Incremental Algorithms for Closeness Centrality

Ahmet Erdem Sarıyüce[1,2], Kamer Kaya[1], Erik Saule[1], Ümit V. Çatalyürek[1,3]

Depts. [1]Biomedical Informatics, [2]Computer Science and Engineering, [3]Electrical and Computer Engineering
The Ohio State University
Email:*sariyuce.1@osu.edu, kamer@bmi.osu.edu, esaule@uncc.edu, umit@bmi.osu.edu*

***Abstract*—Centrality metrics have shown to be highly correlated with the importance and loads of the nodes within the network traffic. In this work, we provide fast incremental algorithms for closeness centrality computation. Our algorithms efficiently compute the closeness centrality values upon changes in network topology, i.e., edge insertions and deletions. We show that the proposed techniques are efficient on many real-life networks, especially on small-world networks, which have a small diameter and spike-shaped shortest distance distribution. We experimentally validate the efficiency of our algorithms on large-scale networks and show that they can update the closeness centrality values of 1.2 million authors in the temporal DBLP-coauthorship network 460 times faster than it would take to recompute them from scratch.**

***Keywords*-closeness centrality; dynamic networks; small-world networks**

I. Introduction

Centrality metrics, such as closeness or betweenness, quantify how central a node is in a network. They have been successfully used to carry analysis for various purposes such as structural analysis of knowledge networks [14, 18], power grid contingency analysis [7], quantifying importance in social networks [12], analysis of covert networks [9], and even for finding the best store locations in cities [15]. Several works on rapid computation of these metrics exist in the literature. The algorithm with the best time complexity to compute centrality metrics [2] is believed to be asymptotically optimal [8]. Research have focused on either approximation algorithms for computing centrality metrics [3, 4, 13] or on high performance computing techniques [11, 19]. Today, the networks one needs to analyze can be quite large and dynamic, and better analysis techniques are always required.

In a dynamic and streaming network, ensuring the correctness of the centralities is a challenging task [5, 10]. Furthermore, for some applications involving a static network such as the contingency analysis of power grids and robustness evaluation of networks, to be prepared and take proactive measures, we need to know how the centrality values change when the network topology is modified by an adversary or outer effects such as natural disasters. As Figure 1 shows, the effect of a local topology modification is usually global. To quantify these effects and find exact centrality scores, existing algorithms are not efficient enough to be used in practice. Novel, incremental algorithms are essential to quickly evaluate the effects of topology modifications on centrality values.

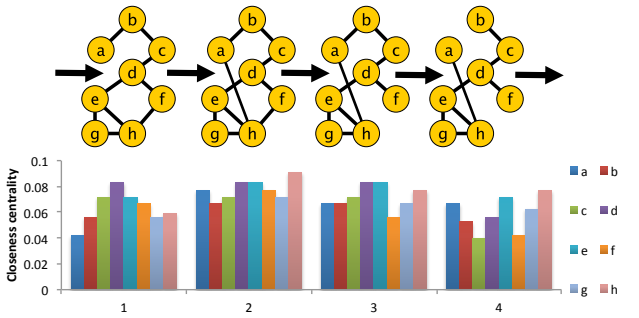

Figure 1. A toy network with eight nodes, three consecutive edge (ah, fh, and ab, respectively) insertions/deletions, and values of closeness centrality.

Our main contributions are incremental algorithms which efficiently update the closeness centralities upon edge insertions and deletions. Compared with the existing algorithms, our algorithms have a low-memory footprint which makes them practical and applicable to very large graphs. For random edge insertions/deletions to the Wikipedia users' communication graph, we reduced the centrality (re)computation time from 2 days to 16 minutes. And for the real-life temporal DBLP coauthorship network, we reduced the time from 1.3 days to 4.2 minutes.

The rest of the paper is organized as follows: Section II introduces the notation and the closeness centrality metric. Our algorithms are explained in detail in Section III. Related works are given in Section IV. An experimental analysis is given in Section V. Section VI concludes the paper.

