A Fast Algorithm for Streaming Betweenness Centrality

Oded Green, Robert McColl, David A. Bader
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA 30332

***Abstract*—Analysis of social networks is challenging due to the rapid changes of its members and their relationships. For many cases it impractical to recompute the metric of interest, therefore, streaming algorithms are used to reduce the total runtime following modifications to the graph. Centrality is often used for determining the relative importance of a vertex or edge in a graph. The vertex Betweenness Centrality is the fraction of shortest paths going through a vertex among all shortest paths in the graph. Vertices with a high betweenness centrality are usually key players in a social network or a bottleneck in a communication network. Evaluating the betweenness centrality for a graph $G = (V, E)$ is computationally demanding and the best known algorithm for unweighted graphs has an upper bound time complexity of $O(V^2 + VE)$. Consequently, it is desirable to find a way to avoid a full re-computation of betweenness centrality when a new edge is inserted into the graph. In this work, we give a novel algorithm that reduces computation for the insertion of an edge into the graph. This is the first algorithm for the computation of betweenness centrality in a streaming graph. While the upper bound time complexity of the new algorithm is the same as the upper bound for the static graph algorithm, we show significant speedups for both synthetic and real graphs. For synthetic graphs the speedup varies depending on the type of graph and the graph size. For synthetic graphs with 16384 vertices the average speedup is between $100X - 400X$. For five different real world collaboration networks the average speedup per graph is in range of $36X - 148X$.**

***Index Terms*—graph algorithms; social networks;**

I. Introduction

Betweenness centrality is computed for graphs $G = (V, E)$ where V represents the set of vertices and E represents the set of links between the vertices. The graph can be directed or undirected and weighted or unweighted.

A path between source vertex $s \in V$ and the destination vertex $t \in V$ is defined as the sequence of vertices $s, v_1, v_2, .., v_k, t$ such that $(v_i, v_{i+1}) \in E$ for the entire sequence. The length of a path is the sum of the weights of all the edges in the path. For an unweighted graph, the length of the path is the number of edges in the sequence. The shortest path between two vertices, also known as the geodesic, is the sequence of vertices that has the smallest summed weight. It is worth noting that there can be more than one shortest path connecting any pair of vertices. This was formalized by Freeman [16]. In his work, Freeman suggests comparing the number of shortest paths going through a vertex v with the total number of the shortest paths (including those that do not go through v).

In this work we show how to compute betweenness centrality for unweighted streaming graphs. If an algorithm supports both edge insertion (incremental) and edge deletion (decremental) then the algorithm is fully dynamic. If the algorithm supports one of these operations, it is partially dynamic. As the algorithm that is presented in this paper supports only insertions, it is an incremental algorithm. To the best of the authors' knowledge this is the first algorithm for incremental streaming betweenness centrality.

Centrality is used for finding important vertices/edges in graphs. In social networks the vertices refer to people/actors and the edges refer to relationships, where the relationship is dependent on the type of social network. In a communication network, the vertices might be servers and the edges might be physical connections between the servers. For email networks, the vertices will be the senders/receivers and the edges refer to emails sent between the sender and receiver.

Related Work

Betweenness centrality is applicable to many fields. Applications that use betweenness centrality as a building block include finding communities within a graph representing information flow [22], detecting communities in social networks [17], analyzing brain network [20], and deploying detection devices in communication networks [8].

In [5], Brandes shows a way to compute betweenness centrality using a dependency accumulation technique rather than doing a pair-wise summation. This algorithm is considerably faster than the pair-wise summation. In Section II we expand on Brandes's approach for computing betweenness centrality as it is crucial for understanding our approach to computing streaming betweenness centrality.

Madduri *et al.* [19] present the first parallel algorithm for computing betweenness centrality. This algorithm uses a two level hierarchy of parallelism to achieve fine-grain parallelism. Edmonds et al [12] give a distributed algorithm for betweenness centrality. Tan et al show several optimization strategies for computing betweenness centrality on the IBM Cyclops64 in [21].

In Bader [3], the authors suggest reducing the complexity requirements of betweenness by computing an approximation. This is done by selecting a subset of vertices and computing

betweenness centrality for these vertices alone. In their paper, the authors show that the approximation can give good results for artificial networks. In [19], [11] execution times are presented for the computation of approximate betweenness centrality for graphs with an edge count of half a billion.

In Buluç and Gilbert [7] the authors show a framework, Combinatorial BLAS, for computing betweenness centrality using algebraic computation. Using Combinatorial BLAS, they show that the computation of betweenness centrality is scalable and can be distributed to multiple cores.

Betweenness Centrality

We denote the number of shortest paths between two vertices s and t using $\sigma_{s,t}$ and the number of shortest paths between two vertices s and t that go through v by $\sigma_{s,t}(v)$. It follows that betweenness centrality is computed as follows [16]:

$$C_B(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}. \quad (1)$$

Finding the shortest path from a single vertex (source) to all the remaining vertices is known as the Single Source Shortest Path (SSSP) problem. Finding the shortest paths from all vertices to all vertices in the graph is known as the All Pairs Shortest Path (APSP) problem. APSP can be solved by running SSSP from every vertex in the graph. The complexity of computing APSP using the Floyd-Warshall algorithm [15] [23] is $O(V^3)$. For a more detailed discussion on SSSP and APSP the reader is referred to [10].

The remainder of the paper is organized as follows: In Section II we show Brandes's algorithm with an explanation of the key stages and focus on the dependency accumulation technique. In Section III the new streaming algorithm is presented. We prove that the new algorithm gives the same results as would a full re-computation and show that the algorithm is deterministic. In Section IV we show empirical speedups of the new algorithm versus doing a full recompute on both synthetic and real networks. In Section V a brief summary of the work will be given.

