cup \{u\}$ such that all the shortest $v \rightsquigarrow w$ paths in the original graph G are passing through u for all $w \in C'$. In G', all the vertices $\mathcal{R}_u \setminus \{u\}$ are disconnected from the vertices in C'. Hence, for each vertex $v \in \mathcal{R}_u$, u will act as a *representative* (or proxy) in C'. During the CC computation, it will be responsible to propagate the impact of v to the CC values of all the vertices in C'. To handle the impact of vertices that u serves as a proxy, we define $\texttt{reach}[u] = |\mathcal{R}_u|$ that is the number of vertices represented by u. `reach` attribute helps to correctly compute the CC scores when the graph is split. Initial `reach` values of all vertices is 1, which implies each vertex only represents itself.

Apart from the split operations, we also compress the graph by removing the vertices and edges. Impact of the removed edges on the CC scores of vertices in G' should be considered. For this purpose, we propose the *forwardable farness* scores to be maintained at each vertex. Considering $\mathcal{R}_u$, and the vertex u, as explained above, we calculate the sum of shortest distances from u to all the vertices in $\mathcal{R}_u$, which is *forwardable* to other vertices if u is also removed in the following steps of preprocessing. We denote the forwardable farness by ff, and formally define it as follows:

$$\mathtt{ff}[u] = \sum_{v \in \mathcal{R}_u} \mathtt{dst}_G(u, v).$$

Initially, all the ff values are set to 0, since there is no compression/split yet, and thus there is no farness to forward.

ALGORITHM 3: Cc-Reach: Modified Closeness Centrality Computation

```
  Data: $G' = (V', E')$, ff[.], reach[.], far[.]
  Output: cc[.]
  for each $s \in V'$ do
      ··· ▷same as Cc-Org
      while $Q$ is not empty do
          $v \leftarrow Q$.pop()
          for all $w \in \Gamma_{G'}(v)$ do
              if dst[$w$] = $\infty$ then
                  $Q$.push($w$)
                  dst[$w$] ← dst[$v$] + 1
1                 fwd ← ff[$w$] + (dst[$w$] × reach[$w$])
2                 far[$s$] ← far[$s$] + fwd
              end
          end
      end
3     far[$s$] ← far[$s$] + ff[$s$]
4     cc[$s$] ← 1/far[$s$]
  end
  return cc[.]
```

Incorporating all the split and compression operations to the CC computation also needs to change the main kernel that is presented in Algorithm 1. We present Algorithm 3 to show the changes made to the original kernel. The first change we need to do is to maintain the farness value of each vertex during computation. In the original kernel (Algorithm 1), we just calculated that value (shown as *far*) for each vertex during the BFS operation when that vertex serves a source. However, in the preprocessing phase, we partially compute each *far* value that requires a temporary storage. We use far array for this purpose, which is used in lines 2, 3, and 4 in Algorithm 3.

The next change to the original kernel happens during the farness accumulation in the most inner loop. Here, we need to consider the vertices that the traversed vertex w represents, which is defined as reach[w], and update the farness for each such vertex. Hence, we multiply the computed distance from source s to w by the reach[w] value, as shown in line 1. Furthermore, we forward the farness score at w (ff[w]) to far[s] in the same line.

Last change is regarding to the forward farness score at source s. We add ff[s] to its total farness score (in line 3) to consider the impact of removed edges that are only reachable through s.

In the following, we present and explain the adjustments to the reach and ff values of vertices upon splits and compressions. After that we present the necessary post-processing steps and introduce the theorem to prove the correctness of CC scores by using reach and ff values.

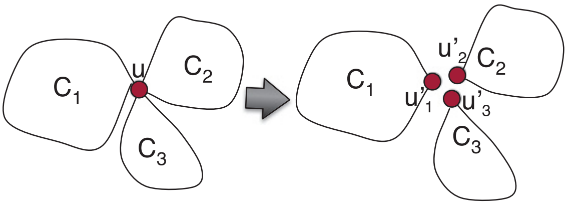


Fig. 2. Articulation vertex cloning on a toy graph with three disconnected components after the graph manipulation.

4.1. Closeness-Preserving Graph Splits

We used two approaches to split the graphs into multiple connected components: *articulation vertex cloning* and *bridge removal*. Indeed, a bridge exists only between two articulation vertices, but we still handle it separately, since we observed that a bridge removal is cheaper than articulation vertex cloning and also the former does not increase the number of vertices but the latter does.

4.1.1. Articulation Vertex Cloning. Let u be an articulation vertex in a component C appeared in the preprocessing phase where we perform graph manipulations. We split C into k components C_i for $1 \leq i \leq k$ by removing u from G and adding a *local clone* u'_i of u to each new component C_i by connecting u'_i to the same vertices u was connected in C_i as shown in Figure 2. For CC computations, to keep the relation between the clones and the original vertex, we use a mapping **org** from V' to V where $\mathbf{org}(u'_i)$ is original vertex $u \in V$ for a clone $u'_i \in V$. At any time of a CC preprocessing phase, a vertex $u \in V$ has exactly one *representative* u' in each component C such that $\texttt{reach}[u']$ is increased due to the existence of u. This vertex is denoted as $\mathbf{rep}(C, u)$. Note that each local clone is a representative of its original.

The cloning operation keeps the number of edges constant but increases the number of vertices in the graph. reach value of a vertex u reflects the number of vertices that u serves as a proxy. Thus, the articulation vertex and each of its clones should be updated to reflect the number of vertices in all the connected components that they are disconnected to. Formally, we update the reach value each clone u_i as follows:

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

Following the same logic, each clone should get the forwarded farness scores from the vertices in all the connected components that they are not connected. Thus, in each component, we need to calculate the sum of distances from the clone vertex u_i to all the other vertices, and forward this farness score to all the other clones. For each clone vertex u_i, update on ff scores is as follows:

$$\texttt{ff}[u'_i] = \texttt{ff}[u] + \sum_{\substack{1 \leq j \leq k \\ j \neq i}} \sum_{v \in C_j} \texttt{dst}_{C_j}(u'_j, v), \tag{5}$$

for $1 \leq i \leq k$. Note that these updates are only local to clone vertices, i.e., only their reach and ff values are affected. For example, a clone vertex u'_i sees the impact of the $\texttt{dst}_C(u, v)$ on $\texttt{ff}[u'_i]$ even though $v \in C_j$, $i \neq j$, is in another component after the split. However, the same is not true for a non-clone vertex $w \notin C_j$. Hence, considering that v and w are not connected anymore, the original CC kernel in Algorithm 1 will not compute the correct CC values. To alleviate this, we modified the original kernel and presented Algorithm 3 to propagate the forwardable farness values of the clone vertices to their components. With the modified kernel, we will have

$$\texttt{cc}[u] = \texttt{cc}'[u'_i], \tag{6}$$

for $1 \leq i \leq k$. That is, all the vertices cloned from the same articulation vertex will have the same CC after the execution of the modified kernel, which will be equal to the actual centrality of the articulation vertex used for splitting.

