2013 IEEE 13th International Conference on Data Mining

An Efficient Approach to Updating Closeness Centrality and Average Path Length in Dynamic Networks

Chia-Chen Yen[†], Mi-Yen Yeh[‡,§], Ming-Syan Chen[†,‡,§]

[†]*Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan*
[‡]*Institute of Information Science, Academia Sinica, Taipei, Taiwan*
[§]*Research Center for Information Technology Innovation, Academia Sinica, Taipei, Taiwan*
ccyen@arbor.ee.ntu.edu.tw, miyen@iis.sinica.edu.tw, mschen@citi.sinica.edu.tw

***Abstract*—Closeness centrality measures the communication efficiency of a specific vertex within a network while the average path length (APL) measures that of the whole network. Since the nature of these two measurements is based on the computation of all-pair shortest path distances, one can perform the breadth-first search method starting at every vertex and obtain the two measurements. However, as the edge counts in the real-world networks like Facebook increase over time, this naive way is obviously inefficient. In this paper, we proposed CENDY, an efficient approach to updating Closeness centrality and avErage path leNgth in DYnamic networks when there is an edge insertion or deletion. In CENDY, we derived some theoretical properties to quickly identify a set of vertices whose shortest path changed after an edge update, and then update the closeness centralities of those vertices only as well as the APL of the graph by a few of single-source shortest path computations. We conducted extensive experiments to show that, when compared to the existing methods of computing exact or approximate values, CENDY outperformed others in significantly low update time while providing exact values of the two measurements on various real-world graph datasets.**

***Keywords*—Closeness centrality; average path length; update algorithm; dynamic networks;**

I. Introduction

Closeness centrality of a specific vertex, which is defined as the reciprocal of the sum of shortest path distances from it to all others in a network, is one of the most important measurements to evaluate its communication efficiency (or how fast it can spread information to the whole network). For example, the vertices can be identified whether they are important boundary spanners in a knowledge network [12], key actors in a terrorist network [18], distinctive actors in a policy network [16], failure-prone software modules in a contribution network [20], authority experts [19] or information propagators [11] in social networks, or sensors with low transmission delay in mobile sensor networks [24]. To further measure network topological structure, a natural measurement is to compute *Average path length* (APL), which is defined as the average number of steps along the shortest paths for all possible vertex pairs in the same network. It also reflects the communication efficiency among vertices of the given network. For example, the APL can be used to estimate the transfer efficiency between all of metabolite pairs in a metabolic network [6], to analyze the topology of real world networks such as World Wide Web [25], or to measure the time needed for evaluating the function in binary decision diagrams [7].

We can see that the common nature of computing the two measures is based on the computation of the all-pair shortest path (APSP) distances in the network. Given a simple undirected connected network, one can find APSP based on the breadth-first search (BFS) method. The time complexity is $O(n(m+n))$ [26], where n and m are the number of vertices and edges in the given graph, respectively. As the edge counts in real-world networks such Facebook increasing over time [9], the desire of an efficient approach to handle the dynamic APSP problem is pressing.

Roditty and Zwick [5] presented an approximation of dynamic APSP algorithm in an undirected graph. For any vertex pair, they provided a query time of $O(t)$, where $t \leq m^{1/2-\gamma}$ and γ is pre-defined parameter. Later, Thorup [4] delivered a solution of a query time of $O(1)$ to answer the shortest distance of a given vertex pair. Those two methods were originally designed to query shortest distance in dynamic networks. If we do not know which vertex pair whose shortest distance has changed, the two methods require additional time of $O(tn^2)$ and $O(n^2)$, respectively, to sum over all shortest path distances for updating the two measures in practice.

To speed up the computation of closeness centrality, some approximate methods with random sampling technique have been proposed in the past decade [15] [13] [17] [14] [21] [22] [23] [10]. Lee et al. [1] used a snowball sampling strategy to approximate the APL of a network. Such approximations of these two measures may lead to an imprecise quantitative analysis of a network. Moreover, all of these methods are designed for static networks. If we apply these methods in a dynamic network, we need to re-consider all vertices to compute these two measures after each edge update, which still leads high computational time. In fact, an edge update may not necessarily cause the shortest paths of all vertex pairs to change. Thus, if we can specifically identify which shortest paths between which vertex pairs are affected due to some new edge connection or disconnection in a network, we can efficiently update the exact closeness centrality of each vertex and the APL of the dynamic network. However, performing such identification in a dynamic graph is far from trivial.

We use an example to illustrate the problem. Fig. 1(a) shows an un-weighted and undirected original graph G consisting of 14 vertices and 19 edges. Suppose an edge $e(a, b)$ is newly added to the graph as shown in Fig. 1(b). Obviously, the direct effect of this edge insertion changes the shortest path between a and b. Also, the shortest paths between the respective neighbors of a and b, such as $\{c, x\}$, change accordingly. However, the original shortest paths of some vertex pairs that do not pass through a and b are affected as well. For example, the original shortest path $\{u, l, o, w, r, s, v\}$ between the vertex

1550-4786/13 $31.00 © 2013 IEEE
DOI 10.1109/ICDM.2013.135

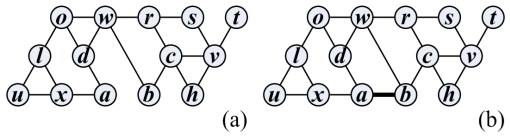

Fig. 1. Example of updating a graph

pair $\{u, v\}$, which does not pass through a or b, changes to $\{u, x, a, b, c, v\}$ due to the newly added edge. Furthermore, some shortest paths are not affected at all, such as the paths from vertex r to all other vertices. The above example shows that the identification of whether a shortest path between a vertex pair changes is not as intuitive as we think. Thus, we need a more systematic way to identify the change of these shortest paths so that we can update the exact value of the closeness centrality of each vertex and the APL of the dynamic network.

In this paper, we proposed an efficient approach, CENDY, to updating the Closeness centrality of each vertex and the avErage path leNgth of the monitored DYnamic network. We consider an undirected and un-weighted dynamic graph, where the number of vertices is fixed but the links between them may change (connect or disconnect) over time. We have an important insight of this problem: *given a graph G and a vertex v, if the difference between the shortest path distances from v to both end vertices of the edge to be inserted or deleted is greater than 1, then the shortest path from v to some vertices must change after an update.* Here, an update refers to an edge insertion or deletion only. Armed with this insight, we derive some theoretical properties to quickly identify a set of vertices whose shortest paths will change after an edge update, and then only update the closeness centralities of those vertices as well as the APL of the graph by only a few of BFS computations.

The main contributions of this paper can be summarized as follows. We first provided a solid analysis of changes in the single-source shortest path when an edge update occurs. Then, we derived two theorems to prove that we can identify the set of vertices whose shortest path changes by only *two* times of BFS on the graph without re-computing APSP distances. Once we identify the affected shortest paths, we can efficiently update the exact closeness centrality of the affected vertices and the APL of a network. Finally, we conduct extensive and comprehensive experiments to compare CENDY with the BFS method and those existing approximate methods for computing closeness centrality and the APL on various real graph datasets. The experimental results demonstrate that CENDY outperforms those algorithms in the updating time of the two measures when providing the exact values.

The remainder of this paper is organized as follows. We first define the problem and outline our main ideas in Section II. Section III presents the methodology and the time complexity of CENDY. In Section IV, we report the experiment results. Finally, we provide a survey of related studies in Section V and conclude the paper in Section VI.

II. Preliminaries

In this section, we first introduce the measurements to be monitored in a dynamic network and then describe the main concept of our idea to update the two measurements efficiently. All discussions are based on un-weighted and undirected graphs.

A. The Two Measures

We consider two measurements that are based on all-pair shortest path distances, the closeness centrality of each vertex and the average path length, in a given graph $G = \{V, E\}$.

Definition 1 (Shortest Path Distance): Given a vertex pair $u, v \in V$, the distance of any path connecting them is the number of edges of that path. The shortest path between u and v in G, denoted as $p(u, v)$, is the path with the minimum distance.

Definition 2 (Closeness Centrality): Given a graph $G = \{V, E\}$, the closeness centrality of a vertex $u \in V$ is defined as the reciprocal of the sum of shortest path distances from u to all other vertices v in G:

$$C_c(u) = \left[\sum_{v \in V, u \neq v} \frac{|p(u, v)|}{|V| - 1} \right]^{-1}, \tag{1}$$

where $|p(u, v)|$ is the distance of $p(u, v)$.

