Utrecht University

Using preprocessing to speed up Brandes' betweenness centrality algorithm

Steven Fleuren

January 2018

BACHELOR THESIS

Supervisor: Prof. Dr. R.H. Bisseling

1. Introduction

In some applications of graph theory it is useful to find out how important each vertex is. For instance, in a road network it can be interesting to know how busy a crossroad is, or in a social network it can be important to know which persons are the most influential. A function that measures the importance of a vertex is called a *centrality measure*. What it means for a vertex to be important depends on the type of network that is being examined, and as such multiple types of centrality measures have been developed. In this paper we will only consider *betweenness centrality*. This centrality measure is based on shortest paths: a vertex has a high centrality score if many shortest paths pass through it. Formally, let $G = (V, E)$ be an unweighted and undirected graph and let $s, t \in V$. Denote the number of shortest paths from s to t by σ_{st} and, for $v \in V$, denote the number of shortest paths from s to t that v lies on by $\sigma_{st}(v)$. Let $\delta_{st}(v) := \sigma_{st}(v)/\sigma_{st}$. Betweenness centrality is introduced in 1977 by Freeman [3] and defined as

$$C_B(v) = \sum_{s \in V \setminus \{v\}} \sum_{t \in V \setminus \{v\}} \delta_{st}(v). \tag{1}$$

When working with multiple graphs we will denote the betweenness centrality of v in G by $C_B^G(v)$. In 2001 Ulrik Brandes [1] published an algorithm that computes the betweenness centrality of every vertex of G in $\mathcal{O}(mn)$ time, where $m = |E|$ and $n = |V|$. In 2012 an article by various authors [5] was published that described two improvements for Brandes' original algorithm. The first improvement was based on the idea that structurally equivalent vertices have the same betweenness centrality value. The second improvement, the subject of this paper, is a divide and conquer technique based on dividing the graph into biconnected components.

The structure of this paper is as follows: In section 2 we will figure out the workings of Brandes' algorithm, using [1] as our guideline. In section 3 we introduce the concept of biconnected components, their relation with betweenness centrality and we show how to construct them. In section 4 we prove the main results of the relevant part of [5]. The proofs in [5] are very compendious, the proofs in our paper will be more comprehensive. In section 5 we will describe the results of our experiments with an implementation of the improved Brandes' algorithm that we wrote in Python, and in appendix A we will share the code of this implementation.

Throughout this paper we will use the definitions of graph theoretical concepts as described in Ray [6], unless specified otherwise.

2. Brandes' Algorithm

In this section we give a description of Brandes' algorithm (Brandes [1]) and the ideas behind it. Let $G = (V, E)$ be an undirected and connected (i.e. for each $u, v \in V$ there exists a path between u and v) graph, and let $n = |V|$ and $m = |E|$. The algorithm can calculate the betweenness centrality of all vertices in V in $\mathcal{O}(mn)$ time if G is unweighted or in $\mathcal{O}(n^2 \log(n) + nm)$ time if G is weighted. We will assume G is undirected, connected and unweighted for the remainder of this paper.

First we make the following observation: Given three vertices $v, s, t \in V$, let p_{sv} be a shortest path between s and v and let p_{vt} be a shortest path between v and t. Let p_{st} be the combination of the other two paths. If p_{st} is not a shortest path between s and t, all shortest paths between s and t do not pass through v. Ergo $\sigma_{st}(v) = 0$. Otherwise p_{st} is a shortest path between s and t, every combination of a shortest path from s to v and a shortest path from v to t is, and $\sigma_{st}(v) = \sigma_{sv}\sigma_{vt}$. So $\sigma_{st}(v)$ can be expressed as

$$\sigma_{st}(v) = \begin{cases} \sigma_{sv}\sigma_{vt} & \text{if } v \text{ lies on a shortest path between } s \text{ and } t, \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Define the *dependency* of a vertex $s \in V$ on a vertex $v \in V$ as[1]

$$\delta_{s\bullet}(v) := \sum_{t \in V \setminus \{v\}} \delta_{st}(v) \tag{3}$$

We will find a recursive relation that will allow us to find $\delta_{s\bullet}(v)$ for all $v \in V$ in $\mathcal{O}(m)$ time. Once we can do that, we can calculate the betweenness centrality of v as

$$C_B(v) = \sum_{s \in V \setminus \{v\}} \delta_{s\bullet}(v).$$

To determine $\delta_{s\bullet}(v)$ for all $v \in V$ we need to find the shortest paths from s to all those vertices. We can use breadth-first search for this, which costs $\mathcal{O}(m)$ time. In the process we find for each shortest path p from s to a $v \in V \setminus \{s\}$ a vertex u such that the edge $\{u, v\}$ lies on p. The vertex u is called a *predecessor* of v on p. Denote the set of predecessors of v on shortest paths from s to v by $P_s(v)$. Define the set of successors of u from s as

$$S_s(u) := \{v \in V \mid u \in P_s(v)\}. \tag{4}$$

We can now prove the following theorem:

[1] Note that this definition deviates from the one given in [1], which is given as

$$\delta_{s\bullet}(v) := \sum_{t \in V} \delta_{st}(v).$$

Theorem 2.1 (Theorem 6 in [1]). *The dependency of $s \in V$ on any $v \in V$ obeys*

$$\delta_{s\bullet}(v) = \sum_{w \in S_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s\bullet}(w)). \tag{5}$$

Proof. Let $s, v \in V$, and let $t \in V \backslash \{v\}$. Note that $\sigma_{st}(v) > 0$ if and only if v lies on a shortest path from s to t. There are two cases. If $\sigma_{st}(v) > 0$, then v lies on a shortest path between s and t, which we denote by p. Since $v \neq t$, the vertex v is not the terminus of p and this path contains exactly one edge $\{v, w\}$ such that v is visited before w. The subpath of p with origin s and terminus w is a shortest path between s and w. Therefore, $w \in S_s(v)$. Denote the number of shortest paths from s to t that contain an edge e by $\sigma_{st}(e)$. We see that

$$\sigma_{st}(v) = \sum_{w \in S_s(v)} \sigma_{st}(\{v, w\}) \tag{6}$$

Now consider the case where $\sigma_{st}(v) = 0$. If $w \in V$ we have $\sigma_{st}(\{v, w\}) \leq \sigma_{st}(v) = 0$, so $\sigma_{st}(\{v, w\}) = 0$. In particular, the equation (6) holds if $\sigma_{st}(v) = 0$ as well.