II. Faster Betweenness Centrality

In [5], Brandes presents a fast algorithm for computing betweenness centrality based on a dependency accumulation technique which accesses the vertices in the reverse order of the BFS (Breadth First Search) traversal. Brandes's [5] algorithm for faster betweenness centrality is key for understanding the work that will be presented in the sections ahead. Therefore, we give a detailed explanation (without the formal proofs) of his work. It is worth noting that the dependency accumulation approach is faster than previous approaches that required summing up all of the pair-wise dependencies.

The algorithm for computing betweenness centrality presented in [5] is made up of four stages, as shown in Algorithm 1. The first two stages, Stage 0 and Stage 1, are data structure initialization stages, where Stage 0 is a global initialization stage and Stage 1 is 'local' initialization that is completed once for each vertex in the graph.

Algorithm 1: The Betweenness centrality algorithm as suggested in [5]. The pseudo-code is divided into 4 stages.

```
Stage 0 - global initialization
C_B[r] ← 0, r ∈ V ;
for r ∈ V do
    Stage 1 - local initialization
    S ← empty stack; Q ← empty queue;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V; σ[r] ← 1;
    d[t] ← ∞, t ∈ V; d[r] ← ∞;
    enqueue r → Q;
    Stage 2 - BFS traversal
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        for all neighbor w of v do
            // w found for the first time
            if d[w] = ∞ then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
    Stage 3 - dependency accumulation
    δ[v] ← 0, v ∈ V;
    while S not empty do
        pop w ← S;
        for all v ∈ P[w] do
            δ[v] ← δ[v] + (σ[v]/σ[w])(1 + δ[w]);
        if w ≠ r then
            C_B[w] ← C_B[w] + δ[w];
```

In Stage 0 the output of the algorithm, the betweenness centrality score of each vertex is initialized to zero. Stages 1, 2, and 3 are executed for each vertex in the graph. Each vertex is considered a root an iteration over all vertices.

In Stage 1, the data structures that will be used in Stages 2 and 3 are initialized. This includes a stack, queue, and three additional arrays. The first array, σ, counts the number of shortest paths from each vertex to the root of the current shortest path tree, r. The second array, d, measures the distance of each vertex from the root. As the graph is unweighted, this is the minimum number of edges between the vertex and the root. We refer to this distance as the level of the vertex in the BFS tree. Initially the distances of all vertices from the root are set to ∞. The third array, P, is an array of linked lists. Each vertex v has a linked list $P[v]$, that contains all the vertices that precede v in the BFS traversal. These are the parent vertices of v in the previous level.

Stage 2 and Stage 3 are the key components of the betweenness centrality computation. Stage 2 is a BFS traversal from

a given root that finds the shortest path to all other vertices. In this stage, each element is placed in a queue when it is found. It is later placed in the stack when it is dequeued from the queue. [1] As part of the BFS traversal the distance from the root vertex, s, to each vertex is also computed. For each vertex, v, found in the BFS traversal there is a list of parental vertices that are all one hop closer to the root. Thus, all of v's shortest paths go through its parents and these are accumulated in $\sigma[v]$.

In Eq. (1), the following two notations are seen: $\sigma_{st}(v)$ and σ_{st}. The latter (denominator) refers to all the shortest path between s and t. The first (numerator) refers to all the shortest paths between s and t that go through vertex v. If there are no paths between s and t that go through v then $\sigma_{st}(v) = 0$. [2] By setting s to be a specific vertex (i.e. the root of the tree) it is possible to compute both numerator and denominator $\sigma_t(v)$ and σ_t using the BFS traversal for each root vertex s.

Stage 3 computes betweenness centrality using the dependency accumulation technique of Brandes [5]. The pair-dependency for a pair of vertices s, t is defined as follows:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}. \tag{2}$$

Using Eq. (2) with Eq. (1) changes the computation of betweenness centrality based on the pair-dependency:

$$C_B(v) = \sum_{s \neq t \neq v} \delta_{st}. \tag{3}$$

In [5] the following relationship is shown and proven:

$$\delta_s(v) = \sum_{\{w|v \in P_s(w)\}} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)). \tag{4}$$

The immediate outcome of this is that it is no longer necessary to sum all the pair-dependencies as they follow a recursive relation. In addition to this, it is possible to compute each of the $\delta_s(w)$, by computing the shortest path from the root , s, to the rest of the graph using a single source shortest path algorithm.

Complexity Analysis

The memory requirements for the stack, queue and the arrays σ and d are $O(V)$ as the sizes of these data structures are bound by the number of vertices in the graph V. The memory needed by the array of linked lists is bound by the number of edges in the graph $O(E)$ as the maximum number of parents a vertex has is bound by the number of edges it has. The sum of all the parents is bound by the total number of edges in the graph.

As each BFS traversal is computed independently, only a single copy of these data structures needs to be maintained, which is $O(V + E)$. The memory required by the array C_B is also bound by the number of vertices V. Therefore, the total memory requirement of this algorithm is $O(V + E)$.

The time complexity of BFS is $O(V + E)$. The time complexity of the dependency accumulation is also $O(V + E)$ as the maximal number of steps is bound by the number of parents, $O(E)$, and the vertices accessed, $O(V)$. As this computation is computed once for each vertex, the time complexity is $O(V^2 + VE)$. Given that in many cases $E > V$, this is simplified to $O(VE)$.

III. Streaming Betweenness Centrality

In this section we will present a novel algorithm for computing betweenness centrality in streaming graphs. Streaming graphs are graphs into which new edges are inserted over time. We show that it is possible to avoid a full re-computation by maintaining some additional data structures. We show that the algorithm does only minimal re-computation. We show the correctness of the algorithm and show that the algorithm is exact and not an approximation.

Additional data structures will be used to store previously computed values and will allow avoiding redundant computation. These structures will be explained further in this section. Our initialization stages are different from Algorithm 1 as data is maintained between iterations of the insertion rather than thrown away with the completion of the computation as is done in [5]. Following the presentation of the algorithm and the data structures, a deeper complexity analysis of time and space requirements of the algorithm and the data structure will be given. In this work, we focused on unweighted graphs. For simplicity, our proofs will be aimed for at undirected graphs; however, they can be augmented for directed graphs.

