Betweenness Centrality – Incremental and Faster*

Meghana Nasre[1], Matteo Pontecorvi[2], and Vijaya Ramachandran[2]

[1] Indian Institute of Technology Madras, India
meghana@cse.iitm.ac.in
[2] University of Texas at Austin, USA
{cavia,vlr}@cs.utexas.edu

Abstract. We present an incremental algorithm that updates the betweenness centrality (BC) score of all vertices in a graph G when a new edge is added to G, or the weight of an existing edge is reduced. Our incremental algorithm runs in $O(\nu^* \cdot n)$ time, where ν^* is bounded by m^*, the number of edges that lie on a shortest path in G. We achieve the same bound for the more general incremental vertex update problem. Even for a single edge update, our incremental algorithm is the first algorithm that is provably faster on sparse graphs than recomputing with the well-known static Brandes algorithm. It is also likely to be much faster than Brandes on dense graphs since m^* is often close to linear in n.

Our incremental algorithm is very simple, and we give an efficient cache-oblivious implementation that incurs $O(n \cdot sort(\nu^*))$ cache misses, where *sort* is a well-known measure for caching efficiency.

1 Introduction

Betweenness centrality (BC) of vertices is a widely-used measure in the analysis of large complex networks. As a classical measure, BC is widely used in sociology [6,22], biology [9], physics [17] and network analysis [32,34]. BC is also useful for critical applications such as identifying lethality in biological networks [31], identifying key actors in terrorist networks [20] and finding attack vulnerability of complex networks [14]. In recent years, BC also had a wide impact in the analysis of social networks [11,33], wireless [25] and mobile networks [4], P2P networks [18] and more.

In this paper we present incremental BC algorithms that are provably faster on sparse graphs than current algorithms for the problem. By an *incremental update on edge* (u, v) we mean the addition of a new edge (u, v) with finite weight if (u, v) is not present in the graph, or a decrease in the weight of an existing edge (u, v); in an *incremental vertex update*, incremental updates can occur on any subset of edges incident to v, and this includes adding new edges.

We now define BC, and describe the widely-used Brandes algorithm [3] for this problem. We then describe our contributions and related work.

* This work was supported in part by NSF grants CCF-0830737 and CCF-1320675.

E. Csuhaj-Varjú et al. (Eds.): MFCS 2014, Part II, LNCS 8635, pp. 577–588, 2014.

Betweenness Centrality (BC) and the Brandes Algorithm. Let $G = (V, E)$ be a (directed or undirected) graph with positive edge weights $\mathbf{w}(e)$, $e \in E$. The distance $d(s,t)$ from s to t is the weight of a shortest path from s to t. Let σ_{st} be the number of shortest paths from s to t in G (with $\sigma_{ss} = 1$) and let $\sigma_{st}(v)$ be the number of shortest paths from s to t that pass through v. Thus, $\sigma_{st}(v) = \sigma_{sv} \cdot \sigma_{vt}$ if $d(s,t) = d(s,v) + d(v,t)$, and $\sigma_{st}(v) = 0$ otherwise.

The *pair dependency* of s, t on an intermediate vertex v is $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$ [3]. For $v \in V$, the *betweenness centrality* $\mathrm{BC}(v)$ is defined by Freeman [6] as:

$$\mathrm{BC}(v) = \sum_{s \neq v, t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{s \neq v, t \neq v} \delta_{st}(v) \tag{1}$$

Let $P_s(v)$ denote the predecessors of v on shortest paths from s. Brandes [3] defined the dependency of a vertex s on a vertex v as $\delta_{s\bullet}(v) = \sum_{t \in V \setminus \{v,s\}} \delta_{st}(v)$, and observed that

$$\delta_{s\bullet}(v) = \sum_{w : v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w)) \quad \text{and} \quad \mathrm{BC}(v) = \sum_{s \neq v} \delta_{s\bullet}(v) \tag{2}$$

Alg. 1 gives Brandes' algorithm to compute $BC(v)$ for all $v \in V$. This algorithm runs in $O(mn + n^2 \log n)$ time, where $|V| = n$ and $|E| = m$.

Algorithm 1. Betweenness-centrality($G = (V, E)$) (from Brandes [3])

1: **for** every $v \in V$ **do** $\mathrm{BC}(v) \leftarrow 0$.
2: **for** every $s \in V$ **do**
3: Run Dijkstra's SSSP from s and compute σ_{st} and $P_s(t), \forall\, t \in V \setminus \{s\}$.
4: Store the explored nodes in a stack S in non-increasing distance from s.
5: Accumulate dependency of s on all $t \in S$ using Eqn. 2.

1.1 Our Contributions

Let E^* be the set of edges in G that lie on shortest paths, let $m^* = |E^*|$, and let ν^* be the maximum number of edges that lie on shortest paths through any single vertex. Here is our main result:

Theorem 1. *After an incremental update on an edge or a vertex in a directed or undirected graph with positive edge weights, the betweenness centrality of all vertices can be recomputed in:*

1. *$O(\nu^* \cdot n)$ time using $O(\nu^* \cdot n)$ space;*
2. *$O(m^* \cdot n)$ time using $O(n^2)$ space.*

Since $\nu^* \leq m^*$ and $m^* \leq m$, the worst case time for both results is bounded by $O(mn + n^2)$, which is a $\log n$ factor improvement over Brandes' algorithm on sparse graphs. Our results also have benefits for dense graphs (when $m = \omega(n \log n)$) similar to the Hidden Paths algorithm of Karger et al. [15] for the all pairs shortest paths (APSP) problem (see also McGeogh [26]), although our techniques are different. This is through the use of ν^* or m^* in place of m, and we comment more on this below. Our algorithms are simple, and only use stack,

queue and linked list data structures. We also give an efficient cache-oblivious implementation which avoids the high caching cost of Dijkstra's algorithm that is present in Alg. 1 (its bound is given in Section 5).

