BETWEENNESS CENTRALITY OF A GRAPH

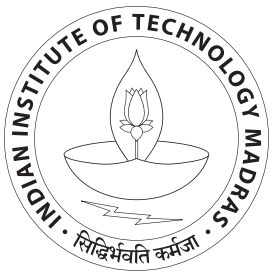
A Project Report

submitted by

TEJAS SHAH

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

September 2017

THESIS CERTIFICATE

This is to certify that the thesis entitled **BETWEENNESS CENTRALITY OF A GRAPH**, submitted by **TEJAS SHAH**, Roll no **CS16M026**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work carried out by her under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Rupesh Nasre
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 9 September 2017

ACKNOWLEDGEMENTS

My foremost gratitude and respects to my guide Dr. Rupesh Nasre for his guidance, motivation and moral support in my execution of this project work. His passionate talks about the subject and the many hour-long discussions renewed the zeal in me to keep trying despite the unsatisfying results. His patience with students has been incredible.

I thank Somesh Singh and other PACE lab students for lending their immense help in my work.

I thank all my classmates and seniors in IITM for their presence that made life happy and incredible here. Their inputs have been significant in every step of these two years.

I thank my parents for being highly accommodating of my erratic schedules and for their love and support. I thank God for making all of this happen.

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.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

BC	Betweenness Centrality
APSP	All Pairs Shortest Path
SSSP	Single Source Shortest Path

CHAPTER 1

INTRODUCTION

1.1 Overview

In social network analysis, graph-theoretic concepts are used to understand and explain social phenomena. A social network consists of actors which may or may not share a relation amongst them. An important metric for the analysis of social networks are centrality which are based on the vertices of the graph. The centrality metric ranks the actors of the network according to their importance in the social structure. One such centrality is betweenness.

In graph theory, betweenness centrality in a graph is a metric based on shortest paths. For each pair of vertices in a connected graph, there exists at least one shortest path between them. This shortest path is determined by the fact that for unweighted graphs, the number of edges that the path passes through is minimized and for weighted graphs, the sum of the weights of the edges is minimized. So for a vertex v, the betweenness centrality $bc[v]$ is sum of ratio of the number of shortest paths between any pair of vertices s and t, other than vertex v which passes through v and total number of shortest paths between s and t. The betweenness centrality of a vertex v is given by the expression:

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

where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v.

1.2 Motivation

Betweenness centrality has applications in various fields such as community detection, power grid contingency analysis, and the study of the human brain. This analyses have high computational cost that prevents the examination of large graphs of size of few million vertices and billion edges. Motivated by the fast-growing need to compute betweenness centrality on such large, yet very sparse graphs, new algorithms for computing betweenness centrality are introduced to reduce the overall execution time.

1.3 Organization of the Thesis

Chapter 2 discusses the previous works in this area and places the proposed work in context. In Chapter 3, 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. As a part of future work, we plan to implement the existing best work for calculating betweenness centrality which is Brandes' Algorithm as described in Algorithm 1 in parallel on GPU using CUDA. We also plan to approximate the betweenness centrality in parallel implementation and further reduce the execution time with some loss of precision in exact values of betweenness centrality. Chapter 8 concludes the work with analysis and future direction.

CHAPTER 2

Brandes' Algorithm for computing Betweenness Centrality

Given a connected graph $G \equiv (V, E)$ where V is total number of vertices and E is total number of edges, let σ_{st} be the number of shortest paths from a source $s \in V$ to a target $t \in V$. Let $\sigma_{st}(v)$ be the number of such $s \rightarrow t$ paths passing through a vertex $v \in V$, $v \neq s, v \neq t$. Let the pair dependency of v to $< s, t >$ pair be the fraction $\Delta_{st}(v) = \sigma_{st}(v)/\sigma_{st}$. The betweenness centrality of v is defined by

