ABSTRACT

KEYWORDS: Betweenneess Centrality, Graph, BFS, APSP, SSSP

The betweenness centrality index is essential in the analysis of social networks, but costly to compute. Currently, the fastest known algorithm requires $O(n * m)$ time and $O(n + m)$ space, where n is the number of actors in the network and m is total number of connections amongst them which are unweighted. The existing Brandes' algorithm does all pairs shortest path and calculates centrality. New algorithms for betweenness of a graph are introduced which tried to improvise the execution time using the same time complexity but failed to do so with respect to existing best available algorithm. However, time complexity has been improved for Trees whereas some execution time improvisation has been achieved for DAGs also.

CHAPTER 1

INTRODUCTION

1.1 Overview

Since the onset of Facebook and Twitter, social media platforms have gained immense popularity and have emerged as an important medium for exchange of ideas. Social networks capture the relationship among the entities on these platforms. Analysis of such networks allows us to study interesting patterns about the social structure and understand key social phenomena by utilizing the vast amount of information available.

Centrality measures capture the importance of nodes in a social network. Some of the popular ones include degree centrality, closeness centrality, betweenness centrality among others [**?**]. Betweenness centrality becomes important if we assume that the flow of information between entities in a network is happening along the shortest paths between them, as in case of communication networks.

In graph theory, betweenness centrality of a vertex is a measure of the number of shortest paths on which it lies. For a vertex v, its betweenness centrality $bc[v]$ is the sum of ratio of the number of shortest paths between any pair of vertices s and t (denoted by $\sigma_{st}(v)$), $s \neq v, t \neq v, s \neq t$, which passes through v and total number of shortest paths between s and t (denoted by σ_{st}) [**?**].

$$bc[v] = \sum_{s \neq v, v \neq t, s \neq t} \sigma_{st}(v)/\sigma_{st} \tag{1.1}$$

1.2 Motivation

Betweenness centrality finds applications in various fields such as community detection [**?**], power grid contingency analysis [**?**], influence maximization [**?**], and the study of the human brain [**?**]. Calculating BC turns out to be computationally expensive [**?**] and takes prohibitively long to run on large graphs having a few million vertices and billion edges. In order to compute BC, in reasonable time, for large graphs, there is a need to improve upon the existing techniques or devise new ones to achieve appreciable speedups. With this goal, new algorithms for computing BC are proposed to reduce the overall execution time.

1.3 Organization of the Thesis

Chapter 2 discusses the state-of-the-art algorithm for BC and places the proposed work in context. In Chapter 3, a new algorithm, specifically for trees, is proposed which reduces the execution time by factor of number of vertices in tree. Another algorithm specifically for DAG is proposed which is detailed in Chapter 4. In chapter 5, another algorithm is proposed for graphs which reuses the partial computed values but doesn't improves the execution time. In chapter 6, we present the results of our proposed algorithm and compare those results with Brandes' algorithm. In chapter 7, we discuss some of the prior works on approximations of BC and implementation on GPUs. Concluding remarks and future directions of work are presented in Chapter 8.

CHAPTER 2

BRANDES' ALGORITHM FOR COMPUTING BETWEENNESS CENTRALITY

For a graph $G \equiv (V, E)$, let, $n = |V|$ and $m = |E|$.

A straight forward implementation for computing BC of a vertex using Equation 1.1 would require $O(n^3)$ operations to compute $bc[v]$ for all $v \in V$ as there are $O(n^2)$ pairs of vertices.

The algorithm proposed by Brandes [**?**], reduced the complexity of node BC computation to $O(mn)$ for unweighted graphs.

The algorithm accumulates the dependencies of pairs over target vertices. After accumulation, the dependency of v to $s \in V$ is

$$\Delta_s(v) = \sum_{t \in V} \Delta_{st}(v) \tag{2.1}$$

Let $P_s(u)$ be the set of parents of u on the shortest paths from s to all the vertices in V.

$$P_s(u) = v \in V : (v, u) \in E, d_s(u) = d_s(v) + 1$$

where $d_s(u)$ and $d_s(v)$ are the shortest distances from s to u and v, respectively.

The accumulated dependency values can be determined using the recurrence Equation 2.2.

$$\delta_s(v) = \sum_{u:v \in P_s(u)} (\delta_s(u) + 1) * \sigma_{sv}/\sigma_{su} \tag{2.2}$$

Computation of $\delta_s(v)$ for all $v \in V, s \neq v$, is done in two phases. First, a breadth first search (BFS) is initiated from s to compute σ_{sv} and $P_s(v)$ for each v. Then, in a back propagation phase, $\delta_s(v)$ is computed for all $v \in V$ in a bottom-up manner using Equation 2.2. In the back propagation phase, the δ values (which are initialized to zero), are calculated and are added to the centrality values.

Each phase of Brandes' algorithm 1 visits each edge at most once, taking $O(m+n)$ time. The phases are repeated, considering each vertex as the source. The overall time complexity is $O(mn)$ [**?**].

2.1 Relevance of the Work Proposed in this Thesis

Brandes' Algorithm is by far the best known, simple and efficient algorithm for computing exact values of betweenness centrality for graphs. Various algorithms are proposed with pre-processing the graphs but this algorithm uses simplistic approach to calculate BC, In this thesis, we have considered this algorithm as benchmark for comparing our results. The work proposed in this thesis aims to improve execution time by proposing variants of the algorithm for trees, DAGs and graphs.

