

# Betweenness Centrality : Algorithms and Lower Bounds

Shiva Kintali\*

## Abstract

One of the most fundamental problems in large-scale network analysis is to determine the importance of a particular node in a network. Betweenness centrality is the most widely used metric to measure the importance of a node in a network. In this paper, we present *a randomized parallel algorithm* and *an algebraic method* for computing betweenness centrality of all nodes in a network. We prove that any path-comparison based algorithm cannot compute betweenness in less than  $O(nm)$  time.

**Keywords:** all-pairs shortest paths, betweenness centrality, lower bounds, parallel graph algorithms, social networks.

## 1 Introduction

One of the most fundamental problems in large-scale network analysis is to determine the *importance* of a particular node (or an edge) in a network. For example, in social networks we wish to know agents that have very short connections to large portions of the population. In communication networks we wish to know the links that carry a lot of traffic, ISPs that attract a lot of business, links that, if disconnected, decrease network performance dramatically, and so on. A particular way to measure the importance of network elements (nodes or edges) is using *centrality* metrics such as closeness centrality [29], graph centrality [19], stress centrality [31] and betweenness centrality ([16], [2]). An impor-

tant application of centrality arises in the study epidemic phenomena in networks when an infectious disease or a computer virus is disseminated. The power of a node to spread the epidemic is related to its centrality [28]. Centrality metrics also find applications in natural language processing [14], to compute relative importance of textual units.

Betweenness centrality (introduced by Freeman [16] and Anthonisse [2]) is the most popular (and computationally expensive) centrality metric. Some recent applications of betweenness include the study of biological networks [20, 26, 12], study of sexual networks and AIDS [24], identifying key actors in terrorist networks [22, 10], organizational behavior [6], supply chain management [9], and transportation networks [18]. Betweenness can also be used as a heuristic to solve NP-hard problems like graph clustering. For example, Newman and Girvan [25] developed a heuristic to find community structure in large networks, based on betweenness of the edges of the network.

Since the networks of interest are huge, it is important to develop algorithms that compute these metrics efficiently. Brandes [4] showed that betweenness centrality can be computed in the same asymptotic time bounds as  $n$  Single Source Shortest Path (SSSP) computations. Brandes and Pich [5] presented experimental results of estimating different centrality measures under various node-selection strategies. Eppstein and Wang [13] presented a randomized approximation algorithm for closeness centrality.

---

\*College of Computing, Georgia Institute of Technology, Atlanta, GA-30332. Email : [kintali@cc.gatech.edu](mailto:kintali@cc.gatech.edu)

## 1.1 Betweenness Centrality

We denote a network by an *undirected* graph  $G(V, E)$ , with vertex set  $\{v_1, v_2, \dots, v_n\}$  (or  $\{1, 2, \dots, n\}$ ), with  $|V| = n$  vertices and  $|E| = m$  edges, representing the relationships between the vertices. In this paper, we refer to *connected undirected* graphs, unless otherwise stated. Each edge  $e \in E$  has a positive integer weight  $w(e)$ . Unweighted graphs have  $w(e) = 1$  for all edges. A *path* from  $s$  to  $t$  is defined as a sequence of edges  $(v_i, v_{i+1})$ ,  $0 \leq i \leq l$ , where  $v_0 = s$  and  $v_l = t$ . The *length* of a path is the sum of weights of edges in this sequence. We use  $d(s, t)$  to denote the *distance* (the minimum length of any path connecting  $s$  and  $t$  in  $G$ ) between vertices  $s$  and  $t$ . We set  $d(i, i) = 0$  by convention. We denote the total number of shortest paths between vertices  $s$  and  $t$  by  $\lambda_{st} = \lambda_{ts}$ . We set  $\lambda_{ss} = 1$  by convention. The number of shortest paths between  $s$  and  $t$ , passing through a vertex  $v$ , is denoted by  $\lambda_{st}(v)$ . Let  $Diam(G)$  be the diameter (the longest shortest path) of the graph  $G$ . Let  $A = (a_{ij})$  be the adjacency matrix of the graph, i.e.,  $A$  is a 0-1 matrix with  $a_{ij} = 1$  iff  $(i, j) \in E$ .

Let  $\delta_{st}(v)$  denote the *fraction* of shortest paths between  $s$  and  $t$  that pass through a particular vertex  $v$  i.e.,  $\delta_{st}(v) = \frac{\lambda_{st}(v)}{\lambda_{st}}$ . We call  $\delta_{st}(v)$  the *pair-dependency* of  $s, t$  on  $v$ . Betweenness centrality of a vertex  $v$  is defined as

$$BC(v) = \sum_{s, t: s \neq v \neq t} \delta_{st}(v)$$

The *dependency* of a source vertex  $s \in V$  on a vertex  $v \in V$  is defined as

$$\delta_{s*}(v) = \sum_{t: t \neq s, t \neq v} \delta_{st}(v).$$

The betweenness centrality of a vertex  $v$  can be then expressed as

$$BC(v) = \sum_{s: s \neq v} \delta_{s*}(v).$$

Define the set of *predecessors* of a vertex  $v$  on shortest paths from  $s$  as  $P_s(v) = \{u \in V :$

$(u, v) \in E, d(s, v) = d(s, u) + w(u, v)\}$ . The following theorem, states that the dependencies of the *closer* vertices can be computed from the dependencies of the *farther* vertices.