A. BFS Tree Data Structure

A BFS tree data structure is maintained for each of vertex in the graph. A BFS tree is the tree created following a BFS traversal from a given root. As these are unweighted graphs, it is possible to maintain the distance of each vertex from the root using an array of size $|V|$. Consequently, at any given time, it is possible to query the level of any vertex in any BFS tree in $O(1)$ time.

While maintaining this structure does indeed increases the space complexity, it will reduce the practical computation requirements and give significant speedups.

B. Edge Insertion

Given a new edge $e = (u, v)$ in the graph, $G = (V, E \cup \{e\})$, the relative position of the edge will be checked in the different BFS trees. For each of the BFS trees denoted T_s, before e connects u and v, one of the following scenarios occurs: [3]

1) $|d_s(u) - d_s(v)| = 0$ - both vertices are in the same level of the tree prior to the addition. The new edge does not create any shorter paths, meaning that in T_s there will

[1] For an array based implementation of the queue and the stack, it is necessary to maintain only one of these data structures, as the order in which the vertices are placed in the queue is the same as that of the stack. As the queue (Stage 2) and stack (Stage 3), are not accessed for computational purposes it is safe to implement this with one array and maintain additional pointers.

[2] If there are no shortest paths to between s and t, $\sigma_{st} = 0$

[3] For simplicity and without the loss of generality, assume that u is closer to the root than v.

Figure 1. Insertion of edge $e = (u, v)$ connects two vertices that are on the same level in the BFS tree of root s.

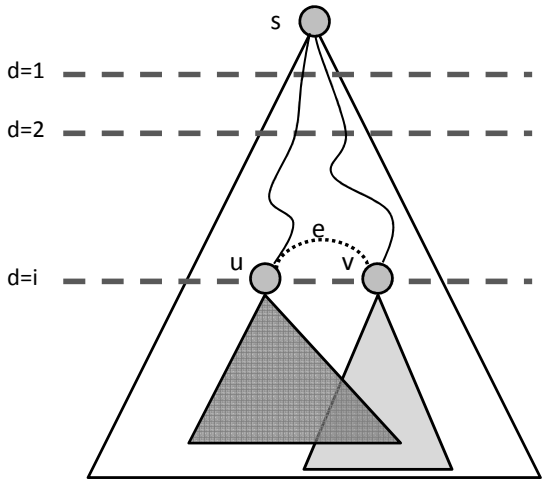
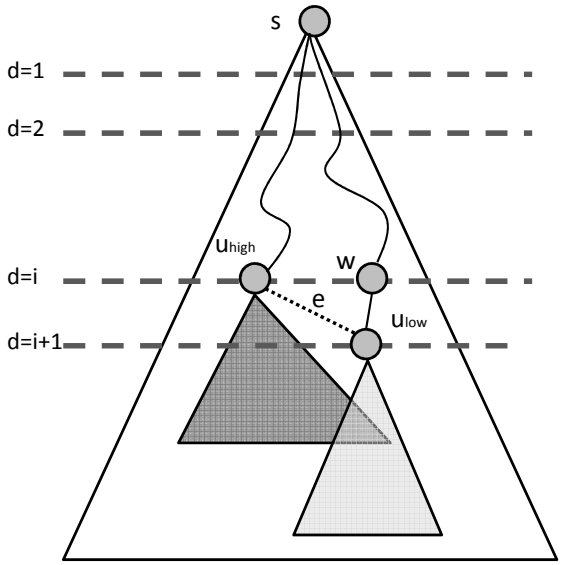


Figure 2. Insertion of edge $e = (u, v)$ connects two vertices that are in adjacent levels in BFS tree of root s. The new edge does not cause any vertex to change its position in the given BFS tree.



be no updates of betweenness centrality. This is denoted in Fig. 1.

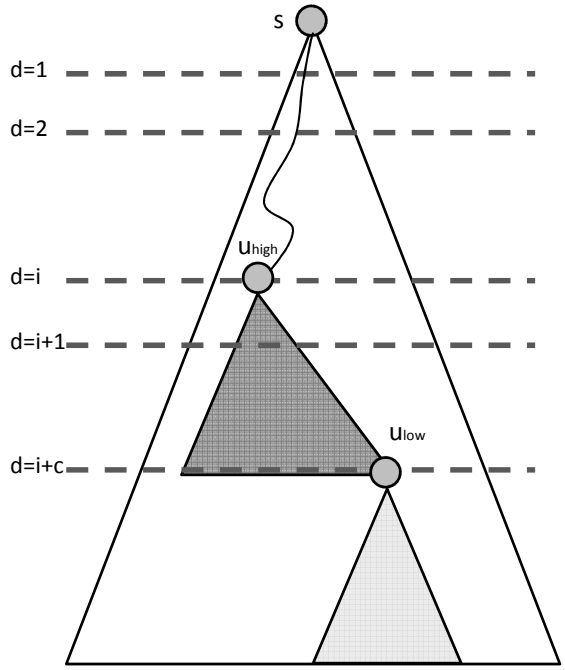
2) $|d_s(u) - d_s(v)| = 1$ - the vertices are in adjacent levels prior to the addition. $d_s(v) = d_s(u) + 1$. This is denoted in Fig. 2.
3) $|d_s(u) - d_s(v)| \geq 2$ - the vertices are not in adjacent levels prior to the addition. $d_s(v) = d_s(u) + \beta$, $\beta \geq 2$. This is denoted in Fig. 3.
4) $(|d_s(u) - d_s(v)| = \infty) \wedge (d_s(v) < |V| \vee d_s(u) < |V|)$ - the vertices do not have a path to each other prior to the addition of the edge. For undirected graphs this means that two components are about to be connected. This is denoted in Fig. 4.

These scenarios will be explained in the following sub sections.

