**Abstract**

The betweenness and closeness metrics have always been intriguing and used in many analyses. 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**, which manipulates the graph by compressing it and splitting into pieces so that the centrality computation can be handled independently for each piece. Although **BADIOS** is designed and fine-tuned for exact betweenness and closeness centrality, it can easily be adapted for approximate solutions as well. 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, we achieve to decrease the closeness computation time from more than 3 days to 6 hours (12.7x speedup).

# Graph Manipulations for Fast Centrality Computation

AHMET ERDEM SARIYÜCE
The Ohio State University
KAMER KAYA
Sabancı University
ERIK SAULE
University of North Carolina at Charlotte
ÜMİT V. ÇATALYÜREK
The Ohio State University

November 21, 2014

## 1 Introduction

Centrality metrics are crucial for detecting the central and influential nodes in various types of networks such as social networks [LdLCL10], biological networks [KS08], power networks [JHC+10], covert networks [Kre02] and decision/action networks [cB08]. The *betweenness* and *closeness* are two intriguing metrics and have been implemented in several tools which are widely used in practice for analyzing networks and graphs [LAB+12]. The betweenness centrality (BC) score of a node is the sum of the fractions of the shortest paths between node pairs that pass through the node of interest [Fre77], whereas the closeness centrality (CC) score of a node is the inverse of the sum of shortest distances from the node of interest to all other nodes. Hence, contribution/load/influence/effectiveness of a node, while disseminating information through a network, is determined with betweenness/closeness metrics. Although BC and CC have been proved to be successful for network analysis, computing the centrality scores of all the nodes in a network is expensive. Brandes proposed an algorithm for computing BC with $\mathcal{O}(nm)$ and $\mathcal{O}(nm + n^2 \log n)$ time and $\mathcal{O}(n + m)$ space complexity for unweighted and weighted networks, respectively, where $n$ is the number of nodes and $m$ is the number of node-node interactions in the network [Bra01]. Brandes' algorithm is currently the best algorithm for BC computations and it is unlikely that general algorithms with better asymptotic complexity can be designed [Kin08]. However, it is not fast enough to handle Facebook's billion or Twitter's 200 million users.

We propose the **BADIOS** framework which uses a set of techniques (based on **B**ridges, **A**rticulation, **D**egree-1, and **I**dentical vertices, **O**rdering, and **S**ide vertices) for faster betweenness and closeness centrality computation. The framework splits the network and reduces its size so that the BC and CC scores of the nodes in two different pieces of network can be computed correctly and independently, and hence, in a more efficient manner. It also preorders the graph to improve cache utilization.

For the sake of simplicity, we consider only standard, shortest-path vertex-betweenness and vertex-closeness centrality 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* [Bra08]. **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.

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 2.8 on small graphs and 3.8 on large ones. In particular, for the largest graph we use, with 2.3M vertices and 4.6M edges, the computation time is reduced from more than 5 days to less than 16 hours. For CC, the average speedup is 2.4 and 3.6 on small and large networks.

The rest of the paper 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 paper with 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 node is represented by a vertex in $V$, and a node-node interaction is represented by an edge in $E$. Let $\Gamma(v)$ be the set of vertices which are 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 $E$, respectively. 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 closeness centrality of $u$ can 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 $\texttt{cc}[u] = 0$.

For a sparse unweighted graph $G = (V, E)$ the complexity of CC computation is $\mathcal{O}(n(m + n))$ [Bra01]. 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.

## 2.2 Betweenness centrality

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

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

**Algorithm 1:** CC-ORG: Closeness centrality computation kernel

```
   Data: G = (V, E)
   Output: cc[.]
 1 for each s ∈ V do
 2     Q ← empty queue
 3     Q.push(s)
 4     dst[s] ← 0
 5     far ← 0
 6     cc[s] ← 0
 7     dst[v] ← ∞, ∀v ∈ V \ {s}
 8     while Q is not empty do
 9         v ← Q.pop()
10         for all w ∈ Γ_G(v) do
11             if dst[w] = ∞ then
12                 Q.push(w)
13                 dst[w] ← dst[v] + 1
14                 far ← far + dst[w]
15     cc[s] ← 1/far
16 return cc[.]
```