II. Background

Let $G = (V, E)$ be a network modeled as a simple graph with $n = |V|$ vertices and $m = |E|$ edges where each node is represented by a vertex in V, and a node-node interaction is represented by an edge in E. Let $\Gamma_G(v)$ be the set of vertices which are connected to v.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A *path* is a sequence of vertices such that there exists an edge between each consecutive vertex pair. A path between two vertices s and t is denoted by $s \rightsquigarrow t$ (or $s \overset{P}{\rightsquigarrow} t$ if a specific path P with endpoints s and t is mentioned). Two vertices $u, v \in V$ are *connected* if

there is a path between u and v. If all vertex pairs in G are connected we say that G is *connected*. Otherwise, it is *disconnected* and each maximal connected subgraph of G is a *connected component*, or a component, of G. We use $d_G(u,v)$ to denote the length of the shortest path between two vertices u, v in a graph G. If $u = v$ then $d_G(u,v) = 0$. If u and v are disconnected, then $d_G(u,v) = \infty$.

Given a graph $G = (V, E)$, a vertex $v \in V$ is called an *articulation vertex* if the graph $G - v$ (obtained by removing v) has more connected components than G. Similarly, an edge $e \in E$ is called a *bridge* if $G - e$ (obtained by removing e from E) has more connected components than G. G is *biconnected* if it is connected and it does not contain an articulation vertex. A maximal biconnected subgraph of G is a *biconnected component*.

A. Closeness Centrality

Given a graph G, the *farness* of a vertex u is defined as

$$\texttt{far}[u] = \sum_{\substack{v \in V \\ d_G(u,v) \neq \infty}} d_G(u,v).$$

And the closeness centrality of u is defined as

$$\texttt{cc}[u] = \frac{1}{\texttt{far}[u]}. \quad (1)$$

If u cannot reach any vertex in the graph $\texttt{cc}[u] = 0$.

For a sparse unweighted graph $G = (V, E)$ the complexity of cc computation is $\mathcal{O}(n(m+n))$ [2]. For each vertex $s \in V$, Algorithm 1 executes a Single-Source Shortest Paths (SSSP), i.e., it initiates a breadth-first search (BFS) from s, computes the distances to the other vertices and $\texttt{far}[s]$, the sum of the distances which are different than ∞. As the last step, it computes $\texttt{cc}[s]$. Since a BFS takes $\mathcal{O}(m+n)$ time, and n SSSPs are required in total, the complexity follows.

Algorithm 1: CC: Basic centrality computation

```
Data: G = (V, E)
Output: cc[.]
1 for each s ∈ V do
      ▷SSSP(G, s) with centrality computation
      Q ← empty queue
      d[v] ← ∞, ∀v ∈ V \ {s}
      Q.push(s), d[s] ← 0
      far[s] ← 0
      while Q is not empty do
          v ← Q.pop()
          for all w ∈ Γ_G(v) do
              if d[w] = ∞ then
                  Q.push(w)
                  d[w] ← d[v] + 1
                  far[s] ← far[s] + d[w]
      cc[s] = 1 / far[s]
return cc[.]
```

III. Maintaining Centrality

Many real-life networks are scale free. The diameters of these networks grow proportional to the logarithm of the number of nodes. That is, even with hundreds of millions of vertices, the diameter is small, and when the graph is modified with minor updates, it tends to stay small. Combining this with the power-law degree distribution of scale-free networks, we obtain the spike-shaped shortest-distance distribution as shown in Figure 2. We use *work filtering with level differences* and *utilization of special vertices* to exploit these observations and reduce the centrality computation time. In addition, we apply *SSSP hybridization* to speedup each SSSP computation.

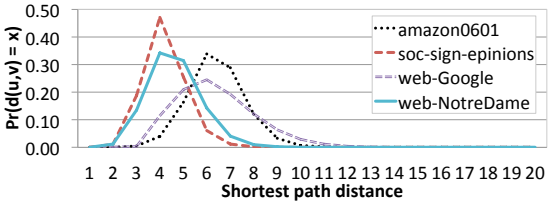

Figure 2. The probability of the distance between two (connected) vertices is equal to x for four social and web networks.

A. Work Filtering with Level Differences

For efficient maintenance of the closeness centrality values in case of an edge insertion/deletion, we propose a *work filter* which reduces the number of SSSPs in Algorithm 1 and the cost of each SSSP by utilizing the level differences.