Both ν^* and m^* are typically much smaller than m in dense graphs. For instance, it is known [7,13,15,23] that $m^* = O(n \log n)$ with high probability in a complete graph where edge weights are chosen from a large class of probability distributions, including the uniform distribution on integers in $[1, n^2]$ or reals in $[0, 1]$. For such graphs, both results in Theorem 1 imply an $O(n^2 \log^2 n)$ algorithm for an incremental update. For the random real weights, the first result would in fact give $O(n^2)$ time and space since shortest paths are unique with probability 1 in this setting, hence $\nu^* = O(n)$.

We observe that Alg. 1 (Brandes) can be made to run faster: In a directed graph, by using the Pettie [29] or the Hidden Paths algorithm in place of Dijkstra in Step 3 of Alg. 1, we can compute BC scores in $O(mn + n^2 \log \log n)$ or $O(m^* n + n^2 \log n)$ time, respectively. In an undirected graph, we can obtain $O(mn \cdot \log \alpha(m, n))$ time, where α is an inverse-Ackermann function, using [30]. Our incremental bounds are better than any of these bounds for sparse graphs.

There are several results on dynamic BC algorithms and heuristics [10,12,19,21], but our time bounds are better than any of these on sparse graphs. In fact, ours is the first incremental BC algorithm that gives a provable improvement over Brandes' algorithm for sparse graphs, which are the type of graphs that typically occur in practice. While the space used by our algorithms is higher than Brandes', which uses only linear space, our second result matches the best space bound obtained by any of these other dynamic BC algorithms and heuristics.

We consider only incremental updates in this paper. Computing decremental and fully dynamic updates efficiently appears to be more challenging (as is the case for APSP [5]). In recent work [28], we have developed decremental and fully dynamic BC algorithms that build on techniques in [5], and run in amortized time $O({\nu^*}^2 \cdot polylog(n))$.

Organization. In Section 2 we discuss related work on dynamic BC. Since the algorithm for a single edge update is simpler than that for a vertex update, we first present our edge update result in Section 3. We describe the $O(n \cdot \nu^*)$ algorithm, and then the simple changes needed to obtain the second $O(n^2)$ space result. We present the vertex update result in Section 4. In Section 5 we sketch our efficient cache-oblivious BC algorithm, and mention some preliminary experimental results.

Step 5 of Alg. 1: For completeness, Alg. 2 below gives the algorithm for Step 5 in Brandes' algorithm (Alg. 1). We will use Alg. 2 unchanged for our first result of Theorem 1, and modified (to eliminate the use of predecessor lists $P_s(t)$) for the second result of Theorem 1.

2 Related Work

Approximation and parallel algorithms for BC have been considered in [2,8], [24] respectively. More recently, the problem of dynamic betweenness centrality has

Algorithm 2. Accumulate-dependency(s, S) (from [3])

Input: For every $t \in V$: $\sigma_{st}, P_s(t)$.
A stack S containing $v \in V$ in a suitable order (non-increasing $d(s, v)$ in [3]).

1: **for** every $v \in V$ **do** $\delta_{s\bullet}(v) \leftarrow 0$.
2: **while** $S \neq \emptyset$ **do**
3: $w \leftarrow \text{pop}(S)$.
4: **for** $v \in P_s(w)$ **do** $\delta_{s\bullet}(v) \leftarrow \delta_{s\bullet}(v) + \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w))$.
5: **if** $w \neq s$ **then** $\text{BC}(w) \leftarrow \text{BC}(w) + \delta_{s\bullet}(w)$.

received attention, and these results for incremental and in some cases, decremental, BC are listed in the table below. All of these results except [16] deal with unweighted graphs as opposed to our results, which are for the weighted case. Further, while all give encouraging experimental results or match the Brandes worst-case time complexity, none prove any worst-case improvement. As mentioned in the Introduction, BC is also widely used in weighted networks (see [4,18,31,32]); however, only the heuristic in Kas et al. [16], which has no worst-case bounds, addresses this version.

Paper	Year	Space	Time	Weights	Update Type
Brandes static [3]	2001	$O(m+n)$	$O(mn)$	NO	Static Alg.
Lee et al. [21]	2012	$O(n^2+m)$	Heuristic	NO	Single Edge
Green et al. [12]	2012	$O(n^2+mn)$	$O(mn)$	NO	Single Edge
Kourtellis+ [19]	2014	$O(n^2)$	$O(mn)$	NO	Single Edge
Singh et al. [10]	2013	–	Heuristic	NO	Vertex update
Brandes static [3]	2001	$O(m+n)$	$O(mn+n^2\log n)$	YES	Static Alg.
Kas et al. [16]	2013	$O(n^2+mn)$	Heuristic	YES	Single Edge
This paper	2014	$O(\nu^* \cdot n)$	$O(\nu^* \cdot n)$	YES	Vertex Update
This paper	2014	$O(n^2)$	$O(m^* \cdot n)$	YES	Vertex Update

Our first algorithm, which takes time $O(\nu^* \cdot n)$ in a weighted graph even for a vertex update, improves on all previous results when $\nu^* = o(m)$. By slightly relaxing the time complexity to $O(m^* \cdot n)$, we are also able to match the best space complexity in any of the previous results, while matching their time complexities and improving on all of them when $m^* = o(m)$.