**Theorem 1.1.** [4] *The dependency of  $s \in V$  on any  $v \in V$  obeys*

$$\delta_{s*}(v) = \sum_{w: v \in P_s(w)} \frac{\lambda_{sv}}{\lambda_{sw}} (1 + \delta_{s*}(w))$$

Brandes's Algorithm [4] is based on the above theorem. First,  $n$  single-source shortest paths (SSSP) computations are done, one for each  $s \in V$ . The predecessor sets  $P_s(v)$  are maintained during these computations. Next, for every  $s \in V$ , using the information from the shortest paths tree and predecessor sets along the paths, compute the dependencies  $\delta_{s*}(v)$  for all other  $v \in V$ . To compute the betweenness value of a vertex  $v$ , we finally compute the sum of all dependency values. The  $O(n^2)$  space requirements can be reduced to  $O(n + m)$  by maintaining a *running* centrality score. Note that the centrality scores need to be divided by two if the graph is undirected, since all shortest paths are considered twice. Brandes's Algorithm runs in  $O(nm)$  time for unweighted graphs and  $O(nm + n^2 \log n)$  time for weighted graphs.

## 1.2 Our Results

Brandes's algorithm is a path-comparison based algorithm. We prove that any path-comparison based algorithm cannot compute betweenness in less than  $O(nm)$  time. Betweenness centrality is closely related to All Pairs Shortest Paths Problems (APSP) and algebraic methods have been very successful in obtaining better running times for APSP ([30], [1], [32], [15], [8], [34]). We present an *algebraic method* for computing betweenness centrality of all nodes in a network. For unweighted graphs, our algorithm runs in time  $O(n^\omega Diam(G))$ , where  $\omega < 2.376$  is the exponent of matrix multiplication and  $Diam(G)$  is the diameter of the graph. For weighted graphs with integer weights taken from

the range  $\{1, 2, \dots, M\}$ , we present an algorithm that runs in time  $O(Mn^\omega \text{Diam}(G))$ . As in [4], our time bounds are true in the model where all arithmetic operations (independent of size of the numbers) take unit time and numbers use unit space. Recent observations, on real-world graph evolution, such as densification and shrinking diameters [23], make our algorithms very relevant to the real-world graphs.

We present a *randomized parallel algorithm* for computing betweenness centrality of all nodes in a network. Our approach is based on the *randomized parallel SSSP* algorithm for unweighted graphs is given by Ullman and Yannakakis [33]. We compute the betweenness in two stages (which we call the forward pass and the backward pass). Our algorithm for forward pass runs in  $O(n)$  time using  $O(m \log n)$  processors for unweighted graphs and  $O(n \log^2 n \log M)$  time using  $O(m)$  processors for weighted graphs with integer weights taken from the range  $\{1, 2, \dots, M\}$ . Our backward pass algorithm runs in  $O(n^2)$  time using  $O(n)$  processors for both weighted and unweighted graphs. For bounded-degree graphs, we present an *optimal* backward pass algorithm that runs in  $O(n \log m)$  time using  $O(m)$  processors for unweighted graphs and  $O(Mn \log m)$  time using  $O(m)$  processors for weighted graphs.

## 2 Lower Bounds

**Definition 2.1.** A PATH-COMPARISON BASED ALGORITHM [11]: A Path-comparison based Algorithm  $\mathcal{A}$  accepts as input a graph  $G$  and a weight function. The algorithm  $\mathcal{A}$  can perform all standard operations. However, the only way it can access the edge weights is to compare the weights of two different paths.

Karger, Koller and Phillips [11] established that  $\Omega(n^3)$  is a lower bound on the complexity of any path-comparison based algorithm for the all-pairs shortest path problem on a graph with  $\Theta(n^2)$  edges. They conjectured that similar lower bounds hold for undirected graphs also.

We use their construction to derive lower bounds on computing betweenness in *directed* graphs. For the details of the construction we refer the reader to [11].

The graph  $G$ , they constructed, is a *directed* tripartite graph on vertices  $u_i, v_j$  and  $w_k$  where  $i, j$  and  $k$  range from 0 to  $n - 1$ . The edge set for  $G$  is  $\{(u_i, v_j)\} \cup \{(v_j, w_k)\}$ . Therefore, the only paths are individual edges and paths  $(u_i, v_j, w_k)$  of length two. A weight function  $W$  is properly chosen so that the unique shortest path from  $u_i$  to  $w_k$  goes through  $v_0$ . Note that the betweenness of the node  $v_0$  is  $n^2$ . Let  $\mathcal{A}$  be any path-comparison-based algorithm. Consider giving  $(G, W)$  as input to  $\mathcal{A}$ , and suppose that  $\mathcal{A}$  runs correctly. It must therefore output  $n^2$  as the betweenness of  $v_0$  based on the set of optimal paths  $L$ . Suppose further that a particular path  $p^* = (u_{i^*}, v_{j^*}, w_{k^*})$  was never one of the operands in any comparison operation which  $\mathcal{A}$  performed. The weight function can be suitably modified (as in [11]) to  $W'$  in which  $p^*$  is the unique shortest path from  $u_{i^*}$  to  $w_{k^*}$ , but the ordering by weight of all the other paths remains the same. Note that the centrality of  $v_0$  decreases with the new weight function  $W'$ . If we run  $\mathcal{A}$  on  $(G, W')$ , all path comparisons not involving  $p^*$  give the same result as they did using  $W$ . Therefore, since  $\mathcal{A}$  never performed a comparison involving  $p^*$  while running on  $W$ , we deduce that it still outputs  $n^2$ , which is now incorrect. The following theorem is immediate.