Algorithm 1: Brandes' Sequential Algorithm

```
bc[v] ← 0, v ∈ V;
for s ∈ V do
    Forward phase: BFS from s
    Q.enqueue(s);
    while Q not empty do
        v ← Q.dequeue();
        S.push(v);
        for each neighbour w of v do
            if d[w] ¡ 0 then
                Q.enqueue(w);
                d[w] ← d[v] + 1;
            end
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                P[w].append(v);
            end
        end
    end
    Backward phase: Back propagation
    δ[v] ← 0, v ∈ V;
    while S not empty do
        w ← S.pop();
        for v ∈ P[w] do
            δ[v] ← δ[v] + (1 + δ[w]) ∗ σ[v]/σ[w];
        end
        if w ≠ s then
            bc[w] ← bc[w] + δ[w];
        end
    end
end
```

CHAPTER 3

ALGORITHM FOR TREES

In this chapter, we propose a new algorithm for computing Betweennness Centrality specifically for trees. Consider a tree $T \equiv (V, E)$ such that $|V| = n$ and $|E| = n - 1$. To compute Betweenness Centrality for a tree, the Brandes' Algorithm by default takes $O(n^2)$ time. We propose an algorithm with time complexity of $O(n)$.

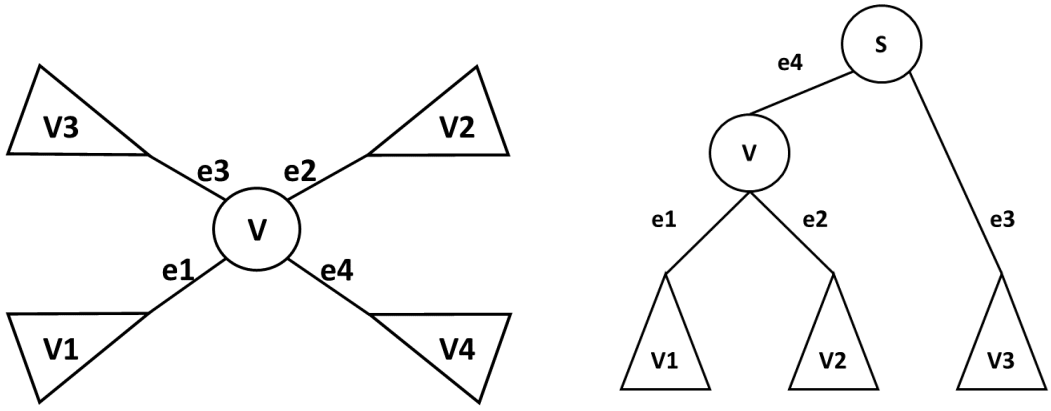


Figure 3.1: Reachable vertices of v

In Figure 3.1(a), a vertex v has four degree with edges as $e1$, $e2$, $e3$, $e4$ and $v1$, $v2$, $v3$, $v4$ corresponds to the number of vertices reachable through those edges respectively. We observe that any vertex from set $v1$ has a shortest path to any vertex in set $v2$ only through v. Same is applicable for $v1$ to $v3$, $v4$. Also other paths to be considered are from $v2$ to $v3$, $v4$ and $v3$ to $v4$. So,

$$bc[v] = v1 * (v2 + v3 + v4) + v2 * (v3 * v4) + v3 * v4.$$

In general we can say that for a vertex v with a degree d, $bc[v]$ can be calculated

using the equation 3.1 where v_i corresponds to number of vertices reachable from i^{th} edge.

$$bc[v] = \sum_{i=1}^{d-1} v_i * (v_{i+1} + .. + v_n) \tag{3.1}$$

Hence for calculating betweenness centrality we need to determine the number of vertices reachable through each edge of that vertex.

The proposed algorithm is divided into two pass.

1. Computes the number of reachable vertices for each edge of a vertex.
2. Computes Betweenness Centrality for a vertex using the values computed in the first pass.

Algorithm 2: Betweenness Centrality of tree

```
Select a root vertex s arbitrarily;
pass1(s,-1);
pass2();
```

The proposed algorithm is as follows:
A vertex s is chosen arbitrarily as root. First pass and second pass are then applied on the tree considering s as root.

3.1 First Pass of Tree Algorithm

Depth First Search(DFS) is performed on tree starting from s. Every vertex v returns the number of vertices in its own subtree including the vertex v to its parent p. The parent vertex adds the returned value from all of its children to its own list. Vertex v with a degree d has a list of size d. So at the end of first pass, the list of each vertex except source contains $d-1$ elements and source s contains d elements. The

