Fully Dynamic Betweenness Centrality Maintenance on Massive Networks

Takanori Hayashi
The University of Tokyo
Tokyo, 113-0033, Japan
thayashi@is.s.u-tokyo.ac.jp

Takuya Akiba
National Institute of Informatics,
CIT STAIR Lab & JST PRESTO
takiba@nii.ac.jp

Yuichi Yoshida
National Institute of Informatics
Tokyo, 101-8430, Japan
yyoshida@nii.ac.jp

ABSTRACT

Measuring the relative importance of each vertex in a network is one of the most fundamental building blocks in network analysis. Among several importance measures, *betweenness centrality*, in particular, plays key roles in many real applications. Considerable effort has been made for developing algorithms for static settings. However, real networks today are highly dynamic and are evolving rapidly, and scalable dynamic methods that can instantly reflect graph changes into centrality values are required.

In this paper, we present the first fully dynamic method for managing betweenness centrality of all vertices in a large dynamic network. Its main data structure is the weighted hyperedge representation of shortest paths called *hypergraph sketch*. We carefully design dynamic update procedure with theoretical accuracy guarantee. To accelerate updates, we further propose two auxiliary data structures called *two-ball index* and *special-purpose reachability index*. Experimental results using real networks demonstrate its high scalability and efficiency. In particular, it can reflect a graph change in less than a millisecond on average for a large-scale web graph with 106M vertices and 3.7B edges, which is several orders of magnitude larger than the limits of previous dynamic methods.

1. INTRODUCTION

Measuring the relative importance of each vertex in a network is one of the most fundamental tasks in network analysis. Among several importance measures, *betweenness centrality* [2, 16], in particular, plays key roles in many applications, e.g., finding important actors in terrorist networks [13, 24], estimating lethality in biological networks [14, 21], detecting hierarchical community structures [19, 29]and identifying crucial intersections in road networks [28]. Owing to its popularity, almost all descriptive literature on network analysis introduces betweenness centrality [28, 37]. Moreover, betweenness centrality is often adopted in various network analysis software such as SNAP [26], WebGraph [9], Gephi, NodeXL, and NetworkX.

Betweenness centrality measures the importance of each vertex in terms of the number of shortest paths passing through the vertex. For static settings, considerable effort has been devoted to developing fast, exact, and approximate algorithms [3, 4, 10, 11, 18, 30, 32, 36]. Moreover, recent approximation algorithms can accurately estimate the betweenness centrality of all vertices in near-linear time, which is sufficiently fast even for today's very large networks with billions of edges.

However, certain real-world networks today are highly dynamic and are evolving rapidly. Examples include, but not limited to, online social networks. Furthermore, an individual's activities are often *bursty* with regard to temporal locality [5]. Therefore, in order to realize and improve real-time network-aware applications (e.g., burst detection, trend analysis, and socially-sensitive search ranking), instantly reflecting graph changes into centrality values is crucial.

In this regard, even if we use the recent sophisticated algorithms, the naive approach of re-running the static algorithms after each update would be too expensive for the large networks of today. Hence, considerable research is being conducted to develop efficient dynamic methods [7, 20, 22, 23, 25]. However, current dynamic methods are still limited in terms of scalability. Exact dynamic methods require $\Omega(n^2)$ preprocessing time and $\Omega(n^2)$ data structure size [20, 22, 23, 25]. The only dynamic approximate method by Bergamini *et al.* [7] is semi-dynamic, i.e., it does not support any removal. Moreover, although its scalability is fairly better than those of exact dynamic methods, it can barely processes graphs with tens of millions of edges, and it cannot handle larger graphs with billions of edges. The method by Bergamini *et al.* is based on shortest-path sampling by Riondato *et al.* [32].