3 Incremental Edge Update

In this section we present our algorithm to recompute BC scores of all vertices in a directed graph $G = (V, E)$ after an incremental edge update (i.e., adding an edge or decreasing the weight of an existing edge). Let $G' = (V, E')$ denote the graph obtained after an edge update to $G = (V, E)$. A path π_{st} from s to t in G has *weight* $\mathbf{w}(\pi_{st}) = \sum_{e \in \pi_{st}} \mathbf{w}(e)$. Let $d(s,t), \sigma_{st}$, $\delta_{s\bullet}(t)$ and $\text{DAG}(s)$ denote the distance from s to t in G, the number of shortest paths from s to t in G, the dependency of s on t and the SSSP DAG rooted at s in G respectively; let $d'(s,t), \sigma'_{st}$, $\delta'_{s\bullet}(t)$ and $\text{DAG}'(s)$ denote these parameters in G'.

Lemma 1. *If weight of edge (u, v) in G is decreased to obtain G', then for any $x \in V$, the set of shortest paths from x to u and from v to x is the same in G and G', and $d'(x, u) = d(x, u)$, $d'(v, x) = d(v, x)$; $\sigma'_{xu} = \sigma_{xu}$, $\sigma'_{vx} = \sigma_{vx}$.*

Proof. Since edge weights are positive, the edge (u, v) cannot lie on a shortest path to u or from v. The lemma follows. □

By Lemma 1, $\text{DAG}(v) = \text{DAG}'(v)$ after the decrease of weight on edge (u, v). The next lemma shows that after the weight of (u, v) is decreased we can efficiently obtain the updated values $d'(s, t)$ and σ'_{st} for any $s, t \in V$.

Lemma 2. *Let the weight of edge (u, v) be decreased to $\mathbf{w}'(u, v)$, and for any given pair of vertices s, t, let $D(s, t) = d(s, u) + \mathbf{w}'(u, v) + d(v, t)$. Then,*

1. *If $d(s, t) < D(s, t)$, then $d'(s, t) = d(s, t)$ and $\sigma'_{st} = \sigma_{st}$.*
   The shortest paths from s to t in G' are the same as in G.
2. *If $d(s, t) = D(s, t)$, then $d'(s, t) = d(s, t)$ and $\sigma'_{st} = \sigma_{st} + (\sigma_{su} \cdot \sigma_{vt})$.*
   The shortest paths from s to t in G' are a superset of the shortest paths G.
3. *If $d(s, t) > D(s, t)$, then $d'(s, t) = D(s, t)$ and $\sigma'_{st} = \sigma_{su} \cdot \sigma_{vt}$.*
   The shortest paths from s to t in G' are new (shorter distance).

Proof. Case 1 holds because the shortest path distance from s to t remains unchanged and no new shortest path is created in this case. In case 2, the shortest path distance from s to t remains unchanged, but there are $\sigma_{su} \cdot \sigma_{vt}$ new shortest paths from s to t created via edge (u, v). In case 3, the shortest path distance from s to t decreases and all new shortest paths pass through (u, v). □

By Lemma 2, the updated values $d'(s, t)$ and σ'_{st} can be computed in constant time for each pair s, t. Once we have the updated $d'(\cdot)$ and $\sigma'_{(\cdot)}$ values, we need the updated predecessors $P'_s(t)$ for every s, t pair for Alg. 2. The SSSP $\text{DAG}(s)$ rooted at a source s is the union of all the $P_s(t), \forall\ t \in V$. Thus, obtaining $\text{DAG}'(s)$ after the edge update is equivalent to computing the $P'_s(t), \forall\ t \in V$. The next section gives a simple algorithm to maintain the SSSP DAGs rooted at every source $s \in V$, after an incremental edge update.

3.1 Updating an SSSP DAG

For each pair s, t we define $flag(s, t)$ to indicate the specific case of Lemma 2 that is applicable.

$$flag(s,t) = \begin{cases} \text{UN-changed} & \text{if } d'(s,t) = d(s,t) \text{ and } \sigma'_{st} = \sigma_{st} \quad \text{(Lemma 2-1)} \\ \text{NUM-changed} & \text{if } d'(s,t) = d(s,t) \text{ and } \sigma'_{st} > \sigma_{st} \quad \text{(Lemma 2-2)} \\ \text{WT-changed} & \text{if } d'(s,t) < d(s,t) \quad \text{(Lemma 2-3)} \end{cases}$$

By Lemma 2, $flag(s, t)$ can be computed in constant time for each pair s, t. Given an input s and the updated edge (u, v), Alg. 3 (Update-DAG) constructs a set of edges H using these $flag$ values, together with $\text{DAG}(s)$ and $\text{DAG}(v)$. We will show that H contains exactly the edges in $\text{DAG}'(s)$. The algorithm

considers edges in DAG(s) (Steps 3–5) and edges in DAG(v) (Steps 6–8), and for each edge (a, b) in either DAG, it decides whether to include it in H based on the value of $flag(s, b)$. For the updated edge (u, v) there is a separate check (Steps 9–10). The algorithm clearly takes time linear in the size of DAG(s) and DAG(v), i.e., $O(\nu^*)$ time.