Definition 3 (Average Path Length): Given a graph $G = \{V, E\}$, the *average path length* is the average distances of the shortest paths for all possible vertex pairs:

$$L_G = \frac{\sum_{u,v \in V, u \neq v} |p(u, v)|}{|V|(|V| - 1)}. \tag{2}$$

B. Our Main Ideas

In a dynamic network, we consider two operations: edge insertion and edge deletion, one at a time. Since an edge insertion or deletion may cause changes in the shortest paths between some vertex pairs, the closeness centrality of some vertices and the APL of the whole network may change as well. Our goal is to obtain the updated values quickly by identifying only those affected shortest paths instead of recomputing all-pair shortest path distances. To achieve this goal, we introduce a specially designed data structure, BFS$^+$ graph, as follows.

Definition 4 (BFS$^+$ Graph): Given a graph $G = (V, E)$ and a vertex $r \in V$, the BFS$^+$ graph $\mathcal{G}_r$ is the breadth-first search (BFS) tree of G starting at vertex r plus two types of edges: the horizontal and interlevel edges. To be more specific, any two vertices at the same level of the BFS tree will be connected by a horizontal edge if they originally have a link in G. Also, for any two vertices at different levels of the BFS tree, we link them using an interlevel edge if they are originally linked in G. For each vertex v in $\mathcal{G}_r$, we associate it with a level number, starting from 0 for the root and incrementing one by one level. Therefore, the shortest distance $|p(r, v)|$ from the root r to the vertex v is equal to the level number of v. It is worth mentioning that the insertion or deletion of interlevel and horizontal edges does not change the level of any vertex in the BFS$^+$ graph as well as the distance from the root to all other vertices.

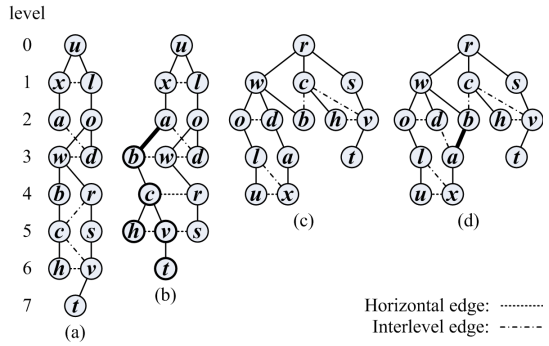

Fig. 2. The illustration of four BFS$^+$ graphs

Example 1: Fig. 2(a) is the BFS$^+$ graph of Fig. 1(a) rooted at u. Solid lines represent the edges in the BFS tree. Dotted lines are horizontal edges connecting the vertex pairs (x, l), (w, d) and (h, v). Dashed dotted lines are interlevel edges connecting (a, d), (r, c) and (c, v). Since the vertices a and d are in level 2 and 3, respectively, $|p(u, a)|$=2 and $|p(u, d)|$=3.

By observing the BFS$^+$ graph, we find that an edge insertion or deletion may cause level changes of some vertices. To reduce the computation cost of updating the two measurements, our intuition is to find a way that can quickly discover how many shortest paths will be affected due to these level-changed vertices.

Suppose we have a graph $G = (V, E)$ and its BFS$^+$ graph $\mathcal{G}_r$ rooted at some vertex r. The updated graph of G is denoted as $\acute{G}$, which is a graph after an edge insertion, i.e., $\acute{G} = G \cup e(a, b)$, or an edge deletion, i.e., $\acute{G} = G \backslash e(a, b)$, where $e(a, b)$ is the edge between vertex a and b. The BFS$^+$ graph rooted at the same vertex r of $\acute{G}$ is denoted as $\acute{\mathcal{G}}_r$.

Definition 5 (Unstable Vertex Pair): Given a vertex pair $\{v, u\} \in V$, we say it is an *unstable vertex pair* (or *unstable vertex* for each) if $|\acute{p}(v, u)|$, the new shortest path distance after an edge insertion or deletion between v and u, is not equal to the original $|p(v, u)|$. Otherwise, it is a *stable vertex pair* (or *stable vertex* for each).

Definition 6 (r-Unstable Vertex Set): Given a BFS$^+$ graph $\mathcal{G}_r$ and the updated one $\acute{\mathcal{G}}_r$, the r-unstable vertex set contains all the vertices $v \in V$ whose level changes in $\acute{\mathcal{G}}_r$, denoted as $\acute{\mathcal{V}}_r = \{v||\acute{p}(r, v)| \neq |p(r, v)|\}$. If $\acute{\mathcal{V}}_r = \emptyset$, we say r is a stable vertex, i.e., $|\acute{p}(r, v)| = |p(r, v)|$ for each vertex $v \in V$.

By identifying the unstable vertex pairs, we only need to take few more computations of the single-source shortest paths among those vertices instead of recomputing the shortest paths of all vertex pairs.

III. Methodology

In this section, we first show how to discover the unstable vertex pairs when there is an edge insertion or deletion. Then we show how to compute the distance changes of the shortest path between these pairs.

A. Unstable Vertex Discovery of An Edge Insertion

Let the graph shown in Fig. 1(a) be the original graph G and Fig. 1(b) be the updated graph $\acute{G} = G \cup e(a, b)$. We first show the case that the edge insertion causes path distance changes from a certain vertex u to some others. Accordingly, the graphs in Fig. 2(a) and 2(b) are $\mathcal{G}_u$ and $\acute{\mathcal{G}}_u$ of G and $\acute{G}$, respectively. Originally, $p(u, b)$ in $\mathcal{G}_u$ passes through $\{l, o, w\}$ so that $|p(u, b)|$ = 4. After the edge $e(a, b)$ is inserted, b is now the child of a and they are now in adjacent levels in $\acute{\mathcal{G}}_u$. This causes some other vertices to move to different levels in $\acute{\mathcal{G}}_u$ so that $\acute{p}(u, b)$ changes as well. We can see that $|p(u, b)| > |\acute{p}(u, b)|$ = 3. Meanwhile, the shortest path distances from u to the vertices c, h, v and t also change. Therefore, in $\acute{\mathcal{G}}_u$, the u-unstable vertex set $\acute{\mathcal{V}}_u = \{b, c, h, v, t\}$.

Next, we show the case that the edge insertion does not cause any change in the shortest paths from a certain vertex r to all others. Fig. 2(c) and 2(d) are the corresponding $\mathcal{G}_r$ and $\acute{\mathcal{G}}_r$, respectively. We can see that the newly inserted edge $e(a, b)$ is an interlevel edge in $\acute{\mathcal{G}}_r$, and the levels of all vertices remain the same. In other words, $|p(r, v)| = |\acute{p}(r, v)|$ always holds for any vertex $v \in V$ so that r is a stable vertex.

From the above observations, we have the following lemma.

Lemma 1: Given a graph $G = (V, E)$ and a disconnected vertex pair $\{a, b\}$, any vertex $v \in V$ is an unstable vertex if and only if $||p(v, b)| - |p(v, a)|| > 1$ after the vertices a and b are connected. Otherwise, v is a stable vertex.

Proof: Suppose we are given an updated BFS$^+$ graph $\acute{\mathcal{G}}_v$ of $\acute{G} = G \cup e(a, b)$ for the vertex v. According to Definition 4, we know that if $||p(v, b)| - |p(v, a)|| > 1$, a and b are absolutely not in the same or adjacent levels in $\mathcal{G}_v$. Connecting a and b with an edge will cause the levels of some vertices u decrease in $\acute{\mathcal{G}}_v$. Therefore, v is an unstable vertex for the edge insertion of $e(a, b)$. On the contrary, if $||p(v, b)| - |p(v, a)|| \leq 1$, a and b must be in the same or adjacent levels in $\mathcal{G}_v$. When a and b are connected, $e(a, b)$ just becomes a horizontal edge or interlevel edge in $\acute{\mathcal{G}}_v$. Since adding interlevel or horizontal edge does not cause any change between the levels of all vertices in $\mathcal{G}_v$ and $\acute{\mathcal{G}}_v$, we can ensure v is the stable vertex to the edge insertion. ■

With Lemma 1, we are capable of identifying whether a vertex v is an unstable vertex to the edge insertion of $e(a, b)$ by checking the difference of $|p(v, a)|$ and $|p(v, b)|$ in the BFS$^+$ graph $\mathcal{G}_v$ rooted at v. However, the time cost of identifying whether all $v \in V$ is stable or unstable to the insertion of $e(a, b)$ is too high to be acceptable. In fact, we only have to find $\acute{\mathcal{V}}_a$ in $\acute{\mathcal{G}}_a$ and $\acute{\mathcal{V}}_b$ in $\acute{\mathcal{G}}_b$. Then, all unstable vertex pairs can be found from $\acute{\mathcal{V}}_a$ and $\acute{\mathcal{V}}_b$.