By arguments similar to the ones we used to determine equation (2), we find that if $w \in S_s(v)$ then $\sigma_{st}(\{v, w\}) = 0 = \sigma_{st}(w)$ provided w does not lie on a shortest path between s and t, and $\sigma_{st}(\{v, w\}) = \sigma_{sv}\sigma_{wt} = \sigma_{st}(w)\sigma_{sv}/\sigma_{sw}$ otherwise. In both cases we have

$$\sigma_{st}(\{v, w\}) = \frac{\sigma_{sv}}{\sigma_{sw}} \sigma_{st}(w). \tag{7}$$

Finally, note that

$$\sum_{w \in S_s(t)} \delta_{st}(\{t, w\}) = 0. \tag{8}$$

Defining $\delta_{st}(e) := \sigma_{st}(e)/\sigma_{st}$ and using the results we found so far we complete the proof:

$$\begin{aligned}
\delta_{s\bullet}(v) &= \sum_{t \in V \backslash \{v\}} \delta_{st}(v), \\
&= \sum_{t \in V \backslash \{v\}} \sum_{w \in S_s(v)} \delta_{st}(\{v, w\}) \text{ using equation (6)}, \\
&= \sum_{t \in V} \sum_{w \in S_s(v)} \delta_{st}(\{v, w\}) \text{ using equation (8)}, \\
&= \sum_{w \in S_s(v)} \sum_{t \in V} \delta_{st}(\{v, w\}), \\
&= \sum_{w \in S_s(v)} \sum_{t \in V} \frac{\sigma_{sv}}{\sigma_{sw}} \delta_{st}(w) \text{ using equation (7)}, \\
&= \sum_{w \in S_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} \left(\delta_{sw}(w) + \sum_{t \in V \backslash \{w\}} \delta_{st}(w) \right), \\
&= \sum_{w \in S_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s\bullet}(w)).
\end{aligned}$$

□

Brandes' algorithm uses this result as follows: Set $C_B(v) \leftarrow 0$ for all $v \in V$. For each $s \in V$, apply breadth-first search (bfs) on G with s as starting point to find $P_s(v)$ and σ_{sv} for all $v \in V$. Each time a vertex is explored with bfs it is put on a stack. Note that if $v \in P_s(w)$, then v is placed on the stack before w is. After the bfs is completed, set $\delta_{s\bullet}(v) \leftarrow 0$ for all $v \in V$. Then pop a vertex w from the stack, add $\sigma_{sv}/\sigma_{sw}(1 + \delta_{s\bullet}(v))$ to all $\delta_{s\bullet}(v)$ with $v \in P_s(w)$ and add $\delta_{s\bullet}(w)$ to $C_B(w)$. Repeat this until the stack is empty. See Algorithm 1 for the pseudocode of the algorithm as provided in [1]:

Algorithm 1: Betweenness centrality in unweighted graphs [1]

```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V; σ[s] ← 1;
    d[t] ← −1, t ∈ V; d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + (σ[v]/σ[w]) · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
```

3. Articulation Points and Biconnected Components

In this section we introduce the definitions of articulation points and of biconnected components, show a connection between these two concepts and betweenness centrality, and we explain how we can determine the articulation points and biconnected components of a given graph. Let $G = (V, E)$ be an undirected and connected graph. A vertex $v \in V$ is called an *articulation point* if removing v from V disconnects G. A graph that contains an articulation point is called *separable*. Let $V_F \subset V$ and let F be the subgraph of G induced by V_F. If F is connected and inseparable but all connected subgraphs H of G with $F \subsetneq H$ are separable, F is called a *biconnected component* of G. We will refer to biconnected components as components from now on. Note that given two different components B and F of G the intersection $B \cap F$ is either an empty graph or it contains exactly one vertex v: otherwise the union $B \cup F$ is inseparable which contradicts the assumption that B and F are components. Furthermore, the vertex v must be an articulation point in G. We can use this knowledge to calculate the betweenness centrality of v in G. After all, the vertex v lies on any path that originates in B and terminates in F, including all the shortest paths. To expand on this idea, we define F^v to be the *hull* around component F of vertex v, which we define as the graph that satisfies:

(1) $F \subset F^v \subset G$,
(2) $v \in V_{F^v}$,
(3) $F^v - v$ is connected,
(4) F^v is maximal with regard to conditions (1)-(3).

We introduce $c(F, v)$ as $c(F, v) := |V_{F^v} \backslash \{v\}|$ and $d(F, v)$ as $d(F, v) = |V \backslash V_{F^v}|$. We have

$$c(F, v) + d(F, v) + 1 = |G| \tag{9}$$

for all components F of G and $v \in V$. We have the following lemma:

Lemma 3.1 ([5]). *Let $G = (V, E)$ be an undirected, connected and separable graph, and let $v \in V$ be an articulation point of G. Let $F_1, \ldots, F_k$ be the components of G that contain v. Then*

$$C_B^G(v) = \left(\sum_{i=1}^{k} C_B^{F_i^v}(v) + c(F_i, v) \cdot d(F_i, v) \right). \tag{10}$$

Proof. Note that $V_{F_i^v} \cap V_{F_j^v} = \{v\}$ for all $1 \leq i < j \leq k$ and

$$\bigcup_{i=1}^{k} V_{F_i^v} = V. \tag{11}$$

Let $s \in V_{F_i^v} \backslash \{v\}$. We see that

$$\begin{aligned}\sum_{t \in V \backslash \{v\}} \delta_{st}(v) &= \sum_{t \in V_{F_i^v} \backslash \{v\}} \delta_{st}(v) + \sum_{t \in V \backslash V_{F_i^v}} \delta_{st}(v), \\ &= \sum_{t \in V_{F_i^v} \backslash \{v\}} \delta_{st}(v) + \sum_{t \in V \backslash V_{F_i^v}} 1, \\ &= \sum_{t \in V_{F_i^v} \backslash \{v\}} \delta_{st}(v) + d(F_i, v).\end{aligned}$$

We use the definition of betweenness centrality to complete the proof:

$$\begin{aligned}C_B^G(v) &= \sum_{s \in V \backslash \{v\}} \sum_{t \in V \backslash \{v\}} \delta_{st}(v), \\ &= \sum_{i=1}^{k} \sum_{s \in V_{F_i^v} \backslash \{v\}} \sum_{t \in V \backslash \{v\}} \delta_{st}(v), \\ &= \sum_{i=1}^{k} \sum_{s \in V_{F_i^v} \backslash \{v\}} \left(\sum_{t \in V_{F_i^v} \backslash \{v\}} \delta_{st}(v) + d(F_i, v) \right), \\ &= \sum_{i=1}^{k} \left(\sum_{s \in V_{F_i^v} \backslash \{v\}} \sum_{t \in V_{F_i^v} \backslash \{v\}} \delta_{st}(v) + \sum_{s \in V_{F_i^v} \backslash \{v\}} d(F_i, v) \right), \\ &= \sum_{i=1}^{k} \left(C_B^{F_i^v} + d(F_i, v) \sum_{s \in V_{F_i^v} \backslash \{v\}} 1 \right), \\ &= \sum_{i=1}^{k} \left(C_B^{F_i^v}(v) + c(F_i, v) \cdot d(F_i, v) \right).\end{aligned}$$