Level-based filtering detects the unnecessary updates and filter them out. Let $G = (V, E)$ be the current graph and uv be an edge to be inserted to G. Let $G' = (V, E \cup uv)$ be the updated graph. The centrality definition in (1) implies that for a vertex $s \in V$, if $d_G(s,t) = d_{G'}(s,t)$ for all $t \in V$ then $\texttt{cc}[s] = \texttt{cc}'[s]$. The following theorem is used to detect such vertices and filter their SSSPs.

Theorem 1: Let $G = (V, E)$ be a graph and u and v be two vertices in V s.t. $uv \notin E$. Let $G' = (V, E \cup uv)$. Then $\texttt{cc}[s] = \texttt{cc}'[s]$ if and only if $|d_G(s,u) - d_G(s,v)| \leq 1$.

Proof: If s is disconnected from u and v, uv's insertion will not change $\texttt{cc}[s]$. Hence, $\texttt{cc}[s] = \texttt{cc}'[s]$. If s is only connected to one of u and v in G the difference $|d_G(s,u) - d_G(s,v)|$ is ∞, and $\texttt{cc}[s]$ needs to be updated by using the new, larger connected component containing s. When s is connected to both u and v in G, we investigate the edge insertion in three cases as shown in Figure 3:

Case 1: $d_G(s,u) = d_G(s,v)$: Assume that the path $s \overset{P}{\rightsquigarrow} u$–$v \overset{P'}{\rightsquigarrow} t$ is a shortest $s \rightsquigarrow t$ path in G' containing uv. Since $d_G(s,u) = d_G(s,v)$, there exists a shorter path $s \overset{P''}{\rightsquigarrow} v \overset{P'}{\rightsquigarrow} t$ with one less edge. Hence, $\forall t \in V$, $d_G(s,t) = d_{G'}(s,t)$.

Case 2: $|d_G(s,u) - d_G(s,v)| = 1$: Let $d_G(s,u) < d_G(s,v)$. Assume that $s \overset{P}{\rightsquigarrow} u$–$v \overset{P'}{\rightsquigarrow} t$ is a shortest path in G' containing uv. Since $d_G(s,v) = d_G(s,u) + 1$,

there exists another path $s \overset{P''}{\rightsquigarrow} v \overset{P'}{\rightsquigarrow} t$ with the same length. Hence, $\forall t \in V$, $d_G(s,t) = d_{G'}(s,t)$.

Case 3: $|d_G(s,u) - d_G(s,v)| > 1$: Let $d_G(s,u) < d_G(s,v)$. The path $s \rightsquigarrow u$–v in G' is shorter than the shortest $s \rightsquigarrow v$ path in G since $d_G(s,v) > d_G(s,u) + 1$. Hence, $\forall t \in V \setminus \{v\}, d_{G'}(s,t) \leq d_G(s,t)$ and $d_{G'}(s,v) < d_G(s,v)$, i.e., an update on $\mathtt{cc}[s]$ is necessary. ■

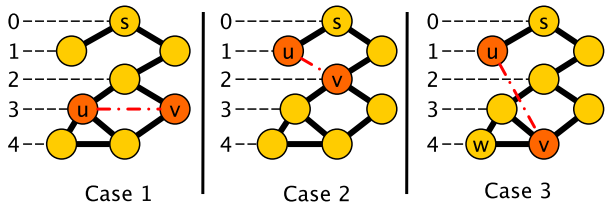

Figure 3. Three cases of edge insertion: when an edge uv is inserted to the graph G, for each vertex s, one of them is true: (1) $d_G(s,u) = d_G(s,v)$, (2) $|d_G(s,u) - d_G(s,v)| = 1$, and (3) $|d_G(s,u) - d_G(s,v)| > 1$.

Although Theorem 1 yields to a filter only in case of edge insertions, the following corollary which is used for edge deletion easily follows.

Corollary 2: Let $G = (V,E)$ be a graph and u and v be two vertices in V s.t. $uv \in E$. Let $G' = (V, E \setminus \{uv\})$. Then $\mathtt{cc}[s] = \mathtt{cc}'[s]$ if and only if $|d_{G'}(s,u) - d_{G'}(s,v)| \leq 1$.

With this corollary, the work filter can be implemented for both edge insertions and deletions. The pseudocode of the update algorithm in case of an edge insertion is given in Algorithm 2. When an edge uv is inserted/deleted, to employ the filter, we first compute the distances from u and v to all other vertices. And, we filter the vertices satisfying the statement of Theorem 1.