**Theorem 2.2.** *There exists a directed graph of  $3n$  vertices on which any path-comparison based algorithm for betweenness must perform at least  $n^3/2$  path weight comparisons.*

A similar argument can be used to show an  $\Omega(nm)$  lower bound on graphs of  $m$  edges. Assume without loss of generality that  $m \geq 4n$  and that  $2n$  divides  $m$ . We perform the same construction, but of the middle vertices we use only  $v_1, \dots, v_{m/2n}$ , connecting each of them to all the vertices  $u_i$  and  $w_k$ . This requires  $m$  edges and creates  $mn/2$  paths.

**Theorem 2.3.** *There exists a directed graph with  $2n + m/2n$  vertices and  $m$  edges, on which any path-comparison-based algorithm for betweenness must perform at least  $mn/2$  path weight comparisons.*

CONJECTURE : Computing betweenness of a single vertex is at least as hard as computing betweenness of all vertices.

We make the following conjecture for computing betweenness centrality in general graphs. If our conjecture is true, then the existing techniques for APSP provide lower bounds for computing betweenness.

CONJECTURE : Computing betweenness of all vertices is at least as hard as computing all-pairs shortest distances.

### 3 An Algebraic Method

We denote matrices by upper case letters and the elements of a matrix by the corresponding lower case letter. Recall that  $A$  is the adjacency matrix of the graph. Let  $\mathbf{0}_{n \times n}$  be an  $n \times n$  zero-matrix. Let  $\mathbf{I}_{n \times n}$  be an  $n \times n$  identity matrix. Let  $D$  be an  $n \times n$  matrix of distances, i.e.,  $d_{ij} = d(i, j)$ . Let  $D_l$  be a 0-1 matrix such that  $(d_l)_{ij} = 1$  iff  $d(i, j) = l$ . Let  $\Lambda$  be an  $n \times n$  matrix, where  $\lambda_{ij}$  is the number of shortest paths between  $i$  and  $j$ . Let  $\Delta$  be an  $n \times n$  matrix of dependencies, i.e.,  $\delta_{ij} = \delta_{i^*}(j)$ . Let  $\Delta_l$  be a matrix such that  $(\delta_l)_{ij}$  is non-zero and equal to  $\delta_{i^*}(j)$  iff  $d(i, j) = l$ . If  $X$  and  $Y$  are two matrices, we let  $X \text{ mult } Y$  ( $X \text{ div } Y$ ) be the matrix obtained by *element-wise* multiplication (division) of the matrices  $X$  and  $Y$ . We let  $X \cdot Y$  denote the product of the two matrices  $X$  and  $Y$ , i.e.,  $(X \cdot Y)_{ij} = \sum_k x_{ik} y_{kj}$ . We call the computation of the distance and the number of shortest paths (between all pairs) as the **forward pass**, since shortest paths are computed using

BFS/Dijkstra’s algorithm. The computation of dependencies is called the **backward pass**, since dependencies are computed in a bottom-up fashion. In other words, the matrices  $D$  and  $\Lambda$  are computed in the forward pass and the matrix  $\Delta$  is computed in the backward pass.

### 3.1 Unweighted Graphs

#### 3.1.1 Forward Pass

The lengths of all shortest paths can be computed using the following theorem of Seidel [30].

**Theorem 3.1.** [30] *All-pairs shortest distances for undirected unweighted graphs can be computed in time  $O(n^\omega \log(\text{Diam}(G)))$ .*

We compute the *number* of shortest paths ( $\lambda_{ij}$  for all  $i, j$ ) using the following algorithm :

---

#### ComputePathCount( $A$ )

Initialize  $Z$  to  $\mathbf{I}_{n \times n}$

Initialize  $\Lambda$  to  $\mathbf{I}_{n \times n}$

Initialize  $\Lambda_{prev}$  and  $\Lambda_{curr}$  to  $\mathbf{0}_{n \times n}$

for  $l \leftarrow 1$  to  $\text{Diam}(G)$

$Z \leftarrow Z \cdot A$

for  $i, j \leftarrow 1$  to  $n$

if  $(\lambda_{prev})_{ij} > 0$

$(\lambda_{curr})_{ij} \leftarrow 0$

else

$(\lambda_{curr})_{ij} \leftarrow z_{ij}$

$\Lambda \leftarrow \Lambda + \Lambda_{curr}$

$\Lambda_{prev} \leftarrow \Lambda_{curr}$

for  $i \leftarrow 1$  to  $n$

$\lambda_{ii} \leftarrow 1$

return  $\Lambda$

---

*Correctness* : Note that  $Z = A^l$  after  $l^{th}$  iteration of the main *for* loop. Let  $A^l = (a_{ij}^l)$ . It is easy to see that  $a_{ij}^l$  equals the number of paths (not necessarily shortest) from  $i$  to  $j$  of length exactly

$l$ . Note that the least  $l$  for which  $a_{ij}^l$  is non-zero, represents the number of *shortest* paths from  $i$  to  $j$ , of length exactly  $l$ . The first time we encounter a non-zero value of  $a_{ij}^l$ , we store the value in  $\Lambda_{curr}$  and eventually in  $\Lambda$ . Also, we make sure that these values are not overwritten in the future iterations. In the end we set all  $\lambda_{ii}$  to 1 by convention. Hence the above algorithm correctly computes the number of shortest paths, for all pairs, in an undirected unweighted graph. As a consequence we get the following lemma :