□

For the remainder of this section, we will show how to determine the articulation points and components of a given graph. The method we use is based on problem 22-2 in [2]. Let $G_\pi = (V, E_\pi)$ be the depth-first search (dfs) tree of G with root u_π. We refer to edges in E_π as tree-edges. Edges in $E \backslash E_\pi$ are called backedges. We can prove the following about the structure of G_π:

Lemma 3.2. *Let $u, v \in V$ and let $e = \{u, v\} \in E$. Then u is either a descendant or an ancestor of v in G_π.*

Proof. Suppose u is expanded before v by the dfs process. Then the edge e is discovered, so v will be added to the tree before u is completed. Any vertex discovered after u is expanded but before u is completed is a descendant of u, so v is a descendant of u. By a similar argument u is a descendant of v if v is discovered first. □

We can generalise the result further:

Lemma 3.3. *Let $u, w \in V$. If there exists a path p from u to w such that u is discovered before the other vertices in p, then w is a descendant of u in G_π.*

Proof. Suppose there exists a path p from u to w such that u is discovered before the other vertices in p. Let d be the length of p. We proceed with induction over d. If $d = 1$, there is an edge between u and w, and by Lemma (3.2) the vertex u is a descendant of w. The induction hypothesis is that Lemma (3.3) holds if $1 \leq d \leq k$. Suppose $d = k + 1$. There exists a path p of length d between u and w. Removing w from this path leaves us with a path with length $d - 1$ between u and v, a neighbour of w. By the induction hypothesis, v is a descendant of u, and by Lemma (3.2), vertex v is either an descendant or an ascendant of w. So w is either an ascendant or a descendant of u. Since u was discovered first, w must be a descendant. □

For a vertex $v \in V$ we define G_π^v to be the subtree of G_π rooted at v. There is a simple test to check if u_π is an articulation point:

Lemma 3.4 (Problem 22-2 a in [2])**.** *The root u_π is an articulation point if and only if it has two or more children in the dfs tree.*

Proof. Suppose u_π has only one child, denoted by v. The subtree G_π^v is connected and contains all vertices in V except u_π. This implies that $G - u_\pi$ is connected, so u_π is not an articulation point.

Now suppose u_π has $k > 1$ children $v_1, \ldots, v_k$, where v_1 was discovered first during the dfs process. Assume u_π is not a articulation point. Then there exists a path p in G from v_1 to v_2 that does not pass through u_π. Since u_π is the only vertex discovered before v_1, all the vertices in p were discovered after v_1 was. It follows from Lemma 3.3 that v_2 is a descendant of v_1. This contradicts v_2 being a child of u_π, so u_π is an articulation point. □

We can use G_π to find the other articulation points of G as well. We introduce $g : V \to \mathbb{N}_{\geq 0}$ as the time stamp of discovery, i.e. $g(u_\pi) = 0$ and $g(v) = g(u) + 1$ if v is expanded immediately after u during the dfs process. Denote the set of neighbours of v in G by N_v and the set of children of v in G_π by C_v. We define h recursively as

$$h(v) = \min_{w \in N_v \cup \{v\}:\ v \notin C_w} \begin{cases} h(w): & w \in C_v, \\ g(w): & w \notin C_v. \end{cases} \tag{12}$$

We will now prove the following theorem:

Theorem 3.5 (based on Problem 22-2 b&c in [2])**.** *Let v be a nonroot vertex of G_π. The following statements are equivalent:*

(1) The vertex v is an articulation point.

(2) The vertex v has a child w such that there is no backedge that connects G_π^w and $G_\pi \backslash G_\pi^v$.

(3) The vertex v has a child w such that $g(v) \leq h(w)$

Proof. We start with proving the implication (1) $\implies$ (2) by contraposition. Suppose that for each child w of v there exists a backedge that connects G_π^w with $G_\pi \backslash G_\pi^v$. We show that for all $u \in V \backslash \{v\}$ there exists a path to u_π that does not pass through v. If $u \in G_\pi \backslash G_\pi^v$, then u_π is an ancestor of u but v is not. So we can construct a path by climbing the tree from u to u_π. If $u \in G_\pi^w$ we can construct a walk by subsequently ascending to w, descending to the vertex that is connected to $G_\pi \backslash G_\pi^v$ with a backedge, and ascending to u_π from there. We can turn the walk into a path by removing the vertices and edges that are visited twice. This path does not pass through v. Every vertex in $V \backslash \{v\}$ is connected to u_π, which implies that $G - v$ is connected and that v is not an articulation point.

We will now prove (2) $\implies$ (1). Suppose their does exist a child w of v such that there is no backedge connecting G_π^w and $G_\pi \backslash G_\pi^v$. There does not exist a backedge between G_π^w and G_π^s for some child $s \neq w$ of v: If it did it would be of the form $\{x, y\}$ for $x, y \in V$ and one of them would be a descendant of the other by Lemma 3.2, which contradicts them being contained in separate subtrees. So a path can only leave G_π^w through a tree-edge or a backedge that ends in v. So there does not exist a path between w and u_π in $G - v$, and therefore v is an articulation point. See Figure 1 for an illustration.

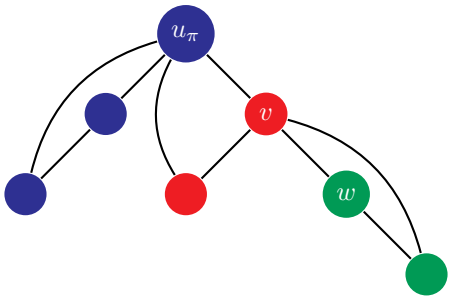

FIGURE 1. An example of a dfs tree. The straight edges are tree-edges and the bent edges are backedges. The red and green vertices are contained in G_π^v, the blue ones are contained in $G_\pi \backslash G_\pi^v$. The set $V_{G_\pi^w}$ consists of the green vertices. In this example v is an articulation point because there is no backedge connecting the green vertices with the blue ones.

We continue with the proof for (2) $\implies$ (3). Suppose statement (2) is true: there are no backedges from w or any of its descendants to $G_\pi \backslash G_\pi^v$. So

$$h(w) \geq \min\{g(u) \mid u \text{ is a descendant of } v\}.$$

If u is a descendant of v, it is expanded later than v, so $g(u) \geq g(v)$ and $h(w) \geq g(v)$.