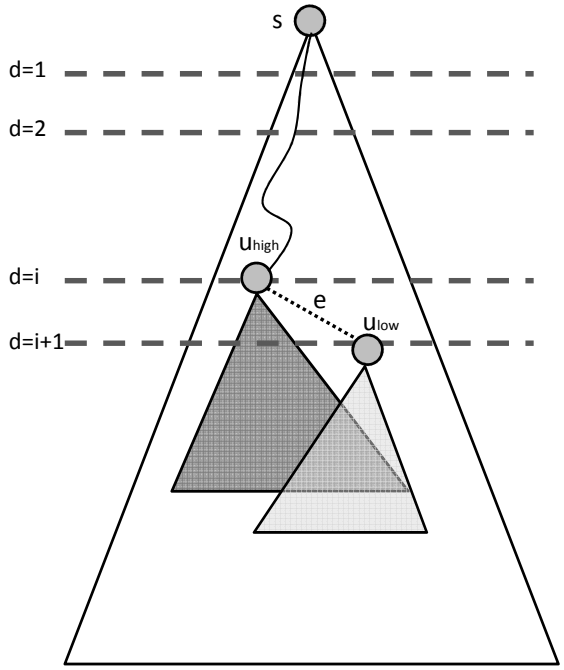
C. Same Level Insertion

In this subsection we show that the insertion of an edge between vertices in the same level of a give BFS tree, as depicted in Fig. 1, does not require an any additional computation.

Figure 3. Insertion of edge $e = (u, v)$ connects two vertices that are not adjacent to each other in the BFS tree of root s. In the simplest case only one vertex is moved (pulled up), v. For other scenarios an entire subtree moves as can be seen in (b).

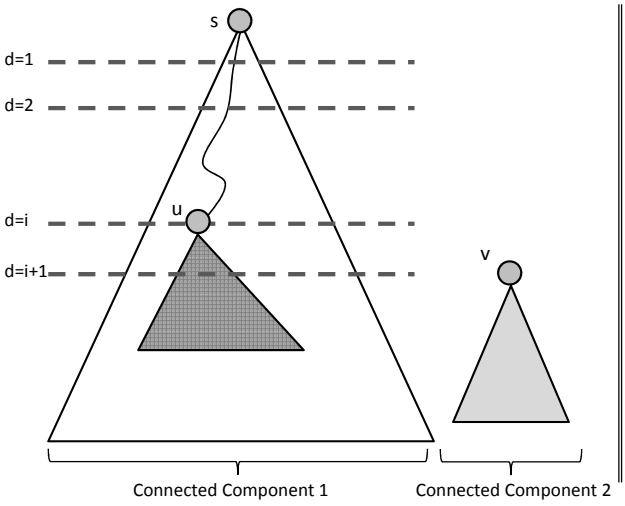


(a) Before edge insertion.

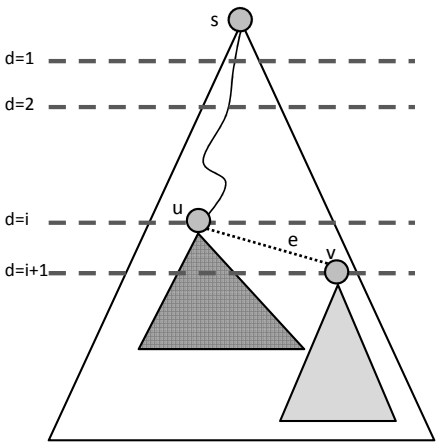


(b) After edge insertion. v has moved closer to the root of the tree. Consequently, additional vertices might be pulled up.

Figure 4. Insertion of the edge $e = (u, v)$ connects two connected components. The BFS tree of s is connected to vertex u and is not connected to vertex v.



(a) Before edge insertion.



(b) After edge insertion. Note the all the shortest paths between vertices in the two connected components go through e.

Lemma 1. *Given an edge $e = (u, v)$ such that $d_s(u) = d_s(v)$, no shortest paths go through e.*

Proof 1. *Assume by contradiction that for some vertex w there is a shortest path between s and w that goes through e. This path is denoted by vertices $p_1, p_2, ..., p_u, p_v, ..., p_w$. Obviously, v has a path to s as well, this path is denoted by $\hat{p_1}, \hat{p_2}, ..., p_v$. By creating an alternate path $\hat{p_1}, \hat{p_2}, ..., p_v, ..., p_w$ we have created a shorter path in contradiction with the assumption.*

Lemma 2. *The BFS structure is maintained and betweenness centrality is updated correctly when a new edge connects vertices in the same level.*

Proof 2. *Following Lemma 1, no new shortest paths are created; therefore, the BFS structure is maintained and there is no change in the betweenness centrality in the given BFS tree.*

Consequently, for edges that connect vertices that are in the same level of the BFS structure, no computation needs to be done.

D. Adjacent Level Insertion

In this subsection we present the algorithm for inserting a new edge between vertices that are in adjacent levels of a given tree with root s, as is depicted in Fig. 2. We denote $u_{high} = u$ and $u_{low} = v$ for this scenario. The BFS tree of s does not change due to the insertion. Prior to the insertion $d(u_{low}) = d(u_{high})+1$. This is still correct after the insertion. While new shorter paths have been created, the distance for all the vertices in the tree stay the same. However, the number of shortest paths going between the root and some of the vertices will change.

The pseudo-code for the new algorithm can be found in Algorithm 2. The justification for the pseudo code made will be presented in the following Lemmas.

Lemma 3. *Given vertex u_{low}, the only vertices that will have new shortest paths from the root, s, are the vertices found in the BFS subtree starting at u_{low} in s's BFS tree. The BFS traversal starting at u_{low} can only move down s's BFS tree.*

Definition 1. *$\hat{\sigma}_s(v)$ is the new number of shortest paths to v.*

In Stage 1 of Algorithm 2 $\hat{\sigma}_s(v) \leftarrow \sigma_s(v)$. After Stage 1, $\hat{\sigma}_s(v)$ is updated if there are new paths, otherwise it remains unchanged. The number of new paths will be maintained in the array dP, where $dP[v]$ is the number of new shortest paths to v.