$$bc[v] = \sum_{s \neq v, v \neq t, s \neq t} \Delta_{st}(v) \tag{2.1}$$

Since there are $O(n^2)$ pairs in V, one needs $O(n^3)$ operations to compute $bc[v]$ for all $v \in V$ by using Equation 2.1. Brandes reduced this complexity and proposed an $O(mn)$ algorithm for unweighted networks. The algorithm is based on the accumulation of pair dependencies over target vertices. After accumulation, the dependency of v to $s \in V$ is

$$\Delta_s(v) = \sum_{t \in V} \Delta_{st}(v) \tag{2.2}$$

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

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

where $d_s(u)$ and $d_s(v)$ are the shortest distances from s to u and v, respectively. P_s defines the shortest paths graph rooted in s. Brandes' observed that the accumulated

dependency values can be computed recursively:

$$\Delta_s(v) = \sum_{u:v \in P_s(u)} (\Delta_s(u) + 1) * \sigma_{sv}/\sigma_{su} \tag{2.3}$$

To compute $\Delta_s(v)$ for all $v \in V$ except s, Brandes' algorithm uses a two-phase approach:

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 by using Equation 2.3. In the first phase, the algorithm computes $\sigma[v]$ for $v \in V$ which is the number of shortest paths from the source vertex s to v. In addition, the parents of v on these shortest paths are stored in $P[v]$. Before the second phase, the algorithm initializes $\delta[v]$ to 0. In the back propagation phase, the δ values are calculated and are added to the centrality values.

Both the phases of Algorithm 1 considers all the edges at most once, taking $O(m + n)$ time. The phases are repeated for 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 $O(n)$.

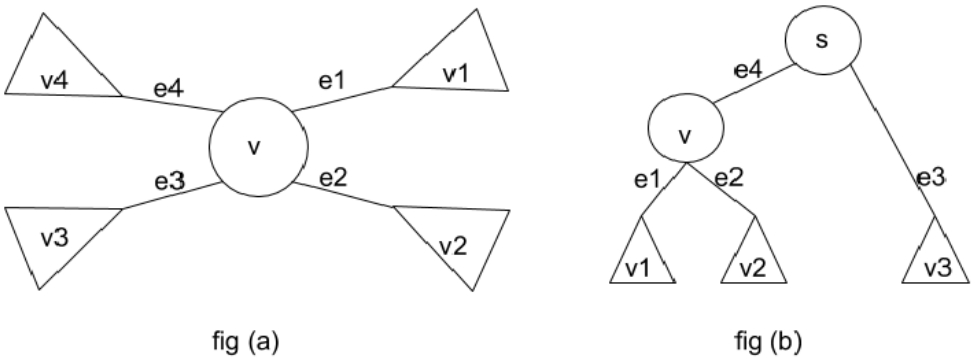


Figure 3.1: Reachable vertices of v

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 observed 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 done on the tree considering s as root.