Algorithm 3. Update-DAG($s, \mathbf{w}'(u, v)$)

Input: DAG(s), DAG(v), and $flag(s, t), \forall t \in V$.
Output: An edge set H after decrease of weight on edge (u, v), and $P'_s(t), \forall t \in V - \{s\}$.
1: $H \leftarrow \emptyset$.
2: **for** each $v \in V$ **do** $P'_s(v) = \emptyset$.
3: **for** each edge $(a, b) \in$ DAG(s) and $(a, b) \neq (u, v)$ **do**
4: **if** $flag(s, b) =$ UN-changed or $flag(s, b) =$ NUM-changed **then**
5: $H \leftarrow H \cup \{(a, b)\}$ and $P'_s(b) \leftarrow P'_s(b) \cup \{a\}$.
6: **for** each edge $(a, b) \in$ DAG(v) **do**
7: **if** $flag(s, b) =$ NUM-changed or $flag(s, b) =$ WT-changed **then**
8: $H \leftarrow H \cup \{(a, b)\}$ and $P'_s(b) \leftarrow P'_s(b) \cup \{a\}$.
9: **if** $flag(s, v) =$ NUM-changed or $flag(s, v) =$ WT-changed **then**
10: $H \leftarrow H \cup \{(u, v)\}$ and $P'_s(v) \leftarrow P'_s(v) \cup \{u\}$.

Lemma 3. *Let H be the set of edges output by Alg. 3. An edge $(a, b) \in H$ if and only if $(a, b) \in DAG'(s)$.*

Proof. Since the update is an incremental update on edge (u, v), we note that for any b, a shortest path π'_{sb} from s to b in G' can be of two types:
(i) π'_{sb} is a shortest path in G. Therefore every edge on such a path is present in DAG(s) and each such edge is added to H in Steps 3–5 of Alg. 3.
(ii) π'_{sb} is not a shortest path in G. However, since π'_{sb} is a shortest path in G', therefore π'_{sb} is of the form $s \rightsquigarrow u \rightarrow v \rightsquigarrow b$. Since shortest paths from s to u in G and G' are unchanged (by Lemma 1), the edges in the sub-path $s \rightsquigarrow u$ are present in DAG(s) and are added to H in Steps 3–5 of Alg. 3. Finally, shortest paths from v to any b in G and G' remain unchanged. Thus, the edges in the sub-path $v \rightsquigarrow b$ are present in DAG(v) and are added to H in Steps 6–8 of Alg. 3.

For the other direction, if the edge (a, b) is added to H by Step 5, this implies that the edge $(a, b) \in$ DAG(s). Thus, there exists a shortest path $\pi_{sb} = s \rightsquigarrow a \rightarrow b$ in G. We execute Step 5 when $flag(s, b) =$ UN-changed or $flag(s, b) =$ NUM-changed. Thus every shortest path from s to b in G is also shortest path in G'. Therefore, $(a, b) \in$ DAG$'(s)$. If the edge (a, b) is added to H by Step 8, then the edge $(a, b) \in$ DAG(v). Thus, there exists a shortest path $\pi_{vb} = v \rightsquigarrow a \rightarrow b$ in G. Since decreasing the weight of the edge (u, v) does not change shortest paths from v to any other vertex, π_{vb} is in G'. We execute Step 8 when $flag(s, b) =$ NUM-changed or $flag(s, b) =$ WT-changed. Therefore, there exists at least one shortest path from s to b in G' that uses the updated edge (u, v). Hence the path $\pi'_{sb} = \pi'_{su} \cdot (u, v) \cdot \pi_{vb}$ is shortest in G', and this establishes that $(a, b) \in$ DAG$'(s)$. Finally, edge (u, v) is added to H by Step 10 only if $flag(s, v)$ is NUM-changed or WT-changed, and in either case, there is at least a new shortest path from s to v through (u, v). Hence $(u, v) \in$ DAG$'(s)$. □

3.2 Updating Betweenness Centrality Scores

The algorithm for updating the BC scores after an edge update (Alg. 4) is similar to Alg. 1, but with the following changes: an extended Step 1 also computes, for every s, t pair, the updated $d'(s,t)$ and σ'_{st}, as well as $flag(s,t)$. Using Lemma 2, we spend constant time for each s, t pair, hence $O(n^2)$ time for all pairs. In Step 3, instead of Dijkstra's algorithm, we run Alg. 3 to obtain the updated predecessor lists $P'_s(t)$, for all s, t. This step requires time $O(\nu^*)$ for a source s, and $O(\nu^* \cdot n)$ over all sources. The last difference is in Step 4: we place in the stack S the vertices in reverse topological order in $\mathrm{DAG}'(s)$, instead of non-increasing distance from s. This requires time linear in the size of the updated DAG. Thus the time complexity of Edge-Update is $O(\nu^* \cdot n)$.

Algorithm 4. Edge-Update$(G = (V, E), \mathbf{w}'(u, v))$

Input: updated edge with $\mathbf{w}'(u, v)$, $d(s,t)$ and σ_{st}, $\forall\ s, t \in V$; $\mathrm{DAG}(s), \forall\ s \in V$.
Output: $\mathrm{BC}'(v)$, $\forall\ v \in V$; $d'(s,t)$ and σ'_{st} $\forall\ s, t \in V$; $\mathrm{DAG}'(s), \forall\ s \in V$.

1: **for** every $v \in V$ **do** $\mathrm{BC}'(v) \leftarrow 0$.
   for every $s, t \in V$ **do** compute $d'(s,t), \sigma'_{st}$, $flag(s,t)$. // use Lemma 2
2: **for** every $s \in V$ **do**
3: Update-DAG$(s, (u, v))$. // use Alg. 3
4: Stack $S \leftarrow$ vertices in V in a reverse topological order in $\mathrm{DAG}'(s)$.
5: Accumulate-dependency(s, S). // use Alg. 2

Undirected Graphs. For an undirected G, we construct the corresponding directed graph G_D in which every undirected edge is replaced with 2 directed edges. An incremental update on an undirected edge (u, v) is equivalent to two edge updates on (u, v) and (v, u) in G_D. Thus, Theorem 1 holds for undirected graphs.