Algorithm 2: Simple work filtering

```
Data: G = (V, E), cc[.], uv
Output: cc'[.]
G' ← (V, E ∪ {uv})
du[.] ← SSSP(G, u) ▷ distances from u in G
dv[.] ← SSSP(G, v) ▷ distances from v in G
for each s ∈ V do
    if |du[s] − dv[s]| ≤ 1 then
        cc'[s] = cc[s]
    else
        ▷ use the computation in Algorithm 1
          with G'
return cc'[.]
```

B. Utilization of Special Vertices

We exploit some special vertices to speedup the incremental closeness centrality computation further. We leverage the articulation vertices and identical vertices in networks. Although it has been previously shown that articulation vertices in real social networks are limited and yield an unbalanced shattering [17], we present the related techniques here to give a complete view.

1) Filtering with biconnected components: Our filter can be assisted by maintaining a biconnected component decomposition (BCD) of $G = (V,E)$. A BCD is a partitioning Π of E where $\Pi(e)$ is the component of each edge $e \in E$. When uv is inserted to G and $G' = (V, E' = E \cup \{uv\})$ is obtained, we check if

$$\{\Pi(uw) : w \in \Gamma_G(u)\} \cap \{\Pi(vw) : w \in \Gamma_G(v)\}$$

is empty or not: if the intersection is not empty, there will be only one element in it, cid, which is the id of the biconnected component of G' containing uv (otherwise Π is not a valid BCD). In this case, $\Pi'(e)$ is set to $\Pi(e)$ for all $e \in E$ and $\Pi'(uv)$ is set to cid. If there is no biconnected component containing both u and v , i.e., if the intersection above is empty, we construct Π' from scratch and set $cid = \Pi'(uv)$. Π can be computed in linear, $\mathcal{O}(m+n)$ time [6]. Hence, the cost of BCD maintenance is negligible compared to the cost of updating closeness centrality. Details can be found in [16].

2) Filtering with identical vertices: Our preliminary analyses show that real-life networks can contain a significant amount of *identical* vertices with the same/a similar neighborhood structure. We investigate two types of identical vertices.

Definition 3: In a graph G, two vertices u and v are *type-I-identical if and only if* $\Gamma_G(u) = \Gamma_G(v)$.

Definition 4: In a graph G, two vertices u and v are *type-II-identical if and only if* $\{u\} \cup \Gamma_G(u) = \{v\} \cup \Gamma_G(v)$.

Both types form an equivalance class relation since they are reflexive, symmetric, and transitive. Hence, all the classes they form are disjoint.

Let $u, v \in V$ be two identical vertices. One can see that for any vertex $w \in V \setminus \{u,v\}$, $d_G(u,w) = d_G(v,w)$. Then the following is true.

Corollary 5: Let $\mathcal{I} \subseteq V$ be a vertex-class containing type-I or type-II identical vertices. Then the closeness centrality values of all the vertices in $\mathcal{I}$ are equal.

C. SSSP Hybridization

The spike-shaped distribution given in Figure 2 can also be exploited for SSSP hybridization. Consider the execution of Algorithm 1: while executing an SSSP with source s, for each vertex pair $\{u,v\}$, u is processed before v if and only if $d_G(s,u) < d_G(s,v)$. That is, Algorithm 1 consecutively uses the vertices with distance k to find the vertices with distance $k+1$. Hence, it visits the vertices in a *top-down* manner. SSSP can also be performed in a a *bottom-up* manner. That is to say, after all distance (level) k vertices are found, the vertices whose levels are unknown can be processed to see if they have a neighbor at level k. The top-down variant is expected to be much cheaper for small k values. However, it can be more expensive for the upper levels where there are much less unprocessed vertices remaining.

Following the idea of Beamer et al. [1], we hybridize the SSSPs. While processing the nodes at an SSSP level, we

simply compare the number of edges need to be processed for each variant and choose the cheaper one.

IV. Related Work

To the best of our knowledge, there are only two works on maintaining centrality in dynamic networks. Yet, both are interested in betweenness centrality. Lee et al. proposed the QUBE framework which uses a BCD and updates the betweenness centrality values in case of edge insertions and deletions in the network [10]. Unfortunately, the performance of QUBE is only reported on small graphs (less than 100K edges) with very low edge density. In other words, it only performs significantly well on small graphs with a tree-like structure having many small biconnected components.