4.1.2. Bridge Removals. As mentioned above, bridges can only exist between two articulation vertices. The graph can be split into three connected components via articulation vertex cloning where one of the components will be a trivial one having a single edge and two clone vertices. Here, we show that the removal of a bridge $\{u, v\}$ can combine these steps and does not form such unnecessary trivial components. Let C_u and C_v be the two components after bridge removal that contain u and v, respectively. Vertex u should represent all the vertices in C_v and symmetrically v needs to represent all in C_u. Thus, `reach` values are updated to reflect these changes as follows:

$$\texttt{reach}[u] = \texttt{reach}[u] + \sum_{w \in C_v} \texttt{reach}[w], \tag{7}$$

$$\texttt{reach}[v] = \texttt{reach}[v] + \sum_{w \in C_u} \texttt{reach}[w]. \tag{8}$$

Another aspect we need to consider is the forwarded farness values. Consider vertex u (all the following applies to v symmetrically). First, we should forward $\texttt{ff}[v]$ to u, which will be further forwarded to the vertices in C_u during the CC computation. Next, we need to consider the distances from v to all the other vertices in C_v, and forward their sum to $\texttt{ff}[u]$, which can be expressed as $\sum_{w \in C_v} \texttt{dst}_{C_v}(v, w)$. And at last, we take the set vertices that vertex v represents, $\mathcal{R}_v$, into account. We do not know the distance from each of those vertices to v, but need to consider their impact on $\texttt{ff}[u]$. The nice thing is that for each such vertex $x \in \mathcal{R}_v$, the difference between its distance to u and v ($\texttt{dst}_G(x, u) - \texttt{dst}_G(x, v)$) is always 1, since v is always on the shortest path to u. Leveraging this fact, we just add the sum of these differences, which is $\texttt{reach}[v]$, to $\texttt{ff}[u]$ and complete all the update operations. All these changes are summarized formally as follows:

$$\texttt{ff}[u] = \texttt{ff}[u] + \left(\texttt{ff}[v] + \sum_{w \in C_v} \texttt{dst}_{C_v}(v, w)\right) + \texttt{reach}[v],$$

$$\texttt{ff}[v] = \texttt{ff}[v] + \left(\texttt{ff}[u] + \sum_{w \in C_u} \texttt{dst}_{C_u}(u, w)\right) + \texttt{reach}[u],$$

where $\texttt{reach}[u]$ and $\texttt{reach}[v]$ are the recently updated values from (7) and (8).

To update the `reach` and `ff` values, both the cloning and removal techniques described above require a traversal within the component of the graph in which the articulation vertex or bridge appears. Although it seems costly, the benefit of such manipulations can be understood if the superlinear complexity of CC computation is considered. Assume that a graph is split into k disconnected components each having equal number of vertices and edges. Considering the $\mathcal{O}(n(m+n))$ time complexity, the CC computation for each of these components will take k^2 times less time. Since there are k of them, the split will provide a k fold speedup in total. Although such articulation vertices and bridges that evenly split the graph do not appear in real-world graphs, even with imbalanced splits, one can obtain significant speedups since the cost of a split is just a single BFS traversal.

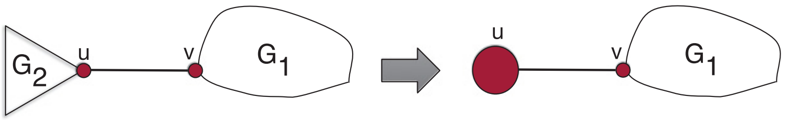


Fig. 3. A toy graph where G_2 is compressed via manipulations and a degree-1 vertex u is obtained.

4.2. Closeness-Preserving Graph Compression

In this section, we present two closeness-preserving techniques that can be used to reduce the number of vertices and edges in a graph: (1) degree-1 vertex removal and (2) side-vertex removal.

4.2.1. Compression with Degree-1 Vertices. A degree-1 vertex is a special instance of a bridge and can be handled in a way that is explained in the previous section. However, the previous approach traverses the entire component once to update the reach and ff values. Here, we propose another approach with $\mathcal{O}(1)$ operations per vertex removal that requires a post-processing after the CC scores of the remaining vertices are computed by the modified kernel.

Figure 3 shows a simple example where a degree-1 vertex u appears after the subgraph G_2 is compressed into a single vertex after a set of graph manipulations. Again we focus on reach and ff values of the vertices that the removed edge is connected to. Regarding vertex v, changes are actually the same as the bridge removal. Since there is no vertex in u's connected component, summation terms are just omitted. Hence, updates are as follows:

$$\texttt{reach}[v] = \texttt{reach}[v] + \texttt{reach}[u], \tag{9}$$

$$\texttt{ff}[v] = \texttt{ff}[v] + \texttt{ff}[u] + \texttt{reach}[u]. \tag{10}$$

Regarding the updates on vertex u, we choose the lazy computation for efficiency, which means that the far[u] is marked to computed once the far[v] is finalized. The only information that needs to be remembered is the difference between those two farness values, which is easy to compute. Considering Figure 3, the final far[u] can be thought as the summation of three terms:

—$\sum_{x \in G_1} \texttt{dst}_{G_1}(x, u) = \sum_{x \in G_1} \texttt{dst}_{G_1}(x, v) + |G_1|$,
—ff[v]; the farness forwarded by v (before removing the edge),
—ff[u]; as shown in line 3 of Algorithm 3.

Note that the first term is expressed in terms of the distances from each vertex to v. For any vertex $x \in G_1$, distance to u is always one more than the distance to v, so $\texttt{dst}_{G_1}(x, u) = \texttt{dst}_{G_1}(x, v) + 1$. Hence, $\sum_{x \in G_1} \texttt{dst}_{G_1}(x, u) = \sum_{x \in G_1} \texttt{dst}_{G_1}(x, v) + |G_1|$.

On the other hand, far[v] can also be thought of combination of two terms:

—$\sum_{x \in G_1} \texttt{dst}_{G_1}(x, v)$,
—updated ff[v].

In summary, we have the following for u:

$$\texttt{far}[u] = \sum_{x \in G_1} \texttt{dst}_{G_1}(x, v) + |G_1| + \texttt{ff}[v] + \texttt{ff}[u]. \tag{11}$$

Substituting $|G_1|$ with $|V| - \texttt{reach}[u]$

$$\texttt{far}[u] = \sum_{x \in G_1} \texttt{dst}_{G_1}(x, v) + |V| - \texttt{reach}[u] + \texttt{ff}[v] + \texttt{ff}[u]. \tag{12}$$

And following for v:

$$\texttt{far}[v] = \sum_{x \in G_1} \texttt{dst}_{G_1}(x, v) + \texttt{ff}[v], \tag{13}$$

where the updated $\texttt{ff}[v]$ can be expanded as follows by Equation (10)

$$\texttt{far}[v] = \sum_{x \in G_1} \texttt{dst}_{G_1}(x, v) + \texttt{ff}[v] + \texttt{ff}[u] + \texttt{reach}[u]. \tag{14}$$

So, the difference by Equations (12) and (14) is

$$\begin{aligned} \texttt{far}[u] - \texttt{far}[v] &= (|V| - \texttt{reach}[u]) - \texttt{reach}[u] & (15)\\ &= |V| - 2 \times \texttt{reach}[u]. & (16) \end{aligned}$$

Hence, once the overall farness value of v is computed, the farness value of u can be computed via a simple addition during the post-processing phase.