Finally we will prove (3) $\implies$ (2) by contraposition. Suppose all children w have a descendant s with a backedge to $t \in G_\pi \backslash G_\pi^v$. By Lemma 3.2 the vertex t is an ancestor of s. Since t is not

a descendant of v it must be an ancestor of v, and

$$h(w) \leq h(s) \leq g(t) < g(v). \tag{13}$$

Since equation (13) holds for all children of v, the proof is complete. □

The method to find all the articulation points works as follows: initialise dfs in some vertex to construct G_π and to calculate $g(v)$ for all $v \in V$. Then calculate $h(v)$, starting from the vertex with the highest $g(v)$ value. Check if the root is an articulation point with Lemma 3.4, and check for each of the nonroot vertices v if it has a child w such that $g(v) \leq h(w)$.

We can find the components of G in a similar manner. We saw that a vector v is an articulation point if $g(v) \leq h(w)$ for some child w. So w does not share a component with the parent of v or with the other children of v. So if $g(v) \leq h(w)$ we can construct a new component that consists of the vertices v and w and the edge between them. If $g(v) > h(w)$ the child w shares a component with v and the parent of v. See Algorithm 2 for the pseudo code of an algorithm that finds the articulation points and components of a given graph.

Def `constructComponent(`v, w`)`
  $V_B \leftarrow [v, w]$;
  $E_B \leftarrow [\{v, w\}]$;
  $B \leftarrow (V_B, E_B)$;
  return B

Algorithm 2: Finding articulation points and biconnected components

```
dfs(G, u_π);
h[v] ← g[v], v ∈ V;
i ← n − 1;
while i ≥ 0 do
    v ← g^{-1}[i];
    foreach neighbour w of v do
        if w is a child of v in G_π then
            h[v] ← min(h[v], h[w]);
        else if w is not the parent of v in G_π then
            h[v] ← min(h[v], g[w]);
    i ← i − 1;
articulationPoints ← empty set;
components ← empty list;
componentOfEdge[e] ← −1, e ∈ E;
for 0 ≤ i ≤ n − 1 do
    v ← g^{-1}[i];
    if v = u_π then
        if v has multiple children then
            articulationPoints.append(v);
            foreach child w of v do
                components.append(constructComponent(v, w);
                componentOfEdge[{v, w}] ← len(components);
        else
            V_B ← [u_π];
            E_B ← empty list;
            B ← (V_B, E_B);
            components.append(B);
    else
        u ← parent of v;
        foreach neighbour w of v do
            if w is a child of v in G_π then
                if g(v) ≤ h(w) then
                    articulationPoints.add(v);
                    components.append(constructComponent(v, w);
                    componentOfEdge[{v, w}] ← len(components);
                else
                    B = components[componentOfEdge[{u, v}]];
                    V_B.append(w);
                    E_B.append({v, w});
            else if w is not the parent of v in G_π then
                B = components[componentOfEdge[{u, v}]];
                E_B.append({v, w});
```

4. Using Biconnected Components To Compute Betweenness Centrality

In the last section we learned how to find the articulation points and biconnected components of G, and we proved Lemma 3.1, which can be used to compute $C_B^G(v)$ if v is an articulation point. In this section we will expand on this idea. To do so, we introduce a more general form of betweenness centrality:[2]

$$C_\tau(v) := \sum_{s \in V \setminus \{v\}} \sum_{t \in V \setminus \{v\}} \delta_{st}(v) \cdot \tau(s) \cdot \tau(t), \tag{14}$$

where $\tau : V \to \mathbb{R}$ is a function. We will show that for all non-articulation points $v \in V$ it is possible to express $C_B^G(v)$ in terms of $C_{\tau_F}^F$ and $d(F, v)$, where F is the component that contains v. Define $\tau_F(v)$ as follows:

$$\tau_F(v) := \begin{cases} d(F, v) + 1 & \text{if } v \text{ is an articulation point,} \\ 1 & \text{otherwise.} \end{cases} \tag{15}$$

We will prove the following relation:

Theorem 4.1 ([5])**.** *Let $G = (V, E)$ be an undirected and connected graph and let $v \in V$ be a vertex that is not an articulation point. Let F be the biconnected component of G that contains v. Then*

$$C_B^G(v) = C_{\tau_F}^F(v). \tag{16}$$

Proof. If G is inseparable, it must be that $F = G$ and $\tau_F(w) = 1$ for all $w \in V$. So in this case $C_B^G(v) = C_{\tau_F}^F(v)$. Suppose G is separable. Define W_F as

$$W_F = \{w \in V_F \mid w \text{ is an articulation point in } F\}.$$

For $w \in W_F$, recall that F^w denotes the maximal subgraph of G that satisfies $F \subset F^w$ and $F^w - w$ is connected. For all $w \in W_F$, define $S_{F,w}$ as

$$S_{F,w} = \{u \in V \mid u \notin V_{F^w}\}.$$

Note that for all $u \in V$, either $u \in V_F$ or $u \in S_{F,w}$ for exactly one $w \in W_F$. Furthermore, the number of vertices in $S_{F,w}$ equals $|V| - |V_{F^w}| = |V| - c(F, w) - 1 = d(F, w)$. Let $s \in V \setminus \{v\}$. Suppose $t \in S_{F,w}$ for some $w \in W_F$. Recall that if a shortest path from s to t passes through v, then $\sigma_{st}(v) = \sigma_{sv}\sigma_{vt}$. Since $t \in S_{F,w}$, any path from v to t passes through w, so $\sigma_{vt} = \sigma_{vw}\sigma_{wt}$. If $s \in S_{F,w}$, the shortest path from s to t does not leave $S_{F,w}$ and $\delta_{st}(v) = 0$. Otherwise any path from s to t passes through w, and $\sigma_{st} = \sigma_{sw}\sigma_{wt}$. Assuming the existence of a shortest path

[2]Here we differ from the method used in [5]. The source combines the two τ factors into a single matrix entry. We found that calculating this $n \times n$ matrix creates unnecessary overhead. We generalise betweenness centrality differently to avoid having to calculate the rank-1 matrix.

from s to t that passes through v we have

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} = \frac{\sigma_{sv}\sigma_{vw}\sigma_{wt}}{\sigma_{sw}\sigma_{wt}} = \frac{\sigma_{sv}\sigma_{vw}}{\sigma_{sv}} = \delta_{sw}(v). \tag{17}$$