Definition 2. *$\hat{\delta}_s(v)$ is the new accumulative sum for vertex v.*

In the beginning of Stage 3, $\hat{\delta}_s(v)$ is initialized to zero for all vertices.

Proof 3. *Assume by contradiction that some vertex w has a shortest path to the root, s, through u_{low} and that w is not found in a BFS traversal starting at v. Because w has a shortest path to the root via u_{low} it has some ancestral*

Algorithm 2: Insertion of a new edge in a specific BFS tree where the vertices are in adjacent levels prior to the insertion.

Stage 1 - local initilization
$Q_{BFS} \leftarrow$ empty queue;
for $level \leftarrow 1$ *to* V **do**
  $Q[level] \leftarrow$ empty queue;
$dP[v] \leftarrow 0, v \in \forall V$;
$t[v] \leftarrow$ Not-Touched , $v \in \forall V$;
$\hat{\sigma}[v] \leftarrow \sigma[v], v \in \forall V$;
enqueue $u_{low} \rightarrow Q[d[u_{low}]]$;
enqueue $u_{low} \rightarrow Q_{BFS}$;
$t[u_{low}] \leftarrow$ Down;
$dP[u_{low}] \leftarrow \sigma[u_{high}]$;
$\hat{\sigma}[u_{low}] \leftarrow \hat{\sigma}[u_{low}] + dP[u_{low}]$;
Stage 2 - BFS traversal starting at u_{low}
while Q *not empty* **do**
  dequeue $v \leftarrow Q$;
  for *all neighbor* w *of* v **do**
    if $d[w] = (d[v] + 1)$ **then**
      if $t[w] =$ *Not-Touched* **then**
        enqueue $w \rightarrow Q_{BFS}$;
        enqueue $w \rightarrow Q[d[w]]$;
        $t[w] \leftarrow$ Down;
        $d[w] \leftarrow d[v] + 1$;
        $dP[w] \leftarrow dP[v]$;
      else
        $dP[w] \leftarrow dP[w] + dP[v]$;
      $\hat{\sigma}[w] \leftarrow \hat{\sigma}[w] + dP[v]$;
Stage 3 - modified dependency accumulation
$\hat{\delta[v]} \leftarrow 0, v \in \forall V$; $level \leftarrow V$;
while *level>0* **do**
  while $Q[level]$ *not empty* **do**
    dequeue $w \leftarrow Q[level]$;
    for *all* $v \in P[w]$ **do**
      if $t[v] =$*Not-Touched* **then**
        enqueue $v \rightarrow Q[level - 1]$;
        $t[v] \leftarrow$ Up;
        $\hat{\delta}[v] \leftarrow \delta[v]$;
      $\hat{\delta}[v] \leftarrow \hat{\delta}[v] + \frac{\hat{\sigma}[v]}{\hat{\sigma}[w]}(1 + \hat{\delta}[w])$;
      if $t[v] =$ *Up* $\wedge (v \neq u_{high} \vee w \neq u_{low})$ **then**
        $\hat{\delta}[v] \leftarrow \hat{\delta}[v] - \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$;
      if $w \neq r$ **then**
        $C_B[w] \leftarrow C_B[w] + \hat{\delta}[w] - \delta[w]$;
  $level \leftarrow level - 1$;
$\sigma[v] \leftarrow \hat{\sigma}[v], v \in \forall V$;
for $v \in V$ **do**
  if $t[v] \neq$ *Not-Touched* **then**
    $\delta[v] \leftarrow \hat{\delta}[v], v \in \forall V$

path to u_{low}. However, this path will be found during the BFS traversal in contradiction to the assumption.

Corollary 1. *If the number of shortest paths from the root to u_{low} has changed, the vertices that are affected from this change are those in the BFS subtree beginning at u_{low}. All vertices above u_{low} are not affected as they don't have any shortest paths to s via u_{low}.*

Lemma 4. *Given the newly inserted edge and that $d[u_{low}] = d[u_{high}] + 1$ prior to the insertion, the only vertices that will have a change in the number of shortest paths to the root are those found in the BFS subtree starting at v.*

Proof 4. *The insertion of the edge does not add any shortest paths from the root to vertex u_{high}. However, u_{low} has new paths to the root through u_{high}. Following Lemma 3 and Corollary 1 it is clear that the only vertices that need to be updated are those in the BFS traversal starting at u_{low}.*

Corollary 2. *If the number of shortest paths to the root has changed for a vertex w then for all $v \in P[w], \delta[v]$ of vertex v needs to be updated. We denote these changes using $\hat{\delta}[v]$*

The immediate result of Corollary 2 is that any vertex that is on an ancestral path from a vertex that has had a change in its σ value will also have a change in its δ value. It is apparent that there are vertices that are not in discovered in the BFS traversal whose δ values need to be updated as one of their children had an update in either its δ or σ value. These vertices are found in the dependency accumulation using the parent lists.

The stack in Algorithm 1 ensures a vertex is not accessed until all vertices in the level below it have been accessed. We note, that unlike the algorithm by Brandes which used a stack for the dependency accumulation, we maintain a queue for each level. This has the following benefits: 1) Allows enqueueing newly discovered vertices (by way of the parent lists) to the adjacent queue in the inverse traversal. This cannot be done using the stack. 2) Ensures that all vertices in a level are accessed before moving on to the next level as is required.