4.2.2. Compression with Side Vertices. Let u be a side vertex appearing in a component during the graph manipulation process. No shortest path is passing through u except the ones starting or ending at u, i.e., u is always on the sideways, since $\Gamma(u)$ is a clique. Hence, we can remove u if we compensate the effect of the all shortest paths where u is either source or target. To do this, we initiate a BFS from u in the original graph G as shown in Algorithm 4.

ALGORITHM 4: Side-Vertex Removal BFS for Closeness Centrality

```
  Data: side vertex $u$, $G = (V, E)$, $\texttt{far}[.]$
  $Q \leftarrow$ empty queue
  $Q$.push($u$)
  $\texttt{dst}[u] \leftarrow 0$
  $\texttt{dst}[v] \leftarrow \infty, \forall v \in V \setminus \{u\}$
  while $Q$ is not empty do
      $v \leftarrow Q$.pop()
      for all $w \in \Gamma_G(v)$ do
          if $\texttt{dst}[w] = \infty$ then
              $Q$.push($w$)
              $\texttt{dst}[w] \leftarrow \texttt{dst}[v] + 1$
              $\texttt{far}[u] \leftarrow \texttt{far}[u] + \texttt{dst}[w]$
1             $\texttt{far}[w] \leftarrow \texttt{far}[w] + \texttt{dst}[w]$
          end
      end
  end
  $\texttt{cc}[u] \leftarrow 1/\texttt{far}[u]$
```

The main difference between the BFS in side-vertex removal and in the original implementation in the main loop of Algorithm 1 is line 1 (of Algorithm 4) that adds $\texttt{dst}[w]$ to $\texttt{far}[w]$ for each traversed vertex w. This handles the shortest paths where the side vertex is the target and each traversed vertex is the source. To do that a single variable to store the farness value (as in Algorithm 1) is not sufficient since side-vertex removals update the farness values partially and these updates need to be stored till the end of the graph manipulation process. That is why we used an additional far array to perform side-vertex removal operations, which is also explained at the beginning of Section 4.

This compression technique has a little impact on the overall time since for a side vertex removal, an additional BFS (Algorithm 4) is necessary and it is almost as expensive as the original BFS (of Algorithm 1) that we try to avoid. However, the important benefit is that these removals can reveal new special vertices during the manipulation process that also enable further splits and compression of the graph in a cheaper way.

4.3. Combination of Techniques and Post-Processing

We continuously process a reduction on the graph with split and compression operations until no further reduction is possible. We first perform degree-1 removals since they are the cheapest to handle. Next, we split the graph by first bridges and then articulation vertex clones. The order is important for efficiency since the former is cheaper. We iteratively use these three techniques until no reduction is possible. After that we remove the side vertices to discover new special vertices. The reason behind delaying the side-vertex removals is that its additional BFS requirement makes it expensive compared to the other graph manipulation techniques. Hence, we do not use them until we really need them.

After all the graph manipulation techniques, the original CC kernel given in Algorithm 1 cannot compute the correct centrality values since it does not forward the `ff` values to the other vertices. We apply the modified version presented Algorithm 3 to compute the CC scores once the split and compression operations are done and `reach` and `ff` attributes are fixed.

THEOREM 4.1. *Let $G = (V, E)$ be the original graph and $G' = (V', E')$ be the reduced graph after split and compression operations with* `reach`*,* `ff`*, and* `far` *attributes computed for each vertex $v \in V'$. For all the vertices in V', the CC scores of G computed by Algorithm 1 is the same with the CC scores of G' computed by Algorithm 3.*

PROOF. For a source vertex $s \in V'$ and another vertex $w \neq s$ that is connected to s in G', $\texttt{ff}[w]$ is forwardable to $\texttt{far}[s]$ by using the equation at lines 1 and 2 of Algorithm 3. Remember that for a vertex $w \in G'$, all the $\texttt{reach}[w]$ vertices in $\mathcal{R}_w$ are not connected to s. Although they are represented by w and from s (and from any vertex in the same component), they are reachable only through w. Since the shortest-path distance between s and w is $\texttt{dst}[w]$, the vertices in $\mathcal{R}_w$ are $\texttt{dst}[w]$ more edges far away from s when compared to w. Thus, an additional $\texttt{dst}[w] \times \texttt{reach}[w]$ farness is required while forwarding the $\texttt{ff}[w]$ value to $\texttt{far}[s]$.

At the end of the algorithm (line 3), we have an extra addition of $\texttt{ff}[s]$ to the total farness value of s. It is required since while computing the total farness of s and its `cc` score, we need to consider the farness due to the vertices in $\mathcal{R}_s$. □

4.3.1. Work Filtering with Identical Vertices. If some vertices in G' are identical, i.e., their adjacency lists are the same, the forwardable farness values from other vertices to their overall farness will be the same. Hence, it is possible to combine these vertices and avoid extra computation in Algorithm 3. We use two types of identical vertices: u and v are type-I (or type-II) identical if and only if $\Gamma(u) = \Gamma(v)$ (or $\Gamma(u) \cup \{u\} = \Gamma(v) \cup \{v\}$), as exemplified in Figure 4.

For each identical vertex set, we compute the farness value for only one representative. Let $G' = (V', E')$ be the reduced graph after preprocessing operations, and let $\mathcal{I} \subset V'$ be a set of identical vertices. We select a proxy vertex $u \in \mathcal{I}$, compute its overall farness ($\texttt{far}[u]$) to other vertices and CC score as shown in the main loop of Algorithm 3. Then, for each vertex $v \in \mathcal{I}$, we just need to reflect the farness differences. Assume the identical vertex set is type-I, so distances between each pair of identical vertex ($\texttt{dst}_G(u, v)$ where $u, v \in \mathcal{I}$) is 2. $\texttt{far}[u]$ includes the term that is shown in line 1

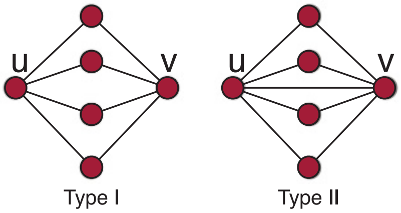


Fig. 4. Type-I (left) and type-II (right) identical vertices u and v.

of Algorithm 3 for all $v \in \mathcal{I}$. Summation of these terms that considers the pairwise distances between all identical vertices can be expressed as follows:

$$\sum_{v \in \mathcal{I} \wedge v \neq u} (2 * \texttt{reach}[v] + \texttt{ff}[v]) = \sum_{v \in \mathcal{I}} (2 * \texttt{reach}[v] + \texttt{ff}[v]) - (2 * \texttt{reach}[u] + \texttt{ff}[u]). \quad (17)$$