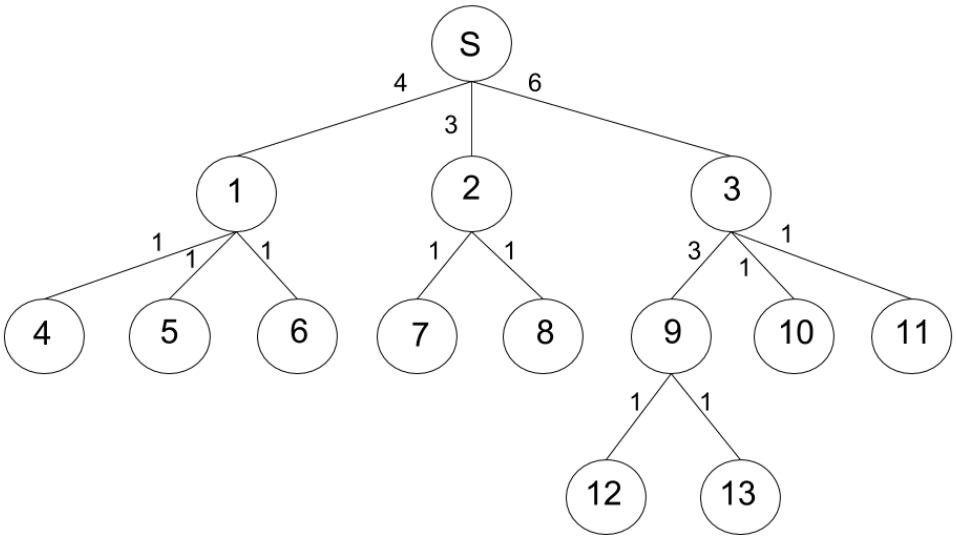


Figure 3.2: Example of Tree

3.1 First Pass of Tree Algorithm

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;
```

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 its children to its own list. Vertex v with a degree d has a list of size d. So at the at end of first pass, the list each vertex except source contains $d-1$ elements and source s contains d elements. The 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

So, in the second pass, the number of vertices reachable through parent s of a vertex v are determined. In fig 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 can 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.
Each pass visits each node once and thus the time complexity of the algorithm is $O(n)$.

3.3 Proof Of Correctness

For a tree, all the vertices are connected and have a single path and thus that path needs to be shortest. So the Betweenness Centrality value of a vertex v becomes the total number of paths between any pair s and t passing through $v, s \neq t, v \neq t, s \neq v$. Now, such a $< s, t >$ pair whose shortest path passes through v, needs to be reachable from different edges of v.

Let us assume that the betweenness centrality values calculated by our proposed algorithm is incorrect. Two cases arise that either our algorithm calculates more value of centrality or it calculates less than the actual value for a vertex v.

1. If our proposed algorithm has calculated more centrality value then for a pair $< s, t >$ whose shortest path does not contain v, some values are added to $bc[v]$. But according to Algorithm 2, it does not considers any pair a, b which does not pass through v and thus won't add any value to $bc[v]$.

2. If our proposed algorithm has calculated less centrality value then for a pair $< s, t >$ whose shortest path contains v, some values are not added to $bc[v]$. Algorithm 2 considers does not considers the vertices reachable from same edge. Since pair $< s, t >$ is not considered, then it should be reachable from v through same edge and thus contradicts the information that v lies on shortest path from s to t

Thus, both this cases are not possible which proves that our proposed algorithm is sound and complete.

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. First, we tried to apply the proposed algorithm for tree on DAGs by making making duplicate vertices 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 then the existing Brandes' Algorithm. The reason being, multiple shortest path 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 algorithm 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();
```