Lemma 2: Given an original graph $G = (V, E)$ and an updated graph $\acute{G} = G \cup e(a, b)$, if $\{v, u\}$ is an unstable vertex pair to the insertion, its new shortest path $\acute{p}(v, u)$ in $\acute{G}$ must pass through $e(a, b)$.

Theorem 1: Given an updated graph $\acute{G} = G \cup e(a, b)$ and a pair of BFS$^+$ graphs $\acute{\mathcal{G}}_a$ with $\acute{\mathcal{V}}_a$ and $\acute{\mathcal{G}}_b$ with $\acute{\mathcal{V}}_b$, each unstable vertex pair $\{v, u\}$ must satisfy the condition that $v \in \acute{\mathcal{V}}_a$ and $u \in \acute{\mathcal{V}}_b$ (or $v \in \acute{\mathcal{V}}_b$ and $u \in \acute{\mathcal{V}}_a$).

Proof: Assume u and v are unstable vertices such that $|\acute{p}(u,v)| < |p(u,v)|$ after the insertion of the edge $e(a,b)$. According to Lemma 2, the path $\acute{p}(u,v)$ must contain $e(a,b)$. For the case that in path $\acute{p}(u,v)$ vertex a is closer to u compared to vertex b, we know that $|\acute{p}(u,a)| = |p(u,a)|$ but $|\acute{p}(u,b)|$ will decrease, and $|\acute{p}(v,b)| = |p(v,b)|$ but $|\acute{p}(v,a)|$ will decrease. Thus, u will appear in $\acute{\mathcal{V}}_b$, and v will appear in $\acute{\mathcal{V}}_a$. Symmetrically, when b is closer to u compared to a in $\acute{p}(u,v)$, u will appear in $\acute{\mathcal{V}}_a$, and v will appear in $\acute{\mathcal{V}}_b$. This completes the proof. ■

With Theorem 1, we can efficiently identify all unstable vertex pairs using only two BFS$^+$ graphs $\acute{\mathcal{G}}_a$ and $\acute{\mathcal{G}}_b$ for the edge insertion of $e(a,b)$.

B. Unstable Vertex Discovery of An Edge Deletion

In this subsection, we look into the possible changes between the original graph and the updated graph after an edge deletion. Let the graphs shown in Fig. 1(b) be the original graph G and 1(a) be the updated graph $\acute{G} = G \backslash e(a,b)$.

We first show the case that the edge deletion causes the changes in the shortest paths from a certain vertex u to others. Now we consider Fig. 2(b) as $\mathcal{G}_u$ and 2(a) as $\acute{\mathcal{G}}_u$. We can see that vertices a and b are in adjacent levels and the shortest path from u to b is $\{u,x,a,b\}$ in $\mathcal{G}_u$. After $e(a,b)$ is deleted from $\mathcal{G}_u$, a and b are not in adjacent levels and $|p(u,b)| = 3 < |\acute{p}(u,b)| = 4$. Moreover, the shortest path distances from u to vertices c,h,v and t change as well. Therefore, in $\acute{\mathcal{G}}_u$, the u-unstable vertex set $\acute{\mathcal{V}}_u = \{b,c,h,v,t\}$.

Next, we show the case that the edge deletion does not lead to any change in the shortest paths from a certain vertex r to others. Now we consider Fig. 2(d) as $\mathcal{G}_r$ and 2(c) as $\acute{\mathcal{G}}_r$. In addition to $e(a,b)$, one can easily see that there is an interlevel edge $e(a,d)$ in $\mathcal{G}_r$. The deletion of the edge $e(a,b)$ does not cause any level changes of all vertices in $\acute{\mathcal{G}}_r$. Thus the vertex r is a stable vertex.

From the above observations, we know that in an original graph G if the vertex v satisfies one of the following conditions: (i) $|p(v,a)| = |p(v,b)|$ or (ii) there exists another edge $e(x,b)$ such that $|p(v,a)| = |p(v,x)| = |p(v,b)| - 1$, then v is a stable vertex to the deletion of $e(a,b)$. This is because that deleting $e(a,b)$ never causes any change in $p(v,u)$ for each vertex $u \neq v \in V$. Otherwise, v is an unstable vertex.

Similar to the case of edge insertion introduced in Sec. III-A, we do not need to check all $\mathcal{G}_u$, $u \in V$, to find all the unstable vertex pairs. In the following, for deleting an edge $e(a,b)$ we show how to identify all unstable vertices based on only $\acute{\mathcal{G}}_a$ with $\acute{\mathcal{V}}_a$ and $\acute{\mathcal{G}}_b$ with $\acute{\mathcal{V}}_b$.

Theorem 2: Given an updated graph $\acute{G} = G \backslash e(a,b)$ and a pair of updated BFS$^+$ graphs $\acute{\mathcal{G}}_a$ with $\acute{\mathcal{V}}_a$ and $\acute{\mathcal{G}}_b$ with $\acute{\mathcal{V}}_b$, each pair $\{v,u\}$ must satisfy the condition that $v \in \acute{\mathcal{V}}_a$ and $u \in \acute{\mathcal{V}}_b$ (or $v \in \acute{\mathcal{V}}_b$ and $u \in \acute{\mathcal{V}}_a$).

Proof: Assume u and v are unstable vertices such that $|\acute{p}(u,v)| > |p(u,v)|$ after the deletion of the edge $e(a,b)$. Now, we know that $\acute{p}(u,v)$ does not include $e(a,b)$ any more. For the case that in path $\acute{p}(u,v)$ the vertex a is closer to u than b is, we can ensure that $|\acute{p}(u,a)| = |p(u,a)|$ but $|\acute{p}(u,b)|$ increases, and $|\acute{p}(v,b)| = |p(v,b)|$ but $|\acute{p}(v,a)|$ increases. Hence, u will appear in $\acute{\mathcal{V}}_b$, and v will appear in $\acute{\mathcal{V}}_a$. Symmetrically, when b is closer to u than a in $\acute{p}(u,v)$, u will appear in $\acute{\mathcal{V}}_a$, and v will appear in $\acute{\mathcal{V}}_b$. This completes the proof. ■

With Theorem 2, we can efficiently identify all unstable vertex pairs by the updated BFS$^+$ graphs rooted at both end vertices a and b of the edge $e(a,b)$ to be deleted from the graph G.

Algorithm 1: findUVSet($a, b, G, \mathcal{G}_a/\acute{\mathcal{G}}_a$)

input : The vertex pair $\{a,b\}$, the original graph G, and the BFS$^+$ graph $\mathcal{G}_a/\acute{\mathcal{G}}_a$
output: The a-unstable vertex set $\acute{\mathcal{V}}_a$

```
1  V́_a = ∅, Q = ∅ ;
2  V́_a = V́_a ∪ b ;
3  Q = Q ∪ b ;
4  b.dis = 1 ;
5  while Q ≠ ∅ do
6      x = Q.dequeue() ;
7      for each vertex c ∈ N_x \ N_a in G do
8          if level of c in G_a(Ǵ_a) > x.dis + 1 then
9              c.dis = x.dis + 1 ;
10             V́_a = V́_a ∪ c ;
11             Q = Q ∪ c ;
12         end
13     end
14 end
15 Return V́_a ;
```

C. Updating the Shortest Path Distances

In Sec. III-A and III-B we prove that the unstable vertex pair must contain one vertex in $\acute{\mathcal{V}}_a$ and the other in $\acute{\mathcal{V}}_b$ when inserting or deleting an edge $e(a,b)$. Here we show how to identify these unstable vertex pairs and the algorithm is provided in Algorithm 1.

The design guideline of Algorithm 1 is as follows. Suppose we are given a vertex pair $\{a,b\}$ to be connected or disconnected and want to find a-unstable vertex set $\acute{\mathcal{V}}_a$. For both cases of edge insertion and deletion, we first examine each neighbor vertex, say c, of b to see whether the original shortest path distance between a and c differs from the updated one. If so, i.e., $|p(a,c)| \neq |\acute{p}(a,c)|$, c is an unstable vertex to be put into $\acute{\mathcal{V}}_a$. We repeat the above steps to examine all such neighbor vertex c and capture all unstable vertices in $\acute{\mathcal{V}}_a$.