Computation of $\hat{\delta}[v]$ is based on the computation of $\delta[v]$ with one minor modification, which will be explained briefly. This modification is that $\hat{\sigma}$ is used instead of σ. As $\hat{\delta}[v]$ contains the new and correct value of the dependency accumulation, $\delta[v]$ is no longer needed and should be removed from the centrality value. Thus:

$$C_B[v] \leftarrow C_B[v] + \hat{\delta}[v] - \delta[v]. \tag{5}$$

The difference in the computation of $\hat{\delta}[v]$ versus $\delta[v]$ is for the vertices that are found during the traversal up the tree. For these vertices only partial recomputation of the dependency accumulation might be needed as some of these vertices might have adjacent (one level below) vertices that have not been impacted by the insertion. For such vertices, $\hat{\delta}[v] \leftarrow \delta[v]$ is set initially. Within the value of $\hat{\delta}[v]$ are all the dependency accumulations made due to vertices in the adjacent level with v as a parent. These values need to be removed based on the previous values of σ and δ as can be seen in:

$$\hat{\delta}[v] \leftarrow \hat{\delta}[v] - \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w]). \tag{6}$$

All of the observations have been placed in the pseudo-code of Algorithm 2.

Lemma 5. *The BFS structure is maintained and betweenness centrality is updated correctly when a newly inserted edge connects vertices in adjacent levels.*

Proof 5. *In Lemmas 3 and 4 we show that the shortest path count is maintained. As there are no vertices that move in the BFS tree following the insertion, the BFS structure is maintained. Based on Eq. (5) and Lemma 2 the betweeness centrality metric is update correctly.*

E. Non-Adjacent Level Insertion

This subsection presents the modifications needed for updating the BFS tree for the insertion of a new edge. Similar to the last section, we denote $u_{high} = u$ and $u_{low} = v$. In this subsection we show how to make updates to the BFS when the inserted edge connects vertices in non adjacent levels. An example of such a tree can be seen in Fig. 3. Fig. 3 (a) shows the BFS tree prior to the insertion. Fig. 3 (b) shows the BFS tree after the insertion.

As can be seen, the BFS tree changes (at least in one place) as the vertex u_{low} is pulled-up the tree due to the distance between the vertices prior to the insertion. Additional vertices might be pulled-up as well, unlike in the adjacent level scenario.

Below we sketch the necessary steps needed for updating the BFS tree, followed by an explanation of how to update betweenness centrality. This subsection will be less formal than the previous one.

Following the insertion of the new edge, $e = (u_{high}, u_{low})$, we know for a fact that the vertex u_{low} will move up the tree and will be one level below u_{high}. As a consequence of this pull-up, additional vertices might be pulled-up as well. For all neighbors of u_{low} we will check if a new shortest path has been creating due to the pulling up of u_{low}.

For immediate neighbors of u_{low}, there are two obvious options: they will be moved up, or they will stay as they are. For both of these scenarios, they will be placed in a BFS-like queue.

After the pull-up, all neighbors of vertices in the queue need to be tested. This is an immediate consequence of the fact that there are new shortest paths to some of the vertices. For all vertices in the queue, except for u_{low}, there is a third scenario: the neighboring vertex will stay in its place and will not be placed in the queue as it is not affected from other pull-ups.

When the BFS-like stage has been completed, the dependency accumulation begins. The difference between dependency accumulation for this scenario and the Adjacent Level scenario is that for some vertices that have stayed in their level in the BFS tree, the number of neighbors they have in the following level has been reduced. Using Eq. (6) fixes this.

F. Insertion Connects Two Components

The newly inserted edge (u, v) connects two different connected components $C_1$and C_2, see Fig. 4. Assume that $u \in C_1$and $v \in C_2$. As the new edge connects two different connected components, it is safe to state that there are no additional edges between any vertices (c_1, c_2) such that $c_1 \in C_1$ and $c_2 \in C_2$.

Without the loss of generality and for simplicity, consider all the BFS trees of the vertices in C_1. It will become apparent that the same explanations holds for all the vertices of C_2 as well.

Given root $s \in C_1$, no new shortest paths to other vertices in C_1 are created following the insertion of the edge. It is, therefore, not necessary to begin the BFS traversal at s as the BFS tree to all the vertices in C_1 will remain unchanged. Instead, it is possible to start the BFS traversal from vertex v. Vertex v will be initialized in the following manner : $\sigma[v] \leftarrow \sigma[u]$, as all the paths between s and v go through u. In subsection III-D, we show that the scenario explained here is an instance of the adjacent level scenario.

Following the completion of the BFS traversal, the dependency accumulation is computed all the way back to v.

At this point $\delta[u] \leftarrow \delta[u] + \delta[v]$ is modified as there are new paths going through u. As $\delta[u]$ has been updated, it is necessary to update all of u's parents using the dependency accumulation concept. This too is explained in subsection III-D.

G. Complexity Analysis

In this subsection we discuss both the work and storage complexity of the new algorithm. For each BFS tree, one of the four scenarios occurs upon insertion. The upper bound complexity for connecting components, adjacent level insertion, and non-adjacent level insertion is similar to the one given by Brandes [5], $O(V + E)$. This includes both the BFS traversal and the dependency accumulation which are similar to the static graph version.

As there are V vertices, the upper bound on the complexity of the insertion is $O(V^2 + VE)$ similar to the one given by Brandes. While the upper bounds of both algorithms is the same, we will see in the next section that the new algorithm offers a substantial speedup in practice.

As for the storage complexity, as it is necessary to maintain the BFS trees, a total of $O(V + E)$ memory is needed for each tree. This includes the distance from the root, number of shortest paths to the root and the parent lists for each vertex. In addition to this, $O(V + E)$ is used in the update process (the updated data is no longer needed upon completion of the reverse accumulation). The upper bound on the memory is $O(V^2 + VE)$.

In summary, we have shown an algorithm with a work complexity of $O(V^2+VE)$ and storage complexity of $O(V^2+VE)$.

IV. Results

In this section we show speedups of the new streaming algorithm against the static algorithm. The algorithm is tested using three types of graph: 1) Erdös-Rènyi [13][14] random graphs, 2) Recursive Matrix (R-MAT) [9] random graphs, and 3) real social networks taken from [18], [1]. For the simulations we ran on an Intel i7-2600K quad core systems with 16GB of memory. The cores' clock frequency is 3.4GHz. The simulations use a single core as the algorithms are sequential. The L2 and L3 caches are 256KB and 8MB respectively.