**Lemma 3.2.** *All-pairs shortest path counts for undirected unweighted graphs can be computed in time  $O(n^\omega \text{Diam}(G))$ .*

### 3.1.2 Backward Pass

**Lemma 3.3.** *If  $d(i, j) = \text{Diam}(G)$ , then  $\delta_{i*}(j) = \delta_{j*}(i) = 0$ . Hence  $\Delta_{\text{Diam}(G)} = \mathbf{0}_{n \times n}$ .*

**Lemma 3.4.** *For unweighted graphs, if  $l = \text{Diam}(G)$  then  $\Delta_{l-1} = (D_l \text{div } \Lambda) \cdot A$ .*

*Proof.* We have the following cases :

**Case I :**  $d(i, j) = l - 1$

$$\begin{aligned}
\sum_{k=1}^n \left( \frac{(d_l)_{ik}}{\lambda_{ik}} \right) \cdot a_{kj} &= \sum_{k: a_{kj}=1, (d_l)_{ik}=1} \left( \frac{(d_l)_{ik}}{\lambda_{ik}} \right) \cdot a_{kj} \\
&= \sum_{d(i,k)=l, a_{kj}=1} \left( \frac{(d_l)_{ik}}{\lambda_{ik}} \right) \cdot a_{kj} \\
&= \sum_{k: j \in P_i(k)} \left( \frac{(d_l)_{ik}}{\lambda_{ik}} \right) \cdot a_{kj} \\
&= \sum_{k: j \in P_i(k)} \left( \frac{1}{\lambda_{ik}} \right) \\
&= \sum_{k: j \in P_i(k)} \left( \frac{1}{\lambda_{ik}} \right) (1 + \delta_{i*}(k)) \\
&= \delta_{i*}(j)
\end{aligned}$$

Note that we have used the fact that, if  $d(i, k) = l = \text{Diam}(G)$  then  $\delta_{i*}(k) = 0$ .

**Case II :**  $d(i, j) < l - 1$

$$\begin{aligned}
\sum_{k=1}^n \left( \frac{(d_l)_{ik}}{\lambda_{ik}} \right) \cdot a_{kj} &= \sum_{k: a_{kj}=1, (d_l)_{ik}=1} \frac{(d_l)_{ik}}{\lambda_{ik}} \\
&= \sum_{k: a_{kj}=1, d(i,k)=l} \frac{(d_l)_{ik}}{\lambda_{ik}} \\
&= 0
\end{aligned}$$

Since if  $d(i, j) < l - 1$ ,  $\nexists k$  such that  $d(i, k) = l$  and  $a_{kj} = 1$ .

**Case III :**  $d(i, j) = l$

In this case, it is easy to see that  $\sum_{k=1}^n \left( \frac{(d_l)_{ik}}{\lambda_{ik}} \right) \cdot a_{kj} = 0$ .  $\square$

**Lemma 3.5.** *For unweighted graphs if  $l < \text{Diam}(G)$  then  $\Delta_{l-1} = ((D_l + \Delta_l) \text{div } (\Lambda)) \cdot A$ .*

*Proof.* This can be proved by induction using the previous lemma as the base case, and the argument is similar to the proof for unweighted trees. In addition we use the fact that shortest path trees have no cross edges (i.e., all the edges of BFS tree join vertices of levels that differ at most by one). Hence, the dependencies computed at distance  $l - 1$  uses only the dependencies at distance  $l$ .  $\square$

---

### ComputeDependency( $A, D, \Lambda$ )

Initialize  $\Delta$  to  $\mathbf{0}_{n \times n}$

Initialize  $\Delta_{\text{Diam}(G)}$  to  $\mathbf{0}_{n \times n}$

for  $l \leftarrow \text{Diam}(G)$  to 1

Construct a 0-1 matrix  $D_l$ , such that

$(d_l)_{ij} = 1$  iff  $d(i, j) = l$ .

$\Delta_{l-1} \leftarrow ((D_l + \Delta_l) \text{div } (\Lambda)) \cdot A$

$\Delta_{l-1} \leftarrow \mathbf{Mask}(\Delta_{l-1}, l - 1)$

$\Delta_{l-1} \leftarrow \Delta_{l-1} \text{ mult } \Lambda$

$\Delta \leftarrow \Delta + \Delta_{l-1}$

return  $\Delta$

### Mask( $X, l$ )

for all  $0 \leq i, j \leq n$

if  $d(i, j) \neq l$   
 $x_{ij} \leftarrow 0.$   
return  $X$

---

From the previous lemma, it is easy to see that the above algorithm runs in  $O(n^\omega \text{Diam}(G))$  using  $O(n^2)$  space. Once the dependencies are computed, the centrality of each node can be computed by adding the corresponding dependencies, in  $O(n^2)$  time.

**Theorem 3.6.** *The betweenness of all vertices of an undirected unweighted graph  $G$ , can be computed in time  $O(n^\omega \text{Diam}(G))$ .*

## 3.2 Weighted Graphs

### 3.2.1 Forward Pass

We make use of a well-known reduction from APSP to the computation of the *distance product* (also known as the *min-plus product*) of two  $n \times n$  matrices.