Space Efficient Implementation. In order to obtain $O(n^2)$ space complexity, we do not store the SSSP DAGs rooted at every source. Instead, we only store the edge set E^*. After an incremental update on edge (u, v) we first construct the updated set E'^* in $O(m^* \cdot n)$ time as follows. For each edge $(a, b) \in E^*$, if $d'(s, b) = d(s, a) + \mathbf{w}(a, b)$ for some source $s \in V$, then $(a, b) \in E'^*$. Using the updated E'^* we can construct $\mathrm{DAG}'(s)$ in $O(m^*)$ time, by using the fact that an edge $(a, b) \in E'^*$ belongs to $\mathrm{DAG}'(s)$ iff $d(s, b) = d(s, a) + \mathbf{w}(a, b)$. Since the construction of each updated DAG takes $O(m^*)$ time and there are n DAGs to be constructed, the $O(m^* \cdot n)$ time complexity follows. The space used is $O(m^* + n^2)$ to store E^* and $d(s,t)$, σ_{st}, for all $s, t \in V$.

4 Incremental Vertex Update

We now consider an incremental update to a vertex v in $G = (V, E)$, which allows an incremental edge update on any subset of edges incoming to and outgoing from v. In this algorithm, we use the graph G and the graph $G_R = (V, E_R)$, which is obtained by reversing every edge in G, i.e., $(a, b) \in E_R$ iff $(b, a) \in E$. Thus, for every $s \in V$, we also maintain $\mathrm{DAG}_R(s)$, the SSSP DAG rooted at s in G_R. We will obtain the same time bound as in Section 3.

4.1 Overview

Let $E_i(v)$ and $E_o(v)$ denote the set of updated edges incoming to v and outgoing from v respectively. Our algorithm is a natural extension, with some new features, of the algorithm for a single edge update, and works as follows. We process $E_i(v)$ in G in Step 1 to form G', G'_R, $\text{DAG}'(s)$ and $\text{DAG}'_R(s)$; we then process $E_o(v)$ in G'_R in a complementary Step 2 to obtain the updated G'', $\text{DAG}''(s)$ and $\text{DAG}''_R(s)$. Step 1, which processes $E_i(v)$, consists of two phases.

Step 1, Phase 1: Constructing the $\text{DAG}'(s)$ for updates in $E_i(v)$.
Since $E_i(v)$ contains updated edges incoming to v, $\text{DAG}(v) = \text{DAG}'(v)$ (as in the single edge update case). In order to handle updates to several edges incoming to v, we strengthen Lemma 2 by introducing $\hat{\sigma}$, which keeps track of new shortest paths from s to v that go through any of the updated edges in $E_i(v)$. This allows us to efficiently recompute the number of shortest paths from a source to any node in G', and thus update all the $\text{DAG}'(s)$ using an algorithm similar to Alg. 3. Parts (A), (B), (C) in Section 4.2 describe Phase 1 in detail.

Step 1, Phase 2: Constructing the $\text{DAG}'_R(s)$ for updates in $E_i(v)$.
We present an efficient algorithm to construct the $\text{DAG}'_R(s)$ for all s in G'. We construct these reverse graphs because the edges in $E_o(v)$ are in fact incoming edges to v in G'_R. Hence our method to maintain DAGs when incoming edges are updated can be applied to G'_R with E_o to obtain $\text{DAG}''_R(s)$, for every s, in Phase 1 of Step 2 (and then we can obtain the $\text{DAG}''(s)$ in Phase 2 of Step 2).

Let (t, a) be the first edge on a shortest path from t to v in G'. Then (t, a) is an outgoing edge from t in $\text{DAG}'(t)$, and its reverse (a, t) is on a shortest path from v to t in G'_R. Further an edge (a, t) is on a new shortest path from v to t in G'_R if and only if its reverse is on a new shortest path from t to v in G'. These edges on new shortest paths are the ones we need to keep track of in order to update the reverse DAGs, and to facilitate this we define a collection of sets R_t, $t \in V$. The set R_t is the set of (reversed) outgoing edges from t in $\text{DAG}'(t)$ that lie on a shortest path from t to v in G' (see also Eqn. 5 in the next section). Thus, if a new shortest path π_{sb} is present in $\text{DAG}'_R(s)$ (π_{sb} must pass through v), its last edge (a, b) is present in R_b. Using the sets $R_t, \forall\ t \in V$, it is possible to quickly build the $\text{DAG}'_R(t)$ after Phase 1 as shown in part (D) in section 4.2.

Step 2: After applying Phase 1 and 2 on the initial DAGs using E_i to obtain the $\text{DAG}'_R(s)$ and G'_R, Step 2 re-applies Phase 1 and Phase 2 on these updated graphs using E_o in order to complete all of the updates to vertex v. We can then apply Alg. 2 to the $\text{DAG}''(s)$ to obtain the BC scores for the updated graph G''.

4.2 Vertex Update Algorithm

We now give details of each phase of our algorithm starting with the graph G.