The Erdös-Rènyi (ER) model uses a uniform distribution for selecting the edges that will appear in the graph. All edges have the same probability of existing in the graph

R-MAT is a graph generator used to create synthetic scale-free graphs that follow properties found in real-world networks. For simplicity, we present R-MAT using an adjacency matrix. Unlike the ER generator, edges in R-MAT do not have uniform probability of being created. Initially, the adjacency matrix is empty, and edges are added one at a time. For each newly inserted edge, the adjacency matrix is divided into equal-size quadrants where each has a different probability of being selected. One of the quadrants is selected using a random number generator. This quadrant is recursively subdivided into smaller equal-size quadrants from which the next random selection is made. This process is repeated until each quadrant contains only a single element in the adjacency matrix. The last round randomly selects a single element and creates the corresponding edge. The probabilities assigned to the quadrants are designated a, b, c, and d. If $a = b = c = d = 0.25$, then RMAT generator will generate an ER graph.

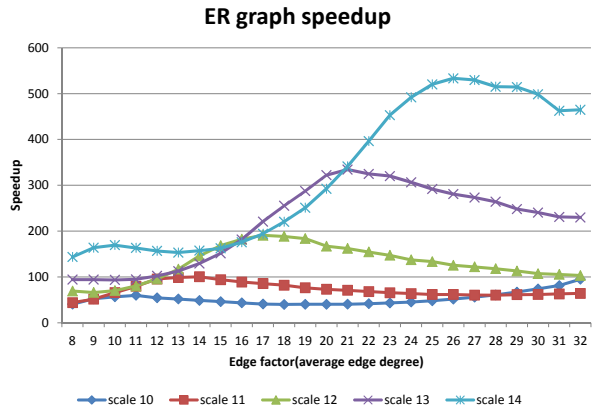
In Albert *et al.* [2] the authors present the small-world phenomena which states the distance between two vertices in the graph is a small number of hops away. Barabasi *et al.* [4] show that the edge distribution follows a power law. In Broder *et al.* [6] the authors show that World Wide Web (WWW) has one huge connected component that contains 90% of the vertices in the graph. The work of Leskovec *et al.* [18] confirmed that many real world networks have these properties.

A. Synthetic Graphs

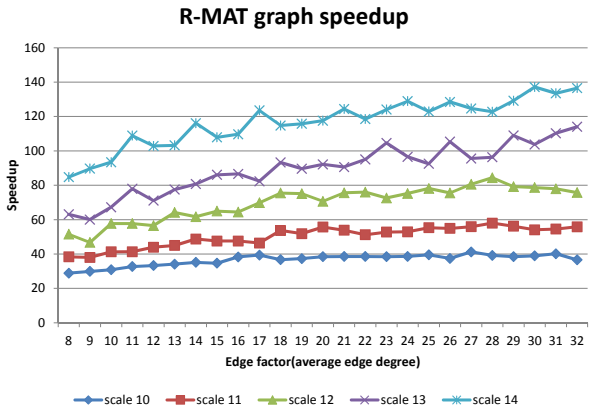
In this subsection we present results of the new algorithm on ER and R-MAT generated graphs. In our tests we created graphs in which the vertex count is a power of 2. We denote this power as the scale. The scales that are tested range from 10 to 14. An edge factor (average number of edges per vertex) of 8 to 32 is checked for each scale size. For each scale and edge factor, 100 different graphs are tested and timed. The speedups in the figures are of the average times. The ordinate denotes speedup and the abscissa denotes the edge factor.

As can be seen in Fig. IV (a), the speedups of the new algorithm for random graphs are considerable despite the same upper bound complexity. This is because redundant computations are avoided for all the insertion scenarios.

In Fig. IV (b) the speedups for R-MAT graphs are presented. While the speedups for the R-MAT graph are not as

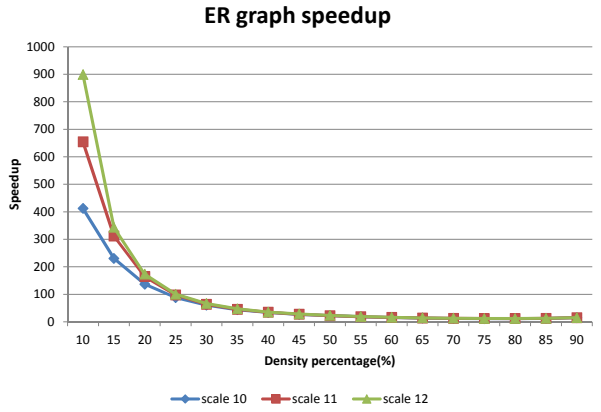


(a) Speedup of the streaming algorithm for Erdös-Rènyi sparse graphs.

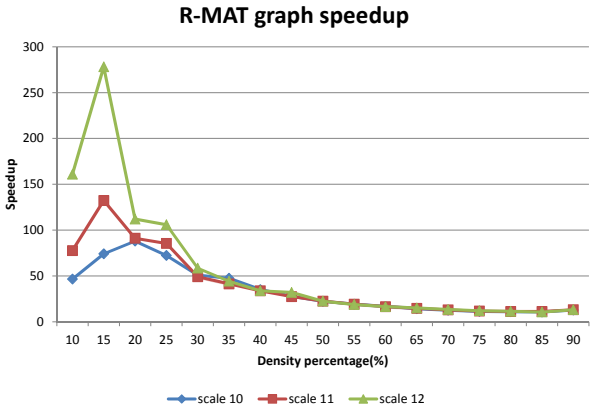


(b) Speedup of the streaming algorithm for R-MAT sparse graphs.

Figure 5. Speedup of the new streaming algorithm versus doing a full recompute for sparse synthetic graphs.



(a) Speedup of the streaming algorithm for Erdös-Rènyi dense graphs.



(b) Speedup of the streaming algorithm for R-MAT dense graphs.