**Definition 3.7.** DISTANCE PRODUCTS: *Let  $X, Y$  be  $n \times n$  matrices. The distance product of  $X$  and  $Y$ , denoted  $X \star Y$ , is an  $n \times n$  matrix  $Z$  such that*

$$z_{ij} = \min_{k=1}^n \{x_{ik} + y_{kj}\}, \text{ for } 1 \leq i, j \leq n.$$

It is well-known that the distance product of two  $n \times n$  matrices, whose elements are taken from the set  $\{-M, \dots, 0, \dots, M\} \cup \{+\infty\}$ , can be computed in time  $O(Mn^\omega)$ . Combining the distance products with our observations for unweighted graphs we get the following theorem.

**Theorem 3.8.** *All-pairs shortest distances and number of shortest paths for undirected weighted graphs with integer weights taken from  $\{1, 2, \dots, M\}$  can be computed in time  $O(Mn^\omega \text{Diam}(G))$ .*

The lengths of all shortest paths can also be computed by the following theorem of Alon, Galil, Margalit [1].

**Theorem 3.9.** [1] *All-pairs shortest distances for undirected weighted graphs with integer weights taken from  $\{1, 2, \dots, M\}$  can be computed in time  $\tilde{O}(Mn^\omega)$ .*

### 3.2.2 Backward Pass

Let  $D, D_l, \Delta, \Delta_l, \Lambda$  be the matrices as defined earlier. Let  $A^*$  be a 0-1 matrix with  $a_{ij}^* = 1$  iff  $w(i, j) = d(i, j)$ . In other words,  $a_{ij}^* = 1$  iff the edge  $(i, j)$  participates in the shortest paths.

**Theorem 3.10. ComputeDependency** *correctly computes the dependencies in a weighted graph with integer weights taken from  $\{1, 2, \dots, M\}$  in time  $O(Mn^\omega \text{Diam}(G))$ .*

*Proof.* Follows from the correctness of the algorithm for unweighted graphs.  $\square$

## 4 A Randomized Parallel Algorithm

We assume a model of parallel computation called OR CRCW PRAM [3], in which multiple processors can simultaneously read and write to a shared memory. If multiple processors attempt to write multiple values to a single location, the value written is the bitwise OR of the values. The most elementary parallel SSSP algorithm is *parallel breadth-first search*, in which the nodes are visited level by level as the search progresses. Level 0 consists of the source. The problem with this approach is that the time required grows linearly with the number of levels traversed. To keep the time small Ullman and Yannakakis [33] use *k-limited search*.

The *size* of a path is the number of nodes in the path and the *minimum path-size* is the shortest distance measured in number of nodes traversed. A *k-limited shortest path* from  $s$  to  $t$  is a path from  $s$  to  $t$  that is no longer than any  $s$ -to- $t$  path of size at most  $k$ . To find *k-limited shortest paths* in unweighted graphs we can run  $k$  iterations of parallel BFS. We call this *k-limited breadth-first*

*search.* The *work* required by a parallel algorithm is defined to be the product of time and number of processors required; this corresponds to the time that would be required if the parallel processors were all simulated by a single processor. If the *work* of a parallel algorithm is equal to the time required by a sequential algorithm for the same problem, then the parallel algorithm is said to be *optimal*.

In the following sections, we present parallel algorithms for the forward and backward passes. Forward pass consists of computing the distance and the number of shortest paths (between all pairs). Backward pass involves computing the dependencies. Once the dependencies are known, to compute the betweenness value of a vertex  $v$ , we can simply compute the sum of all the dependencies for each vertex. This can be done in time  $O(n \log n)$  time using  $O(n)$  processors.

## 4.1 Forward Pass

### 4.1.1 Unweighted Graphs

Ullman and Yannakakis's algorithm [33] for parallel BFS, uses  $k$ -limited search using random sampling of *distinguished* vertices based on the following well known observation (see, e.g., Greene and Knuth [17]). Their algorithm uses about  $\sqrt{n} \log n$  distinguished nodes, and therefore needs to search forward for about  $\sqrt{n}$  distance from each distinguished node. Our algorithm for parallelizing the forward pass is based on their technique.

**Theorem 4.1.** [17] *Given a path of length  $k$  in a graph, a random sample of  $\frac{n \log n}{k}$  vertices will have at least one vertex belonging to the path with probability  $1 - \frac{1}{n^c}$ .*

**Theorem 4.2.** *With high probability, Algorithm 1 computes correctly the shortest paths from the source  $s$  to all the other nodes in  $V$ . The parallel global time  $O(\sqrt{n})$  using  $m \log n$  processors.*

*Proof.* Given any  $v \in V$ , let  $P_v$  be an arbitrary shortest path from  $s$  to  $v$ . From Theorem 5.1, with high probability, each subpath of  $P_v$  of size  $\sqrt{n}$  contains at least a node  $x \in S$ . Hence,  $P_v$  can be seen as a sequence of subpaths of size not larger than  $\sqrt{n}$ , whose extremal nodes belonging to  $S$  (except for the last node  $v$ ). Such subpaths are computed in the  $\sqrt{n}$ -limited search in Step 2. Thus, the shortest path from  $s$  to the last  $S$ -vertex  $x$  in  $P_v$  is correctly computed in Step 4 and the shortest path from the latter to node  $v$  is correctly computed in Step 2. The  $\sqrt{n}$ -limited search, in Step 2, can be performed in  $O(\sqrt{n})$  time using  $m \log n$  processors. The total work of Step 4 is  $O((\sqrt{n})^3 \log n)$  and can be done in  $O(\sqrt{n})$  using  $m \log n$  processors. Correctness of the number of shortest paths follows.  $\square$

Since we need the distances and number of shortest paths between *all* pairs of vertices, we can simply run the above algorithm for  $n$  times, once for each source vertex. This approach *duplicates* many computations. Since we choose  $\Theta(\sqrt{n} \log n)$  distinguished nodes, we can compute the shortest path distances from each of these distinguished nodes (treating them as source nodes), with a *single* run of **Algorithm 1**. The following theorem states that we need to run the algorithm for only  $O(\sqrt{n})$  times. This results in an *optimal* parallel algorithm (modulo log-factors) for the forward pass.