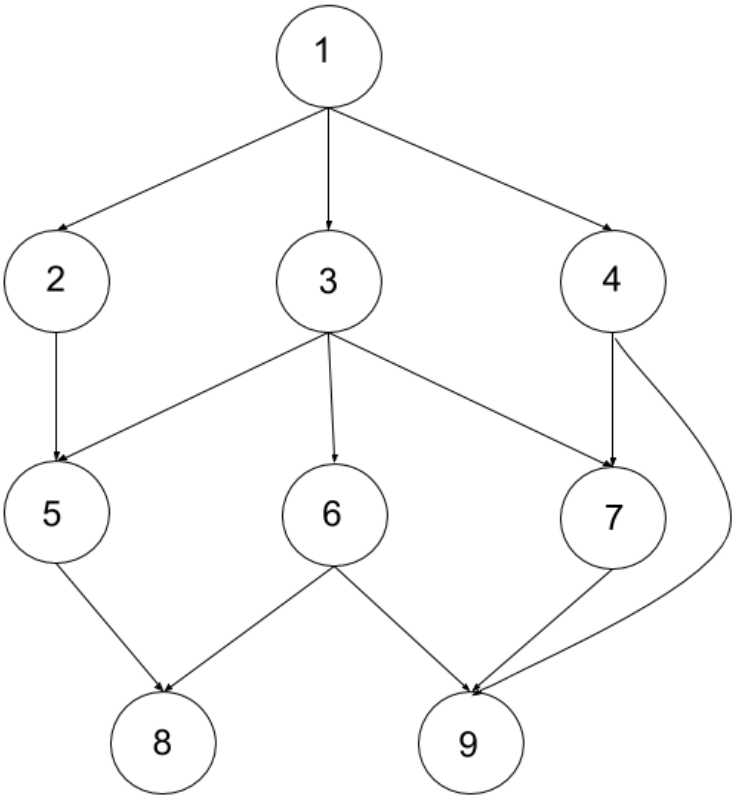


Figure 4.1: Example of DAG

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

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 following information: in following table.

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

Table 4.1: Values stored at Vertex 1

vertex	length	count
5	1	1
8	1	1

Table 4.2: Values stored at Vertex 2

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

Table 4.3: Values stored at Vertex 3

vertex	length	count
7	1	1
9	1	1

Table 4.4: Values stored at Vertex 4

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, u doesn't updates 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 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 shorter paths 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 has 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