Here, far[u] also has the ff[u] as the last addition (line 3 of Algorithm 3). If we subtract that and the term in Equation (17) from far[u], the residual will be a common term that takes place in the farness value of each identical vertex, which is the following:

$$\texttt{far}[u] - \left(\sum_{v \in \mathcal{I}} (2 * \texttt{reach}[v] + \texttt{ff}[v]) - (2 * \texttt{reach}[u] + \texttt{ff}[u]) + \texttt{ff}[u] \right) \quad (18)$$

$$= \texttt{far}[u] - \sum_{v \in \mathcal{I}} (2 * \texttt{reach}[v] + \texttt{ff}[v]) + 2 * \texttt{reach}[u]. \quad (19)$$

Now we find far[w] for an identical vertex $w \in \mathcal{I}$ s.t. $w \neq u$. We actually just add the sum of the terms in line 1 of Algorithm 3 for each identical vertex (Equation (17)) and also include ff[w] (line 3 of Algorithm 3). In summary,

$$\texttt{far}[w] = \texttt{far}[u] - \sum_{v \in \mathcal{I}} (2 * \texttt{reach}[v] + \texttt{ff}[v]) + 2 * \texttt{reach}[u] \quad (20)$$

$$+ \sum_{v \in \mathcal{I}} (2 * \texttt{reach}[v] + \texttt{ff}[v]) - (2 * \texttt{reach}[w] + \texttt{ff}[w]) + \texttt{ff}[w], \quad (21)$$

which gives

$$\texttt{far}[w] = \texttt{far}[u] + 2 * (\texttt{reach}[u] - \texttt{reach}[w]). \quad (22)$$

Hence, difference of farness values just depends on the reach values of vertices. For type-II identical vertices, we replace 2 by 1, since the distance among each identical pair is 1.

4.3.2. Post-Processing for the Degree-1 Vertices. Once Algorithm 3 is done, the only remaining part is computing the CC scores of removed degree-1 vertices since they are not in G' anymore. To do that, we resolve the dependencies created when the degree-1 vertices are being removed. We do a loop on the vertices, and for each vertex u we visit, we check if u's CC score is already computed. If not, we recursively follow the dependencies to find the final representative vertex in G'. While coming back from the recursion path, we use Equation (16) to find the farness and the CC score(s) of the removed degree-1 vertices. Since the dependencies form a tree and at most $\mathcal{O}(1)$ operations are performed per vertex, we need at most $\mathcal{O}(|V|)$ operations to resolve all the dependencies.

5. *BADIOS* FOR BETWEENNESS CENTRALITY

Here, we propose a set of *betweenness-preserving* graph manipulation techniques similar to the ones described for CC. The proposed techniques will make the original graph $G = (V, E)$ smaller and disconnected while preserving the information required to compute the distance-based metrics by using some auxiliary arrays.

5.1. Betweenness-Preserving Graph Splits

To correctly compute the BC scores after splitting G, we use the `reach` attribute as described above and set $\texttt{reach}[v] = 1$ for all $v \in V$ before the manipulations.

5.1.1. Articulation Vertex Cloning. Let u be an articulation vertex in a component C obtained during the preprocessing phase whose removal splits C into k (connected) components C_i for $1 \leq i \leq k$. As in CC, we remove u and keep a local clone u'_i at each component C_i. For BC on *BADIOS*, the `reach` values for each local clone are set with

$$\texttt{reach}[u'_i] = \sum_{v \in C \setminus C_i} \texttt{reach}[v] \tag{23}$$

for $1 \leq i \leq k$.

Algorithm 5 computes the BC scores of the vertices in a split graph. Note that the only differences 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 split graph obtained via preprocessing. Let `bc` and $\texttt{bc}'$ be the scores computed by Bc-Org(G) and Bc-Reach(G'), respectively. We will prove that

$$\texttt{bc}[v] = \sum_{v' \in V' | \mathbf{org}(v') = v} \texttt{bc}'[v'], \tag{24}$$

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

ALGORITHM 5: Bc-Reach: Modified Betweenness Centrality Computation

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

LEMMA 5.1. *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), $\delta_s(v) = \delta_u(v)$. □

LEMMA 5.2. *For any vertex pair $s, t \in V$, there exists exactly one component C of G' that contains a clone of t and a representative of s as two distinct vertices.*

PROOF (BY INDUCTION ON THE NUMBER OF SPLITS). Given $s, t \in V$, the statement is true for the initial (connected) graph G since it contains one clone of each vertex. Assume that it is also true after the ℓth splitting. Let C be this component. When C is further split via t's clone, all but one newly formed (sub)components contains a clone of t as the representative of s. For the remaining component C', $\mathbf{rep}(C', s) = \mathbf{rep}(C, s)$ that is not a clone of t.

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

The local clones of an articulation vertex v, created while splitting, are acting as the original vertex v in their components. Once the `reach` value for each clone is set as in (23), 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 5.3. *Equation (24) is correct after splitting 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. Note that $\texttt{reach}[v'] - 1$ is the number of vertices $t \neq v$ such that t does not have a clone 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 5.1, $\delta_s(v) = \delta_u(v) = \delta_{u'}(v')$. 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 5.2, $\delta_{s'}(v')$ will be added to exactly one clone v' of v. Hence, (24) is correct. □

5.1.2. Bridge Removals. Let $\{u, v\}$ be a bridge in a component C formed during graph manipulations. Let $u' = \mathbf{org}(u)$ and $v' = \mathbf{org}(v)$. As stated in previous section, a bridge removal operation is similar to a splitting via an articulation vertex; however, no new clones of u' or v' are created. Instead, we let u and v act as a clone of v' and u' in the newly created components C_u and C_v that contain u and v, respectively. Similar to (23), 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 5.2 correct. No component contains a distinct representative of u' (v') and clone of v' (u') anymore. Hence, $\delta_v(u')$ and $\delta_u(v')$ will not be added to any clone 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 add $\delta_u(v)$ to $\texttt{bc}'[v]$, where $\delta_u(v)$ is computed by interchanging u and v on the right side of the above equation. Note that Lemma 5.2 is correct for all other vertex pairs.

Corollary 1. *Equation (24) is correct after splitting G with articulation vertices and bridges.*

5.2. Betweenness-Preserving Graph Compression

Here, we present *BADIOS*'s betweenness-preserving compression techniques: (1) degree-1 vertex removal, (2) compression by identical vertices, and (3) side-vertex removal.

5.2.1. Compression with Degree-1 Vertices. As stated before, although a degree-1 vertex removal is a special instance of a graph split with a bridge, we handle them separately to avoid trivial components. Let u be a degree-1 vertex connected to v and appeared in a component C formed during the preprocessing. To remove u, we add $\texttt{reach}[u]$ to $\texttt{reach}[v]$ and increase $\texttt{bc}'[u]$ and $\texttt{bc}'[v]$, 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. *Equation (24) is correct after splitting G with articulation vertices and bridges and compressing it with degree-1 vertices.*

5.2.2. Compression with Identical Vertices. Instead of basic work filtering applied for CC, *BADIOS* uses the type-I and type-II identical vertices to compress the graph further for BC. Hence, it exploits these vertices in a more complex way. To handle the complexity, an $\texttt{ident}$ attribute is assigned 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 $v \in V$.