**Theorem 4.3.** *With high probability, Algorithm 1 is run only  $O(\sqrt{n})$  times to compute all-pairs shortest distances and number of shortest paths.*

*Proof.* Let us say, we run the **Algorithm 1** independently for  $k$  times. Each time the algorithm picks  $\sqrt{n} \log n$  vertices. Then the probability that a vertex  $v \in V$  is *not* picked in any of these iterations is given by

$$\begin{aligned} \Pr [v \text{ not picked}] &= \left(1 - \frac{\sqrt{n} \log n}{n}\right)^k \\ &< e^{-\frac{k \sqrt{n} \cdot \log n}{n}} \end{aligned}$$

Choosing  $k = c\sqrt{n}$ , for some constant  $c > 0$ , we get

$$\begin{aligned} \Pr[v \text{ not picked}] &< e^{-\frac{c\sqrt{n}\sqrt{n}\cdot\log n}{n}} \\ &= e^{-c\log n} \\ &= e^{-c'\ln n} \\ &= \frac{1}{n^{c'}} \end{aligned}$$

Hence the probability that a vertex  $v \in V$  is not picked in any of the  $O(\sqrt{n})$  iterations is very small, inverse polynomial in  $n$ .  $\square$

**Theorem 4.4.** *With high probability, we can compute the  $D$  and  $\Lambda$  matrices for an unweighted graph in  $O(n)$  time using  $O(m \log n)$  processors.*

---

**Algorithm 1 :**

*Input :* An undirected graph  $G(V, E)$ , a source  $s \in V$ .

*Output :*  $d(s, v)$  and  $\lambda_{sv}$  for all  $v \in V$ .

1. Choose uniformly at random a subset  $S$  of  $V$ , together with  $s$ ; the size of  $S$  must be  $\Theta(\sqrt{n} \log n)$ .
2. From any  $x \in S$  perform, in parallel, a  $\sqrt{n}$ -limited search, generating the shortest path  $P'_{x,v}$  from  $x$  to every node  $v \in V$ .
3. An auxiliary *weighted* graph  $H$  is computed on the vertex set  $S$ , where the weight of an edge is defined to be the length computed by the previous  $\sqrt{n}$ -limited search.
4. Compute the *all-pairs* shortest paths  $P_{x,y}$  in  $H$ , with no-limited search.
5. The shortest distance  $d(s, v) = |P_v|$ , from  $s$  to a node  $v \in V$ , is computed in the following way:

$$P_v \equiv P_{s, \min} \cup P_{\min, v}$$

where  $\min$  is a vertex in  $H$  for which:

$$|P_{s, \min}| + |P'_{\min, v}| = \min_{x \in H} \{|P_{s, x}| + |P'_{x, v}|\}$$

6. The number of shortest paths  $\lambda_{sv}$ , can be computed by counting the number of such *min* nodes.

---

## 4.1.2 Weighted Graphs

Ullman and Yannakakis's approach cannot be directly applied to weighted graphs, indeed there is no apparent way to perform efficiently the  $\sqrt{n}$ -limited search, especially when the weights are large. On the other hand, it is easy to verify that the remaining steps of **Algorithm 1** works also for weighted graphs, thus the crucial problem is to find a *weighted* version of the  $\sqrt{n}$ -limited search. A useful method for solving optimization problems which involve numerical inputs is to uniformly shrink all weights; but this, in itself, is not sufficient since the search is strongly based on the fact that weights are integers. Klein and Subramanian [21] proposed a  $\sqrt{n}$ -limited search for weighted graphs which uses the integer shrinking together with the well-established technique, due to Raghavan and Thompson [27], for rounding weights without changing their sums "too much". The key idea is that a non-integral value is rounded up or down according to a probability function which reflects how close the value is to the next higher integer and next lower one. By applying this approach to the basic techniques of Ullman and Yannakakis, Klein and Subramanian provided a randomized parallel algorithm for SSSP in weighted graphs. Their algorithm runs in  $O(\sqrt{n} \log^2 n \log M)$  time and using  $O(m)$  processors to compute an SSSP tree. We enhance their algorithm to compute *all-pairs* shortest paths (and number of shortest paths). The modifications needed are similar to those presented in the previous section. We mention our main theorem here.

**Theorem 4.5.** *With high probability, we can compute the  $D$  and  $\Lambda$  matrices for a weighted graph, with integer weights taken from the range*

$\{1, 2, \dots, M\}$ , in  $O(n \log^2 n \log M)$  time using  $O(m)$  processors.

## 4.2 Backward Pass

### 4.2.1 General Graphs

After the forward pass is performed, we may assume that the matrices  $D$  and  $\Lambda$  are available in the shared memory. The following algorithm computes the *betweenness centralities* (without actually computing the dependencies) in  $O(n^2)$  time using  $O(n)$  processors.