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
```

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

4.3 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 a vertex as a source and performs SSSP for each and every vertex. But if we select an order such that we can reuse partial SSSP graph which are same for both the vertices. Such an ordering is possible, if we compute SSSP 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 SSSP. Similarly, u's graph can be reused for all its neighbours if they are not considered for SSSP.

For a source w, the graph which gets formed while SSSP has edges classified at every vertex as *parent*, *cousins*, *children*. While performing SSSP, 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 SSSP 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 SSSP graph which consists of dotted edges representing *cousins* and directed edges represent *parent* to *child* relation.

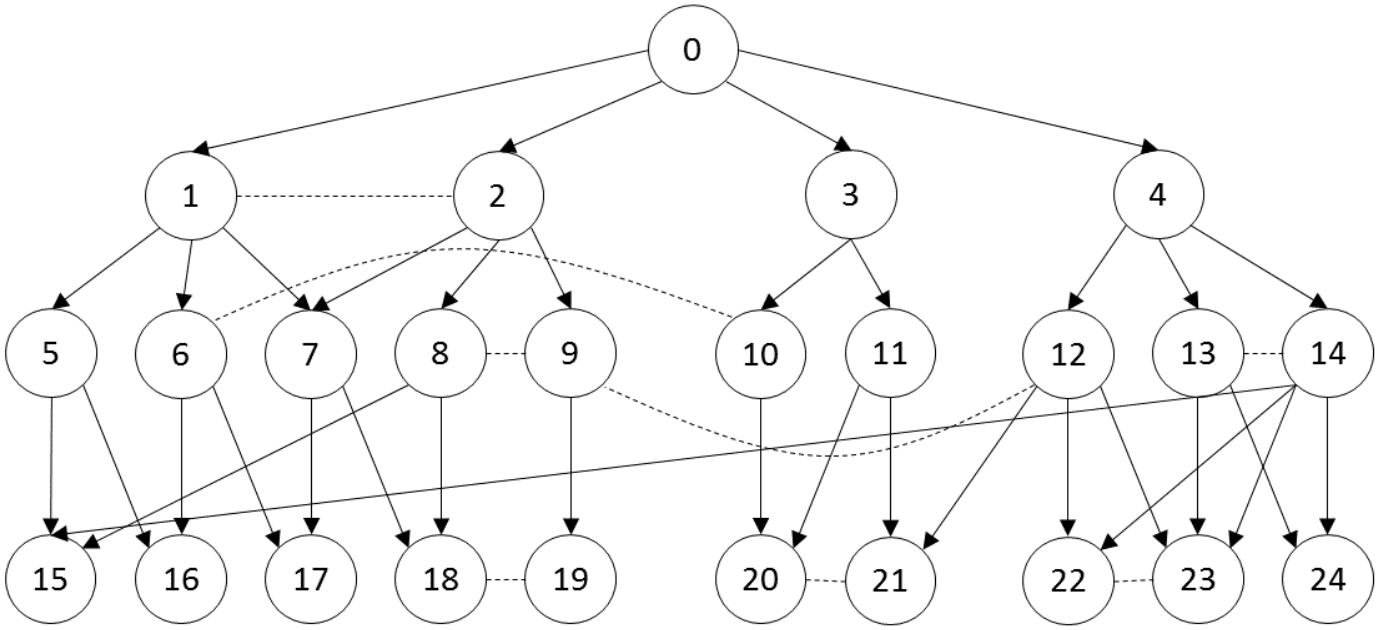


Figure 5.1: SSSP with vertex 0 as source

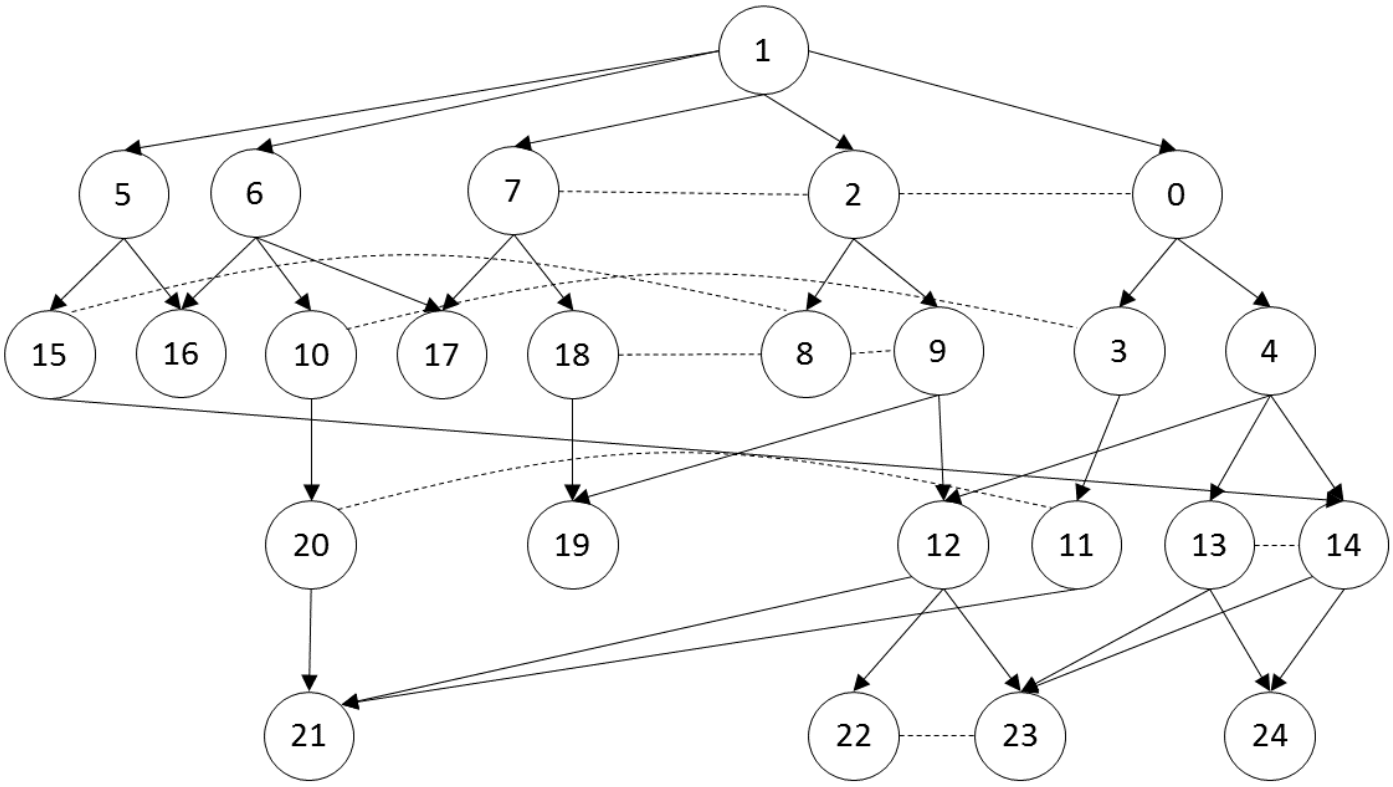


Figure 5.2: SSSP with vertex 1 as source

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 SSSP on vertex 0 and obtain information about each vertex's *parent*, *children*, *cousins*. Now we re-use these information to perform SSSP and compute betweenness centrality for vertex 1 which is shown in figure 5.2.

Now to compute SSSP 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 SSSP graph becomes $sub-tree$ which remains 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. These 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 SSSP graph using for u using SSSP 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 shortcomings 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 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 paper 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. 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 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.

No.of vertices	Proposed algorithm on tree	Brandes' algorithm
	Time in ns	Time in ns
2000	108175589	76791051421
4000	110450761	387301630614
6000	120499273	1093484042473
8000	134970756	2005514757385

Table 6.1: Comparison of Brandes' algorithm with our algorithm

6.2.2 Results for Algorithm on DAGs

The trees used for the experimental purposes are synthetic trees made through random functions. 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. As the number of vertices are increased, 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 ns	Time in ns
20000	91207	57927664569	120239411956