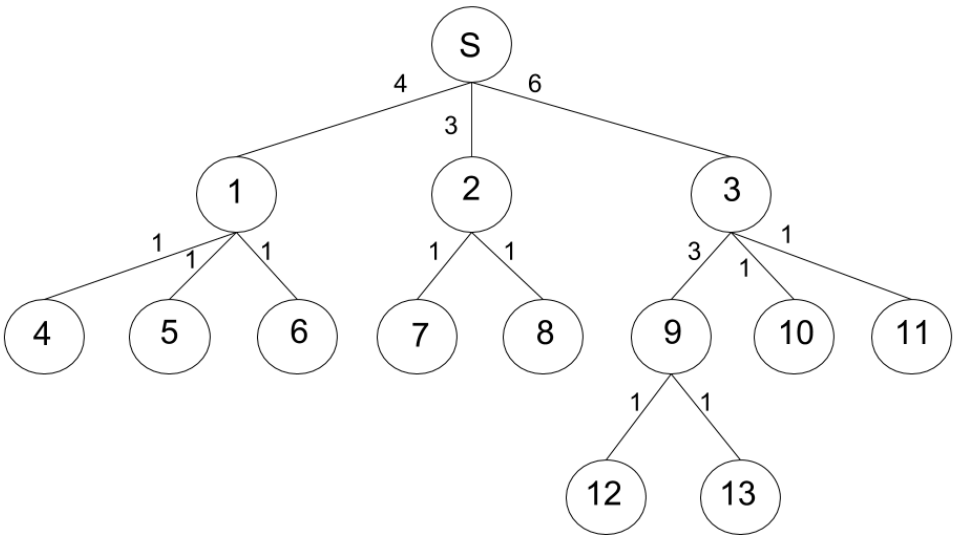


Figure 3.2: Example of Tree

Algorithm 3: Pass1 of Tree Algorithm

```
dfs(child, parent)
sum ← 0;
for each neighbour vertex v of child do
    if v ≠ parent then
        temp ← dfs(v, child);
        sum ← sum + temp;
        Push temp to list[child];
    end
end
return sum + 1;
```

reason is that the source s has no parent while every other vertex has exactly one parent. Hence, we need to add total number of vertices reachable through parent for all vertices except s. For example, in Figure 3.2 vertex '9' returns a value 3 to its parent '3' and vertex 3 adds to its list. Similarly at the end of pass 1, vertex S has values 4, 3, 6 in its list whereas vertex '3' has values 3, 1, 1.

3.2 Second Pass of Tree Algorithm

In the second pass of the algorithm, the number of vertices reachable through parent s of a vertex v are determined. In Figure 3.1(b), we know values of $v1$, $v2$ but to calculate the number of vertices reachable from parent s that is through edge $e4$, we use the formula as
no of vertices reachable through parent = total no of vertices - 1 - ($v1$ + $v2$)

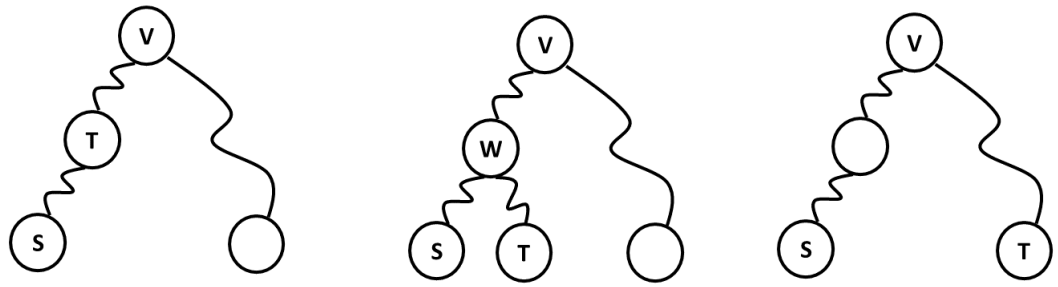
In general we can say that,

$$v_d = n - 1 + \sum_{i=1}^{d-1} list[v_i] \tag{3.2}$$

where n is total number of vertices in graph, v_d is number of reachable vertices through parent s.

For example, in Figure 3.2, for vertex '3' its list contains values 3, 1, 1. So to calculate the number of reachable vertices through its parent, applying formula we get (14 - 1 - (3+1+1)) = 8.

After calculating v_d, it is added to $list[v]$. Thus the calculated value 8 is added to list of vertex '3'. Since we know the total no. of vertices reachable from all edges of a vertex, we can calculate Betweenness Centrality using the equation 3.1.



(a) (b) (c)

Figure 3.3: Cases for shortest path from s to t

3.3 Time Complexity of the Algorithm

First pass of the algorithm visits each node once and calculates the number of vertices reachable through its children. Whereas in second pass of the algorithm, the number of vertices reachable through its parent are calculated using the formula for each and every vertex using the equation 3.2 and the betweenness centrality is calculated using the equation 3.1. Hence its time complexity is $\theta(n)$ where n is total number of vertices.

3.4 Proof Of Correctness

In a tree, since there is exactly one path between a pair of vertices, the path is also the shortest path between them. So the Betweenness Centrality value of a vertex v becomes the total number of paths between any pair s and t passing through v, $s \neq t, v \neq t, s \neq v$ using Equation 1.1.

Lemma 1. *The shortest path from s to t through vertex v should include two distinct edges incident on v such that* $s \neq t, v \neq t, s \neq v$.

Proof. Figure 3.3 shows the possible scenarios to be considered while computing

the shortest path between s and t in the tree. In Figure 3.3(a), the shortest path between s and t won't pass through vertex v. Also in Figure 3.3(b), shortest path between s and t passes through w and not v. In figure 3.3(c), shortest path between s and t does passes through vertex v. And this is the only case possible in which s and t are both incident through two distinct edges of v.

□

Lemma 2. *Consider any vertex $v \in V$, the number of shortest path between any pairs $< s, t >$ such that $s \in S, t \in T$, passing through v are $|S| * |T|$ where S and T are set consisting of vertices reachable through two distinct edges incident on v.*

Proof. By lemma 1, any shortest path between s and t passing through v must be like figure 3.3(c). Also, the set S and T consists of reachable vertices through two distinct edges incident on v, $S \cap T = \Phi$ as it is a tree. Also total number of such $< s$, $t >$ pairs possible are $|S| * |T|$. All these pairs will be unique pairs whose shortest path passes through v. □

Lemma 2 holds for any vertex v having $|S|$ and $|T|$ vertices through two distinct edges. Now, consider any vertex v with d edges. We can choose any pair of edges from this d edges in $\binom{d}{2}$ ways. Hence for computing $bc[v]$, we consider all the edges incident on v, and choose any two edges and apply lemma 2, and add those values to $bc[v]$. This proves that our Equation 3.1 is correct. The proposed algorithm calculates betweenness centrality for vertices using 3.1. Hence the proposed algorithm is correct.