Step 1, Phase 1
(A) **Compute $d'(s, v)$ and σ'_{sv} for any s.** We show how to compute in G' the distance and number of shortest paths to v from any s. Let $(u_j, v) \in E_i(v)$

and let $D_j(s,v) = d(s,u_j) + \mathbf{w}'(u_j,v)$. Since the updates on edges in $E_i(v)$ are incremental, it follows that:

$$d'(s,v) = \min\{d(s,v), \min_{j:(u_j,v)\in E_i(v)}\{D_j(s,v)\}\} \tag{3}$$

Further, if $d'(s,v) = d(s,v)$, we define:

$$\widehat{\sigma}'_{sv} = |\{\pi'_{sv} : \pi'_{sv} \text{ is a shortest path in } G' \text{ and } \pi'_{sv} \text{ uses } e \in E_i(v)\}| \tag{4}$$

We also need to compute σ'_{sv}, the number of shortest paths from s to v in G'. It is straightforward to compute $d'(s,v)$, σ'_{sv}, and $\widehat{\sigma}'_{sv}$ in $O(|E_i(v)|)$ time. Alg. 5 gives the details of this step.

Algorithm 5. Dist-to-v $(s, E_i(v))$

Input: $E_i(v)$ with updated weights $\mathbf{w}'$. $d(s,t)$ and σ_{st}, $\forall\ s,t \in V$.
Output: $d'(s,v), \sigma'_{sv}, \widehat{\sigma}'_{sv}$.

1: $\hat{\sigma}'_{sv} \leftarrow 0$, $\sigma'_{sv} \leftarrow \sigma_{sv}$, $D' \leftarrow d(s,v)$.
2: **for** each edge $(u_i,v) \in E_i(v)$ **do**
3: **if** $D' = d(s,u_i) + \mathbf{w}'(u_i,v)$ **then**
4: $\sigma'_{sv} \leftarrow \sigma'_{sv} + \sigma_{su_i}$.
5: $\hat{\sigma}'_{sv} \leftarrow \hat{\sigma}'_{sv} + \sigma_{su_i}$.
6: **else if** $D' > d(s,u_i) + \mathbf{w}'(u_i,v)$ **then**
7: $D' \leftarrow d(s,u_i) + \mathbf{w}'(u_i,v)$.
8: $\sigma'_{sv} \leftarrow \sigma_{su_i}$.
9: $d'(s,v) \leftarrow D'$.

Algorithm 6. Upd-Rev-DAG$(s, E_i(v))$

Input: $\mathrm{DAG}_R(s)$; $R_t, flag(s,t), \forall t \in V$.
Output: An edge set X after update on edges in $E_i(v)$.

1: $X \leftarrow \emptyset$.
2: **for** each edge $(a,b) \in \mathrm{DAG}_R(s)$ **do**
3: **if** $flag(b,s)$ = UN-changed or $flag(b,s)$ = NUM-changed **then**
4: $X \leftarrow X \cup (a,b)$.
5: **for** each $b \in V \setminus \{s\}$ **do**
6: **if** $flag(b,s)$ = NUM-changed or $flag(b,s)$ = WT-changed **then**
7: $X \leftarrow X \cup R_b$.

(B) **Compute $d'(s,t)$ and $\sigma'(s,t)$ for all s,t.** After computing $d'(s,v), \sigma'_{sv}$ and $\hat{\sigma}'_{sv}$, we show that the values $d'(s,t)$ and $\sigma'(s,t)$ can be computed efficiently. We state Lemma 4 which captures this computation. The proof of this lemma is similar to Lemma 2 in the edge update case.

Lemma 4. *Let $E_i(v)$ be the set of updated edges incoming to v. Let G' be the graph obtained by applying the updates in $E_i(v)$ to G. For any $s \in V$ and $t \in V \setminus \{v\}$, let $D(s,t) = d'(s,v) + d(v,t)$, $\Sigma_{st} = \sigma_{st} + \widehat{\sigma}'_{sv} \cdot \sigma_{vt}$, $\Sigma'_{st} = \sigma_{st} + \sigma'_{sv} \cdot \sigma_{vt}$.*

1. *If $d(s,t) < D(s,t)$, then $d'(s,t) = d(s,t)$ and $\sigma'_{st} = \sigma_{st}$.*
2. *If $d(s,t) = D(s,t)$ and $d(s,v) = d'(s,v)$, then $d'(s,t) = d(s,t)$ and $\sigma'_{st} = \Sigma_{st}$.*
3. *If $d(s,t) = D(s,t)$ and $d(s,v) > d'(s,v)$, then $d'(s,t) = d(s,t)$ and $\sigma'_{st} = \Sigma'_{st}$.*
4. *If $d(s,t) > D(s,t)$, then $d'(s,t) = D(s,t)$ and $\sigma'_{st} = \sigma'_{sv} \cdot \sigma_{vt}$.*

The value $flag(s,t)$ for every s,t can be computed using the updated distances and number of shortest paths ($flag(s,t)$ is UN-changed for 1, NUM-changed for both 2 and 3, and WT-changed for 4, in Lemma 4).
(C) **Compute $\mathrm{DAG}'(s)$ for every s.** Given $d'(s,t)$ and $\sigma'(s,t)$ updated for all $s,t \in V$, the algorithm to compute $\mathrm{DAG}'(s)$ for any $s \in V$ is similar to Alg. 3 in the edge update case. The only modification we need is in Steps 9–10 where instead of a single edge (u,v), we consider every edge $(u_1,v) \in E_i(v)$.