**Algorithm 2:** BC-ORG: Betweenness centrality computation kernel

```
   Data: G = (V, E)
 1 bc[v] ← 0, ∀v ∈ V
 2 for each s ∈ V do
 3     S ← empty stack, Q ← empty queue
 4     P[v] ← empty list, σ[v] ← 0
 5     dst[v] ← ∞, ∀v ∈ V \ {s}
 6     Q.push(s), σ[s] ← 1, dst[s] ← 0
       ▷Phase 1: BFS from s
 7     while Q is not empty do
 8         v ← Q.pop(), S.push(v)
 9         for all w ∈ Γ(v) do
10             if dst[w] = ∞ then
11                 Q.push(w)
12                 dst[w] ← dst[v] + 1
13             if dst[w] = dst[v] + 1 then
15                 σ[w] ← σ[w] + σ[v]
16                 P[w].push(v)
       ▷Phase 2: Back propagation
17     δ[v] ← 1/σ[v], ∀v ∈ V
18     while S is not empty do
19         w ← S.pop()
20         for v ∈ P[w] do
22             δ[v] ← δ[v] + δ[w]
23         if w ≠ s then
25             bc[w] ← bc[w] + (δ[w] × σ[w] − 1)
26 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 [Bra01]. 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 $\texttt{d}_s(u)$ and $\texttt{d}_s(v)$ are the shortest distances from $s$ to $u$ and $v$, respectively. $\texttt{P}_s$ defines the *shortest paths graph* rooted in $s$. Brandes observed that the accumulated dependency values can be computed recursively:

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

To compute $\delta_s(v)$ for all $v \in V \setminus \{s\}$, Brandes' algorithm uses a two-phase approach (Algorithm 2). First, a breadth first search (BFS) is initiated from $s$ to compute $\sigma_{sv}$ and $\texttt{P}_s(v)$ for each $v$. Then, in a *back propagation* phase, $\delta_s(v)$ is computed for all $v \in V$ in a bottom-up manner by using (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 the introduction, closeness- and betweenness-based network and 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 which 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 it 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, $\texttt{bc}[u]$ and $\texttt{bc}[v]$ ($\texttt{cc}[u]$ and $\texttt{cc}[v]$) 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 $G_\ell$ be the one after the $\ell$th splitting/compression operation. The $\ell + 1$th operation modifies a single connected component of $G_\ell$ 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 $v \in V$, $\texttt{bc}[v]$ is equal to the number of pairs communicating via $v$, i.e., $\texttt{bc}[v] = 2 \times ((l_v r_v) + (n - l_v - r_v - 1)(l_v + r_v))$ where $l_v$ and $r_v$ are the number of vertices in the left and right subtrees of $v$, respectively. This approach takes only $\mathcal{O}(n)$ time. These equations can also be modified for closeness-centrality computations and a linear-time CC algorithm can easily 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

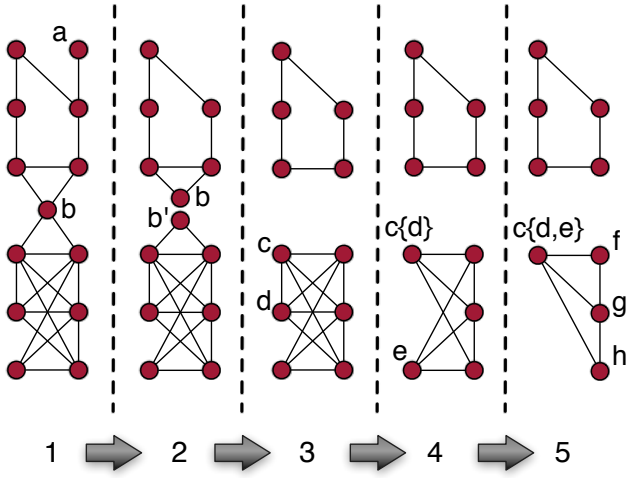



Figure 1: (1) $a$ is a degree-1 vertex and $b$ is an articulation vertex. The framework removes $a$ and create a clone $b'$ to represent $b$ in the bottom component. (2) There is no degree-1, articulation, or identical vertex, or a bridge. Vertices $b$ and $b'$ are now side vertices and they are removed.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

identical, degree-1, articulation, and side vertices can appear. And lastly, new identical and degree-1 vertices can be discovered when a side vertex is removed from the graph. To fully reduce the graph 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 closeness centrality which will be our main application while describing the proposed approach.

For simplicity, we assume that graph is initially connected. In order to correctly compute the shortest-path distances and closeness centrality values after reduction, we keep a representative vertex id for some of the vertices removed from the graph during the process. We also assign two auxiliary attributes to all the vertices: `reach` and `ff` (*forwardable farness*).

As explained above, **BADIOS** compresses the graph $G$, splits it into multiple disconnected 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 closeness centrality values of all the vertices in $C'$. We use $\texttt{reach}[u] = |\mathcal{R}_u|$ to denote the number of vertices represented by $u$.

In addition to `reach`, we assign another attribute `ff` to each vertex where at any time of the graph

manipulation process

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

The correctness of the proposed approach heavily depends on the correctness of the updates on these attributes during the process. Before the manipulations, $\texttt{reach}[u]$ is set to 1 for each $u \in V$ since there is only one vertex (itself) in $\mathcal{R}_u$. Similarly, $\texttt{ff}[u]$ is set to 0 since $\texttt{dst}_G(u, u) = 0$.

## 4.1 Closeness-preserving graph splits

We used two approaches to split the graphs into multiple disconnected 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 and more effective than articulation vertex cloning and 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 and BC computations, to keep the relation between the clones and the original vertex, we use a mapping **org** from $V'$ to $V$ where $\textbf{org}(u'_i)$ is original vertex $u \in V$ for a clone $u'_i \in V$. At any time of a (BC or 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 $\textbf{rep}(C, u)$. Note that each local clone is a representative of its original.

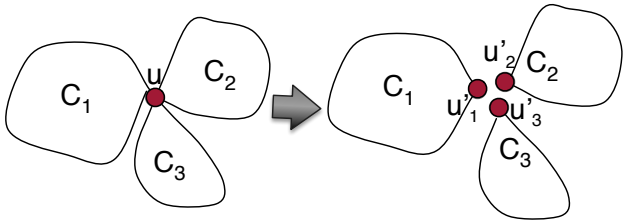