1.1 Contributions

In this paper, we present the first fully dynamic approximate betweenness centrality algorithm, which maintains the betweenness centrality of all vertices along with the internal data structures. It is fully dynamic in the sense that it can instantly reflect insertions and deletions of vertices and edges. Our method has several orders of magnitude better scalability, and unlike previous methods, it can handle networks with billions of edges, while maintaining a low error rate. Moreover, even on very large networks, the proposed method can update the internal data structure and centrality values of all the affected vertices in milliseconds, which is an order of magnitude faster than Bergamini's method [7].

Proceedings of the VLDB Endowment, Vol. 9, No. 2

Our method is also based on sampling. However, unlike Bergamini's method [7], which selects a single shortest path between each sampled pair of vertices, our focus is to maintain all the shortest paths between each sampled pair of vertices. Thereby, the necessary number of pairs of vertices are saved. We represent the shortest paths in a data structure named *hypergraph sketch*, which is inspired from a previous static method [36]. The betweenness centrality of any vertex can be easily obtained from the hypergraph sketch.

Accordingly, our main technical challenge is correctly and efficiently updating the hypergraph sketch. This is non-trivial because the part of the hypergraph sketch that requires modifications cannot be detected locally. Moreover, even if we overlook efficient updating, it is far from trivial to correctly update the hypergraph sketch without reconstruction, because it is important to consider the probability distribution of sampled pairs for accurate centrality estimation.

We summarize our main technical contributions for addressing these challenges as follows:

- **Hypergraph Sketch:** Our main data structure is the weighted hyperedge representation of shortest paths called hypergraph sketch. We carefully design an update procedure that probabilistically replaces vertex pairs using non-uniform distribution. We theoretically guarantee that it ensures estimation accuracy against any sequence of graph changes.
- **Two-Ball Index:** To efficiently detect the parts of hypergraph sketches that require modification, we propose an auxiliary data structure named *two-ball index*. It basically maintains subtrees of shortest path trees and is designed to exploit *small-world property* of networks [27,34] to achieve a small data structure size.
- **Special-Purpose Reachability Index:** Although solely using the two-ball index is sufficiently efficient for (almost) strongly connected graphs, its performance degrades terribly when handling unreachable pairs. To cope with this issue, as another auxiliary data structure, we propose a special-purpose dynamic reachability index, which maintains the reachability of fixed pairs of vertices rather than all pairs.

We conducted extensive experiments using real large networks for demonstrating the performance of our method and for empirically analyzing our new techniques. The results validate the high scalability of our method, in that the proposed method can handle large-scale complex networks with over 1B edges in memory. In addition, each update is processed in several milliseconds on all the datasets that were examined in this study. Moreover, the proposed method's processing time is faster than that of previous state-of-the-art algorithms for dynamic networks.

Organization. The remainder of this paper is organized as follows. We discuss related work in Section 2. In Section 3, we explain definitions and notations. We present our hypergraph sketches, two-ball index and special-purpose reachability index in Sections 4, 5 and 6, respectively. In Section 7, we present experimental results. We analyze the time and space complexity of our method in Section 8. Finally in Section 9, we present our conclusion.

2. RELATED WORK

2.1 Algorithms for Static Networks

As explained in the previous section, many algorithms have been proposed for static settings. The asymptotically fastest exact betweenness centrality algorithm was proposed by Brandes [10]. This algorithm computes, for each vertex v, the contribution of v to the centrality of all other vertices by conducting a single BFS and a backward aggregation. It runs in $\Theta(nm)$ time on unweighted networks and $\Theta(nm + n^2 \log n)$ time on weighted networks, and requires $\Theta(n+m)$ space. There have been a few attempts to speedup Brandes' algorithm by exploiting the characteristics of complex networks [4,15,30], but these methods also need $\Theta(nm)$ time to compute betweenness centrality in the worst case. Therefore, computing exact betweenness centrality on large real-world networks is prohibitively expensive.