It is worth mentioning that we use different BFS$^+$ graphs as input of Algorithm 1 for edge insertion and deletion to speed up the examining process. In the case of edge insertion, since we will examine each neighbor vertex c of b to find $\acute{\mathcal{V}}_a$, $|\acute{p}(a,c)|$ can be obtained during the algorithm. Thus, we only prepare the BFS$^+$ graph $\mathcal{G}_a$ by running one breadth-first search in advance as the input of Algorithm 1 and use the level of the vertex c in $\mathcal{G}_a$ as $|p(a,c)|$. On the other hand, for the case of edge deletion we only prepare $\acute{\mathcal{G}}_a$ in advance as the input after deleting $e(a,b)$ and then use the level of the vertex c in $\acute{\mathcal{G}}_a$ as $|\acute{p}(a,c)|$, since $|p(a,c)|$ can be obtained during the algorithm.

To be more specific, in Algorithm 1, we first put the vertex b into both $\acute{\mathcal{V}}_a$ and an empty queue Q, and then set b.dis as 1,

there shouldn't be mod here, right?

Algorithm 2: CENDY($G, C_c(v), L_G, a, b$)

input : The original graph G, the original closeness centrality $C_c(v)$ of each vertex $v \in V$, the APL L_G and the vertex pair $\{a, b\}$ to be connected or disconnected

output: The updated closeness centrality $\acute{C}_c(v)$ of each vertex $v \in V$ and the updated average path length $\acute{L}_G$ of G

1 **if** $e(a,b)$ *is inserted into* G **then**
2 Perform BFS starting at a and b in G to get $\mathcal{G}_a$ and $\mathcal{G}_b$, respectively ;
3 $\acute{\mathcal{V}}_a$ = findUVSet($a, b, G, \mathcal{G}_a$) ;
4 $\acute{\mathcal{V}}_b$ = findUVSet($b, a, G, \mathcal{G}_b$) ;
5 **else if** $e(a,b)$ *is deleted from* G **then**
6 $\acute{G} = G \backslash e(a,b)$;
7 Perform BFS starting at a and b in $\acute{G}$ to get $\acute{\mathcal{G}}_a$ and $\acute{\mathcal{G}}_b$, respectively ;
8 $\acute{\mathcal{V}}_a$ = findUVSet($a, b, G, \acute{\mathcal{G}}_a$) ;
9 $\acute{\mathcal{V}}_b$ = findUVSet($b, a, G, \acute{\mathcal{G}}_b$) ;
10 **end**
11 Suppose $|\acute{\mathcal{V}}_a| \leq |\acute{\mathcal{V}}_b|$;
12 **for** *each* $v \in \acute{\mathcal{V}}_a$ **do**
13 Perform BFS starting at v in G and $\acute{G}$ to get $\{\mathcal{G}_v, \acute{\mathcal{G}}_v\}$;
14 **for** *each* $u \in \acute{\mathcal{V}}_b$ **do**
15 $\delta^v = \delta^v + (||\acute{p}(v,u)| - p|(v,u)||)$;
16 $\delta^u = \delta^u + (||\acute{p}(v,u)| - p|(v,u)||)$;
17 **end**
18 **end**
19 **if** $e(a,b)$ *is inserted into* G **then**
20 Calculate $\acute{C}_c^i(v)$ of each unstable vertex v by Eq. (5) ;
21 Calculate $\acute{L}_G^i$ by Eq. (7) ;
22 **else if** $e(a,b)$ *is deleted from* G **then**
23 Calculate $\acute{C}_c^d(v)$ of each unstable vertex v by Eq. (6) ;
24 Calculate $\acute{L}_G^d$ by Eq. (8) ;
25 **end**

which indicates the updated shortest path distance from a to b after the insertion (or the original shortest path distance before the deletion). Next, from Line 5 to 14, for each vertex x in Q we examine each vertex c belonging to neighborhood N_x of x but not neighborhood N_a of a to see whether the original shortest path distance from a to c differs from the updated one. If so, i.e., $|p(a,c)| \neq |\acute{p}(a,c)|$, we put c into both $\acute{\mathcal{V}}_a$ and Q. Please note that in Line 8, for the case of edge insertion we use the level of c in $\mathcal{G}_a$ as $|p(a,c)|$ and x.dis + 1 as $|\acute{p}(a,c)|$. For the case of the deletion we use x.dis + 1 as $|p(a,c)|$ and the level of c in $\acute{\mathcal{G}}_a$ as $|\acute{p}(a,c)|$. The above steps repeat until Q is empty. Finally, we return $\acute{\mathcal{V}}_a$ and stop the process.

After running Algorithm 1 twice for vertices a and b, we have $\acute{\mathcal{V}}_a$ and $\acute{\mathcal{V}}_b$. The next step is to compute the distance difference over each unstable vertex pair $\{v, u\}$ between the changed shortest path and the original one for latter updating process.

According to Theorem 1 and Theorem 2, we know that each unstable vertex pair $\{v, u\}$ satisfies the condition that $v \in \acute{\mathcal{V}}_a$ and $u \in \acute{\mathcal{V}}_b$ for an edge $e(a,b)$ to be connected or disconnected. Therefore, we can ensure that $\acute{\mathcal{V}}_v \subseteq \acute{\mathcal{V}}_b$ for each unstable vertex $v \in \acute{\mathcal{V}}_a$ and $\acute{\mathcal{V}}_u \subseteq \acute{\mathcal{V}}_a$ for each unstable vertex $u \in \acute{\mathcal{V}}_b$, and define the distance difference δ^v for v:

$$\delta^v = \sum_{u \in \acute{\mathcal{V}}_b} ||p(v,u)| - |\acute{p}(v,u)||, \tag{3}$$

or δ^u for u:

$$\delta^u = \sum_{v \in \acute{\mathcal{V}}_a} ||p(v,u)| - |\acute{p}(v,u)||. \tag{4}$$

To compute both $|p(v,u)|$ and $|\acute{p}(v,u)|$, we need to prepare an BFS+ graph pair $\{\acute{\mathcal{G}}_v, \mathcal{G}_v\}$ or $\{\acute{\mathcal{G}}_u, \mathcal{G}_u\}$ in advance to deal with both edge insertions and deletions in a dynamic network. To speed up the computation, we prepare the BFS$^+$ graph pair of the vertex in the unstable vertex set with minimum size. For example, if $|\acute{\mathcal{V}}_a| \leq |\acute{\mathcal{V}}_b|$, we prepare $\{\mathcal{G}_v, \acute{\mathcal{G}}_v\}$ for each $v \in \acute{\mathcal{V}}_a$ to compute both δ^v and δ^u. We will empirically show $|\acute{\mathcal{V}}_a| \ll |\acute{\mathcal{V}}_b|$ in most real graph networks in Sec. IV.

Now we can update the closeness centrality as follows. Since the original closeness centrality $C_c(v)$ of each unstable vertex v is recorded off-line, we have the updated one $\acute{C}_c(v)$ for the case of the insertion by the following equation

$$\acute{C}_c^i(v) = \frac{|V| - 1}{(|V| - 1)/C_c(v) - \delta^v}, \tag{5}$$

or the deletion by

$$\acute{C}_c^d(v) = \frac{|V| - 1}{(|V| - 1)/C_c(v) + \delta^v}. \tag{6}$$

For the updated APL $\acute{L}_G$, we have the following equation for the case of the insertion by

$$\acute{L}_G^i = \frac{L_G \times (|V|(|V| - 1)) - \sum_{v \in \acute{\mathcal{V}}_a \cup \acute{\mathcal{V}}_b} \delta^v}{|V|(|V| - 1)}, \tag{7}$$

or the deletion by:

$$\acute{L}_G^d = \frac{L_G \times (|V|(|V| - 1)) + \sum_{v \in \acute{\mathcal{V}}_a \cup \acute{\mathcal{V}}_b} \delta^v}{|V|(|V| - 1)}. \tag{8}$$

The entire process of CENDY is summarized in Algorithm 2. Suppose that we have the edge $e(a,b)$ to be inserted into or deleted from G. For the case of the edge insertion, from Line 2 to 4, we perform BFS starting at a and b in G to obtain $\mathcal{G}_a$ and $\mathcal{G}_b$. Next, we use $\mathcal{G}_a$ and $\mathcal{G}_b$ as the input of Algorithm 1 to find $\acute{\mathcal{V}}_a$ and $\acute{\mathcal{V}}_b$. For the case of the edge deletion, we obtain $\acute{\mathcal{G}}_a$ and $\acute{\mathcal{G}}_b$ by BFS and use them as the input of Algorithm 1 to find corresponding $\acute{\mathcal{V}}_a$ and $\acute{\mathcal{V}}_b$ from Line 6 to 9. Next, from Line 11 to 18, we first compare the size of $\acute{\mathcal{V}}_a$ with that of $\acute{\mathcal{V}}_b$. Without loss of generality, we suppose $|\acute{\mathcal{V}}_a| \leq |\acute{\mathcal{V}}_b|$. Then for each unstable vertex $v \in \acute{\mathcal{V}}_a$, we prepare the graph pair $\{\mathcal{G}_v, \acute{\mathcal{G}}_v\}$ and calculate δ^v and δ^u for the each unstable vertex pair $\{v, u\}$, $u \in \acute{\mathcal{V}}_b$. Finally, we update the closeness centrality of each unstable vertex v by Eq. (5) or Eq. (6) and the APL of G by Eq. (7) or Eq. (8), depending on the case of the insertion or the deletion.