CHAPTER 4

ALGORITHM FOR DAGs

In this chapter, we propose a new algorithm for Betweenness Centrality specifically for DAGs. Consider a Directed Acyclic Graph $G \equiv (V, E)$ where set V contains the vertices and set E contains edges of graph G. First, we tried to apply the proposed algorithm for tree i.e. Algorithm 2 on DAGs by making duplicate vertices only if a vertex has two or more parents. Converting DAG into a tree with duplicating vertices and then applying the algorithm for trees was less efficient than the existing Brandes' Algorithm. The reason being, multiple shortest paths exists in DAG from vertex s to t where $s, t \in V$, which was not the case for trees. And even after converting DAG to tree, the merging of vertices to compute the exact values becomes the bottleneck for the computing betweenness centrality and no improvement is achieved in terms of execution time.

So we proposed another algorithm specifically for DAG, which reduces the execution time with respect to Brandes' Algorithm. The outline of proposed algorithm is given in Algorithm 4.

Algorithm 4: Betweenness Centrality of DAG

```
Topological sort and Reverse topological sort;
Backward Propagation();
Forward Propagation();
```

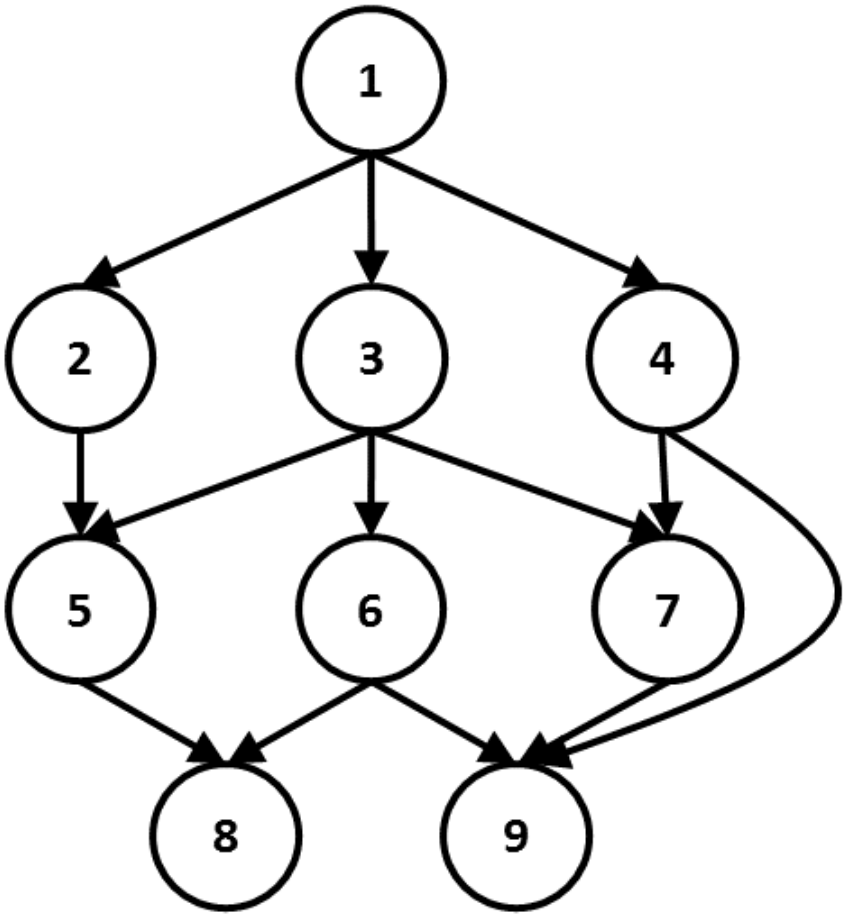


Figure 4.1: Example of DAG

Algorithm 5: Backward Propagation

child $\rightarrow v$, parent $\rightarrow p$
for *each reachable vertex t from v* **do**
  if $d(v,t) + 1 = d(p,t)$ **then**
    $c(p,t) \leftarrow c(p,t) + 1$;
  else
    if $d(p,t) > d(v,t) + 1$ *OR* $d(p,t) < 0$ **then**
      $d(p,t) \leftarrow d(v,t) + 1$;
      $c(p,t) \leftarrow c(v,t)$;
    end
  end
end

vertex	length	count
2	1	1
3	1	1
4	1	1
5	2	2
6	2	1
7	2	2
8	3	3
9	2	1

(a)

vertex	length	count
5	1	1
8	1	1

(b)

vertex	length	count
5	1	1
6	1	1
7	1	1
8	2	2
9	2	2

(c)

vertex	length	count
7	1	1
9	1	1

(d)

Table 4.1: Values stored at vertices 1-(a), 3-(b), 2-(c), 4-(d)

4.1 Backward Propagation Phase

As mentioned in Algorithm 4, the first step of the proposed algorithm is to compute the topological sort and reverse topological sort of the given graph.

The second step is backward propagation where we use reverse topological sort computed in the first step. The outline of Backward propagation step is given in Algorithm 5. At the end of this step, for any vertex t which is reachable from vertex v, the number of shortest paths from v to t and the length of shortest path are stored at vertex v. For example given in Figure 4.1, after backward propagation phase, vertex 1, 2, 3, 4 contains information which is described in Table 4.1.

Table 4.1(a) describes the information stored at vertex 1. So, the first tuple 2, 1, 1 implies that vertex 2 is at a shortest distance of length 1 and only 1 such path exists. Whereas tuple 8, 3, 3 implies that vertex 8 is at a shortest distance of length 3 and 3 such path exists. Similarly for other vertices, values get computed and are stored at each vertex.