Let $\mathcal{I}$ be a set of identical vertices formed during the preprocessing phase. We remove all vertices in $\mathcal{I}$ except one, which acts as a proxy for the others. Let v be the proxy vertex for $\mathcal{I}$. 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 2: 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 `reach` attribute into account. Both attributes can be maintained simultaneously. The details are not presented here for brevity. 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. *Equation (24) is correct after splitting G with articulation vertices and bridges, and compressing it with degree-1, and identical vertices.*

5.2.3. Compression with Side Vertices. Let u be a side vertex in a component C formed after a set of manipulations on the original graph G. Since $\Gamma(u)$ is a clique, no shortest path is passing through u. Hence, we can remove u from C by 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 6 shows, the only differences are two additional lines 1 and 2. Note that this extra BFS is as expensive as the original one that we avoid by removing u. As in CC, *BADIOS* performs the side vertex removals since they can yield new special vertices in the graph, which will be used to improve the performance.

ALGORITHM 6: Side-Vertex Removal BFS for Betweenness Centrality

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

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

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

since none of the $\texttt{reach}[w] - 1$ vertices represented by w lies on a $v \rightsquigarrow u$ path and $\delta_{vu}(w) = \delta_{uv}(w)$. The same dependency appears for all vertices represented by u. Line 1 of Algorithm 6 takes into account all these dependencies.

Let $s \in V$ be a vertex s.t. $\mathbf{rep}(C, s) = v \neq u$. When we remove u from C, due to Lemma 5.2, $\delta_s(u) = \delta_v(u)$ will not be added to any clone of $\mathbf{org}(u)$. 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 that 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 Algorithm 6 compensates this loss.

Corollary 4. *Equation (24) is correct after splitting G with articulation vertices and bridges, and compressing it with degree-1, identical, and side vertices.*

5.3. Combining the Techniques

For BC, ordering of heuristics is similar to the CC case, as explained in Section 4.3. *BADIOS* first applies degree-1 removal since it is the cheapest to handle. Next, it splits the graph by first removing the bridges, and then articulation vertices. It then removes 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. Similar to CC, the framework does not use side vertices until it really needs them.

6. EXPERIMENTS

We implemented our framework in `C++`. The code is compiled with `gcc v4.8.1` and optimization flag `-O2`. The graph is kept in memory in the Compressed Storage by Row format (essentially adjacency list that is compact in memory). The experiments are run on a computer with 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 13 real-world networks from the UFL Sparse Matrix Collection (http://www.cise.ufl.edu/research/sparse/matrices/). Their properties are summarized in Table I. They are from different application areas, such as grid (*power*), router (*as-22july06*, *p2p-Gnutella31*), social (*PGPgiantcompo*, *astro-ph*, *cond-mat-2005*, *soc-sign-epinions*, *loc-gowalla*, *amazon0601*, *wiki-Talk*), protein interaction (*protein*), and web networks (*web-NotreDame*, *web-Google*). We symmetrized the directed graphs. We categorized the graphs into two classes: small and large ones (separately shown in Table I).

Our proposed techniques can be combined in many different ways. In this section, we use lower case abbreviations for representing these combined methods. We will use lower case letters *"o"* for the BFS *o*rdering, *"d"* for *d*egree-1 vertices, *"b"* for *b*ridge, *"a"* for *a*rticulation vertices, *"i"* for *i*dentical vertices, and *"s"* for *s*ide vertices. The ordering is performed to improve the cache locality during centrality computation by initiating a BFS from a random source vertex as in Algorithm 1 and renumbering the vertices as their visit order. Using this scheme, for example, abbreviation *das* means that the degree-1 removal is followed by the articulation vertex cloning, which is followed by the side-vertex removal. This pattern is repeated until no further modification is possible.

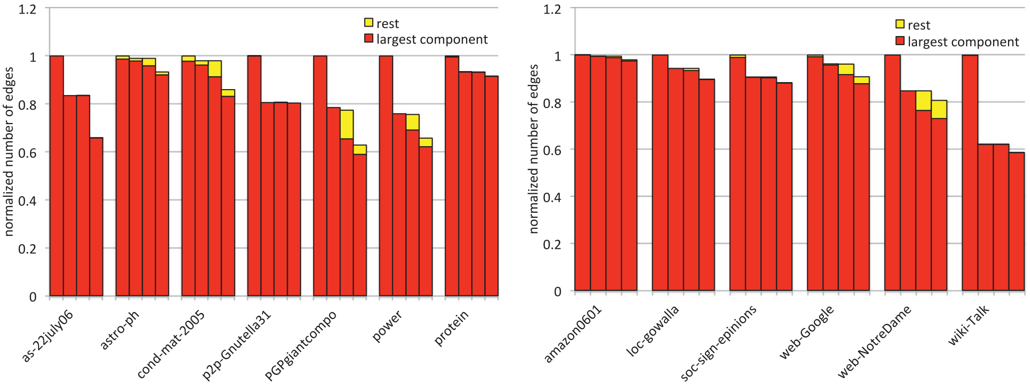
6.1. Closeness Centrality Experiments

We first investigate the efficiency of *BADIOS* on reducing the graphs. We check the number of remaining edges by applying our techniques on the test graphs. Figure 5(a) and (b) shows the number of remaining edges in the reduced graph normalized with respect to the original number of edges in G. We chose the variants, d, da, and das since these manipulations are the only ones that reduce the number of edges or make new articulation vertices appear. We measured the remaining number of edges in the largest

Table I. The Graphs Used in the Experiments

Graph					Time (in seconds)					
Name	$\|V\|$	$\|E\|$	Diam.	Max deg.	BC org.	BC best	BC Sp.	CC org.	CC best	CC Sp.
as-22july06	22.9K	48.4K	11	2390	43.72	8.78	4.9	17.03	5.49	3.1
astro-ph	16.7K	121.2K	14	504	40.56	19.41	2.0	14.10	9.15	1.5
cond-mat-2005	40.4K	175.6K	18	278	217.41	97.67	2.2	79.16	46.21	1.7
p2p-Gnutella31	62.5K	147.8K	11	95	422.09	188.14	2.2	180.27	65.13	2.8
PGPgiantcompo	10.6K	24.3K	24	205	10.99	1.55	7.0	4.63	0.75	6.2
power	4.9K	6.5K	46	19	1.47	0.60	2.4	0.78	0.27	2.8
protein	9.6K	37.0K	14	270	11.76	7.33	1.6	4.12	2.33	1.7
					geometric mean		**2.8**	geometric mean		**2.5**
amazon0601	403K	2,443K	19	2,752	42,656	36,736	1.1	17,653	11,901	1.5
loc-gowalla	196K	950K	12	14,730	5,926	3,692	1.6	2,117	1,138	1.9
soc-sign-epinions	131K	711K	14	3,558	2,193	839	2.6	889	264	3.4
web-Google	875K	4,322K	18	6,332	153,274	27,581	5.5	83,821	22,935	3.7
web-NotreDame	325K	1,090K	27	10,271	7,365	965	7.6	2,736	517	5.3
wiki-Talk	2,394K	4,659K	10	100,029	452,443	56,778	7.9	279,548	22,029	12.7
					geometric mean		**3.4**	geometric mean		**3.7**