D. Time Complexity of CENDY

For the time complexity of the unstable vertex discovery in Algorithm 1, from Line 1 to 4 each step takes constant time. Then, from Line 5 to 14, we examine each neighbor vertex c of the vertex x in Q to check the distance differences recursively. Since this checking process runs BFS, its time complexity is $O(m+n)$.

For the time complexity of CENDY in Algorithm 2, at Line 2 and Line 7 we perform BFS twice starting at both end vertices of the edge to be inserted into or deleted from G. Then, we use Algorithm 1 for both end vertices to get corresponding unstable vertex sets. Hence, the time cost from Line 1 to 10 is $O(m+n)$. From Line 11 to 18, we first compare $|\acute{\mathcal{V}}_a|$ with $|\acute{\mathcal{V}}_b|$. Next, we perform BFS twice starting at each vertex $v \in \acute{\mathcal{V}}_a$ in G and $\acute{G}$ to get the graph pair $\{\mathcal{G}_v, \acute{\mathcal{G}}_v\}$, which takes $O((m+n)|\acute{\mathcal{V}}_a|)$ time. Then, for each unstable vertex pair $\{v,u\}$, $u \in \acute{\mathcal{V}}_b$, we calculate δ^v and δ^u, which takes $O(|\acute{\mathcal{V}}_a| \times |\acute{\mathcal{V}}_b|)$ time. In practice, $|\acute{\mathcal{V}}_b| < (m+n)$. Thus, the cost from Line 11 to 18 takes $O((m+n)|\acute{\mathcal{V}}_a|)$. Finally, from Line 19 to 25, the update of closeness centralities of the unstable vertices and the average path length can be completed in $O(|\acute{\mathcal{V}}_a|+|\acute{\mathcal{V}}_b|)$. Therefore, the total cost of Algorithm 2 takes $O(\min\{|\acute{\mathcal{V}}_a|, |\acute{\mathcal{V}}_b|\} \times (m+n))$. We will show in experiments that in real-world networks, the number of $\min\{|\acute{\mathcal{V}}_a|, |\acute{\mathcal{V}}_b|\}$ is very small and thus CENDY can update the two measurements very efficiently.

IV. Experiments

A. Settings

To evaluate CENDY, we measured the update time of the closeness centrality of all vertices and the APL of the entire network by conducting extensive experiments on six real unit-weighted graph datasets of different types. The information of each dataset was summarized in Table I. Among these data sets, five of them were Wiki-Vote, CA-CondMat, Cit-HepPh, p2p-Gnutella31 and Email-EuAll obtained from [8], while the other one was DBLP[1] with a time interval from Jan. 2002 to Dec. 2011. For some datasets, e.g., Wiki-Vote, that were directed graphs originally, we converted all the directed edges into undirected ones. For each dataset, we extracted the largest connected component as the test graph.

In comparison, we implemented methods including Naive-BFS, the approximate method of RAND [15], and SNOWBALL [1]. The Naive-BFS method always found all-pair shortest paths by performing BFS for each vertex pair, and then summed over all distances from each vertex to all others to calculate its closeness centrality and the APL of the entire network. The RAND method was adapted from [15]. First, we selected a set of sample vertices $\{s_1, s_2, ..., s_k\}$ randomly and performed BFS starting at each s_i. For each vertex $v \neq s_i$, its estimated closeness centrality was:

$$\acute{C}_c^R(v) = \frac{1}{\sum_{i=1}^{k} \frac{|V| \times |p(s_i,v)|}{k(|V|-1)}}, \tag{9}$$

and the estimate APL of the given graph was:

$$\acute{L}_G^R = \frac{\sum_{v \in V} \frac{|V|-1}{\acute{C}_c^R(v)}}{|V|(|V|-1)}. \tag{10}$$

In the above process we set $k = \frac{\log n}{\epsilon^2}$ as indicated in [15]. However, the authors did not give guidelines to select ϵ quantitatively. To find a better trade-off between the two estimate measurements and the computation time, we set ϵ as 0.06 for RAND on different graph datasets. The SNOWBALL method was implemented to estimate the APL of a network. It first sampled a subgraph $\tilde{G} = \{\tilde{V}, \tilde{E}\}$ from the original graph G by the snowball sampling method [1]. The APL was then estimated as follows.

$$\acute{L}_G^S = \frac{\sum_{u,v \in \tilde{V}, u \neq v} |p(u,v)|}{|\tilde{V}|(|\tilde{V}|-1)}, \tag{11}$$

where the number of sampled vertices $|\tilde{V}|$ was set as $|V| \times 60$ %.

To evaluate the accuracy of the estimated APL calculated by RAND and SNOWBALL after each edge update, we used the following metrics.

$$\text{Acc}(\acute{L}_G^*) = \left[1 - \frac{|\acute{L}_G^* - L_G|}{L_G}\right] \times 100\%. \tag{12}$$

where $*$ represents S or R.

The accuracy of the estimated closeness centrality computed by RAND was evaluated as:

$$\text{Acc}(\acute{C}_c^R) = \left[1 - \frac{\sum_{v \in V} \frac{|\acute{C}_c^R(v) - C_c(v)|}{C_c(v)}}{|V|}\right] \times 100\%. \tag{13}$$

Based on the assumption that the total number of vertices was fixed, we considered two operations, edge insertion and deletion, separately. Since the number of edges in a real network may vary over time, for the insertion operation, we inserted 200,000 edges, one at a time, randomly on each dataset. For the deletion operation, we deleted 50,000 edges, one at a time, randomly from Wiki-Vote, CA-CondMat, Cit-HepPh and p2p-Gnutella31, 100,000 edges from Email-EuAll, and 200,000 edges from DBLP. The different number of edge deletion was to make sure the test graph was still connected. Initially, CENDY computed the two exact measurements in each graph dataset and then updated them after each insertion/deletion operation using Eq.(5) and Eq.(7) / Eq.(6) and Eq.(8), respectively. Since other methods were not designed for dynamic updating, we re-run them after each insertion/deletion operation.

All algorithms were implemented in Java and the experiments were conducted on a PC running Windows 7 with 2GB RAM and 2.0 GHz CPU.

[1] http://dblp.uni-trier.de/xml/

TABLE I. STATISTICS OF REAL GRAPH DATASETS

Dataset	Wiki-Vote	CA-CondMat	Cit-HepPh	p2p-Gnutella31	Email-EuAll	DBLP
$\|V\|$	7,066	21,363	34,401	62,561	224,832	460,413
$\|E\|$	100,736	91,314	420,806	147,878	340,360	1,853,734
type	Wikipedia	Collaboration	Citation	Internet Peer-to-Peer	Communication	Collaboration
max. vertex degree	1,065	280	846	95	7636	454
min. vertex degree	1	1	1	1	1	2
avg. degree	28	8	24	4	3	8