---

#### Algorithm 2 :

*Input* :  $D$  and  $\Lambda$  matrices.

*Output* : Betweenness centrality ( $BC(v)$ ) of all vertices.

Let  $n$  processors represent the vertices.

Each processor maintains a running centrality score  $BC(v)$ , initialized to zero

For each pair of vertices  $s, t \in V$ , processor  $v$  ( $v \neq s \neq t$ ) does the following :

if  $d(s, t) = d(s, v) + d(v, t)$

$$BC(v) += \frac{\lambda_{sv} \cdot \lambda_{vt}}{\lambda_{st}}$$

else

$$BC(v) += 0$$


---

### 4.2.2 Bounded Degree Graphs

In this section we present a faster parallel algorithm for backward pass in *bounded-degree* graphs. Backward pass involves computing the dependencies (i.e., computing the matrix  $\Delta$ ). Recall the following lemma.

**Lemma 3.3** : If  $d(i, j) = \text{Diam}(G)$ , then  $\delta_{i*}(j) = \delta_{j*}(i) = 0$ .

Brandes's theorem (*Theorem 1.1*) states that the dependencies of the *closer* vertices can be computed from the dependencies of the *farther* vertices. The following algorithm (**ComputeDependency**) uses this fact (and a small trick) to compute the dependencies in parallel. The *main idea* behind the algorithm is to compute

dependencies of pairs of vertices (taking a maximum of  $n/2$  pairs) which are at distance  $d$ . Distance  $d$  is decreased from  $n$  to 1.

---

#### ComputeDependency( $A, D, \Lambda$ )

For  $d \leftarrow n$  to 1

Let  $V_d = \{v \in V : \exists u \in V \text{ with } d(u, v) = d\}$

While  $|V_d| \neq 0$

Select a maximum of  $n/2$  pairs of vertices (with no two pairs having a common vertex) from  $V_d$  such that each pair is at a distance  $d$  from each other. Let  $V'_d$  be such a set.

$$V_d \leftarrow V_d \setminus V'_d$$

$$\Delta = \text{ParallelCompute}(A, D, \Lambda, V'_d)$$

return  $\Delta$

---

*Correctness* : **ParallelCompute** computes the dependencies of (at most  $n/2$  pairs of) vertices (such that each pair of vertices are at a distance of  $d$  from each other) in parallel. This can be done in  $O(\log k)$  time, since this involves computing sum of  $k$  values. When there are multiple vertices at distance  $d$  (from a vertex  $v$ ) the algorithm is repeated until all the pairs's dependencies are calculated. Note that there can be at most  $O(\text{maxdeg}(G))$  such nodes, where  $\text{maxdeg}(G)$  is the maximum degree of any vertex in the graph. Hence **ParallelCompute** takes  $O(\text{maxdeg}(G) \log k) = O(\log m)$  time (since we are interested in *bounded-degree* graphs). Since there are at most  $n$  different distances and  $O(n^2)$  pairs of dependencies to be computed, **ParallelCompute** is called at most  $O(n)$  times. The constant in  $O(n)$  depends on the distribution of ( $n$  possible) distances among the  $O(n^2)$  pairs of vertices. Note that this gives an *optimal* algorithm.

---

#### ParallelCompute( $A, D, \Lambda, V'_d$ )

Let  $m$  processors represent the edges.

For each pair  $u, v \in V'_d$  (such that  $d(u, v) = d$ ) do the following in *parallel*

◦ Let  $w_1, w_2, w_3, \dots, w_k$  be the vertices such that  $v \in P_u(w_i)$ .

- The processor representing edge  $(v, w_i)$  calculates  $\frac{1}{\lambda_{uw_i}}(1 + \delta_{u^*}(w_i))$ .

- The  $k$  processors (representing the edges  $(v, w_i)$ ) compute the sum  $\sum_{i=1}^k (1 + \delta_{u^*}(w_i))$ .

- This sum is multiplied by  $\lambda_{uv}$  and stored in the shared memory as  $\delta_{uv}$ .

- Compute  $\delta_{vu}$  similarly.

- If there are multiple vertices at distance  $d$  from  $v$  then repeat the algorithm **Parallel Compute** for the remaining pairs of vertices.

return  $\Delta$

**Theorem 4.6.** *The dependencies in an unweighted graph can be computed in  $O(n \log m)$  time using  $O(m)$  processors.*

For weighted graphs with integer weights taken from the range  $\{1, 2, \dots, M\}$ , the distances vary from  $nM$  to 1.

**Theorem 4.7.** *The dependencies in a weighted graph with integer weights taken from the range  $\{1, 2, \dots, M\}$ , can be computed in  $O(Mn \log m)$  time using  $O(m)$  processors.*

## 5 Open Problems

1. Is there an algorithm to compute (exactly or approximately) the betweenness of all (or even top  $k$ ) vertices in *sub-cubic* (or  $o(mn)$ ) time?
2. Since the networks of interest are huge and dynamic, it is expensive to recompute betweenness for every addition/deletion of edge. Is there a fully dynamic algorithm to maintain betweenness in  $O(n^2)$  amortized time per update (edge insertion or deletion), using only  $O(n^2)$  space. Here, it is crucial to observe that betweenness centrality of *all* vertices can be changed by deleting (hence

adding) a single edge to the graph. For example, let  $C_{4k+1}$  be a cycle on  $4k + 1$  vertices. The centrality of any vertex in  $C_{4k+1}$  is  $k^2$ . Removing an edge from  $C_{4k+1}$  results in a path  $P_{4k+1}$  on  $4k + 1$  vertices. Betweenness of vertices of  $P_{4k+1}$  are  $0, 4k - 1, 2(4k - 2), \dots, 4k^2, \dots, 2(4k - 2), 4k - 1, 0$ .