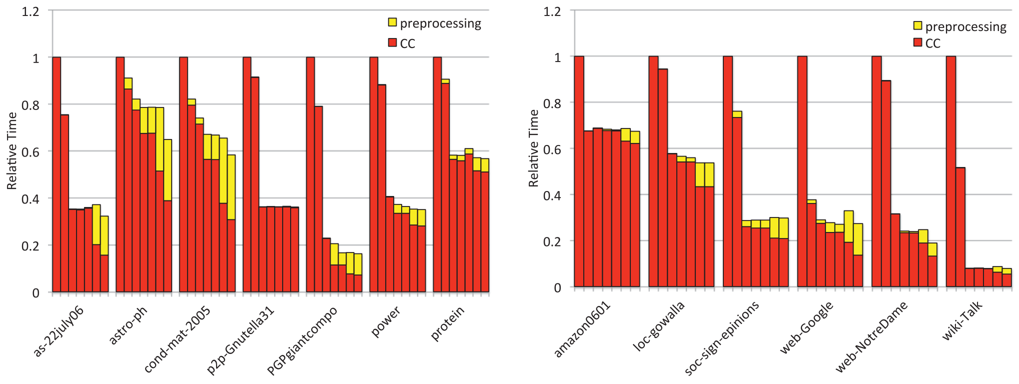
Notes: Diameters and maximum degrees are given along with the size information. Columns *BC org.* and *CC org* show the original execution times of BC and CC computations without any modification, and *BC best* and *CC best* are the minimum execution times achievable via our framework for BC and CC. The names of the graphs are kept short where the full names can be found in the text.
Geometric means of speedups are shown in bold.



(a) Normalized remaining edges for small graphs (b) Normalized remaining edges for large graphs

Fig. 5. The plots on the left and right show the number of remaining edges on the graphs that initially have less than and more than 500K edges, respectively. They show the ratio of remaining edges of the variants, which consecutively reduce the number of edges: base, *d*, *da*, and *das*. The number of remaining edges is normalized w.r.t. the total number of edges in the graph and divided into two: largest connected component and rest of the graph. Compression by removing degree-1 and side vertices reduces the number of edges in the graph and the decrease of the number of edges helps to understand the impact of those heuristics. Articulation vertex cloning helps to split the graph into multiple components and the share of edges in each graph component is also given to show the structure of graphs.

connected component as well as the other components (shown as "rest"). Degree-1 vertex removal (going from first bar to second bar) provides 13% and 14% average reductions in the sizes of small and large graphs, respectively. This result shows that there is a significant amount of degree-1 vertices in real-world graphs, and they can be efficiently utilized by our techniques. When we measure the impact of articulation vertex cloningon the total number of connected components, we observe two facts: (1) there is usually one giant (strongly) connected component in real-world social networks, and



(a) Normalized execution times for small graphs (b) Normalized execution times for large graphs

Fig. 6. The plots on the left and right show the CC computation times on graphs with less than and more than 500K edges, respectively. They show the normalized runtime of the variants: base, *o*, *do*, *dao*, *dbao*, *dbaos*, and *dbaosi*. The times are normalized w.r.t. base and divided into two: preprocessing and the CC computation.

(2) other components are small in size. As can be seen from the second and third bars, articulation-vertex cloning increases the yellow-colored regions in the graph, i.e., splits the graphs. At last, we measure the effect of side vertex removal. The differences between the third and fourth bars show the reduction by side vertex removal. We observe 9% and 5% average reductions in small and large graphs.

Next, we measure the performance of *BADIOS* on CC computation time. We evaluate the preprocessing and computation time separately. Figure 6(a) and (b) presents the runtimes for each combination normalized w.r.t. the implementation of Algorithm 1. For each graph, we tested six different combinations of the improvements proposed in this work: They are denoted with *o*, *do*, *dao*, *dbao*, *dbaos*, and *dbaosi*. For each graph, each figure has seven stacked bars for the six combinations in the order described above plus the base implementation.

In many graph kernels, the order of edge accesses is important due to cache locality. Therefore, we order our graphs after split and compression operations. The second bars for each graph at Figure 6(a) and (b) shows the improvement gained by ordering the graphs. We have 13% and 34% improvements (over the baseline) with ordering for small and big graphs, respectively. Especially, larger graphs benefit more from the graph ordering and the cache is utilized more efficiently.

In general, the preprocessing phase takes little time for all graphs. At most 7% of the overall execution time is spent for graph manipulations on small graphs, and this value is 6% for large graphs. With split and compression operations, *BADIOS* can obtain significant speedup values. When we only remove the degree-1 vertices, we have 16% runtime improvement for small graphs and 54% improvement for large graphs. When Figure 5(a) and (b) is compared with Figure 6(a) and (b), the correlation between the reduction on the number of edges and the improvement on the performance becomes more clear. In addition to degree-1 removal, if we split the input graph with articulation vertex cloning, the speedups increase: In large graphs, this reduces the overall execution time up to 5%. As expected, when there are more articulation vertices in the graph, the speedups are higher. As explained in Section 4.1.2, a bridge always exists between two articulation vertices, but bridge removal is cheaper than articulation vertex cloning. We see the effect of cheap bridge removals when we look at the combination *odab* (fifth bar): In small graphs, we have 4% improvement with articulation vertex cloning plus bridge removal over only articulation vertex cloning.

The side vertex removals turn out to be not efficient. We cannot observe significant speedups when we remove the side vertices in graphs. On the other hand, filtering the work via identical vertices brings good improvements. We gain 8% and 10% in small and large graphs with identical vertex filtering. This shows that there are significant amount of identical vertices in the reduced graph, and they can be utilized for faster solutions.

Overall, we have decent speedup numbers for CC when all the techniques are applied. Table I shows the runtime of the base algorithm, runtime of the combination where all techniques are used, and the speedup obtained by that combination. For the largest graph we have, *wiki-Talk* with 2.3M vertices and 4.6M edges, we reach a speedup of 12.7 over the base implementation.

6.2. Betweenness Centrality Experiments

Here, we experimentally evaluate the performance of *BADIOS* for BC computations. As we did for CC, we measure the preprocessing time and BC computation time separately. Figure 7(a) and (b) presents the runtimes for each combination normalized w.r.t. Brandes' algorithm. For each graph, each figure has seven stacked bars for the seven combinations in the order described in the caption. To compare the reductions on the execution times with the reductions on the number of edges and vertices, in Figure 7(c) and (d), the number of edges remaining in the graph after the preprocessing phase is given for the combinations *d*, *da*, *dai*, and *dasi*.