In reverse topological sort order, any vertex t pushes the information it possess to all its parents. Reverse topological sort order is used because we are pushing information from a child to a parent so the child's information should be computed first and it should not change, otherwise the changed value propagates to the other vertices in case the order is not considered.

A vertex u updates the values at its parent w. While updating values for vertex v such that shortest path length from u to v is x with count $c1$ then based on values present at vertex w three cases arises:

1. $x + 1 >$ path length from w to v
2. $x + 1 =$ path length from w to v
3. $x + 1 <$ path length from w to v

4.1.1 Case 1

Since a shorter path exists through other vertex, so vertex u doesn't change the values for v at vertex w. For example given in Figure 4.1, vertex '3' tries to update the value at its parent vertex '1' for vertex '9'. The shortest path length from vertex '3' to vertex '9' is 2. Whereas Vertex '4' has already updated its values at vertex '1'. Hence vertex '1' already contains a shortest path to vertex '9' of length 2 i.e. 1-4-9. When vertex '3' tries to update value 3 as a path length to vertex '9' at vertex '1', it won't update because there already exists a shorter path through some other vertex and in this case vertex '4'.

4.1.2 Case 2

Since there exists other paths which are of same length as w to v through other vertices, so u only increments the existing number of paths from vertex w to v by

$c1$. In Figure 4.1, vertex '3' tries to update the value at its parent vertex '1' for vertex '7'. The shortest path length from vertex '3' to vertex '7' is '1'. Whereas Vertex '4' has already updated its values at vertex '1'. Hence vertex '1' already contains a shortest path to vertex '7' of length 2 with number of paths as 1, i.e. 1-7-9. So, vertex '3' increments count of number of paths from vertex '1' to vertex '9' by the number of shortest path from vertex '3' to vertex '9' which is 1. Thus vertex '3' has a shortest path of length 2 and number of such paths are 2.

4.1.3 Case 3

Since the shortest path from vertex w to v doesn't exist or is of length greater than the path through u, so u updates the shortest path length as $x + 1$ and also the number of such paths as $c1$. In Figure 4.1, vertex '5' updates vertex '3' for vertex '8' as path length 1 and number of such paths as 1 since vertex '3' doesn't have any path to vertex '8' before any updates by vertex '6'.

After backward propagation phase, any vertex v will have total number of shortest paths to each reachable vertex from v and their path length.

4.2 Forward Propagation Phase

In Algorithm 6, $d(v, t)$ is shortest distance calculated in second step from v to t whereas, $c(v, t)$ is count of shortest path from v to t. Δ_{vt} is ratio of number of shortest paths to t which passes through v. The third step is to calculate Betweenness Centrality in the topological sorted order of vertices computed in the first step. For each parent p of vertex v, we check whether there is shortest path to any node t reachable from v such that distance from p is (distance from $v + 1$). If such a path

Algorithm 6: Forward Propagation

```
bc[v] ← 0, v ∈ V;
for each parent p of vertex v do
    for each reachable vertex t from v do
        if d(v, t) + 1 = d(p, t) then
            Δ_vt ← c(v, t)/c(p, t);
            δvt ← δvt + Δ_vt * (1 + δpt);
            bc[v] ← bc[v] + Δ_vt * (1 + δpt);
        end
    end
end
```

exists then we calculate Betweenness Centrality which is given in Algorithm 6.

4.3 Space Complexity

The algorithm requires storing information at each and every vertex v. This information is a tuple of each reachable vertex from vertex v, shortest length possible, count of number of shortest path. So in worst case in DAG, all higher numbered vertices are reachable from lower numbered vertices. This makes n entries in the table stored at each vertex. Since there are n vertices and each uses n memory, so worst case space required be $O(n^2)$.

4.4 Summary

The betweenness centrality values are calculated correctly with reduced execution time with respect to Brandes' Algorithm. The issue with this algorithm is that it requires a huge amount of space since we store count and distance for each reachable vertex t from v at vertex v.

CHAPTER 5

ALGORITHM FOR GRAPHS

In this chapter we discuss about our proposed algorithm for graphs. Given is a Graph $G \equiv (V, E)$ where V is total number of vertices and E is total number of edges.

The drawback of proposed algorithm of DAG was huge memory requirement so it is not feasible to apply the same algorithm for graphs. What we observed is that Brandes' algorithm makes each and every vertex as a source in any order and performs BFS. But we can select an order such that we can reuse partial BFS graph which are commmon for two vertices. Such an ordering is possible, if we compute BFS for vertex v then for any neighbour u of v we can reuse that graph partially, provided that u has not yet been considered for BFS. Similarly, u's graph can be reused for all its neighbours if they are not considered for BFS.

For a source w, the graph which gets formed while BFS has its edges classified at every vertex as *parent*, *cousins*, *children*. While performing BFS, if there is a vertex v at height h and another vertex u at height $h + 1$ and there is an edge between v and u then for vertex v, vertex u is *child* and for vertex u, vertex v is *parent*. If there is an edge in original graph between any two vertices and not included in BFS graph, then they are *cousins* of each other. For source w, its *parent* and *cousins* remains to be empty set.

In Figure 5.1 vertex 0 is considered as source and in Figure 5.2, vertex 1 is considered as source. These graphs are BFS graph which consists of dotted edges representing *cousins* and directed edges represent *parent* to *child* relation.