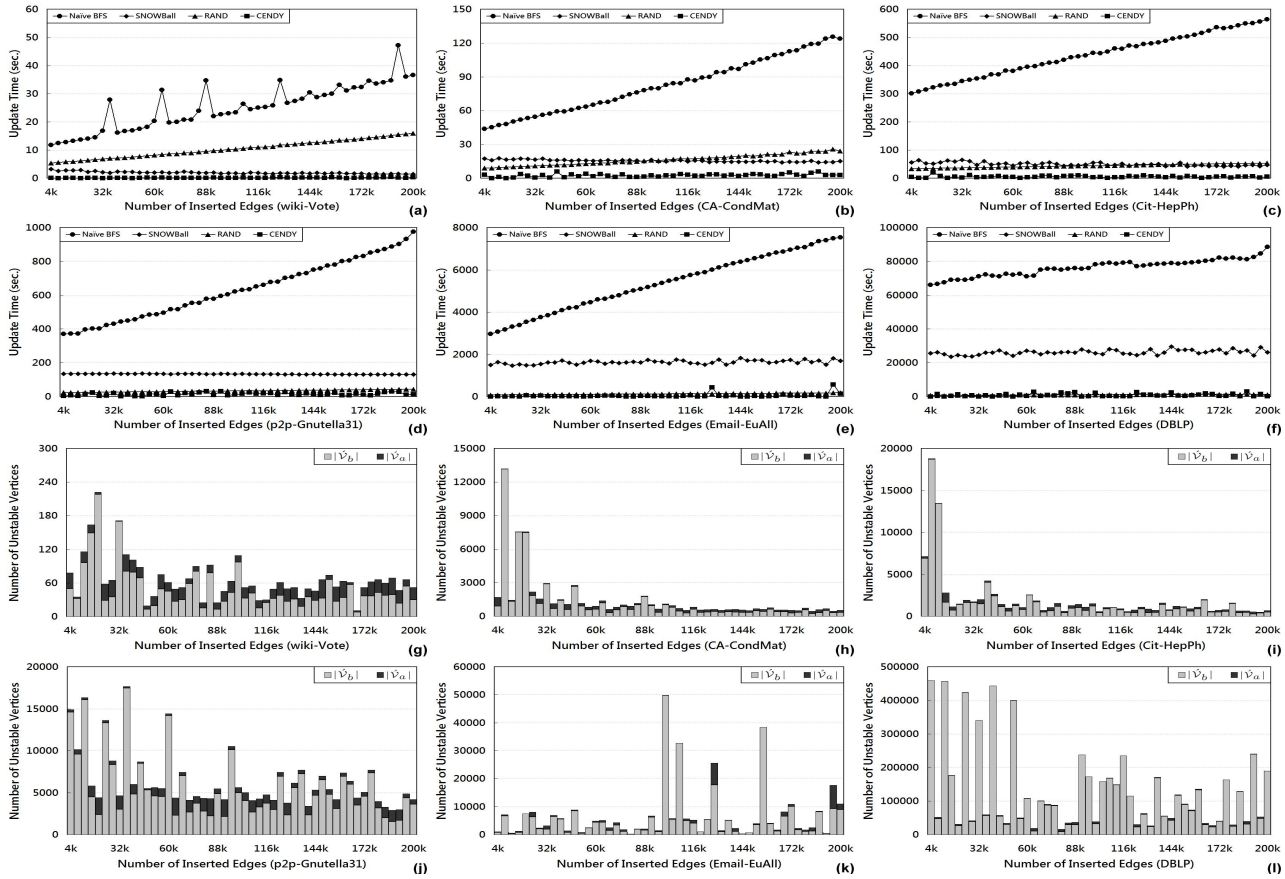

Fig. 3. (a) to (f) The update time required to compute closeness centrality and the APL by all methods at every 4,000th edge insertion on six different real datasets; (g) to (l) The number of unstable vertices ($|\dot{\mathcal{V}}_a| + |\dot{\mathcal{V}}_b|$) produced by CENDY corresponding to (a) to (f)

TABLE II. THE AVERAGE ACCURACY OF CORRESPONDING MEASUREMENTS CALCULATED BY SNOWBALL AND RAND ON DIFFERENT GRAPH DATASETS, IN THE OPERATION OF EDGE INSERTION

dataset	SNOWBALL	RAND	
	avg. Acc($\acute{L}_G^S$)	avg. Acc($\acute{L}_G^R$)	avg. Acc($\acute{C}_c^R$)
(1)	95.33 %	99.31 %	99.92 %
(2)	94.65 %	99.93 %	99.01 %
(3)	96.18 %	99.88 %	99.12 %
(4)	95.12 %	99.52 %	99.1 %
(5)	96.25 %	99.14 %	99.79 %
(6)	96.6 %	99.29 %	99.91 %

(1)Wiki-Vote (2)CA-CondMat (3)Cit-HepPh
(4)p2p-Gnutella31 (5)Email-EuAll (6)DBLP

TABLE III. THE AVERAGE PERCENTAGE OF UNSTABLE VERTICES OVER THE SEQUENCE OF EDGE INSERTIONS ON DIFFERENT GRAPH DATASETS

dataset	avg. $\frac{\|\dot{\mathcal{V}}_a\|}{\|V\|}$	avg. $\frac{\|\dot{\mathcal{V}}_b\|}{\|V\|}$
Wiki-Vote	0.2 %	0.6 %
CA-CondMat	1 %	5.8 %
Cit-HepPh	0.7 %	5 %
p2p-Gnutella31	1.3 %	8.9 %
Email-EuAll	0.3 %	2.6 %
DBLP	0.5 %	27.8 %

B. Evaluation on Edge Insertion

Fig. 3(a) to 3(f) show the time required for each method to update the closeness centrality of each vertex and the APL of the network at every 4,000th edge insertion. Please note that SNOWBALL only computed the APL value. We also reported the average accuracy of the two corresponding estimate measurements calculated by SNOWBALL and RAND on different real graph datasets in Table II.

We can see that CENDY always outperformed other methods in consuming least updating time. Not surprisingly, Naive-BFS required the most computational time at each edge insertion compared with all other methods. Moreover, the time consumed by Naive-BFS increased as the number of inserted

edges grew on every graph dataset. This was because that the BFS method to compute APSP required computation time linear to the number of edges when the number of vertex was fixed.

On the contrary, the time consumed by SNOWBALL was not affected significantly with the grow of edge counts. This was because that the sampled subgraph only included partial newly inserted edges. However, SNOWBALL was not capable of calculating the exact shortest path distance from the sampled subgraph. To achieved the high accuracy of the estimated APL around 95 % as shown in Table II, the SNOWBALL method in fact sampled 60 % vertices of the original graph. Like Naive-BFS, the time consumed by RAND at each edge insertion increased slightly as the edge counts grew. From Fig. 3(a) to 3(f) and Table II, we can see that RAND always reported the APL and the closeness centrality values of higher accuracy (around 99.5 %) at a smaller cost of computation time in most cases compared to SNOWBALL. This was because that RAND performed BFS in the original graph starting at the sampled vertices to calculate the exact single source shortest paths. Thus, RAND required fewer sampled vertices and consumed less computation time but estimated the two measurements with higher accuracy. Compared to CENDY, however, the computation time consumed by RAND was still high. The average speed-up achieved by CENDY ranged from about 1.1 to even 68.6 times faster compared to RAND among the six datasets.

We next gave a more detailed analysis about why CENDY always outperformed other methods in consuming least update time over all datasets. Suppose $e(a, b)$ was the newly inserted edge and $|\acute{\mathcal{V}}_a| \leq |\acute{\mathcal{V}}_b|$ without loss of generality. We computed the sum of unstable vertices, $(|\acute{\mathcal{V}}_a| + |\acute{\mathcal{V}}_b|)$, of CENDY corresponding to Fig. 3(a) to 3(f), and showed the results in Fig. 3(g) to 3(l). All the corresponding average percentage of $\frac{|\acute{\mathcal{V}}_a|}{|V|}$ and $\frac{|\acute{\mathcal{V}}_b|}{|V|}$ over the sequence of edge insertions on different graph datasets was shown in Table III.

Recall that CENDY performed only two times of the BFS computations to identify all unstable vertices, which took negligible time compared to the overall updating time. As aforementioned, the performance of CENDY to update closeness centrality and the APL was mainly dominated by the size of the minimum unstable vertex set, i.e., $\acute{\mathcal{V}}_a$. From Fig. 3(g) to 3(l) and Table III, we can observe that the vertex count of $\acute{\mathcal{V}}_a$ was remarkably smaller than $|\acute{\mathcal{V}}_b|$ on each graph dataset. The average percentage of $|\acute{\mathcal{V}}_a|$ ranged only from 0.2 % to 1.3 %. With such a small $\acute{\mathcal{V}}_a$, only a few of BFS computations was required to update the two measurements. Therefore, CENDY was able to quickly update the two measurements when the number of edges was increased.

C. Evaluation on Edge Deletion

Fig. 4(a) to 4(d) illustrate the update time required to compute closeness centrality and the APL by all methods at every 1,000$^{\text{th}}$ edge deletion, while 4(e) and 4(f) at every 2,000$^{\text{th}}$ and 4,000$^{\text{th}}$ edge deletion, respectively, on the six real graph datasets. The average accuracy of the two corresponding estimate measurements calculated by SNOWBALL and RAND on different real graph datasets was reported in Table IV.