Figure 2: Articulation vertex cloning on a toy graph with three disconnected components after the graph manipulation.

The cloning operation keeps the number of edges constant but increases the number of vertices in the graph. The `reach` value for each clone $u'_i$ is set to

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

and its forwardable farness is set to

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

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 $v$ and $w$ are not connected anymore, the original CC kernel in Algorithm 1 will not

compute the correct closeness centrality values. To alleviate this, we will modify the original kernel later 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. Furthermore, this value will be equal to 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 disconnected 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 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 which contain $u$ and $v$, respectively. We update the `reach` values of $u$ and $v$ as follows:

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

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

Consecutively, the `ff` values are updated as

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

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

where $\texttt{reach}[v]$ and $\texttt{reach}[v]$ are the recently updated values from (7) and (8). Note that the above equations add the forwardable farness value to each other in addition to the total distance we lose by disconnecting a connected component into two. The last `reach` term is required since $\texttt{reach}[v]$ ($\texttt{reach}[u]$) vertices added to $\mathcal{R}_u$ ($\mathcal{R}_v$), and for all these vertices, $v$ ($u$) is one edge closer than $u$ ($v$). Again these values will be propagated to the other vertices in $C_u$ and $C_v$ by the modified CC kernel that will be described later.

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 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.

### 4.2 Closeness-preserving graph compression

In this section, we present two closeness-preserving techniques which 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 as 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 which requires a post-processing after the CC scores of the remaining vertices are computed by the modified kernel.

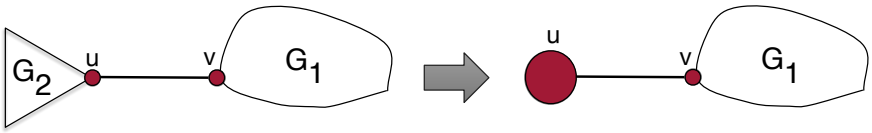



Figure 3: A toy graph where $G_2$ is compressed via manipulations and a degree-1 vertex $u$ is obtained.

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. To remove $u$, which is connected to $v$, three operations need to be performed: (1) an update on $\texttt{reach}[v]$, (2) an update on $\texttt{ff}[v]$, and (3) setting $u$ as a dependent of $v$ for post-processing. When $u$ is removed, all the vertices that were being represented by $u$ (which are the vertices in $G_2$) will be represented by $v$. Hence, the new value of $\texttt{reach}[v]$ is updated as

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

The forwardable farness of $u$, i.e., $\texttt{ff}[u]$, needs to be added to $\texttt{ff}[v]$ as

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

Similar to the bridge removal case, the last term $\texttt{reach}[u]$ is required in the equation since all the $\texttt{reach}[u]$ vertices that changed their representative to $v$ were one edge closer to $u$ compared to $v$. As the last operation, we mark that $u$ is dependent to $v$ and the difference between the overall farness values of $u$ and $v$ is set to

$$\texttt{far}[u] - \texttt{far}[v] = (|V| - \texttt{reach}[u]) - \texttt{reach}[u] \tag{11}$$

$$= |V| - 2 \times \texttt{reach}[u]. \tag{12}$$

The first term $(|V| - \texttt{reach}[u])$ in the summation is added since all the vertices in $V$ are one edge far away, except the ones in $\mathcal{R}_u$, to $u$ compared to $v$. Similarly, all the vertices in $\mathcal{R}_u$ are one edge closer to $u$. Thus we have an additional $-\texttt{reach}[u]$ in (11). Sum of these two terms give the dependency equation in (12), i.e., the difference in $u$ and $v$'s farness. Hence, once the overall farness value of $v$ is computed, the farness value of $u$ can be computed via a simple addition via a 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. Since $\Gamma(u)$ is a clique, except the ones starting or ending at $u$, no shortest path is passing through $u$, i.e., $u$ is always on the sideways. Hence, we can remove $u$ if we compensate the effect of the shortest $s \rightsquigarrow t$ paths where $u$ is either $s$ or $t$. To do this, we initiate a BFS from $u$ in the original graph $G$ as shown in Algorithm 3.

The main difference between a BFS in side-vertex removal and in the original implementation in Algorithm 1 is line 13 (of Algorithm 3) which adds $\texttt{dst}[w]$ to $\texttt{far}[w]$ for each traversed vertex $w$. 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. Hence, we used an additional `far` array to perform side-vertex removal operations.

This compression technique has a little impact of the overall time since for a side vertex removal, an additional BFS (Algorithm 3) is necessary and it is almost as expensive as the original BFS (of Algorithm 1) we try to avoid. However, these removals can make new special vertices appear during the manipulation process which enable further splits and compression of the graph in a cheaper way.

**Algorithm 3:** Side-vertex removal BFS for closeness centrality