Green et al. proposed a technique to update the betweenness centrality scores rather than recomputing them from scratch upon edge insertions (can be extended to edge deletions) [5]. The idea is to store the whole data structure used by the previous computation. However, as the authors stated, it takes $\mathcal{O}(n^2 + nm)$ space to store all the required values. Compared to their work, our algorithms are much more practical since the memory footprint of linear.

V. Experimental Results

We implemented the algorithms in `C` and compiled with `gcc v4.6.2` with the optimization flags `-O2 -DNDEBUG`. The graphs are kept in the compressed row storage (CRS) format. The experiments are run in sequential on a computer with two Intel Xeon E5520 CPU clocked at 2.27GHz and equipped with 48GB of main memory.

For the experiments, we used 10 networks from the UFL Sparse Matrix Collection[1] and also extracted the coauthor network from the current set of DBLP papers. Properties of the graphs are summarized in Table I. They are from different application areas, such as social (*hep-th*, *PGPgiantcompo*, *astro-ph*, *cond-mat-2005*, *soc-sign-epinions*, *loc-gowalla*, *amazon0601*, *wiki-Talk*, *DBLP-coauthor*), and web networks (*web-NotreDame*, *web-Google*). The graphs are listed by increasing number of edges and a distinction is made between small graphs (with less than 500K edges) and the large graphs (with more than 500K) edges.

Although the filtering techniques can reduce the update cost significantly in theory, their practical effectiveness depends on the underlying structure of G. Since the diameter of the social networks are small, the range of the shortest distances is small. Furthermore, the distribution of these distances is unimodal. When the distance with the peak (mode) is combined with the ones on its right and left, they cover a significant amount of the pairs (56% for *web-NotreDame*, 65% for *web-Google*, 79% for *amazon0601*, and 91% for *soc-sign-epinions*). We expect the filtering procedure to have a significant impact on social networks because of their structure. Besides, that specific structure is also important for the SSSP hybridization.

Graph			**Time** (in sec.)		
name	$\|V\|$	$\|E\|$	Org.	Best	Speedup
hep-th	8.3K	15.7K	1.41	0.05	29.4
PGPgiantcompo	10.6K	24.3K	4.96	0.04	111.2
astro-ph	16.7K	121.2K	14.56	0.36	40.5
cond-mat-2005	40.4K	175.6K	77.90	2.87	27.2
				geometric mean	**43.5**
soc-sign-epinions	131K	711K	778	6.25	124.5
loc-gowalla	196K	950K	2,267	53.18	42.6
web-NotreDame	325K	1,090K	2,845	53.06	53.6
amazon0601	403K	2,443K	14,903	298	50.0
web-Google	875K	4,322K	65,306	824	79.2
wiki-Talk	2,394K	4,659K	175,450	922	190.1
DBLP-coauthor	1,236K	9,081K	115,919	251	460.8
				geometric mean	**99.8**

Table I
THE GRAPHS USED IN THE EXPERIMENTS. COLUMN *Org.* SHOWS THE INITIAL CLOSENESS COMPUTATION TIME OF CC AND *Best* IS THE BEST UPDATE TIME WE OBTAIN IN CASE OF STREAMING DATA.

A. Handling topology modifications

To assess the effectiveness of our algorithms, we need to know when each edge is inserted to/deleted from the graph. Our datasets from the UFL collection do not have this information. To conduct our experiments on these datasets, we delete 1,000 edges from a graph chosen randomly in the following way: A vertex $u \in V$ is selected randomly (uniformly), and a vertex $v \in \Gamma_G(u)$ is selected randomly (uniformly). Since we do not want to change the connectivity in the graph (having disconnected components can make our algorithms much faster and it will not be fair to CC), we discard uv if it is a bridge. If this is not the case we delete it from G and continue. We construct the initial graph by deleting these 1,000 edges. Each edge is then re-inserted one by one, and our algorithms are used to recompute the closeness centrality scores after each insertion.

In addition to the random insertion experiments, we also evaluated our algorithms on a real temporal dataset of the DBLP coauthor graph[2]. In this graph, there is an edge between two authors if they published a paper together. We used the publication dates as timestamps and constructed the initial graph with the papers published before January 1, 2013. We used the coauthorship edges of the later papers for edge insertions. Although we used insertions in our experiments, a deletion is a very similar process which should give comparable results.