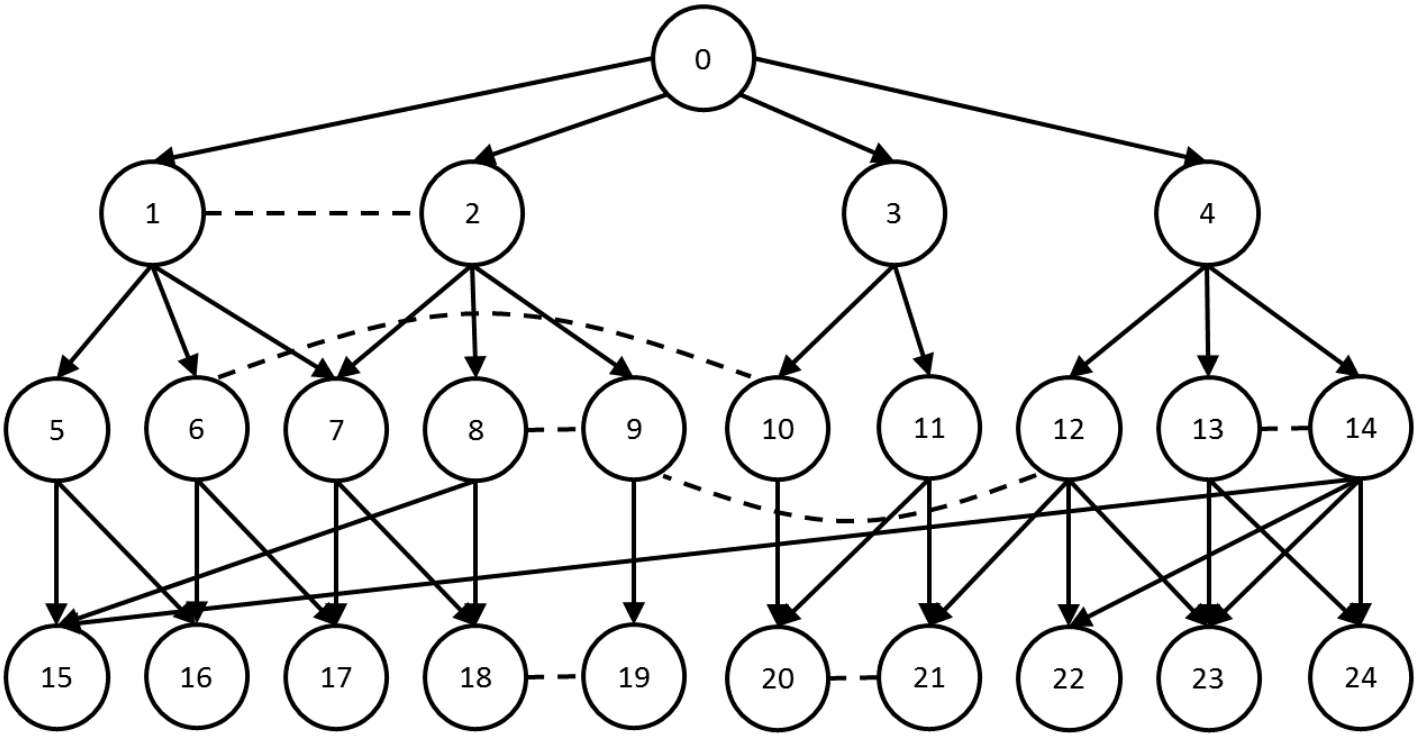


Figure 5.1: BFS with vertex 0 as source

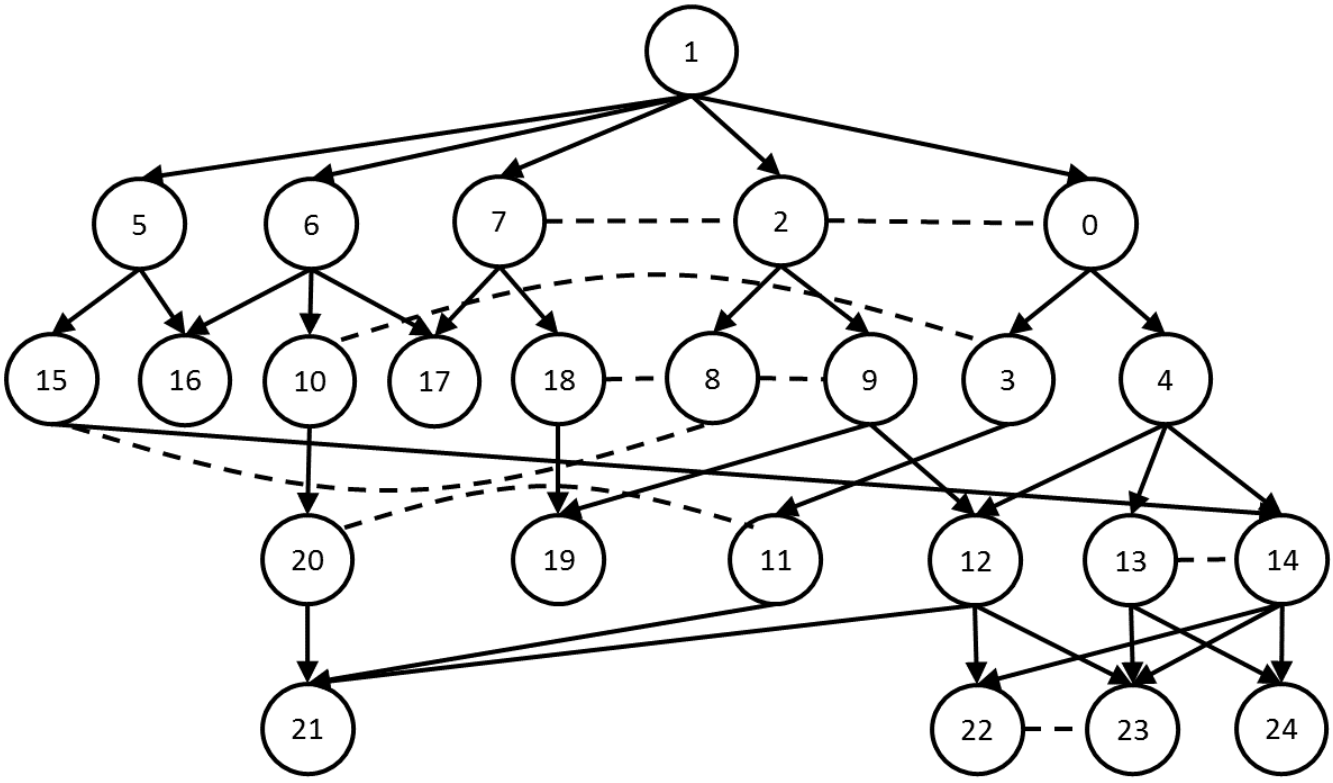


Figure 5.2: BFS with vertex 1 as source

In Figure 5.1, For vertex 6, vertex 1 is a *parent*, vertex 10 is *cousin* and vertices 16 and 17 are *children*. Similarly, For vertex 1, vertex 0 is *parent*, vertex 2 is *cousin* and vertices 5, 6 and 7 are *children*.

For source vertex w, *parent* and *cousin* remain to be an empty set. For example, in Figure 5.1, vertex 0 has no *parent* and no *cousin*. We perform BFS on vertex 0 and obtain information about each vertex's *parent*, *children*, *cousins*. Now we re-use these information to perform BFS and compute betweenness centrality for vertex 1 which is shown in Figure 5.2.

Now to compute BFS graph of u, u becomes the source and its $sub-tree$ in graph of v remains unchanged. Vertex 1 and all its reachable vertices in original BFS graph become $sub-tree$ in this BFS graph of u as they remain un-changed in Figure 5.2.

If there are any *cousins* from $sub-tree$ to outside of $sub-tree$ then they will be *children* in the new graph. This new relation of *parent* with their *children* makes the *children* to form $extended\ sub-tree$. *Cousins* from $extended\ sub-tree$ gets changed to their *children* and thus a new parent child link is formed. In Figure 5.2, vertex 2 becomes *child* of vertex 1, because vertex 2 was *cousin* in Figure 5.1 and vertex 1 belongs to $sub-tree$. Also, vertex 10 becomes *child* of vertex 6, because vertex 10 was *cousin* in Figure 5.1 and vertex 6 belongs to $sub-tree$. This makes vertex 2 and 10 and their reachable vertices which are not included in $sub-tree$ in Figure 5.1 to be included to $extended\ sub-tree$. Thus vertices 8, 9, 10, 20 are included in $extended\ sub-tree$. Also cousins from $extended\ sub-tree$ not belonging to $sub-tree$ and $extended\ sub-tree$ becomes their *children*. For example, vertex 12 becomes *child* of vertex 9 in Figure 5.2 because vertex 12 is not included in any $sub-tree$.