```
   Data: side vertex u, G = (V, E), far[.]
 1 Q ← empty queue
 2 Q.push(u)
 3 dst[u] ← 0
 4 dst[v] ← ∞, ∀v ∈ V \ {u}
 5 while Q is not empty do
 6 |   v ← Q.pop()
 7 |   for all w ∈ Γ_G(v) do
 8 |   |   if dst[w] = ∞ then
 9 |   |   |   Q.push(w)
10 |   |   |   dst[w] ← dst[v] + 1
11 |   |   |   far[u] ← far[u] + dst[w]
13 |   |   |   far[w] ← far[w] + dst[w]
14 cc[u] ← 1/far[u]
```

## 4.3 Combining and post-processing

We continuously process a reduction on the graph with split and compression operations until no further reduction 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 than the latter. 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 a modified version as shown in Algorithm 4 to compute the CC scores once the split and compression operations are done and `reach` and `ff` attributes are fixed.

**Algorithm 4:** CC-REACH: Modified closeness centrality computation

```
   Data: G' = (V', E'), ff[.], reach[.], far[.]
   Output: cc[.]
 1 for each s ∈ V' do
 2 |   ··· ▷same as CC-ORG
 3 |   while Q is not empty do
 4 |   |   v ← Q.pop()
 5 |   |   for all w ∈ Γ_G'(v) do
 6 |   |   |   if dst[w] = ∞ then
 7 |   |   |   |   Q.push(w)
 8 |   |   |   |   dst[w] ← dst[v] + 1
10 |   |   |   |   fwd ← ff[w] + (dst[w] × reach[w])
12 |   |   |   |   far[s] ← far[s] + fwd
14 |   far[s] ← far[s] + ff[s]
15 |   cc[s] ← 1/far[s]
16 return cc[.]
```

**Theorem 1.** *Let $G = (V, E)$ be the original graph and $G' = (V', E')$ is the reduced graph after split and compression operations with* `reach`*,* `ff`*, and* `far` *attributes computed for each vertex $v \in V'$. Assuming these*

*attributes are correct, 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 4.*

*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 10 and 12 of Algorithm 4. Remember that for a vertex $w \in G'$, all the $\texttt{reach}[w]$ vertices in $\mathcal{R}_w$ are not connected to $s$. Hence, 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 14), 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 $\texttt{cc}$ 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 4. We use 2 types of identical vertices: Vertices $u$ and $v$ are type-I (or type-II) identical if and only if $\Gamma(u) = \Gamma(v)$ (or $\Gamma(u) \cup \{u\} = \Gamma(v) \cup \{v\}$), as exemplified in Figure 4.

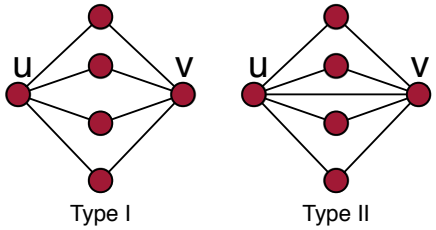



Figure 4: Type-I (left) and type-II (right) identical vertices $u$ and $v$.

The compression works as follows: 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 to other vertices and CC score as shown in Algorithm 4. Then for a vertex $v \in \mathcal{I}$, we compute

$$\texttt{far}[v] = \texttt{far}[u] - k \times (\texttt{reach}[v] - \texttt{reach}[u]), \tag{13}$$

$$\texttt{cc}[v] = 1/\texttt{far}[v], \tag{14}$$

where $k$, the shortest distance between two identical vertices, is 2 for a type-1 identical vertex set, and 1 for a type-2 identical vertex set. Note that the only difference between the farness values is $k \times (\texttt{reach}[v] - \texttt{reach}[u])$ according to the lines 10, 12, and 14.

#### 4.3.2 Post-processing for the degree-1 vertices

Once Algorithm 4 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 (12) 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 closeness centrality. 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 betweenness centrality on **BADIOS**, the `reach` values for each local clone is set with