To handle large real-world networks, many approximate algorithms have been proposed [3,11,18,32,36]. These methods are based on random sampling and allow us to estimate betweenness centralites of vertices with acceptable accuracy in almost linear time. Brandes and Pich [11] and Geisberger *et al.* [18] estimate the betweenness centralities by sampling a small set of vertices and accumulating their contributions to the betweenness centralities of other vertices. Bader *et al.* [3] estimate the centrality of one specified vertex by adaptive sampling. Riondato and Kornaropoulos [32] approximate the centrality values by sampling peer-to-peer shortest paths instead of vertices. Yoshida [36] estimates the centrality values by sampling vertex pairs. In algorithms by [11,36], in order to guarantee that the error of estimated centrality of every vertex is less than ϵ with probability $1-\delta$, we need to sample $\Omega(\frac{1}{\epsilon^2} \log \frac{n}{\delta})$ vertices or vertex pairs. In contrast, the method due to Riondato *et al.* only have to sample $\Omega(\frac{1}{\epsilon^2} \log \frac{VD(G)}{\delta})$ shortest paths, where $VD(G)$ is the diameter of a graph G when we ignore the weight of edges. We note that $VD(G)$ is often much smaller than n on complex networks.

2.2 Algorithms for Dynamic Networks

In recent years, several studies have proposed on exact dynamic algorithms [20, 22, 23, 25]. These algorithms store auxiliary data structures to avoid re-computing the betweenness centralities from scratch. Lee *et al.* [25] decompose a graph into subgraphs and reduce the search space for each update. They achieve several times faster updates in comparison with Brandes' exact algorithm. Kas *et al.* [22] extend the dynamic all-pair shortest path algorithm by Ramalingam *et al.* [31] to enable dynamic updates of betweenness centrality. Green *et al.* [20] accelerates dynamic updates by maintaining a shortest path tree (SPT) rooted at each vertex in a graph and updating distance and the number of shortest paths from a root. Kourtellis *et al.* [23] further improve the scalability by storing SPTs on disk in a compact format and distributing the computation. Among exact dynamic algorithms, only this algorithm can handle million-node networks, although it uses 100 machines and 16TB of space. However, all of these exact algorithms require $\Omega(n^2)$ space to store their data structures and it is difficult to handle large networks in memory.

The first semi-dynamic approximate algorithm was proposed by Bergamini *et al.* [7], which can handle each edge

insertion on million-node networks in less than ten milliseconds and yield further speedups by processing batches of edge insertions. The betweenness centrality estimation method of this algorithm is based on [32]. For each shortest path, this algorithm stores an SPT rooted at one endpoint of it. By maintaining the shortest path distance and the number of shortest paths from the root of each SPT, the contribution of each shortest path is kept up to date. However, $\Omega(n)$ space is required to store the information in each SPT and thousands of SPTs might be used to estimate betweenness centrality. As a result, this algorithm faces difficulty in handling billion-scale real-world networks in memory.

3. PRELIMINARIES

Notations. In this study, a network is represented by an unweighted directed graph $G = (V, E)$, where V is a set of n vertices and E is a set of m edges. For a vertex $v \in V$, the set of out-neighbors and in-neighbors are denoted by $\overrightarrow{N}(v)$ and $\overleftarrow{N}(v)$, respectively. Let $d(u, v)$ be the (shortest path) distance from a vertex u to a vertex v. If there is no path from u to v, $d(u, v)$ is considered as ∞. For two vertices s and t, let $P(s, t)$ be the set of vertices on at least one of the shortest paths from s to t, i.e., $P(s, t) = \{v \in V \mid d(s, v) + d(v, t) = d(s, t)\}$.

Betweenness Centrality. The number of shortest paths from s to t is denoted by $\sigma(s, t)$, and the number of shortest paths from s to t passing through v is denoted by $\sigma(s, t \mid v)$. If $s = v$ or $t = v$, let $\sigma(s, t \mid v)$ be zero. The *betweenness centrality* of a vertex v on a graph G is

$$C(v) = \sum_{s,t \in V} \frac{\sigma(s, t \mid v)}{\sigma(s, t)}.$$

3.1 Dynamic Graphs and Problem Definitions

When we consider dynamic networks, G denotes the latest graph, and G_τ denotes the graph at time τ. For simplicity, we assume that time is described by positive integers (i.e., graph snapshots are $G_1, G_2, \ldots$). We add subscripts to indicate the time we consider. For example, $d_\tau(u, v)$ denotes the distance between u and v in the graph G_τ.