Hence, we can construct BFS graph of u using BFS graph for v. σ values can

be easily calculated while making and breaking the links. Then we apply back propagation phase of Brandes' algorithm to calculate Betweenness Centrality.

The shortcoming of this algorithm is that it takes up a huge amount of space as we need to store graphs for each vertex. Other reason is due to random access nature of memory requirement, improvement in execution time was not achieved. The time complexity remains same as Brandes' algorithm but execution time does not decrease compared to Algorithm 1.

CHAPTER 6

EXPERIMENTAL RESULTS

We conducted an experimental evaluation of our algorithms, with two major driving goals in mind: study the behavior of the algorithms presented in this thesis and compare it with that of state of art algorithm 1 in terms of execution time.

6.1 Implementation and environment

We implemented our algorithms and the one presented in algorithm 1 on IITM's Libra cluster node which has 74GB RAM size and using 18GB as maximum heap space.

6.2 Results

We have compared our algorithm with Brandes' algorithm and we found that there is huge reduction in execution time.

6.2.1 Results for Algorithm on Tree

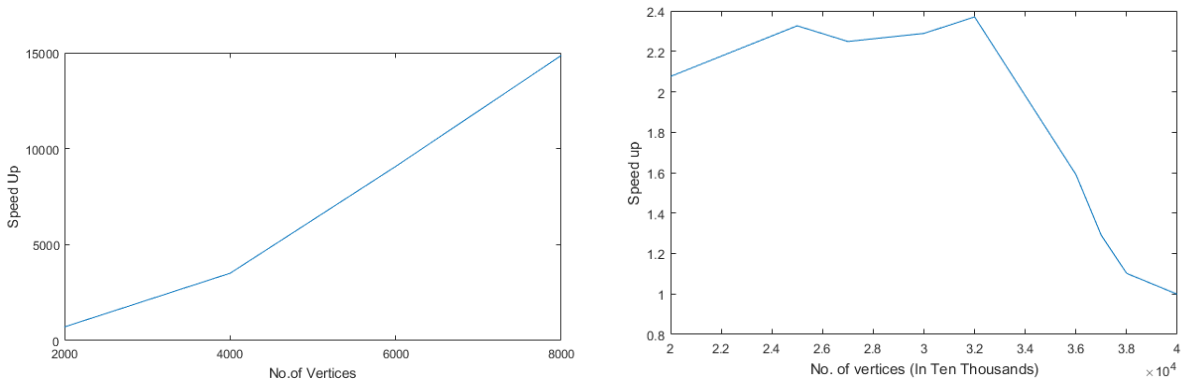
The trees used for the experimental purposes are synthetic trees made through random functions. The synthetic trees which were developed used a random function which generated a random number and that generated number was used as a degree for that particular vertex. And correspondingly new edges were added

from that vertex. It was made sure that, the generated graph is a tree or a forest and does not have any cycle

As we can see in table 6.1, that as per increase in the number of vertices of the tree, the time taken by Brandes' algorithm increases to a large extent as compared to proposed algorithm. This proves the basis which we stated to reduction in time complexity for computing betweenness centrality for trees. As we can see that in Figure 6.1(a), the speed up increases linearly with respect to number of vertices.