$$\texttt{reach}[u'_i] = \sum_{v \in C \setminus C_i} \texttt{reach}[v] \tag{15}$$

for $1 \leq i \leq k$.

**Algorithm 5:** Bc-Reach: Modified betweenness centrality computation

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

Algorithm 5 computes the BC scores of the vertices in a split graph. Note that the only difference with Bc-Org are lines 7 and 15, 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 `bc'` be the scores computed by Bc-Org$(G)$ and Bc-Reach$(G')$, respectively. We will prove that

$$\texttt{bc}[v] = \sum_{v' \in V' | \mathbf{org}(v') = v} \texttt{bc}'[v'], \tag{16}$$

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

**Lemma 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 2.** *For any vertex pair $s, t \in V$, there exists exactly one component $C$ of $G'$ which 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 $\ell$-th 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 $C'$, $\mathbf{rep}(C', s) = \mathbf{rep}(C, s)$ which 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 (15), line 7 of Bc-Reach handles the BC contributions from each new component (except the one containing the source), and line 15 of Bc-Reach fixes the contribution of vertices reachable only via the source $s$.

**Theorem 2.** *Eq. 16 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 $V$, respectively. 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 7 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 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 15 of Bc-Reach. Furthermore, according to Lemma 2, $\delta_{s'}(v')$ will be added to exactly one clone $v'$ of $v$. Hence, (16) 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 above, 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 $C_v$ which contain $u$ and $v$, respectively. Similar to (15), 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 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 $v$ in the right side of the above equation. Note that Lemma 2 is correct for all other vertex pairs.

**Corollary 1.** *Eq. 16 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 $\texttt{bc}'[v]$, respectively, with

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

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

**Corollary 2.** *Eq. 16 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 `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 15 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 22 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 25 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.** *Eq. 16 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 12 and 14. Note that this extra BFS is as expensive as the original one 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'`
1 $\cdots$ ▷same as Bc-Reach
2 **while** $Q$ *is not empty* **do**
3 $\quad\cdots$ ▷same as the BFS in Bc-Reach
4 $\delta[v] \leftarrow \texttt{reach}[v] - 1, \forall v \in V_\ell$
5 **while** $S$ *is not empty* **do**
6 $\quad w \leftarrow S$.pop()
7 $\quad$ **for** $v \in \texttt{P}[w]$ **do**
8 $\quad\quad \delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
9 $\quad$ **if** $w \neq s$ **then**
10 $\quad\quad \texttt{bc}'[w] \leftarrow \texttt{bc}'[w] + (\texttt{reach}[s] \times \delta[w]) +$
12 $\quad\quad\quad\quad (\texttt{reach}[s] \times (\delta[w] - (\texttt{reach}[w] - 1))$