If such a path does not exist, there does not exist a shortest path from s to w that passes through v either, so $\delta_{st}(v) = 0 = \delta_{sw}(v)$. We apply what we found so far to rewrite the inner sum of $C_B^G(v)$:

$$\begin{aligned}
\sum_{t\in V\setminus\{v\}} \delta_{st}(v) &= \sum_{w\in W_F} \left(\delta_{sw}(v) + \sum_{t\in S_{F,w}} \delta_{st}(v) \right) + \sum_{t\in (V_F\setminus W_F)\setminus\{v\}} \delta_{st}(v), \\
&= \sum_{w\in W_F} \left(\delta_{sw}(v) + \sum_{t\in S_{F,w}} \delta_{sw}(v) \right) + \sum_{t\in (V_F\setminus W_F)\setminus\{v\}} \delta_{st}(v), \\
&= \sum_{w\in W_F} \left(\delta_{sw}(v) + \delta_{sw}(v) \sum_{t\in S_{F,w}} 1 \right) + \sum_{t\in (V_F\setminus W_F)\setminus\{v\}} \delta_{st}(v), \\
&= \sum_{w\in W_F} (d(F,w)+1)\delta_{sw}(v) + \sum_{t\in (V_F\setminus W_F)\setminus\{v\}} \delta_{st}(v), \\
&= \sum_{t\in V_F\setminus\{v\}} \tau_F(t)\cdot\delta_{st}(v).
\end{aligned}$$

We complete the proof by applying this equality twice:

$$\begin{aligned}
C_B^G(v) &= \sum_{s\in V\setminus\{v\}} \sum_{t\in V\setminus\{v\}} \delta_{st}(v), \\
&= \sum_{s\in V\setminus\{v\}} \sum_{t\in V_F\setminus\{v\}} \tau_F(t)\cdot\delta_{st}(v), \\
&= \sum_{t\in V_F\setminus\{v\}} \tau_F(t) \sum_{s\in V\setminus\{v\}} \delta_{st}(v), \\
&= \sum_{t\in V_F\setminus\{v\}} \tau_F(t) \sum_{s\in V\setminus\{v\}} \delta_{ts}(v), \\
&= \sum_{t\in V_F\setminus\{v\}} \tau_F(t) \sum_{s\in V_F\setminus\{v\}} \tau_F(s)\cdot\delta_{ts}(v), \\
&= \sum_{s\in V\setminus\{v\}} \sum_{t\in V\setminus\{v\}} \delta_{st}(v)\cdot\tau_F(s)\cdot\tau_F(t), \\
&= C_{\tau_F}^G(v).
\end{aligned}$$

□

Combining Lemma 3.1 and Theorem 4.1 gives the following result:

Theorem 4.2 ([5]). *Let $G = (V, E)$ be an undirected, connected and separable graph, and let $v \in V$ be an articulation point of G. Let $F_1, \ldots, F_k$ be the components of G that contain v. Then*

$$C_B^G(v) = \sum_{i=1}^{k} C_{\tau_{F_i}}^{F_i}(v) + c(F_i, v) \cdot d(F_i, v). \tag{18}$$

Proof. Lemma 3.1 states that

$$C_B^G(v) = \sum_{i=1}^{k} C_B^{F_i^v}(v) + c(F_i, v) \cdot d(F_i, v). \tag{19}$$

Let $1 \leq i \leq k$. By construction, the graphs F_i^v and $F_i^v - v$ are connected, so v is not an articulation point in F_i^v and we can apply Theorem 4.1: $C_B^{F_i^v}(v) = C_{\tau_{F_i}}^{F_i}(v)$. Combining with (19) gives the desired result. □

Let us recapitulate. Our goal is to efficiently calculate the betweenness centrality of all $v \in V$ using the theorems we have just proven. To do so, we need to determine which v are articulation points and which component(s) the vertices are in. We can use the method in the last section to find this information. What we are still lacking is a method to calculate $c(F, v)$ and $d(F, v)$ for a component F and vertex $v \in V_F$ if v is an articulation point in G. We also need a method to calculate $C_\tau(v)$ for given τ.

We proceed with the former problem. Let v be an articulation point of G. Recall that $c(F, v) + d(F, v) + 1 = n$ for all components F that contain v, so once we know $c(F, v)$ we can easily calculate $d(F, v)$. We defined $c(F, v) = |F^v \backslash \{v\}|$. Since every $u \in V \backslash \{v\}$ is contained in exactly one graph F^v, we have

$$1 + \sum_{F : v \in W_F} c(F, v) = n. \tag{20}$$

So if we know $c(F, v)$ for all components F that contain v except for F', we can calculate

$$c(F', v) = n - 1 - \sum_{F : v \in W_F, F \neq F'} c(F, v). \tag{21}$$

There is another relation we can use: let F be a component of G. Every $v \in V$ is contained in either V_F or in $S_{F,w}$ for exactly one $w \in W_F$. The number of vertices in $S_{F,w}$ can be expressed as

$$|S_{F,w}| = n - |V_{F^w}| = n - c(F, w) - 1. \tag{22}$$

So

$$n = |V_F| + \sum_{w \in W_F} \left(n - c(F, w) - 1\right). \tag{23}$$

If we know $c(F, v)$ for all $w \in W_F$ except for w', we can rewrite the last equation as

$$c(F, w') = |V_F| - 1 + \sum_{w \in W_F, w \neq w'} \left(n - c(F, w) - 1\right). \tag{24}$$

We can calculate $c(F, v)$ for all combinations F and $v \in W_F$ using the equations (21) and (24). To see this, consider the undirected graph $T = (V_T, E_T)$ where T_V consists of the components and the articulation points of G, and $v, w \in V_T$ are connected if and only if one of them is a component, the other is an articulation point, and the articulation point is contained in the component. First note that T must be a tree: If there was a cycle there would be two components with two connections to each other. Secondly, any articulation point has at least two neighbours in T because it is contained by at least two components. Ergo, all the leaves are components F with $W_F = \{w\}$ for some articulation point w of G, and $c(F, w) = |V_F| - 1$. In general, we can calculate $c(F, w)$ if we earlier calculated it for either all the neighbours of w in T or all the neighbours of F in T. See Algorithm 3 for pseudo code of the process.