No.of vertices	Proposed algorithm on tree	Brandes' algorithm
	Time in sec	Time in sec
2000	0.10	76.7
4000	0.11	387.3
6000	0.12	1093.4
8000	0.13	2005.5

Table 6.1: Comparison of Brandes' algorithm with our algorithm



(a) (b)

Figure 6.1: Speed Up of our algorithm (a) Tree (b) DAG

6.2.2 Results for Algorithm on DAGs

The DAGs used for the experimental purposes are synthetic DAGs made through random functions. The synthetic DAGs which were developed used a random function which generated a random number and that generated number was used

as a degree for that particular vertex v. And correspondingly new edges were added from that vertex v to a vertex u randomly chosen within a range with a higher value that vertex v. It was made sure that, the generated graph is a DAG does not have any cycle.

As we can see in table 6.2, that as per increase in the number of vertices of the tree, the time taken by Brandes' algorithm increases and the time required by our algorithm increases to a large extent but still is less than Brandes' algorithm. In Figure 6.1(b), the speed up achieved is around 2 and then it decreases as the number of vertices increases and becomes 1. As the number of vertices increases, not much performance gain is achieved because we use huge memory and once the memory is used up, the program takes huge amount of time swapping the memory in and out of RAM. Also for more higher number of vertices, our algorithm doesn't work the heap space provided. Hence we can conclude that, if given enough memory to compute betweenness centrality, our proposed algorithm beats the Brandes' algorithm.

No.of vertices	No.of edges	Proposed algorithm on DAG	Brandes' algorithm
		Time in sec	Time in sec
20000	91207	57.9	120.2
25000	111852	88.9	206.8
27000	121079	109.5	246.4
30000	134769	124.9	285.9
32000	144339	141.9	336.4
36000	163172	246.0	391.1
37000	165877	333.3	429.9
38000	170896	435.1	479.5
40000	179765	625.0	626.2

Table 6.2: Comparison of Brandes' algorithm with our algorithm

CHAPTER 7

RELATED WORK ON BETWEENNESS CENTRALITY IN GRAPHS

Betweenness centrality was devised as a general measure of centrality way back in 1977. It applies to a wide range of problems in network theory, including problems related to social networks, biology, transport and scientific cooperation. Then in 1994, Douglas R, White et al. generalized the centrality measures for betweenness on undirected graphs to the more general directed case [**?**]. Then in 2011, Brandes devised an algorithm for calculating betweenness centrality which by far remains the state of art solution [**?**]. After his work, lots of research has been done in reducing the execution time and also approximating the betweenness centrality.

One work where J.Newman relaxes the assumption of betweenness centrality and counts just the number of shortest path was published in 2005 where the centrality was based on the random walks of the graph [**?**]. M Barthlemy states in his paper that the in general, the BC is increasing with connectivity as a power law of an exponent [**?**].

First work on approximating betweenness centrality was done by David Bader et al. in 2007 where they approximated on basis of an adaptive sampling technique [**?**]. Robert Geisberger et al. approximated the BC values such that the unimportant nodes also had a good approximation which was not achieved by other work [**?**]. In 2016 Matteo Riondato proposed approximation techniques where the algorithms are based on random sampling of shortest paths and offer probabilistic

guarantees on the quality of the approximation [**?**].

Then the work began on approximating as well as running the program on GPUs to parallelize the algorithm. Keshav Pingali et. al in 2013 calculated the exact BC of a graph where they ares able to extract large amounts of parallelism and showed it can be applied to large graphs [**?**]. Kamer Kaya et. al computed Betweenness centrality on GPUs and on heterogeneous architectures in 2013 where they showed that heterogeneous computing, i.e., using both architectures at the same time, is a promising solution for betweenness centrality [**?**]. McLaughlin and Bader further increased the speed up in parallel setting using hybid approach of using vertex and edge parallelism instead of only vertex or only edge parallelism and beat the then fastest algorithm and achieved high performance [**?**].

CHAPTER 8

CONCLUSION AND FUTURE WORK

We have proposed algorithms on trees, DAGs and graphs. The proposed algorithms on trees improvises time complexity from $\theta(n^2)$ to $\theta(n)$. Whereas the algorithm on DAGs reduces the execution time with respect to Brandes' algorithm. We plan to implement the Brandes' algorithm in parallel and also approximate the values of centrality further reducing the execution time with allowing some loss in precision.

The proposed algorithm for DAG comes with requirement for huge amount of space and thus won't compute for large graphs of size few million vertices and billion edges. The proposed algorithm for graph reuses the partial computed values but fails to improve execution time. As overall future work in this area is calculating betweenness centrality for a given single vertex only. Brandes' algorithm calculates the values incrementally and will calculate for all the vertices, but it would be challenging to compute without incrementally and directly for a given vertex. Also optimization based on recent GPUs would lead to reducing the execution time.