14 $\texttt{bc}'[s] \leftarrow \texttt{bc}'[s] + (\texttt{reach}[s] - 1) \times \delta[s]$
15 **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 12 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 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 which are represented by a vertex in $C - u$, we add

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

to $\texttt{bc}'[u]$ after removing $u$ from $C$. Line 14 of Algorithm 6 compensates this loss.

**Corollary 4.** *Eq. 16 is correct after splitting $G$ with articulation vertices and bridges, and compressing it with degree-1, identical, and side vertices.*

## 5.3 Combining and post-processing

For betweenness centrality, **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

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

Table 1: The graphs used in the experiments. Columns *BC org.* and *CC org.* show the original execution times of BC and CC computation 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.

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.

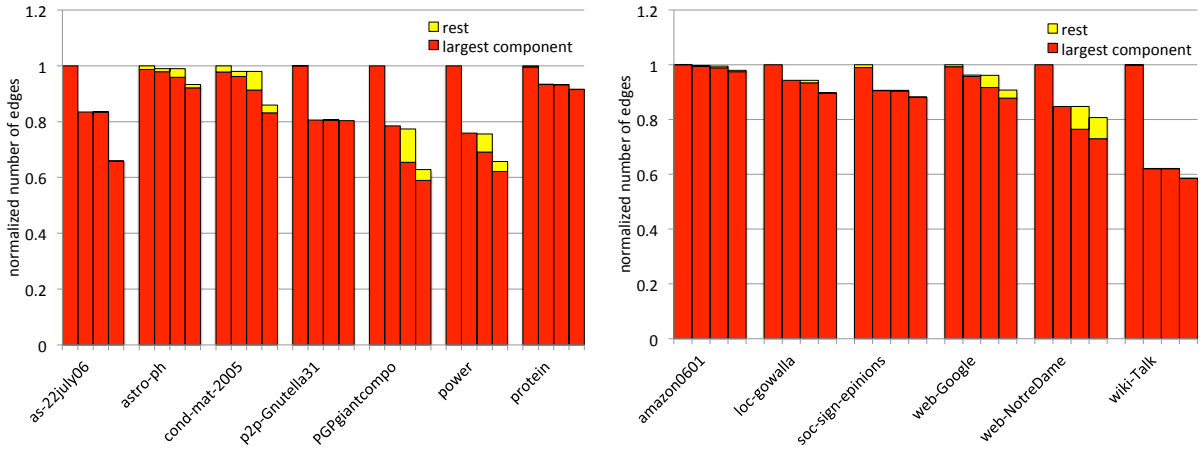
# 6 Experiments

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 adjacency list format. 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 1. They are from different application areas, such as grid (*power*), router (*as-22july06*, *p2p-Gnutella31*), social (*PGP-giantcompo*, *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 1).

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.

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. Figures 5a and 5b show 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 connected component as well as the other components (shown as "rest"). Degree-1 vertex removal (going from 1st bar to 2nd 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 cloning on total




(a) Normalized remaining edges for small graphs

(b) Normalized remaining edges for large graphs

Figure 5: The plots on the left and right show the number of remaining edges on the graphs which 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*, *das*. The number of remaining edges are normalized w.r.t. total number of edges in the graph and divided into two: largest connected component and rest of the graph.

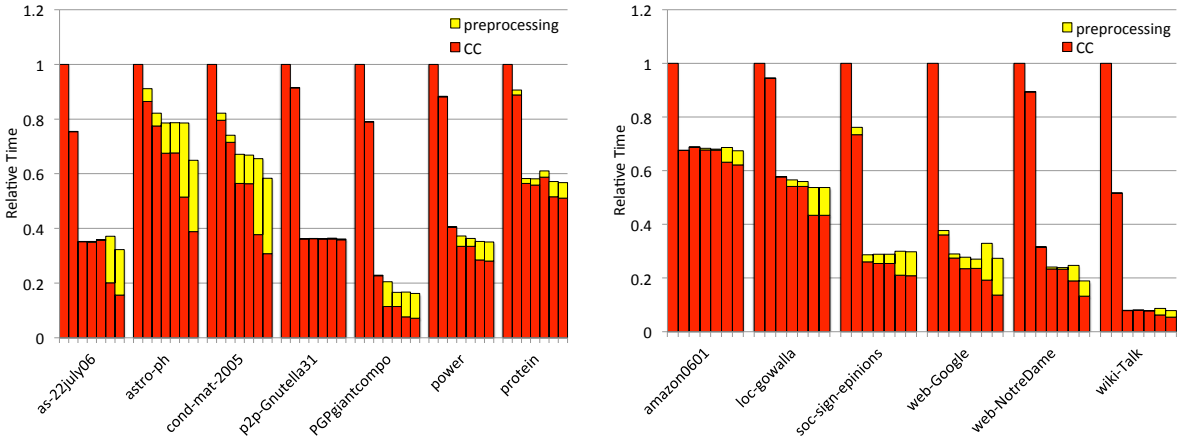
number of connected components, we observe two facts: (1) there is usually one giant (strongly) connected component in real-world social networks, and (2) other components are small in size. As can be seen from the 2nd and 3rd bars, articulation-vertex cloning increases the yellow colored regions in the graph, i.e., splits the graphs. Lastly, we measure the effect of side vertex removal. The differences between the 3rd and 4th bars show the reduction by side vertex removal. We observe 9% and 5% average reductions in small and large graphs.

## 6.1 Closeness centrality experiments

We measure the performance of **BADIOS** on CC computation time. We evaluate the preprocessing and computation time separately. Figures 6a and 6b present the runtimes for each combination normalized w.r.t. the implementation of Algorithm 1. For each graph, we tested 6 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 7 stacked bars for the 6 combinations in the order described above plus the base implementation.