25000	111852	88914295841	206825800469
27000	121079	109585676633	246413985818
30000	134769	124939881885	285926048584
32000	144339	141941467652	336415443364
36000	163172	246004717907	391164367990
37000	165877	333386605803	429938532146
38000	170896	435153434414	479553177044
40000	179765	625092042171	626296047344

Table 6.2: Comparison of Brandes' algorithm with our algorithm

CHAPTER 7

Related Work on Betweenness Centrality on 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 M.E. 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 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. David A. Bader further increased the speed up in parallel setting and beat the then fastest algorithm and achieved high performance.

CHAPTER 8

CONCLUSION AND FUTURE WORK

We have proposed algorithms on trees and DAGs which improvises execution time with respect to Brandes' Algorithm. 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. 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.

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.

REFERENCES

Barthélemy, M. (2004). Betweenness centrality in large complex networks. *The European Physical Journal B*, **38**(2), 163–168. ISSN 1434-6036. URL https://doi.org/10.1140/epjb/e2004-00111-4.

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

Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry*, **40**(1), 35–41. ISSN 00380431.

Geisberger, R., **P. Sanders**, and **D. Schultes**, Better approximation of betweenness centrality. *In Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

McLaughlin, A. and **D. A. Bader**, Scalable and high performance betweenness centrality on the gpu. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14. IEEE Press, Piscataway, NJ, USA, 2014. ISBN 978-1-4799-5500-8. URL https://doi.org/10.1109/SC.2014.52.

Newman, M. J. (2005). A measure of betweenness centrality based on random walks. *Social Networks*, **27**(1), 39 – 54. ISSN 0378-8733.

Riondato, M. and **E. M. Kornaropoulos** (2016). Fast approximation of betweenness centrality through sampling. *Data Mining and Knowledge Discovery*, **30**(2), 438–475. ISSN 1573-756X. URL https://doi.org/10.1007/s10618-015-0423-0.

Sariyüce, A. E., **K. Kaya**, **E. Saule**, and **U. V. Çatalyürek**, Betweenness centrality on gpus and heterogeneous architectures. *In Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-2017-7. URL http://doi.acm.org/10.1145/2458523.2458531.

White, D. R. and **S. P. Borgatti** (1994). Betweenness centrality measures for directed graphs. *Social Networks*, **16**(4), 335 – 346. ISSN 0378-8733.