Step 1, Phase 2

(D) **Compute $\mathrm{DAG}'_R(s)$ for every s.** We update $\mathrm{DAG}_R(s)$, for every s, for which we use Alg. 6. Recall the sets $R_t, \forall\, t \in V$ defined as:

$$R_t = \{(a,t) \mid (t,a) \in \mathrm{DAG}'(t) \text{ and } \mathbf{w}'(t,a) + d'(a,v) = d'(t,v)\} \tag{5}$$

The set R_t is the set of (reversed) outgoing edges from t in $\mathrm{DAG}'(t)$ that lie on a shortest path from t to v in G'. Consider an edge $e = (a,b)$ in the updated $\mathrm{DAG}'_R(s)$. If e is in $\mathrm{DAG}_R(s)$, it is added to $\mathrm{DAG}'_R(s)$ by Steps 2–4. If e lies on a new shortest path present only in G'_R, its reverse must also lie on a shortest path that goes through v in G', and it will be added to $\mathrm{DAG}'_R(s)$ by the R_b during Steps 5–7 (R_b could also contain edges on old shortest paths through v already processed in Steps 2–4, but even in that case each edge is added to $\mathrm{DAG}'_R(s)$ at most twice by Alg. 6). Note that we do not need to process edges (u_j, v) in E_i separately (as with edge (u,v) in Alg. 2), because these edges will be present in the relevant R_{u_j}. The correctness of Alg. 6 follows from Lemma 5, whose proof is similar to Lemma 3, and is omitted.

Lemma 5. *In Alg. 6, an edge (a,b) is placed in X if and only if $(a,b) \in DAG'_R(s)$ after the incremental update of the set $E_i(v)$.*

Step 2: To process the updates in $E_o(v)$, we re-apply Phase 1 and 2 over G'_R. Since we are processing incoming edges in G'_R, our earlier steps apply unchanged, and we obtain modified values for $d''(\cdot)$, $\sigma''_{(\cdot)}$, and $\mathrm{DAG}''_R(s)$ for every s. Then, using Alg. 6 we obtain the $\mathrm{DAG}''(s)$ for every s. Finally, to compute the updated BC values, we apply Alg. 2.

Performance: Computing $d'(s,v), \sigma'_{sv}$ and $\hat{\sigma}'_{sv}$ requires time $O(|E_i(v)|) = O(n)$ for each s, and hence $O(n^2)$ time for all sources. Applying Lemma 4 to all pairs of vertices takes time $O(n^2)$. The complexity of modified Alg. 3 applied to all DAGs is again $O(\nu^* \cdot n)$. Creating set R_t requires at most $O(E^* \cap \{\text{outgoing edges of } t\})$, so the overall complexity for all the sets is $O(m^*)$. Finally, we bound the complexity of Algorithm 6: the algorithm adds (a,b) in a reverse DAG edge set X at most twice. Since $\sum_{s\in V} |E(\mathrm{DAG}'(s))| = \sum_{s\in V} |E(\mathrm{DAG}'_R(s))|$, at most $O(\nu^* \cdot n)$ edges can be inserted into all the sets X when Algorithm 6 is executed over all sources. Finally, applying the updates in $E_o(v)$ requires a symmetric procedure starting from the reverse DAGs, the final complexity bound of $O(\nu^* \cdot n)$ follows.

5 Efficient Cache Oblivious Algorithm

We give a cache-oblivious implementation with $O(n \cdot sort(\nu^*))$ cache misses. Here, for a size M cache that can hold B blocks, $sort(r) = \frac{r}{B} \cdot \log_M r$ when $M \geq B^2$; *sort* is a measure of good caching performance (even though $sort(r)$ performs $r \log r$ operations, the base of M in the log makes $sort(r)$ preferable to, say, r cache misses). In contrast, the Brandes algorithm calls Dijkstra's algorithm, which is affected by unstructured accesses to adjacency lists that lead to large caching costs (see, e.g., [27]).

We consider the basic edge update algorithm. The main change is in the cache-oblivious (CO) implementation of Alg. 2, which is the last step of Alg. 4.

Instead of the stack S, we use an optimal CO max-priority queue Z [1], that is initially empty. Each element in Z has an ordered pair $(d'(s,v),v)$ as its key value, and also has auxiliary data as described below. Consider the execution of Step 4 in Alg. 2 for vertices $v \in P_s(w)$. Instead of computing the contribution of w to $\delta_{s\bullet}(v)$ for each $v \in P_s(w)$ when w is processed, we insert an element into Z with key value $(d'(s,v),v)$ and auxiliary data $(w, \sigma_{sw}, \delta_{s\bullet}(w))$. With this scheme, entries will be extracted from Z in nonincreasing values of $d'(s,v)$, and all entries for a given v will be extracted consecutively. We compute $\delta_{s\bullet}(v)$ as these extractions for v occur from Z, and also update $BC(v)$. Initially, for each sink t in DAG(s), we insert an element with key value $(d'(s,t),t)$ and NIL auxiliary data. Using [1], Alg. 2 (which is Step 6 in Alg. 4) takes $sort(\nu^*)$ cache misses for source s, and hence $O(n \cdot sort(\nu^*))$ over all sources. The earlier steps in Alg. 4 can be performed in $O(n \cdot sort(\nu^*))$ cache misses by suitably storing and rearranging data for cache-efficiency.

Preliminary experimental results for our basic edge update algorithm (in Section 3) on random graphs generated using the Erdős-Rényi model give 2 to 15 times speed-up over Brandes' algorithm for graphs with 256 to 2048 nodes, with the larger speed-ups on dense graphs.

Acknowledgment. We thank Varun Gangal and Aritra Ghosh at IIT Madras for implementing the algorithms.