Algorithm 3: Compute $c(F, w)$ (Algorithm 1 in [5])

```
Find the articulation points and components of G;
articulationPoints ← list of all the articulation points of G;
components ← list of biconnected components of G;
W[B] ← list of articulation points of G that B contains, B ∈ components;
c[B, w] ← −1, B ∈ components, w ∈ W[B];
Q ← empty queue;
foreach B ∈ components do
    if W[B] contains exactly one vertex w then
        enqueue (B, w) → Q;
while Q not empty do
    dequeue pair ← Q;
    if pair is of the form (B, w) then
        size ← |V_B| − 1;
        foreach v ∈ W[B] do
            if v ≠ w then
                size ← size + n − c[B, v] − 1;
        c[B, w] ← size;
        if There is exactly one F with c[F, w] = −1 then
            enqueue (w, F) → Q;
    else
        size ← 1;
        foreach F ∈ {F ∈ components | w ∈ W[F]} do
            if F ≠ B then
                size ← size +c[B, v];
        c[B, w] ← n− size;
        if There is exactly one v with c[B, v] = −1 then
            enqueue (B, v) → Q;
```

We will now show how to compute $C_\tau(v)$ for $v \in V$ with τ given. The process is very similar to Brandes' algorithm, only a few adjustments are necessary to account for τ. Define the

$\tau-$dependency of vertex $s \in V$ as

$$\delta^{\tau}_{s\bullet} := \sum_{t \in V \setminus \{v\}} \tau(t) \cdot \delta_{st}(v). \tag{25}$$

We can generalise Theorem 2.1 as follows:

Theorem 4.3. *The τ-dependency of $s \in V$ on any $v \in V$ obeys*

$$\delta^{\tau}_{s\bullet}(v) = \sum_{w \in S_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} (\tau(w) + \delta^{\tau}_{s\bullet}(w)). \tag{26}$$

Proof. Recall from the proof of Theorem 2.1 that

$$\sigma_{st}(v) = \sum_{w \in S_s(v)} \sigma_{st}(\{v, w\}) \tag{27}$$

and

$$\sigma_{st}(\{v, w\}) = \frac{\sigma_{sv}}{\sigma_{sw}} \sigma_{st}(w). \tag{28}$$

Note that

$$\sum_{w \in S_s(t)} \tau(t) \cdot \delta_{st}(\{t, w\}) = 0. \tag{29}$$

We complete the proof in a similar manner as we completed the proof for Theorem 2.1:

$$\begin{aligned}
\delta_{s\bullet}(v) &= \sum_{t \in V \setminus \{v\}} \tau(t) \cdot \delta_{st}(v), \\
&= \sum_{t \in V \setminus \{v\}} \tau(t) \sum_{w \in S_s(v)} \delta_{st}(\{v, w\}) \text{ using equation (27)}, \\
&= \sum_{t \in V} \sum_{w \in S_s(v)} \tau(t) \cdot \delta_{st}(\{v, w\}) \text{ using equation (29)}, \\
&= \sum_{w \in S_s(v)} \sum_{t \in V} \tau(t) \cdot \delta_{st}(\{v, w\}), \\
&= \sum_{w \in S_s(v)} \sum_{t \in V} \frac{\sigma_{sv}}{\sigma_{sw}} \tau(t) \cdot \delta_{st}(w) \text{ using equation (28)}, \\
&= \sum_{w \in S_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} \left(\tau(w) \delta_{sw}(w) + \sum_{t \in V \setminus \{w\}} \tau(t) \cdot \delta_{st}(v) \right), \\
&= \sum_{w \in S_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} (\tau(w) + \delta^{\tau}_{s\bullet}(w)).
\end{aligned}$$

□

The pseudo code for a more general form of Brandes' algorithm can be found in Algorithm 4. We now have assembled all the pieces we need to compute the betweenness centrality for all $v \in V$ using Theorem 4.1, Theorem 4.2 and Theorem 4.3. See Algorithm 5 for the complete algorithm.

Algorithm 4: Generalised Brandes' algorithm (derived from the Algorithm 1 in [1])

// The first part is exactly the same as Algorithm 1, and is omitted here to avoid repetition.
// The difference lies in the last while loop:
while *S not empty* **do**
  pop $w \leftarrow S$;
  for $v \in P[w]$ **do** $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (\tau[w] + \delta[w])$;
  if $w \neq s$ **then** $C_B[w] \leftarrow C_B[w] + \tau[s] \cdot \delta[w]$;

Algorithm 5: Betweenness computation using biconnected components (derived from Algorithm 2 in [5])

Find the articulation points and components of G;
articulationPoints $\leftarrow$ list of all the articulation points of G;
components $\leftarrow$ list of biconnected components of G;
$W[B] \leftarrow$ list of articulation points of G that B contains, $B \in$ components;
compute $c[B, w], B \in$ components, $w \in W[B]$;
$C_B[v] \leftarrow 0, v \in V$;
foreach $B \in$ *components* **do**
  foreach $v \in V_B$ **do**
    if $v \in W[B]$ **then**
      $\tau[v] \leftarrow n - c[B, v]$;
      $C_B[v] \leftarrow C_B + c[B, v] \cdot (n - c[B, v] - 1)$;
    else
      $\tau[v] \leftarrow 1$;
    compute $C_\tau[v], v \in V_B$;
    foreach $v \in V_B$ **do**
      $C_B[v] \leftarrow C_B + C_\tau[v]$

5. Complexity and Test Results

In this section we briefly discuss the time complexity of Algorithm 5, and we will share the results of some experiments we conducted. We implemented the standard Brandes' algorithm and Algorithm 5 in Python; the code can be found in Appendix A.

Calculating the dfs-tree needed for Algorithm 2 costs $\mathcal{O}(m)$ time. Algorithm 2 contains two other loops. Both use each edge of G twice, so Algorithm 2 takes $\mathcal{O}(m)$ time to complete. The complexity of the first loop of Algorithm 3 depends on the number of components in G, which is bounded by n. The second loop is a bit harder to analyse. Denote the number of articulation points of G by n_W. Given two articulation points there can be at most one component that contains both. Since the number of components is bounded by n, the number of pairs that are processed in the while loop is bounded by $2n$. The size of W_B is bounded by n_W, so the time complexity of Algorithm 3 is $\mathcal{O}(n \cdot n_W)$. The time complexity of Algorithm 4 is the same as the time complexity of standard Brandes, so $\mathcal{O}(mn)$. In our main algorithm the input for Algorithm 4 will be the connected components of G. The total time complexity of Algorithm 5 is

$$\mathcal{O}(m + n \cdot n_w + \max_{\text{component } B \subset G}(|V_B| \cdot |E_B|)). \tag{30}$$

In Table 1 one can find the graphs we used for the experiments. We denote the component of G with the largest number of vertices by $B_{\max}$, and the number of its vertices and edges by $n_{B_{\max}}$ and $m_{B_{\max}}$ respectively. The graphs are ordered by the number of vertices they contain. All the graphs are connected and undirected. The first two graphs were downloaded from NeuroData's Graph Database[3], the others originate from the Stanford Large Network Dataset Collection [4]. Note that all the graphs with articulation points contain one large component that contains the majority of the edges of G.