Figure 6. Speedup of the new streaming algorithm versus doing a full recompute for dense synthetic graphs.

considerable as those for ER graphs, it is worth noting that R-MAT graphs have a different structure than ER and in many cases are more challenging. Also, the variance in the speedups between the different edge factors is significantly smaller for R-MAT graphs.

For both ER and R-MAT graphs we measure the performance for dense graph insertions as well. We use graphs with density of 5% to 90% with intervals of 5%. The speedups can be seen in Figure IV. Initially when the graphs are still relatively sparse, the speedup of the new algorithm gradually increases. At some point, the graph becomes better connected such that more edges need to be traversed for both the BFS and the dependency accumulation. From this point onwards, the speedups gradually decrease. However, the densification offers an additional benefit - the effective(average) diameter decreases. The benefit from this is that for many of the trees, no re-computation is needed as the newly inserted edge connects vertices that are in the same level. For both the ER graphs in Figure IV (a) and the R-MAT graphs in Figure IV (b), when the graph density goes above 55%-60% the speedups come down. However, the speedups stay in the 2-digit region of $12X - 18X$.

B. Real graph

For real social networks, we used five collaboration networks supplied by Leskovc *et al.* [18] and his software [1]. Using terminology defined in [18], the effective diameter is defined as the 90th percentile distance of all the vertices.

The networks that were used are collaboration networks for

Table I
SPEEDUP OF STREAMING ALGORITHM ON REAL CITATION NETWORKS.

Collaboration network	Vertices	Edges	Speedup
Astro-physics	18772	198080	$148X$
Condensed matter	23133	93468	$91X$
General relativity	5242	14490	$40X$
High energy physics	12008	118505	$108X$
High energy physics theory	9877	51970	$36X$

Arxiv in the following fields: astro-physics, condensed matter, general relativity, high energy physics, and high energy physics theory. The effective diameters of these collaboration networks are 5.1, 6.6, 7.6, 5.8, and 7.5 (respectively). The diameter (maximal distance) for these graphs is 14, 15, 17, 13, and 17 (respectively).

In our tests, we create the graph with all but 200 edges. These 200 edges are inserted one at a time using the new streaming algorithm. As with the synthetic graphs, we make sure that the new edge is not connecting two separate components.

The average speedup for the five collaboration networks can be seen in Table I.

V. CONCLUSIONS

In this paper we present the first algorithm for computing streaming betweenness centrality. The new algorithm avoids computing betweenness centrality scores for vertices that have no new paths going through them due to the insertion of a new edge. The new algorithm has the same complexity bounds as the static algorithm $O(V^2 + VE)$; however, we demonstrated considerable speedup for both random graphs and for real-world networks. For synthetic random graphs the speedup depends on both the type and size of random graph. For random graphs with 16K vertices the average speedup is between $100X - 400X$. For 5 different collaborations networks the average speedup is in range of $36X - 148X$

The new algorithm for betweenness centrality is not limited to social networks and can be used on all types of graphs.

VI. ACKNOWLEDGMENTS

This work was supported in part by NSF Grant CNS-0708307 and by the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program.

REFERENCES

[1] *Stanford Network Analysis Package*, 2012 (accessed April 2012). [Online]. Available: http://snap.stanford.edu/data/

[2] R. Albert, H. Jeong, and A. Barabási, "Internet: Diameter of the world-wide web," *Nature*, vol. 401, no. 6749, pp. 130–131, Sep 09 1999.

[3] D. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *Algorithms and Models for the Web-Graph*, ser. Lecture Notes in Computer Science, A. Bonato and F. Chung, Eds., vol. 4863. Springer Berlin / Heidelberg, 2007, pp. 124–137.

[4] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[5] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[6] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Computer Networks*, vol. 33, pp. 309 – 320, 2000.

[7] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: design, implementation, and applications," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[8] R. Bye, S. Schmidt, K. Luther, and S. Albayrak, "Application-level simulation for network security," in *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, ICST, Brussels, Belgium, Belgium, 2008, pp. 33:1–33:10.

[9] D. Chakrabarti, Y. Zhany, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SIAM Proceedings Series*, 2004, pp. 442–446.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. New York: The MIT Press, 2001.

[11] D. Ediger, K. Jiang, J. Riedy, D. Bader, C. Corley, R. Farber, and W. Reynolds, "Massive social network analysis: Mining Twitter for social good," in *39th International Conference on Parallel Processing (ICPP), 2010*, Sept. 2010, pp. 583 –593.

[12] N. Edmonds, T. Hoefler, and A. Lumsdaine, "A space-efficient parallel algorithm for computing betweenness centrality in distributed memory," in *International Conference on High Performance Computing (HiPC), 2010*, Dec. 2010, pp. 1 –10.

[13] P. Erdös and A. Rényi, "On random graphs I," *Publicationes Mathematicae*, pp. 290–297, June 1959.

[14] ——, "The evolution of random graphs," *Magyar Tud. Akad. Mat.*, pp. 17–61, 1960.

[15] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, pp. 345–345, June 1962.

[16] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. pp. 35–41, 1977.

[17] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[18] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 1.

[19] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda, "A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS) 2009*.

[20] M. Rubinov and O. Sporns, "Complex network measures of brain connectivity: Uses and interpretations," *NeuroImage*, vol. 52, no. 3, pp. 1059 – 1069, 2010, Computational Models of the Brain.

[21] G. Tan, V. Sreedhar, and G. Gao, "Analysis and performance results of computing betweenness centrality on IBM Cyclops64," *The Journal of Supercomputing*, vol. 56, pp. 1–24, 2011.

[22] J. R. Tyler, D. M. Wilkinson, and B. A. Huberman, *Email as spectroscopy: automated discovery of community structure within organizations*. Deventer, The Netherlands: Kluwer, B.V., 2003, pp. 81–96.

[23] S. Warshall, "A theorem on Boolean matrices," *J. ACM*, vol. 9, pp. 11–12, Jan. 1962.