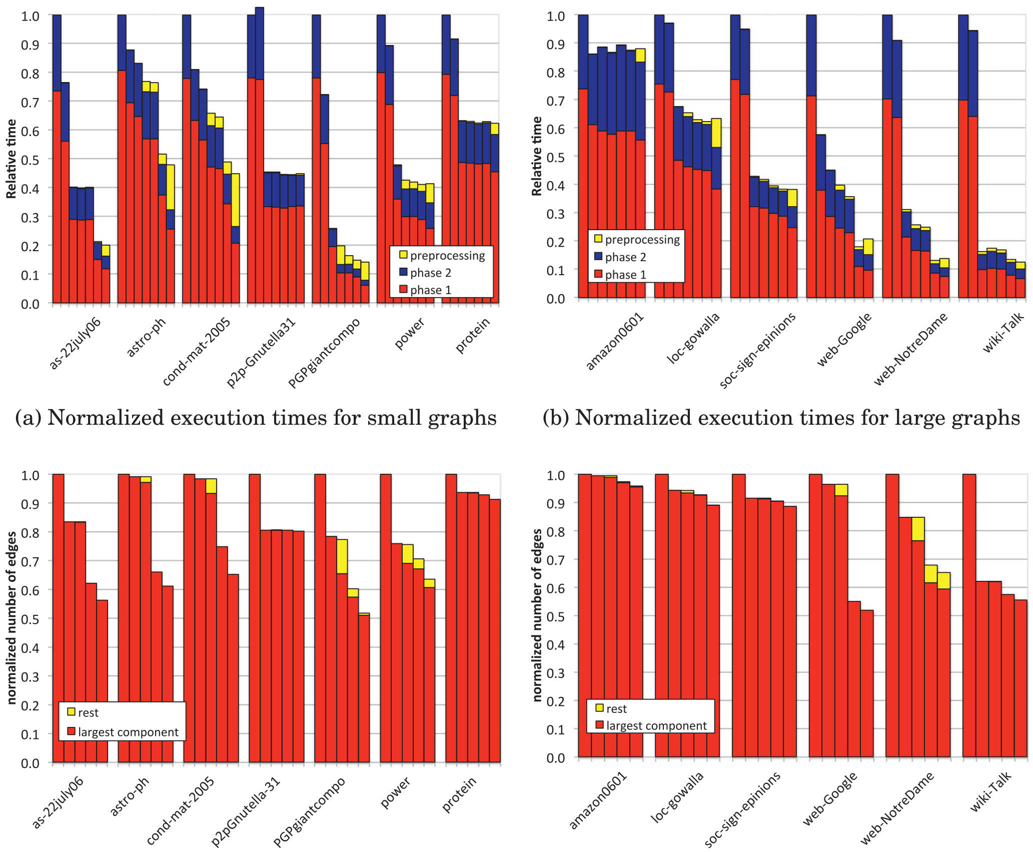
As Figure 7 shows, there is a direct correlation between the amount of edges remaining after the graph manipulations and the overall execution time (except for *soc-sign-epinions* and *loc-gowalla* with 12% and 11% decrease in the number of vertices, respectively). This proves that our rationale behind investigating splitting and compression techniques is valid also for BC.

Table I shows the runtime of the base BC 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 7.9 speedup on large graphs. For *wiki-Talk*, applying all techniques reduced the runtime from 5 days to 16 hours.

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 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.

6.3. Comparison to Previous Work

There exist some other graph manipulation approaches in the literature to speed up the exact BC computation [Baglioni et al. 2012; Puzis et al. 2012], which were published after the release of our technical report (as noted in Sarıyüce et al. [2013b]). Baglioni et al. [2012] proposed to find the trees in a graph in which the BC of nodes are trivial to compute. Furthermore, removal of those trees reduces the size of the graph that in turn speeds up the computation. It is actually same as the compression of degree-1 vertices in *BADIOS*. Another work that leverages special structures in a graph is by Puzis et al. [2012]. They introduce two heuristics. First is about utilizing the vertices with the same neighbor sets, named as structurally equivalent vertices, which we also



(a) Normalized execution times for small graphs

(b) Normalized execution times for large graphs

(c) #remaining edges for small graphs

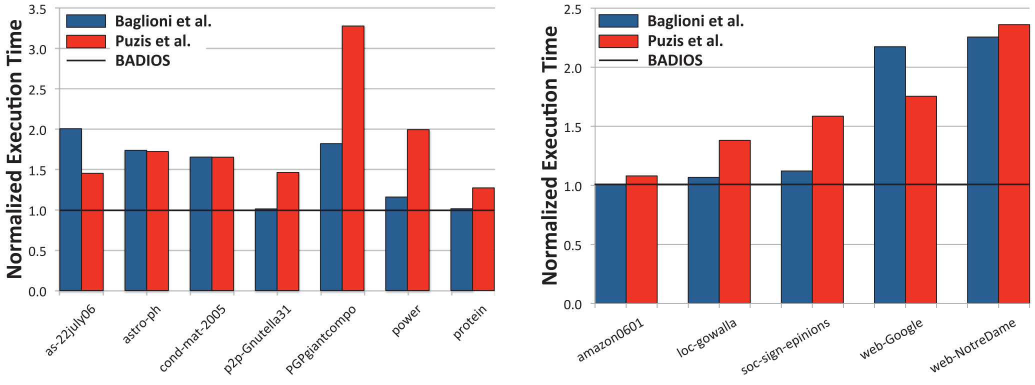
(d) #remaining edges for large graphs

Fig. 7. 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*, and *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 for the combinations *base*, *d*, *da*, *dai*, and *dasi*. Compression by removing degree-1 vertices, identical vertices, and side vertices are useful to reduce the number of edges, and articulation vertex cloning and bridge elimination help to split the graph into multiple components.

defined as the type-I identical vertices. Second heuristic is about partitioning the graph by using articulation points, which is also used by *BADIOS*.

We compare *BADIOS* with those two papers. Both works are implemented in Java, and it is unfair to compare them with *BADIOS* code that is in C++. There are actually some common graphs used in Baglioni et al. [2012], Puzis et al. [2012], and our article that show the huge difference between the implementations of the same Brandes' algorithm [Brandes 2001; Puzis et al. 2012] processes p2p-Gnutella31 graph in more than 52,000 seconds (Figure 7(a) in Puzis et al. [2012]), whereas our implementation finishes the computation in 422 seconds (shown in Table I). Situation is similar for Baglioni et al. [2012]: BC computation by their base implementation takes 564,343 seconds (Table III in Baglioni et al. [2012]) to process soc-sign-epinions graph while our implementation is able to finish the computation in 2,193 seconds (Table I).[1] Actually, both

[1] Different CPU powers and memory sizes can also contribute to the gap between runtimes.



(a) Comparison for small graphs

(b) Comparison for large graphs

Fig. 8. Comparison of *BADIOS* with Baglioni et al. [2012] and Puzis et al. [2012]. Each bar shows corresponding competitor work's normalized time with respect to *BADIOS*. *BADIOS* is faster on all graphs; up to 3.25x and 2.25x speedups are observed over Puzis et al. [2012] and Baglioni et al. [2012], respectively.

works can be implemented by using certain parts of the *BADIOS* framework: Baglioni et al. [2012] uses only degree-1 vertex elimination (same as the *do* variant in Figure 7(a) and (b)—second bar), and [Puzis et al. 2012] uses the articulation vertex cloning and compression based on type-I identical vertices. Both apply the heuristics only once before the BC computation. We implemented those works within the *BADIOS* framework for a fair comparison.