TABLE IV. THE AVERAGE ACCURACY OF CORRESPONDING MEASUREMENTS CALCULATED BY SNOWBALL AND RAND ON DIFFERENT GRAPH DATASETS, IN THE OPERATION OF EDGE DELETION

dataset	SNOWBALL	RAND	
	avg. Acc($\acute{L}_G^S$)	avg. Acc($\acute{L}_G^R$)	avg. Acc($\acute{C}_c^R$)
(1)	96.46 %	99.98 %	99.99 %
(2)	95.32 %	99.2 %	99.21 %
(3)	96.24 %	99.85 %	99.75 %
(4)	95.23 %	99.12 %	99.91 %
(5)	96.98 %	99.74 %	99.89 %
(6)	97.1 %	99.6 %	99.92 %

(1)Wiki-Vote (2)CA-CondMat (3)Cit-HepPh
(4)p2p-Gnutella31 (5)Email-EuAll (6)DBLP

TABLE V. THE AVERAGE PERCENTAGE OF UNSTABLE VERTICES OVER THE SEQUENCE OF EDGE DELETIONS ON DIFFERENT GRAPH DATASETS

dataset	avg. $\frac{\lvert\acute{\mathcal{V}}_a\rvert}{\lvert V\rvert}$	avg. $\frac{\lvert\acute{\mathcal{V}}_b\rvert}{\lvert V\rvert}$
Wiki-Vote	0.04 %	3.7 %
CA-CondMat	0.3 %	8 %
Cit-HepPh	0.1 %	2.5 %
p2p-Gnutella31	1.9 %	16.4 %
Email-EuAll	0.02 %	5.4 %
DBLP	0.01 %	3.2 %

We can see that CENDY always outperformed other methods in consuming the least updating time. Although the time consumed by Naive-BFS decreased as the number of deleted edges increased on every graph dataset, Naive-BFS required the most computational time after each edge deletion compared with all other methods. To achieve the high accuracy of the estimated APL around 96.2 % as shown in Table IV, SNOWBALL needs to sample 60 % vertices of the original graph as it did in the operation of edge insertion. Like Naive-BFS, the time consumed by RAND at each edge deletion decreased slightly as the edge counts decreased. From Fig. 4(a) to 4(f) and Table IV, we can see that RAND always reported the APL and the closeness centrality values with higher accuracy (around 99.6 %) at a smaller cost of computation time in most cases compared to SNOWBALL. Compared to CENDY, however, the computation time consumed by RAND was still high. CENDY achieved a average speed-up ranging from 1.8 to 516.7 times faster on the six datasets compared to RAND.

We also gave a more detailed analysis about why CENDY always outperformed other methods in consuming least update time on all datasets in the operation of edge deletion. In Fig. 4(g) to 4(l), we reported the sum of unstable vertices, $(|\acute{\mathcal{V}}_a| + |\acute{\mathcal{V}}_b|)$, of CENDY corresponding to Fig. 4(a) to 4(f). All the corresponding average percentage of $\frac{|\acute{\mathcal{V}}_a|}{|V|}$ and $\frac{|\acute{\mathcal{V}}_b|}{|V|}$ over the sequence of edge deletions on different datasets was shown in Table V. From Fig. 4(g) to 4(l) and Table V, we can observed that $|\acute{\mathcal{V}}_a|$ was remarkably smaller than $|\acute{\mathcal{V}}_b|$ on each dataset. The average percentage of $|\acute{\mathcal{V}}_a|$ ranged only from 0.01 % to 1.9 %. With such a small $\acute{\mathcal{V}}_a$, only a few of BFS computations was required to update the two measurements. Therefore, CENDY indeed had better update efficiency when the number of edges was decreased.

V. RELATED WORK

Closeness centrality and the APL of a network have drawn wide attention in diverse research areas. They allow us to understand the importance of an vertex and the network flow efficiency of a network. For example, Halatchliyski

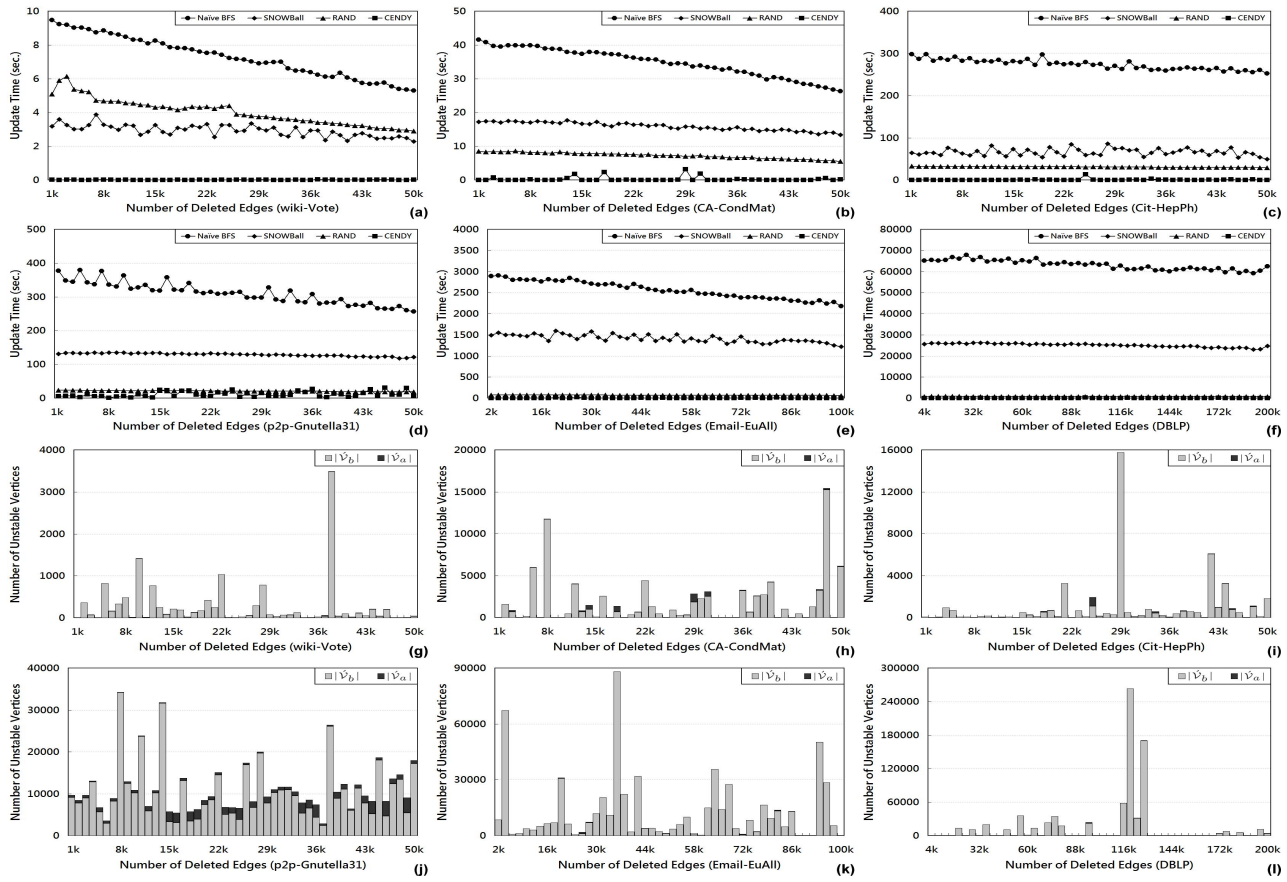

Fig. 4. (a) to (d) The update time required to compute closeness centrality and the APL by all methods at every 1,000$^{\text{th}}$ edge deletion, while (e) and (f) at every 2,000$^{\text{th}}$ and 4,000$^{\text{th}}$ edge deletion, respectively, on the six real graph datasets; (g) to (l) The number of unstable vertices ($|\hat{\mathcal{V}}_a| + |\hat{\mathcal{V}}_b|$) produced by CENDY corresponding to (a) to (f)

et al. [12] used closeness centrality to analyze the role of boundary spanners in a knowledge-building community such as Wikipedia. In terrorist (criminal) social network, Yang et al. [18] utilized the estimated generalized information and closeness centrality to identify suspects, terrorist or criminal subgroups. With computation of closeness centrality, Brandes et al. [16] identified the distinctive actors in a policy network. Zhang et al. [11] analyzed the influence of the central figure in a reply network such as bulletin board system by studying the closeness centralities of the vertices. To help users to find out the most reachable experts in a social network, Jin et al. [19] utilized closeness centrality to analyze the relevance of an expert to the topic that the user is interested in. Ma and Zeng [6] used APL to estimate the transfer efficiency between all of metabolite pairs in a metabolic network. Albert et al. [25] employed APL as one of the tools of statistical mechanics to analyse the topology and dynamics of real world networks such as World Wide Web and the chemical network. In binary decision diagrams (BDDs), Butler et al. [7] utilized APL to measure the time needed to evaluate the functions.