In addition to CC, we configure our algorithms in four different ways: CC-B only uses BCD, CC-BL uses BCD and filtering with levels, CC-BLI uses all three work filtering techniques including identical vertices. And CC-BLIH uses all the techniques described in this paper including the SSSP hybridization.

Table II presents the results of the experiments. The second column, CC, shows the time to run the full base

[1] http://www.cise.ufl.edu/research/sparse/matrices/

[2] http://www.informatik.uni-trier.de/~ley/db/

algorithm for computing the closeness centrality values on the original version of the graph. Columns 3–6 of the table present absolute runtimes (in seconds) of the centrality computation algorithms. The next four columns, 7–10, give the speedups achieved by each configuration. For instance, on the average, updating the closeness values by using CC-B on *PGPgiantcompo* is 11.5 times faster than running CC. Finally the last column gives the overhead of our algorithms per edge insertion, i.e., the time necessary to filter the source vertices and to maintain BCD and identical-vertex classes. Geometric means of these times and speedups are also given to provide a comparison across all the instances.

The times to compute the closeness values using CC on the small graphs range between 1 to 77 seconds. On large graphs, the times range from 13 minutes to 49 hours. Clearly, CC is not suitable for real-time network analysis and management based on shortest paths and closeness centrality. When all the techniques are used (CC-BLIH), the time necessary to update the closeness centrality values of the small graphs drops below 3 seconds per edge insertion. The improvements range from a factor of 27.2 (*cond-mat-2005*) to 111.2 (*PGPgiantcompo*), with an average improvement of 43.5 across small instances and a factor of 42.6 (*loc-gowalla*) to 458.8 (*DBLP-coauthor*), on large graphs, with an average of 99.7. For all graphs, the time spent for overheads is below one second which indicates that the majority of the time is spent for SSSPs. Note that this part is pleasingly parallel since each SSSP is independent from each other. Hence, by combining the techniques proposed in this work with a straightforward parallelism, one can obtain a framework that can maintain the closeness centrality values within a dynamic network in real time.

The overall improvement obtained by the proposed algorithms is significant. The speedup obtained by using BCDs (CC-B) are 3.5 and 3.2 on the average for small and large graphs, respectively. The graphs *PGPgiantcompo*, and *wiki-Talk* benefits the most from BCDs (with speedups 11.5 and 6.8, respectively). Clearly using the biconnected component decomposition improves the update performance. However, filtering by level differences is the most efficient technique: CC-BL brings major improvements over CC-B. For all social networks, when CC-BL is compared with CC-B, the speedups range from 4.8 (*web-NotreDame*) to 64 (*DBLP-coauthor*). Overall, CC-BL brings a 7.61 times improvement on small graphs and a 13.44 times improvement on large graphs over CC.

For each added edge uv, let X be the random variable equal to $|d_G(u,w) - d_G(v,w)|$. By using 1,000 uv edges, we computed the probabilities of the three cases we investigated before and give them in Fig. 4. For each graph in the figure, the sum of the first two columns gives the ratio of the vertices not updated by CC-BL. For the networks in the figure, not even 20% of the vertices require an update ($Pr(X > 1)$). This explains the speedup achieved by filtering using level differences. Therefore, level filtering is more useful for the graphs having characteristics similar to small-world networks.

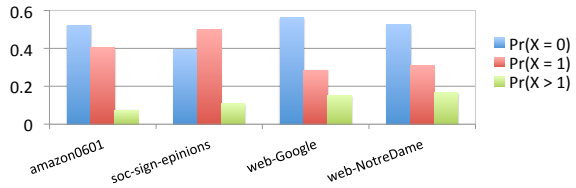

Figure 4. The bars show the distribution of random variable $X = |d_G(u,w) - d_G(v,w)|$ into three cases we investigated when an edge uv is added.

Filtering with identical vertices is not as useful as the other two techniques in the work filter. Overall, there is a 1.15 times improvement with CC-BLI on both small and large graphs compared to CC-BL. For some graphs, such as *web-NotreDame* and *web-Google*, improvements are much higher (30% and 31%, respectively).