[3] http://openconnecto.me/graph-services/download/

name	n	m	n_W	$n_{B_{\max}}$	$m_{B_{\max}}$
mouse_brain_1	213	16242	0	213	16242
rattus.norvegicus_brain_3	493	25988	0	493	25988
facebook/107	1034	26749	20	1007	26717
facebook_combined	4039	88234	11	3698	85963
as-caida20070917	8020	18203	629	5234	15412
p2p-Gnutella06	8717	31525	1343	6727	29535
p2p-Gnutella04	10876	39994	1757	8379	37497
Oregon2_010526	11461	32730	828	8085	29342
as-caida20040105	16301	32955	1377	10424	27061

TABLE 1. Graph attributes

The results of the experiments can be found in Table 1. The run time values are measured in seconds, and the ratio values are equal to the run time of Brandes' algorithm divided by the run time for Algorithm 5. Note that the graphs without articulation points have a longer run time for Algorithm 5, but only slightly so. On very sparse graphs like the two as-caida graphs the speed up achieved by the algorithm is significant.

name	run time Brandes	run time Algorithm 5	speedup ratio
mouse_brain_1	2.20	2.39	0.92
rattus.norvegicus_brain_3	10.09	10.58	0.95
facebook/107	17.80	17.41	1.02
facebook_combined	199.19	183.94	1.08
as-caida20070917	223.20	109.22	2.04
p2p-Gnutella06	288.47	196.56	1.46
p2p-Gnutella04	456.36	342.70	1.33
Oregon2_010526	428.42	250.34	1.71
as-caida20040105	891.41	409.02	2.18

TABLE 2. Experiment results

6. Conclusion

It is possible to speed up Brandes' betweenness centrality algorithm for graphs with multiple biconnected components. In this paper we described how to find all the biconnected components of a given graph in $\mathcal{O}(m)$ time, and showed that it is possible to apply an modified version of Brandes' algorithm on all the biconnected components to find the betweenness centrality values of every vertex in the whole graph. We implemented this method in Python, and experiments showed that it can calculate betweenness centrality twice as fast as Brandes' original algorithm for some graphs.

References

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

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

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

[4] Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

[5] Puzis, R., Zilberman, P., Elovici, Y., Dolev, S., and Brandes, U. (2012). Heuristics for speeding up betweenness centrality computation. In *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Conference on Social Computing (PASSAT-SocialCom 2012)*, pages 302–311, Piscataway, NJ. IEEE.

[6] Ray, S. S. (2014). *Graph Theory with Algorithms and Its Applications: In Applied Science and Technology*. Springer Publishing Company, Incorporated.