Traditionally, these two measures were determined by first solving the APSP problem, and then summing up pair-dependencies of all vertex pairs. Thus, they can be calculated by various algorithms including the BFS-base method in $O(n(m+n))$ time [26], [3], [2]. However, these algorithms were not efficient enough for large-scale networks with millions or more vertices, especially re-performing them at each edge update. To handle the dynamic APSP problem, Roditty and Zwick [5] presented an approximate version of the dynamic APSP algorithm in an undirected graph. For every $\gamma > 0$ and $t \leq m^{1/2-\gamma}$, where γ is pre-defined parameter, the algorithm took an amortized update time of $\tilde{O}(mn/t)$ and a query time of $O(t)$ for any vertex pair. Later, Thorup [4] provided a solution to update a complete distance matrix of a directed graph in $\tilde{O}(n^{2.75})$ time, and thus answering the shortest distance of a given vertex pair in $O(1)$ time. Since those methods were designed for quickly answering a shortest path distance query, they required $O(tn^2)$ and $O(n^2)$ time, respectively, to sum over all shortest path distances for updating the closeness centrality values and the APL, which was still infeasible in applications dealing with a large-scale dynamic network.

To compute the closeness centrality and the APL efficiently, a number of studies were proposed. For the purpose of efficiently computing closeness centrality, Eppstein and Wang [15] utilized a random sampling technique to approximate the closeness centrality of all vertices in a weighted and undirected graph. They demonstrated that the computation

time could be low on $O(\frac{\log n}{\epsilon^2}(n \log n + m))$ with an additive error of at most $\epsilon\phi$, where ϵ was a fixed constant and ϕ was the diameter of the graph. Inspired from [15], Brandes and Pich [17] tested several strategies to take just a few single source shortest path computations on a selected small set of source vertices (pivots). Instead of computing the exact shortest paths on a sampled set of vertices, Rattigan et al. [13] proposed an approximate shortest path distance query technique termed NSI and utilized NSI on a sampled set of vertices in the graph to compute closeness centrality. For each vertex pair, NSI took $O(ld)$ time to calculate its shortest path distance where l was the number of disjoint subsets of vertices in G and d was the number of dimensions. To compute closeness centralities, NSI required $O(snld)$ time when the number of the vertices in the sampled set is s. Recently, Chan et al. [14] proposed a new framework to assess the network centrality in a distributed way without requiring the knowledge of the network topology. By exploiting modularity of community structures in a social network, i.e., the presence of dense subgraphs, they observed that separate calculation for those dense subgraphs was much faster than the calculation for the complete network. In a modular network of size n with $\sqrt{n}$ evenly sized communities, their method took $O(m)$ and $O(\sqrt{n}m)$ time to approximate the closeness and betweenness centralities, respectively. Rather than computing closeness centrality of all vertices in a network, several methods have been proposed for identifying k-top network centralities. Later, Wehmuth and Ziviani [22] computed localized volume-based centrality at each vertex considering only a limited neighborhood around every vertex. Based on the idea of bounded distances of all pair shortest path calculations and focusing on sub-graphs created by limited distance paths from the vertices, Pfeffer and Carley [21] proposed a k-measure approach as an approximation for calculating closeness and betweenness centralities. For the purpose of efficiently computing the APL of a network, Lee et al. [1] used a snowball sampling strategy to get the approximate APL.

With such approximate computation of these two measures, we may get an imprecise quantitative analysis of a network. Moreover, all of those methods are designed for the static network. If we apply those methods in dynamic networks, we need to re-consider all vertices to compute these two measures after each edge update, which still leads a high computational time.

VI. Conclusions

In this paper, we presented *CENDY*, an efficient approach to updating the closeness centrality of each vertex and the APL of a network, where the edge changes dynamically. CENDY identified a set of unstable vertices whose shortest paths changed and efficiently updated the two measurements only by a few of single-source shortest path computations. We conducted extensive experiments to show that, compared to the existing methods of computing exact or approximate values, CENDY outperformed others in better update efficiency while providing exact values of the two measurements on various real-world graph datasets.

References

[1] Sang Hoon Lee, Pan-Jun Kim and Hawoong Jeong, "Statistical properties of sampled networks," in *Physical Review E 73*, 2006, pp. 1-7.

[2] Michael L. Fredman and Robert Endre Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," in *Journal of the ACM*, 1987, pp. 596-615.

[3] Donald B. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks," in *Journal of ACM*, 1977, pp. 1-13.

[4] Mikkel Thorup, "Worst-Case Update Times for Fully-Dynamic All-Pairs Shortest Paths," in *Proceedings of ACM STOC*, 2005, pp. 112-118.

[5] Liam Roditty and Uri Zwick, "Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs," in *Proceedings of IEEE FOCS*, 2004, pp. 499-508.

[6] Hongwu Ma and An-Ping Zeng, "Reconstruction of metabolic networks from genome data and analysis of their global structure for various organisms," in *Bioinformatics 19*, 2003, pp. 270-277.

[7] Jon T. Butler, Tsutomu Sasao and Munehiro Matsuura, "Average path length of binary decision diagrams," in *IEEE Transactions on Computers*, 2005, pp. 1041-1053.

[8] Jure Leskovec, Jon Kleinberg and Christos Faloutsos, "Stanford large network dataset collection, Website," in *http://snap.stanford.edu/data/index.html*.

[9] Jure Leskovec, Jon Kleinberg and Christos Faloutsos, "Graph evolution: densification and shrinking diameters," in *ACM Transactions on Knowledge Discovery from Data*, 2007.

[10] Kazuya Okamoto, Wei Chen and Xiang-Yang Li, "Ranking of closeness centrality for large-scale social networks," in *Proceedings of FAW*, 2008, pp. 186-195.

[11] Ke Zhang, Hui Li, Lijuan Qin and Min Wu, "Closeness centrality on BBS reply network," in *Proceeding of IEEE ICM*, 2011, pp. 80-82.

[12] Iassen Halatchliyski, Johannes Moskaliuk, Joachim Kimmerle and Ulrike Cress, "Who integrates the networks of knowledge in Wikipedia," in *Proceedings of ACM WikiSym*, 2010.

[13] Matthew J. Rattigan, Marc Maier and David Jensen, "Using structure indices for efficient approximation of network properties," in *Procedding of ACM SIGKDD*, 2006, pp. 357-366.

[14] Shu Yan Chan and Ian X. Y. Leung and Pietro Lio, "Fast centrality approximation in modular networks," in *Proceedings of ACM CNIKM*, 2009, pp. 31-38.

[15] David Eppstein and Joseph Wang, "Fast approximation of centrality," in *Journal of Graph Algorithms and Applications*, 2004, pp. 39-45.

[16] Ulrik Brandes, Patrick Kenis and Dorothea Wagner, "Communicating centrality in policy network drawings," in *IEEE Transactions on Visualization and Computer Graphics*, 2003, pp. 241-253.

[17] Ulrik Brandes and Christian Pich, "Centrality estimation in large networks," in *International Journal of Bifurcation and Chaos in Applied Sciences and Engineering*, 2007, pp. 2303-2318.

[18] Christopher C. Yang and Xuning Tang, "Social networks integration and privacy preservation using subgraph generalization," in *Proceedings of ACM CSI-KDD*, 2009, pp. 53-61.

[19] Ling Jin, Jae Yeol Yoon, Young Hee Kim and Ung Mo Kim, "Based on analyzing closeness and authority for ranking expert in social network," in *Proceedings of ICIC*, 2011, pp. 277-283.

[20] Martin Pinzger, Nachiappan Nagappan and Brendan Murphy, "Can developer-module networks predict failures," in *Proceedings of ACM SIGSOFT*, 2008, pp. 2-12.

[21] Jurgen Pfeffer and Kathleen M. Carley, "k-Centralities: Local approximations of global measures based on shortest paths," in *Proceedings of ACM WWW Companion*, 2012, pp. 1043-1050.

[22] Klaus Wehmuth and Artur Ziviani, "Distributed assessment of the closeness centrality ranking in complex networks," in *Proceedings of ACM SIMPLEX*, 2012, pp. 43-48.

[23] Yeon-Sup Lim, Daniel S. Menasche, Bruno Ribeiro, Don Towsley and Prithwish Basu, "Online estimating the k central nodes of a network," in *Proceedings of IEEE NSW*, 2011, pp. 118-122.

[24] Jianwei Niu, Bin Dai, Jinkai Guo and Chao Tong, "DGCCF: data gathering based on closeness centrality forwarding in opportunistic mobile sensor networks," in *Proceeding of IEEE WiCOM*, 2011, pp. 1-6.

[25] Reka Albert and Albert-Laszlo Barabasi, "Statistical mechanics of complex networks," in *Reviews of Modern Physics*, 2002, pp. 47-97.

[26] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to algorithms," in *the MIT Press*, 2009.