The algorithm with the hybrid SSSP implementation, CC-BLIH, is faster than CC-BLI by a factor of 1.42 on small graphs and by a factor of 1.96 on large graphs. Although it seems to improve the performance for all graphs, in some few cases, the performance is not improved significantly. This can be attributed to incorrect decisions on SSSP variant to be used. Indeed, we did not benchmark the architecture to discover the proper parameter. CC-BLIH performs the best on social network graphs with an improvement ratio of 3.18 (*soc-sign-epinions*), 2.54 (*loc-gowalla*), and 2.30 (*wiki-Talk*).

All the previous results present the average single edge update time for 1,000 successively added edges. Hence, they do not say anything about the variance. Figure 5 shows the runtimes of CC-B and CC-BLIH per edge insertion for *web-NotreDame* in a sorted order. The runtime distribution of CC-B clearly has multiple modes. Either the runtime is lower than 100 milliseconds or it is around 700 seconds. We see here the benefit of BCD. According to the runtime distribution, about 59% of *web-NotreDame*'s vertices are inside small biconnected components. Hence, the time per edge insertion drops from 2,845 seconds to 700. Indeed, the largest component only contains 41% of the vertices and 76% of the edges of the original graph. The decrease in the size of the components accounts for the gain of performance.

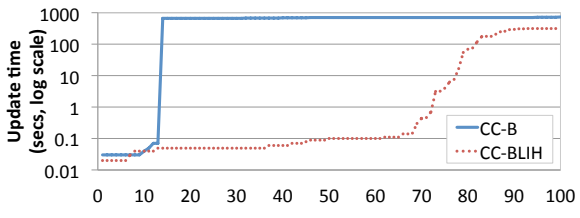

Figure 5. Sorted list of the runtimes per edge insertion for the first 100 added edges of *web-NotreDame*.

Graph	Time (secs)					Speedups				Filter time (secs)
	CC	CC-B	CC-BL	CC-BLI	CC-BLIH	CC-B	CC-BL	CC-BLI	CC-BLIH	
hep-th	1.413	0.317	0.057	0.053	0.048	4.5	24.8	26.6	29.4	0.001
PGPgiantcompo	4.960	0.431	0.059	0.055	0.045	11.5	84.1	89.9	111.2	0.001
astro-ph	14.567	9.431	0.809	0.645	0.359	1.5	18.0	22.6	40.5	0.004
cond-mat-2005	77.903	39.049	5.618	4.687	2.865	2.0	13.9	16.6	27.2	0.010
Geometric mean	9.444	2.663	0.352	0.306	0.217	3.5	26.8	30.7	43.5	0.003
soc-sign-epinions	778.870	257.410	20.603	19.935	6.254	3.0	37.8	39.1	124.5	0.041
loc-gowalla	2,267.187	1,270.820	132.955	135.015	53.182	1.8	17.1	16.8	42.6	0.063
web-NotreDame	2,845.367	579.821	118.861	83.817	53.059	4.9	23.9	33.9	53.6	0.050
amazon0601	14,903.080	11,953.680	540.092	551.867	298.095	1.2	27.6	27.0	50.0	0.158
web-Google	65,306.600	22,034.460	2,457.660	1,701.249	824.417	3.0	26.6	38.4	79.2	0.267
wiki-Talk	175,450.720	25,701.710	2,513.041	2,123.096	922.828	6.8	69.8	82.6	190.1	0.491
DBLP-coauthor	115,919.518	18,501.147	288.269	251.557	252.647	6.2	402.1	460.8	458.8	0.530
Geometric mean	13,884.152	4,218.031	315.777	273.036	139.170	3.2	43.9	50.8	99.7	0.146

Table II

EXECUTION TIMES IN SECONDS OF ALL THE ALGORITHMS AND SPEEDUPS WHEN COMPARED WITH THE BASIC CLOSENESS CENTRALITY ALGORITHM CC. IN THE TABLE CC-B IS THE VARIANT WHICH USES ONLY BCDS, CC-BL USES BCDS AND FILTERING WITH LEVELS, CC-BLI USES ALL THREE WORK FILTERING TECHNIQUES INCLUDING IDENTICAL VERTICES. AND CC-BLIH USES ALL THE TECHNIQUES DESCRIBED IN THIS PAPER INCLUDING SSSP HYBRIDIZATION.