In many graph kernels, the order of edge accesses is important to due to cache locality. Therefore, we order our graphs after split and compression operations. The second bars for each graph at Figures 6a and 6b show 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 Figures 5a and 5b, are compared with Figures 6a and 6b, the correlation between the reduction on the number of edges and the improvement on the performance becomes more clear. Furthermore, we observe larger speedup values for the smaller graphs. 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




(a) Normalized execution times for small graphs

(b) Normalized execution times for large graphs

Figure 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*, *dbaosi*. The times are normalized w.r.t. base and divided into two: preprocessing, and the CC computation.

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 is 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* (5th 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 can not 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.

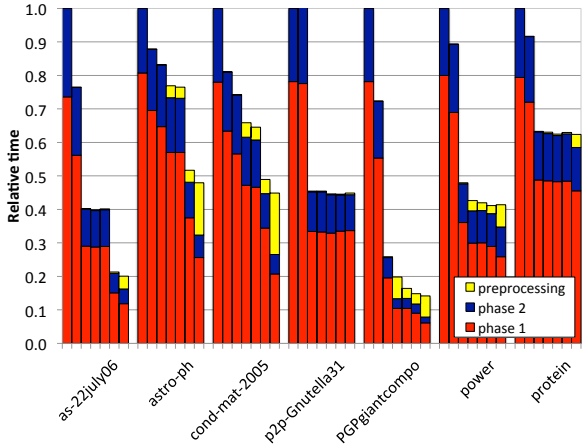
Overall, we have decent speedup numbers for CC when all the techniques are applied. Table 1 shows the runtime of the base algorithm, runtime of the combination where all techniques are used, and the speedup obtained by that combination. For the largest graph we have, *wiki-Talk* with 2.3M vertices and 4.6M edges, we reach 12.7 speedup over base implementation.

## 6.2 Betweenness centrality experiments

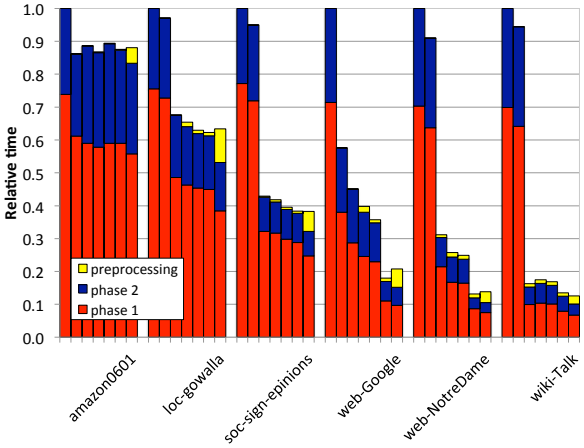
Here we experimentally evaluate the performance of **BADIOS** for betweenness centrality computations. As we did for CC, we measure the preprocessing time and BC computation time separately. Figures 7a and 7b present the runtimes for each combination normalized w.r.t. Brandes' algorithm. For each graph, each figure has 7 stacked bars for the 7 combinations in the order described in the caption. To compare the reductions on the execution times with the reductions on the number of edges and vertices, in Figures 7c–7d, the number of edges remaining in the graph after the preprocessing phase are 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*. This proves that our rationale behind investigating splitting and compression techniques is valid also for BC.

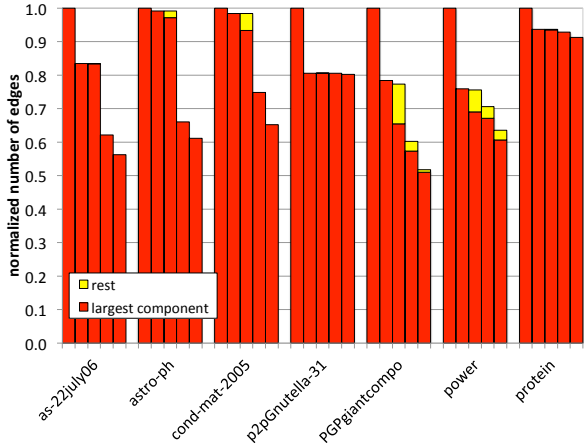
Table 1 shows the runtime of the base BC algorithm as well as the runtime of the combination that lead



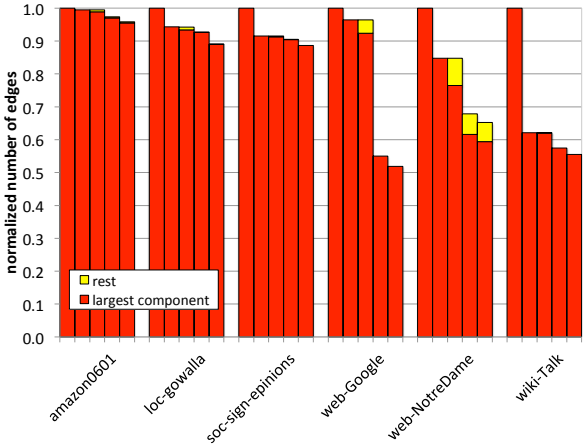

(a) Normalized execution times for small graphs




(b) Normalized execution times for large graphs




(c) #remaining edges for small graphs




(d) #remaining edges for large graphs

Figure 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*, *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.

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.

# 7 Related Work

Several techniques have been proposed to cope with large networks with limited success either by using approximate computations [BP07, GSS08], or by throwing hardware resources to the problem by parallelizing the computations on distributed memory architectures [LC11], multicore CPUs [MEJ+09], and GPUs [SZ11, JLH+11].