Figure 8 presents the speedup of *BADIOS* over competitors. It is faster than both competitors on all graphs (>1 speedups). In average, Baglioni et al. [2012] is 1.5 times slower on both small and large graphs and for some graphs speed-up is around 2.25 (web-Google and web-NotreDame). Puzis et al. [2012] is even slower than Baglioni et al. [2012] for most graphs. For PGPgiantcompo graph, *BADIOS* is 3.25 times faster. In general, Baglioni et al. [2012] is more efficient than Puzis et al. [2012] although it uses only one heuristic. It is mostly due to the fact that removing trees recursively is more efficient than handling each vertex in them as articulation points. On the other hand, there can be some articulation points that are not degree-1 vertex, but it has been shown that real-world graphs do not contain good articulation points that can separate large components, which limits the benefit of such vertices.

7. RELATED WORK

Several techniques have been proposed to enable fast BC and CC computations in large networks in memory. Here, we summarize the efforts in different directions and show the context of our work in the literature. Note that our scope is limited in algorithms that deal with stationary data; thus, the literature on dynamic, incremental, and streaming algorithms are excluded.

Parallel algorithms have been studied well for the centrality computation thanks to the opportunities in coarser and finer execution levels. These include shared memory solutions for multicore architectures [Madduri et al. 2009], distributed memory algorithms [Lichtenwalter and Chawla 2011], GPU implementations [Shi and Zhang 2011; Jia et al. 2011; Sarıyüce et al. 2013a], vectorization [Sarıyüce et al. 2014], and combination of multiple parallelization approaches [Sarıyüce et al. 2015].

Another direction for faster solutions is approximation algorithms. Brandes and Pich [2007] proposed the first algorithm to estimate centrality values, and Geisberger et al. [2008] generalized that approach. Recently, Riondato and Upfal [2016] proposed very efficient algorithms to approximate BC values of vertices, which significantly outperforms previous approaches with better accuracies.

In this article, our focus is on *sequential* and *exact* solutions for faster centrality computation, which is different than the above approaches. To the best of our knowledge, there are two concurrent works on BC computation since our first release, as noted in our technical report [Sarıyüce et al. 2013b]. However, their focus is limited to BC computation only. The first work introduces degree-1 vertex removal for BC [Baglioni et al. 2012]. In the second, Puzis et al. [2012, 2015] propose to remove articulation vertices and structurally equivalent vertices that correspond to our type-I identical vertices. Comparison of *BADIOS* with those two approaches is presented in Section 6.3.

8. CONCLUSION AND FUTURE WORK

In this work, we proposed the *BADIOS* framework to reduce the execution time of BC and CC computations. The proposed framework employs techniques to split graphs into pieces while keeping and organizing all the information to recompute the shortest path distances, farness values, and pair dependencies that are the building blocks of CC and BC computations. *BADIOS* also employs a set of compression techniques to reduce the number of vertices and edges in the graphs. Combining these techniques provides great reductions in graph sizes and improvements on the performance. 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 CC and BC computations. For BC, we show an average speedup of 2.8 on small graphs and of 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. For CC, the average speedup is 2.4 and 3.6 on small and large networks and 12.7 on the largest graph in our experiments.

As a future work, we plan to leverage further special structures in graphs to speed up the centrality computation. For example, two connected vertices, each with degree of 2, have the exact same BC scores. This property can be utilized for faster BC computation by removing one of the vertices with its adjacent edges.

REFERENCES

M. Baglioni, F. Geraci, M. Pellegrini, and E. Lastres. 2012. Fast exact computation of betweenness centrality in social networks. In *Proceedings of IEEE/ACM International Conference on Advances in Social Network Analysis and Mining (ASONAM)*.

U. Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology* 25, 2 (2001), 163–177.

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

U. Brandes and C. Pich. 2007. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos* 17, 7 (2007), 2303–2318.

Ö. Şimşek and A. G. Barto. 2008. Skill characterization based on betweenness. In *Neural Information Processing Systems*.

L. Freeman. 1977. A set of measures of centrality based upon betweenness. *Sociometry* 4 (1977), 35–41.

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

Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart. 2011. Edge vs. node parallelism for graph centrality metrics. In *Proceedings of GPU Computing Gems: Jade Edition*.

S. Jin, Z. Huang, Y. Chen, D. Chavarria-Miranda, J. Feo, and P. C. Wong. 2010. A novel application of parallel betweenness centrality to power grid contingency analysis. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium*.

S. Kintali. 2008. Betweenness Centrality: Algorithms and Lower Bounds. Arxiv preprint arXiv:0809.1906 (2008).

D. Koschützki and F. Schreiber. 2008. Centrality analysis methods for biological networks and their application to gene regulatory networks. *Gene Regulation and Systems Biology* 2 (2008), 193–201.

V. Krebs. 2002. Mapping networks of terrorist cells. *Connections* 24, 3 (2002), 43–52.

R. Lichtenwalter and N. V. Chawla. 2011. DisNet: A framework for distributed graph computation. In *Proceedings of IEEE/ACM International Conference on Advances in Social Network Analysis and Mining (ASONAM)*.

J.-K. Lou, S. D. Lin, K.-T. Chen, and C.-L. Lei. 2010. What can the temporal social behavior tell us? An estimation of vertex-betweenness using dynamic social information. In *Proceedings of IEEE/ACM International Conference on Advances in Social Network Analysis and Mining (ASONAM)*.

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

K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarria-Miranda. 2009. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium*.

R. Puzis, Y. Elovici, P. Zilberman, S. Dolev, and U. Brandes. 2015. Topology manipulations for speeding betweenness centrality computation. *Journal of Complex Networks* 3, 1 (2015), 84–112.

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

M. Riondato and E. Upfal. 2016. ABRA: Approximating betweenness centrality in static and dynamic graphs with rademacher averages. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1145–1154.

A. E. Sarıyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek. 2013a. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of Workshop on General Purpose Processing Using GPUs (GPGPU), in Conjunction with ASPLOS*.

A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. 2013b. Shattering and compressing networks for betweenness centrality. In *Proceedings of the SIAM International Conference on Data Mining, SDM*. An extended version is available as a Tech Rep on ArXiv http://arxiv.org/abs/1209.6007.

A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. 2014. Hardware/software vectorization for closeness centrality on multi-/many-core architectures. In *Proceedings of the 28th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications (MTAAP)*.

A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. 2015. Regularizing graph centrality computations. *Journal of Parallel and Distributed Computing* 76, C (Feb. 2015), 106–119.

Z. Shi and B. Zhang. 2011. Fast network centrality analysis using GPUs. *BMC Bioinformatics* 12 (2011), 149.

Received March 2015; revised September 2016; accepted November 2016