The impact of level filtering can also be seen on Figure 5. 60% of the edges in the main biconnected component do not change the closeness values of many vertices and the updates that are induced by their addition take less than 1 second. The remaining edges trigger more expensive updates upon insertion. Within these 30% expensive edge insertions, using identical vertices and SSSP hybridization provide a significant improvement (not shown in the figure).

Better Speedups on Real Temporal Data: The best speedups are obtained on the DBLP coauthor network which uses real temporal data. Using CC-B, we reach 6.2 speedup w.r.t. CC, which is bigger than the average speedup on all networks. Main reason for this behavior is that 10% of the inserted edges are actually the new vertices joining to the network, i.e., authors with their first publication, and CC-B handles these edges quite fast. Applying CC-BL gives a 64.8 speedup over CC-B, which is drastically higher than all other graphs. Indeed, only 0.7% of the vertices require to run a SSSP algorithm when an edge is inserted on the DBLP network. For the synthetic cases, this number is 12%. Overall, speedups obtained with real temporal data reach 460.8, i.e., 4.6 times greater than the average speedup on all graphs. Our algorithms appear to perform much better on real applications than on synthetic ones.

VI. CONCLUSION

In this paper, we propose the first algorithms to achieve fast updates of exact closeness centrality values on incremental network modification at such a large scale. Our techniques exploit the spike-shaped shortest-distance distributions of these networks, their biconnected component decomposition, and the existence of nodes with identical neighborhood. In large networks with more than $500K$ edges, the proposed techniques bring 99 times speedup on average. For the temporal DBLP coauthorship graph, which has the most edges, we reduced the centrality update time from 1.3 days to 4.2 minutes.

VII. ACKNOWLEDGMENTS

This work was partially supported by the NHI/NCI grant R01CA141090; the NSF grant OCI-0904809; and the NPRP grant 4-1454-1-233 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

REFERENCES

[1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proc. of Supercomputing*, 2012.
[2] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.
[3] S. Y. Chan, I. X. Y. Leung, and P. Liò. Fast centrality approximation in modular networks. In *Proc. of CIKM-CNIKM*, 2009.
[4] D. Eppstein and J. Wang. Fast approximation of centrality. In *Proc. of SODA*, 2001.
[5] O. Green, R. McColl, and D. A. Bader. A fast algorithm for streaming betweenness centrality. In *Proc. of SocialCom*, 2012.
[6] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, June 1973.
[7] S. Jin, Z. Huang, Y. Chen, D. G. Chavarría-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *Proc. of IPDPS*, 2010.
[8] S. Kintali. Betweenness centrality : Algorithms and lower bounds. *CoRR*, abs/0809.1906, 2008.
[9] V. Krebs. Mapping networks of terrorist cells. *Connections*, 24, 2002.
[10] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung. QUBE: a Quick algorithm for Updating BEtweenness centrality. In *Proc. of WWW*, 2012.
[11] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarría-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proc. of IPDPS*, 2009.
[12] E. L. Merrer and G. Trédan. Centralities: Capturing the fuzzy notion of importance in social graphs. In *Proc. of SNS*, 2009.
[13] K. Okamoto, W. Chen, and X.-Y. Li. Ranking of closeness centrality for large-scale social networks. In *Proc. of FAW*, 2008.
[14] M. C. Pham and R. Klamma. The structure of the computer science knowledge network. In *Proc. of ASONAM*, 2010.
[15] S. Porta, V. Latora, F. Wang, E. Strano, A. Cardillo, S. Scellato, V. Iacoviello, and R. Messora. Street centrality and densities of retail and services in Bologna, Italy. *Environment and Planning B: Planning and Design*, 36(3):450–465, 2009.
[16] A. E. Sarıyüce, K. Kaya, E. Saule, and Ümit V. Çatalyürek. Incremental algorithms for network management and analysis based on closeness centrality. *CoRR*, abs/1303.0422, 2013.
[17] A. E. Sarıyüce, E. Saule, K. Kaya, and Ümit V. Çatalyürek. Shattering and compressing networks for betweenness centrality. In *Proc. of SDM*, 2013.
[18] X. Shi, J. Leskovec, and D. A. McFarland. Citing for high impact. In *Proc. of JCDL*, 2010.
[19] Z. Shi and B. Zhang. Fast network centrality analysis using GPUs. *BMC Bioinformatics*, 12:149, 2011.