Appendix A. Python Code

The methods below only work for connected and undirected graphs. The dfs-method will cause stack overflow errors for large graphs, it is probably better to rewrite this method to a non-recursive form.

```
from collections import deque

class Graphs:
    def __init__(self, edges):
        n = len(edges)
        self.n = n
        self.edges = edges
        # Calculated in Brandes/BCBCC:
        self.bc = [0] * n
        # Calculated in dfs:
        self.time = 0  # counter for dfs
        self.g = [-1] * n  # g[vertex] = time discovered
        self.parent = [-1] * n  # parent in dfs tree
        self.dfs_completed = False
        # Calculated in findArticulationPoints:
        self.g_inverse = [-1] * n  # g_inverse[time discovered] = vertex
        self.h = [-1] * n  # Used for finding articulation points
        self.articulation_points = []
        self.find_articulationpoints_completed = False
        # Calculated in constructBlocks:
        self.component_of_u = [0] * n  # Keeps track of which biconnected component
        # a vertex belongs to. Articulation points belong too multiple components
        # and use lists. For example, if edges == [[1], [0, 2], [1]],
        # then Graphs(edges).component_of_u == [0, [0, 1], 1]
        self.component_count = 0
        self.component_contains = []
        self.component_AP_count = []
        self.index_in_component = [0] * n  # Can't be used for articulation points
        self.index_AP_in_component = {}
        self.construct_blocks_completed = False
        # Calculated in constructWeightedBlockTree
        self.c = {}
        self.construct_weighted_block_tree_completed = False

    # Main functions

    def brandes(self):
```

```
    n = self.n
    V = [x for x in range(n)]
    for s in V:
        S = []
        P = [[] for i in range(n)]
        sigma = [0] * n
        sigma[s] = 1
        d = [-1] * n
        d[s] = 0
        Q = deque([s])
        while len(Q) > 0:
            v = Q.popleft()
            S.append(v)
            for w in self.edges[v]:
                if d[w] < 0:
                    Q.append(w)
                    d[w] = d[v] + 1
                if d[w] == d[v] + 1:
                    # v is a predecessor of w on a shortest path from s to w
                    sigma[w] += sigma[v]
                    P[w].append(v)
        delta = [0] * n
        while len(S) > 0:
            w = S.pop()
            for v in P[w]:
                delta[v] += (sigma[v] / sigma[w]) * (1 + delta[w])
            if w != s:
                self.bc[w] += delta[w]
    return

def BCBCC(self):
    self.constructWeightedBlockTree()
    for u in self.articulation_points:
        for B in self.component_of_u[u]:
            self.bc[u] += self.c[(B, u)] * (self.n - self.c[(B, u)] - 1)
    for B in range(self.component_count):
        tau = self.compute_tau(B)
        self.brandesForBCBCC(B, tau)
    return

# Functions needed to perform BCBCC
```

```
def dfs(self, u, resetTime=False):  # see Introduction to Algorithms page 621
    if self.dfs_completed:
        return
    if resetTime:
        self.time = 0
    self.g[u] = self.time
    for v in self.edges[u]:
        if self.g[v] == -1:
            self.time += 1
            self.dfs(v)
            self.parent[v] = u
    if resetTime:
        self.dfs_completed = True
    return

def findArticulationPoints(self, start=0):
    if self.find_articulationpoints_completed:
        return
    n = self.n
    counter = n
    self.dfs(start, True)
    self.h = list(self.g)
    for u in range(n):
        self.g_inverse[self.g[u]] = u
    while counter > 0:
        counter -= 1
        u = self.g_inverse[counter]
        for v in self.edges[u]:
            if v == self.parent[u]:  # v is parent in dfs tree
                pass
            elif self.g[v] < self.g[u]:  # backedge
                if self.g[v] < self.h[u]:
                    self.h[u] = self.g[v]
            else:  # child
                if self.h[v] < self.h[u]:
                    self.h[u] = self.h[v]
    u = start
    children = 0
    for v in self.edges[u]:
        if u == self.parent[v]:
            children += 1
    if children > 1:
```

```
            self.articulation_points.append(u)
            self.component_of_u[u] = []
        for i in range(1, n):
            u = self.g_inverse[i]
            for v in self.edges[u]:
                if u == self.parent[v]:  # v is a child of u
                    if self.h[v] >= i:  # This means u is an articulation point
                        if not self.isArticulationPoint(u):
                            # This means it's the first time it's determined
                            # that u is an articulation point
                            self.articulation_points.append(u)
                            self.component_of_u[u] = []
        self.find_articulationpoints_completed = True
        return

    def constructBlocks(self, start=0):
        if self.construct_blocks_completed:
            return
        self.findArticulationPoints(start)
        n = self.n
        if not self.isArticulationPoint(start):
            self.addNewComponent(start)
        for i in range(1, n):
            u = self.g_inverse[i]
            parent = self.parent[u]
            j = self.g[parent]
            if self.isArticulationPoint(parent):
                if self.h[u] >= j or j == 0:
                    self.component_of_u[parent].append(self.component_count)
                    self.addNewComponent(u)
                else:
                    self.assignComponent(u, self.component_of_u[parent][0])
            else:
                self.assignComponent(u, self.component_of_u[parent])
        for u in range(n):
            if isinstance(self.component_of_u[u], int):
                self.component_contains[self.component_of_u[u]].append(u)
            else:
                for i in self.component_of_u[u]:
                    self.component_contains[i].append(u)
                    self.component_AP_count[i] += 1
        for B in range(self.component_count):
```

```
            for i in range(len(self.component_contains[B])):
                u = self.component_contains[B][i]
                if self.isArticulationPoint(u):
                    self.index_AP_in_component[(B, u)] = i
                else:
                    self.index_in_component[u] = i
        self.construct_blocks_completed = True
        return

    def constructWeightedBlockTree(self):
        # See Algorithm 1 in Heuristics for Speeding up Betweenness
        # Centrality Computation
        if self.construct_weighted_block_tree_completed:
            return
        self.constructBlocks()
        treeEdgesBlockToAP = [[] for x in range(self.component_count)]
        unknownNeighboursOfAP = {}
        unknownNeighboursOfBlock = [0] * self.component_count
        for u in self.articulation_points:
            unknownNeighboursOfAP[u] = len(self.component_of_u[u])
            for B in self.component_of_u[u]:
                treeEdgesBlockToAP[B].append(u)
                self.c[(B, u)] = -1
        Q = deque()
        for B in range(self.component_count):
            unknownNeighboursOfBlock[B] = self.component_AP_count[B]
            if unknownNeighboursOfBlock[B] == 1:
                Q.append((B, treeEdgesBlockToAP[B][0], True))
        while len(Q) > 0:
            triple = Q.popleft()
            if triple[2]:  # Pair is of the form (B, v)
                B = triple[0]
                u = triple[1]
                if self.c[(B, u)] == -1:
                    size = len(self.component_contains[B]) - 1
                    for v in treeEdgesBlockToAP[B]:
                        if self.c[(B, v)] != -1:
                            size += self.n - self.c[(B, v)] - 1
                    self.c[(B, u)] = size
                    unknownNeighboursOfAP[u] -= 1
                    if unknownNeighboursOfAP[u] == 1:
                        for C in self.component_of_u[u]:
```

```
                        if self.c[(C, u)] == -1:
                            Q.append((u, C, False))
                            break
            else:
                B = triple[1]
                u = triple[0]
                if self.c[(B, u)] == -1:
                    size = 1
                    for C in self.component_of_u[u]:
                        if self.c[(C, u)] != -1:
                            size += self.c[(C, u)]
                    self.c[(B, u)] = self.n - size
                    unknownNeighboursOfBlock[B] -= 1
                    if unknownNeighboursOfBlock[B] == 1:
                        for v in treeEdgesBlockToAP[B]:
                            if self.c[(B, v)] == -1:
                                Q.append((B, v, True))
                                break
        self.construct_weighted_block_tree_completed = True
        return

    def compute_tau(self, B):
        length = len(self.component_contains[B])
        h = [1] * length
        for i in range(length):
            u = self.component_contains[B][i]
            if self.isArticulationPoint(u):
                h[i] = self.n - self.c[(B, u)]
        return h

    def brandesForBCBCC(self, B, tau):
        n = len(self.component_contains[B])
        V = [x for x in range(n)]
        edges = [[] for x in range(n)]
        bc = [0] * n
        for i in range(n):
            u = self.component_contains[B][i]
            if self.isArticulationPoint(u):
                for v in self.edges[u]:
                    if self.isInBlock(v, B):
                        j = self.findIndex(v, B)
                        edges[i].append(j)
```

```
            else:
                for v in self.edges[u]:
                    j = self.findIndex(v, B)
                    edges[i].append(j)
        for s in V:
            S = []
            P = [[] for x in range(n)]
            sigma = [0] * n
            sigma[s] = 1
            d = [-1] * n
            d[s] = 0
            Q = deque([s])
            while len(Q) > 0:
                i = Q.popleft()
                S.append(i)
                for j in edges[i]:
                    if d[j] < 0:
                        Q.append(j)
                        d[j] = d[i] + 1
                    if d[j] == d[i] + 1:
                        # v is a predecessor of w on a shortest path from s to w
                        sigma[j] += sigma[i]
                        P[j].append(i)
            delta = [0] * n
            while len(S) > 0:
                j = S.pop()
                for i in P[j]:
                    delta[i] += (sigma[i] / sigma[j]) * (tau[j] + delta[j])
                if j != s:
                    bc[j] += tau[s] * delta[j]
        for i in range(n):
            u = self.component_contains[B][i]
            self.bc[u] += bc[i]
        return

    # Utility functions

    def isArticulationPoint(self, u):
        return isinstance(self.component_of_u[u], list)

    def isInBlock(self, u, B):
        if self.isArticulationPoint(u):
```

```
            return B in self.component_of_u[u]
        else:
            return B == self.component_of_u[u]

    def findIndex(self, u, B):
        if self.isArticulationPoint(u):
            return self.index_AP_in_component[(B, u)]
        else:
            return self.index_in_component[u]

    def addNewComponent(self, u):
        if self.isArticulationPoint(u):
            self.component_of_u[u].append(self.component_count)
        else:
            self.component_of_u[u] = self.component_count
        self.component_contains.append([])
        self.component_AP_count.append(0)
        self.component_count += 1
        return

    def assignComponent(self, u, component):
        if self.isArticulationPoint(u):
            self.component_of_u[u].append(component)
        else:
            self.component_of_u[u] = component
        return

    def is_connected(self):
        self.dfs(0)
        if self.n - self.time == 1:
            return True
        return False

    def is_undirected(self):
        for u in range(self.n):
            for v in self.edges[u]:
                if u not in self.edges[v]:
                    return False
        return True
```