References

1. Arge, L., Bender, M.A., Demaine, E.D., Holland-Minkley, B., Munro, J.I.: An optimal cache-oblivious priority queue and its application to graph algorithms. SIAM J. Comput. 36(6), 1672–1695 (2007)
2. Bader, D.A., Kintali, S., Madduri, K., Mihail, M.: Approximating betweenness centrality. In: Bonato, A., Chung, F.R.K. (eds.) WAW 2007. LNCS, vol. 4863, pp. 124–137. Springer, Heidelberg (2007)
3. Brandes, U.: A faster algorithm for betweenness centrality. J. of Mathematical Sociology 25(2), 163–177 (2001)
4. Catanese, S., Ferrara, E., Fiumara, G.: Forensic analysis of phone call networks. Social Network Analysis and Mining 3(1), 15–33 (2013)
5. Demetrescu, C., Italiano, G.F.: A new approach to dynamic all pairs shortest paths. J. ACM 51(6), 968–992 (2004)
6. Freeman, L.C.: A set of measures of centrality based on betweenness. Sociometry 40(1), 35–41 (1977)
7. Frieze, A., Grimmett, G.: The shortest-path problem for graphs with random arclengths. Discrete Applied Mathematics 10(1), 57–77 (1985)
8. Geisberger, R., Sanders, P., Schultes, D.: Better approximation of betweenness centrality. In: Proc. ALENEX, pp. 90–100 (2008)
9. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proc. the National Academy of Sciences 99(12), 7821–7826 (2002)
10. Goel, K., Singh, R.R., Iyengar, S., Sukrit: A faster algorithm to update betweenness centrality after node alteration. In: Bonato, A., Mitzenmacher, M., Prałat, P. (eds.) WAW 2013. LNCS, vol. 8305, pp. 170–184. Springer, Heidelberg (2013)
11. Goh, K.-I., Oh, E., Kahng, B., Kim, D.: Betweenness centrality correlation in social networks. Phys. Rev. E 67, 017101 (2003)

12. Green, O., McColl, R., Bader, D.A.: A fast algorithm for streaming betweenness centrality. In: Proc. PASSAT, pp. 11–20 (2012)
13. Hassin, R., Zemel, E.: On shortest paths in graphs with random weights. Mathematics of Operations Research 10(4), 557–564 (1985)
14. Holme, P., Kim, B.J., Yoon, C.N., Han, S.K.: Attack vulnerability of complex networks. Phys. Rev. E 65, 056109 (2002)
15. Karger, D.R., Koller, D., Phillips, S.J.: Finding the hidden path: Time bounds for all-pairs shortest paths. SIAM J. Comput. 22(6), 1199–1217 (1993)
16. Kas, M., Wachs, M., Carley, K.M., Carley, L.R.: Incremental algorithm for updating betweenness centrality in dynamically growing networks. In: Proc. ASONAM, pp. 33–40. ACM (2013)
17. Kitsak, M., Havlin, S., Paul, G., Riccaboni, M., Pammolli, F., Stanley, H.E.: Betweenness centrality of fractal and nonfractal scale-free model networks and tests on real networks. Phys. Rev. E 75, 056115 (2007)
18. Kourtellis, N., Iamnitchi, A.: Leveraging peer centrality in the design of socially-informed peer-to-peer systems. CoRR, abs/1210.6052 (2012)
19. Kourtellis, N., Morales, G.D.F., Bonchi, F.: Scalable online betweenness centrality in evolving graphs. CoRR, abs/1401.6981 (2014)
20. Krebs, V.: Mapping networks of terrorist cells. Connections 24(3), 43–52 (2002)
21. Lee, M.-J., Lee, J., Park, J.Y., Choi, R.H., Chung, C.-W.: Qube: a quick algorithm for updating betweenness centrality. In: Proc. WWW, pp. 351–360 (2012)
22. Leydesdorff, L.: Betweenness centrality as an indicator of the interdisciplinarity of scientific journals. J. Am. Soc. Inf. Sci. Technol. 58(9), 1303–1319 (2007)
23. Luby, M., Ragde, P.: A bidirectional shortest-path algorithm with good average-case behavior. Algorithmica 4(1-4), 551–567 (1989)
24. Madduri, K., Ediger, D., Jiang, K., Bader, D.A., Chavarría-Miranda, D.G.: A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In: Proc. IPDPS, pp. 1–8 (2009)
25. Maglaras, L., Katsaros, D.: New measures for characterizing the significance of nodes in wireless ad hoc networks via localized path-based neighborhood analysis. Social Network Analysis and Mining 2(2), 97–106 (2012)
26. McGeoch, C.C.: All-pairs shortest paths and the essential subgraph. Algorithmica 13(5), 426–441 (1995)
27. Mehlhorn, K., Meyer, U.: External-memory breadth-first search with sublinear I/O. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 723–735. Springer, Heidelberg (2002)
28. Nasre, M., Pontecorvi, M., Ramachandran, V.: Decremental and fully dynamic all pairs all shortest paths and betweenness centrality. Manuscript (2014)
29. Pettie, S.: A new approach to all-pairs shortest paths on real-weighted graphs. Theoretical Computer Science 312(1), 47–74 (2004)
30. Pettie, S., Ramachandran, V.: A shortest path algorithm for real-weighted undirected graphs. SIAM J. Comput. 34(6), 1398–1431 (2005)
31. Pinney, J.W., McConkey, G.A., Westhead, D.R.: Decomposition of biological networks using betweenness centrality. In: Proc. RECOMB. Poster session (2005)
32. Puzis, R., Altshuler, Y., Elovici, Y., Bekhor, S., Shiftan, Y., Pentland, A.S.: Augmented betweenness centrality for environmentally aware traffic monitoring in transportation networks. J. of Intell. Transpor. Syst. 17(1), 91–105 (2013)
33. Ramírez: The social networks of academic performance in a student context of poverty in Mexico. Social Networks 26(2), 175–188 (2004)
34. Singh, B.K., Gupte, N.: Congestion and decongestion in a communication network. Phys. Rev. E 71, 055103 (2005)