3. Betweenness centrality implicitly assumes that communications in the network use *shortest* paths. Shortest paths are sensitive to *local* changes (addition/deletion of edges). One possible way to address this issue is to consider  $\delta$ -stretch paths, instead of shortest paths [7]. A  $\delta$ -stretch path is a path from  $s$  to  $t$  of length  $\leq (1 + \delta)d(s, t)$ . What is the complexity of computing betweenness based on  $\delta$ -stretch paths?
4. Our conjectures mentioned in Section 2 are open.

## Acknowledgements

This project is funded by ARC (Algorithms and Randomness Center) of the College of Computing at Georgia Institute of Technology.

## References

- [1] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all-pairs shortest path problem. *J. Compu. Syst. Sci.*, 54:255–262, 1997.
- [2] J. M. Anthonisse. The rush in a directed graph. In *Technical Report BN 9/71, Stichting Mathematisch Centrum*, Amsterdam, 1971.
- [3] H. Bast, M. Dietzfelbinger, and T. Hagerup. A perfect parallel dictionary. In *17th Symposium on Mathematical Foundations of Computer Science*, 1992.
- [4] U. Brandes. A faster algorithm for betweenness centrality. *J. Mathematical Sociology*, 25(2):163177, 2001.

- [5] U. Brandes and C. Pich. Centrality estimation in large networks. To appear in Intl. Journal of Bifurcation and Chaos, Special Issue on Complex Networks' Structure and Dynamics, 2007.
- [6] N. Buckley and M. van Alstyne. Does email make white collar workers more productive? Technical report, University of Michigan, 2004.
- [7] T. Carpenter, G. Karakostas, and D. Shallcross. Practical issues and algorithms for analyzing terrorist networks. *Invited paper at WMC*, 2002.
- [8] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *In Proc. STOC*, 2007.
- [9] D. Csic, B. Kesic, and L. Jakomin. Research of the power in the supply chain. International Trade, Economics Working Paper Archive EconWPA, April 2000.
- [10] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Communications of the ACM*, 47(3):45–47, 2004.
- [11] S. J. Phillips D. R. Karger, D. Koller. Finding the hidden path : time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22:1199–1217, 1993.
- [12] A. del Sol, H. Fujihashi, and P. O'Meara. Topology of small-world networks of protein-protein complex structures. *Bioinformatics*, 21(8):1311–1315, 2005.
- [13] D. Eppstein and J. Wang. Fast approximation of centrality. *Journal of Graph Algorithms and Applications*, 8(1):39–45, 2004.
- [14] G. Erkan and D. R. Radev. Lexrank: Graph-based centrality as salience in text summarization. *Journal of Artificial Intelligence Research (JAIR)*, 22:457–479, 2004.
- [15] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, 5:49–60, 1976.
- [16] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.
- [17] D. H. Greene and D. E. Knuth. Mathematics for the analysis of algorithms. *Birkhauser, Boston*, 1982.
- [18] R. Guimerà, S. Mossa, A. Turttschi, and L.A.N. Amaral. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. 102(22):7794–7799, 2005.
- [19] P. Hage and F. Harary. Eccentricity and centrality in networks. *Social Networks*, 17:57–63, 1995.
- [20] H. Jeong, S.P. Mason, A.-L. Barabási, and Z.N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411:41–42, 2001.
- [21] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest-paths. *In Proc. of the 24th Annual ACM-STOC*, pages 750–758, 1992.
- [22] V.E. Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002.
- [23] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (ACM TKDD)*, 1(1), 2007.
- [24] F. Liljeros, C.R. Edling, L.A.N. Amaral, H.E. Stanley, and Y. Åberg. The web of human sexual contacts. *Nature*, 411:907–908, 2001.
- [25] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69, 026113, 2004.

- [26] J.W. Pinney, G.A. McConkey, and D.R. Westhead. Decomposition of biological networks using betweenness centrality. In *Proc. 9th Ann. Int'l Conf. on Research in Computational Molecular Biology (RECOMB 2005)*, Cambridge, MA, May 2005. Poster session.
- [27] P. Raghavan and C.D. Thompson. Provably good routing in graphs: regular arrays. In *Proc. of the 17th Annual ACM-STOC*, pages 79–87, 1985.
- [28] A. H. Rustam. Epidemic network and centrality. *Master Thesis, University of Oslo*, May 2006.
- [29] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31:581–603, 1966.
- [30] R. Seidel. On the all-pairs-shortest-path problem. In *Proc. of STOC*, 1992.
- [31] A. Shimbel. Structural parameters of communication networks. *Bulletin of Mathematical Biophysics*, 15:501–507, 1953.
- [32] A. Shoshan and U. Zwick. All-pairs shortest paths in undirected graphs with integer weights. *Proc. of 40th FOCS*, pages 605–614, 1999.
- [33] J. D. Ullman and M. Yannakakis. High probability parallel transitive closure algorithms. *SIAM J. of Computing*, 20:100–125, 1991.
- [34] U. Zwick. Exact and approximate distances in graphs - a survey. In *Proc. of 9th ESA*, pages 33–48, 2001.