BETWEENNESS CENTRALITY OF A GRAPH

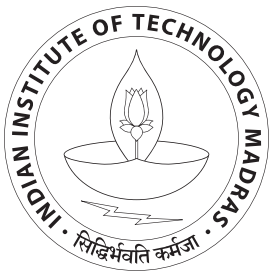
A Project Report

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 section 1.1 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 6 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 $s \to t$ 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 (1.1). 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 (1.3). 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
                enqueue w → Q;
                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
        pop w ← S;
        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 $O(n)$.

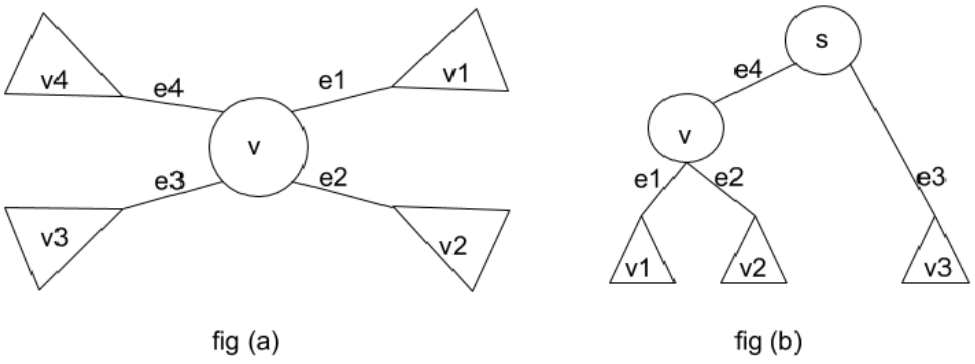


Figure 3.1: Reachable vertices of v

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.

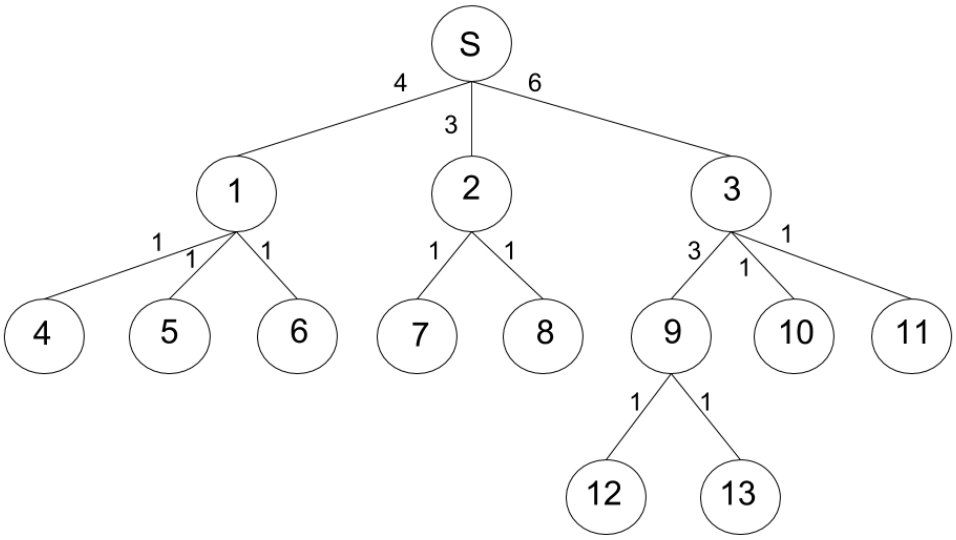


Figure 3.2: Example of Tree

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,

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

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();
```

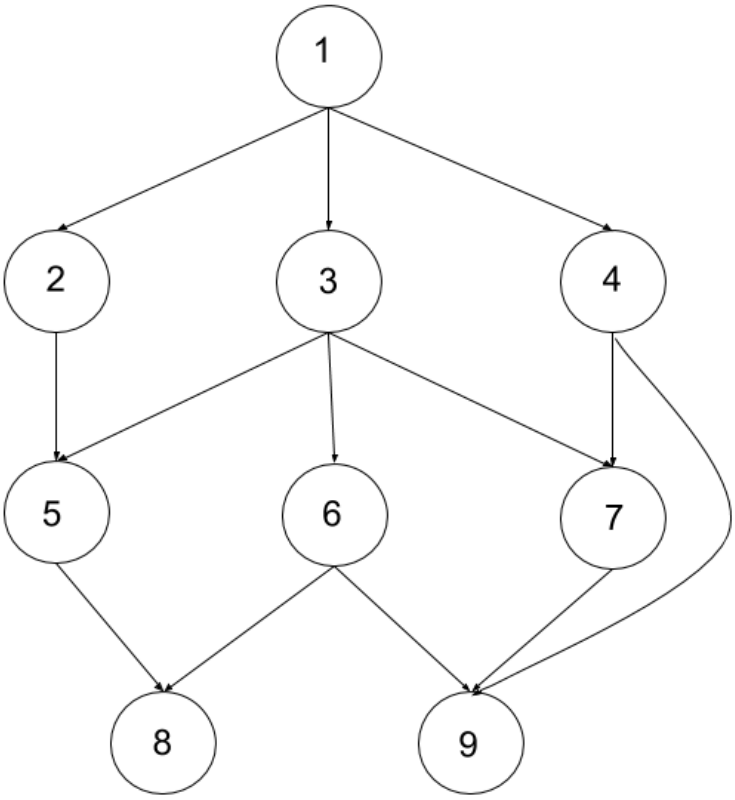


Figure 4.1: Example of DAG

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 table 1.

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.

Since the drawback of proposed algorithm of DAG was huge memory requirement so it is not feasible to apply the same algorithm for graphs. We observed that Brandes' algorithm makes a vertex as a source and performs SSSP for every vertex. But if we select an order such that we can reuse partial 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 garph 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 will be formed while SSSP has edges classified at every vertex as *parent*, *cousins*, *children*. For source w, its *parent* and *cousins* will remain empty set. So a vertex u which is not yet considered for SSSP and is neighbour of source v can be ordered such that it becomes the next source. So to make SSSP graph of u, we will reuse SSSP graph of v.

Now to compute SSSP graph of u, u becomes the source and its subtree in graph of v remains unchanged. If there are any *cousins* from subtree to outside of subtree then they will be *children* in the new graph. These newly converted *children*'s *cousins* will change to *children*. If there is an edge from a vertex which

does not belong to its subtree to a vertex which is part of subtree then they will be cousins. 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 will apply back propagation phase of Brandes' algorithm to calculate Betweenness Centrality.

The issue with this algorithm is that it takes up a huge amount of space as we need to store graphs for each vertex.

CHAPTER 6

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

Adam McLaughlin, D. A. B. (2014). Scalable and high performance betweenness centrality on the gpu. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, **22**(15), 6549–6559.

Ahmet Erdem Sariyce, E. S., Kamer Kaya (2013). Betweenness centrality on gpus and heterogeneous architectures. *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*.

Brandes, U. (2001). A faster algorithm for betweenness centrality.

Douglas R.White, S. P. (1994). Betweenness centrality measures for directed graphs.

Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *American Sociological Association*.

J.Newman, M. (2005). A measure of betweenness centrality based on random walks.

Robert Geisberger, P. S. (2008). Better approximation of betweenness centrality.