To the best of our knowledge, there are two concurrent works since our first release, noted in our technical report [SSKÇ13]. However, their focus is limited to BC computation only. The first work introduces degree-1 vertex removal for BC [BGPL12]. In the second, Puzis et al. propose to remove articulation vertices and structurally equivalent vertices which correspond to our type-I identical vertices [PZE+12]. We did not compare our speedups with theirs for three reasons: the techniques they use form only a subset of the techniques we proposed in this work, they are not well integrated as we did in **BADIOS**, and even our base implementation is already 40–45 times faster than their fastest algorithm (see the results for *soc-sign-epinions* [BGPL12] and *p2p-Gnutella31* [PZE+12]). We believe that an efficient implementation of a novel algorithm is mandatory to evaluate any improvement.

# 8 Conclusion and Future Work

In this work, we proposed the **BADIOS** framework to reduce the execution time of betweenness and closeness centrality 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 which 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 2.8 on small graphs and 3.8 on large ones. In particular, for the largest graph we use, with 2.3M vertices and 4.6M edges, the computation time is reduced from more than 5 days to less than 16 hours. For CC, the average speedup is 2.4x and 3.6x on small and large networks and 12.7x 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

[BGPL12] M. Baglioni, F. Geraci, M. Pellegrini, and E. Lastres. Fast exact computation of betweenness centrality in social networks. In *ASONAM*, 2012.

[BP07] U. Brandes and C. Pich. Centrality estimation in large networks. *I. J. Bifurcation and Chaos*, 17(7), 2007.

[Bra01] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2), 2001.

[Bra08] U. Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2), 2008.

[cB08] Ö. Şimşek and A. G. Barto. Skill characterization based on betweenness. In *NIPS*, 2008.

[Fre77] L. Freeman. A set of measures of centrality based upon betweenness. *Sociometry*, 4, 1977.

[GSS08] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, 2008.

[JHC+10] 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*, 2010.

[JLH+11] 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*. 2011.

[Kin08] S. Kintali. Betweenness centrality : Algorithms and lower bounds. *CoRR*, abs/0809.1906, 2008.

[Kre02] V. Krebs. Mapping networks of terrorist cells. *Connections*, 24, 2002.

[KS08] Dirk Koschützki and Falk Schreiber. Centrality analysis methods for biological networks and their application to gene regulatory networks. *Gene Regulation and Systems Biology*, 2, 2008.

[LAB+12] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *Proc. of SDM*, 2012.

[LC11] R. Lichtenwalter and N. V. Chawla. DisNet: A framework for distributed graph computation. In *ASONAM*, 2011.

[LdLCL10] J-K. Lou, S d. Lin, K-T. Chen, and C-L. Lei. What can the temporal social behavior tell us? An estimation of vertex-betweenness using dynamic social information. In *ASONAM*, 2010.

[MEJ+09] 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*, 2009.

[PZE+12] R. Puzis, P. Zilberman, Y. Elovici, S. Dolev, and U. Brandes. Heuristics for speeding up betweenness centrality computation. In *SocialCom*, sep. 2012.

[SSKÇ13] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Shattering and compressing networks for betweenness centrality. In *SIAM International Conference on Data Mining (SDM)*, May 2013. An extended version is available as a Tech Rep on ¡a href="http://arxiv.org/abs/1209.6007"¿ArXiv¡/a¿.

[SZ11] Z. Shi and B. Zhang. Fast network centrality analysis using GPUs. *BMC Bioinformatics*, 12:149, 2011.