In this paper, we study methods that, given a dynamic graph, construct and manage a data structure to manage the approximate value of betweenness centrality $C_\tau(v)$ for each vertex $v \in V$, where τ denotes the latest time.

4. HYPERGRAPH SKETCH AND ITS EXTENSION TO THE DYNAMIC SETTING

In this section, we first introduce the data structure, called the hypergraph sketch, which is used to estimate the betweenness centrality of vertices. Then, we explain how to update the hypergraph sketch when the graph is dynamically updated. For simplicity, we only explain a high-level description of our method in this section, which is sufficient to verify its correctness. Although we use several auxiliary data structures to achieve scalability, the detailed implementation is deferred to Sections 5 and 6.

4.1 Hypergraph Sketch for Static Graphs

We now describe the hypergraph sketch for static graphs. Given a graph $G = (V, E)$ and an integer M, we sample a set S of M vertex pairs uniformly at random. For each chosen vertex pair $(s, t) \in S$, a hyperedge with an auxiliary information on each vertex is added to a hypergraph H. Specifically, $e_{st} := \{(v, \sigma(s, v), \sigma(v, t)) \mid v \in P(s, t)\}$ is added to H. We call H a hypergraph sketch. In the following sections, we simply call this set a *hyperedge*. For a hyperedge e_{st}, $V(e_{st})$ denotes the set of vertices contained in e, that is, $P(s, t)$.

By conducting a (bidirectional) BFS from s to t, the set $P(s, t)$ and the number of shortest paths $\sigma(s, v)$ and $\sigma(v, t)$ for every v can be computed in $O(m)$ time. Hence, we can construct a hypergraph sketch with M hyperedges in $O(Mm)$ time.

We define the *weight* of v on a hypergraph sketch H by $w_H(v) := \sum_{(s,t) \in S} \frac{\sigma(s,t|v)}{\sigma(s,t)} = \sum_{(s,t) \in S} \frac{\sigma(s,v) \cdot \sigma(v,t)}{\sigma(s,t)}$. Given a vertex v, we output $\widetilde{C}_H(v) = \frac{n^2}{M} w_H(v)$ as an estimation of $C(v)$. The value $\widetilde{C}_H(v)$ is an unbiased estimator of $C(v)$ that is well concentrated:

THEOREM 1. *[36] We have $\mathbf{E}_H[\widetilde{C}_H(v)] = C(v)$ for any $v \in V$. Moreover for $\epsilon, \delta > 0$, by choosing $M = \Omega(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$, we have $\Pr[|\widetilde{C}_H(v) - C(v)| \geq \epsilon n^2] \leq \delta$ for any $v \in V$.*

In our method, we always keep the value of $w_H(v)$ with respect to the latest graph for every $v \in V$. Hence, we can answer the estimated betweenness centrality of any vertex in $O(1)$ time.

We also note that complex networks have vertices though which most vertex pairs have a shortest path pass [6]. In other words, betweenness centralities of those vertices are $\Omega(n^2)$. Since we are interested in only those vertices in many applications, the guarantee given by Theorem 1 is sufficient for practical purpose.

4.2 Dynamic Updates of Hypergraph Sketch

In this section, we describe how to dynamically update the hypergraph sketch and verify its correctness.

4.2.1 Algorithm Description

Suppose that a graph G_τ is obtained by updating a graph $G_{\tau-1}$. We want to efficiently obtain the hypergraph sketch for G_τ using the hypergraph sketch H for $G_{\tau-1}$.

Adding or deleting an edge is trivial. We simply need to update the distance information of hyperedges. What is non-trivial is *efficiently* updating it, and we will look at this issue in Sections 5 and 6.

When adding or deleting vertices, we sometimes need to replace vertex pairs in the sampled set S since we need a set of pairs that are uniformly sampled from the current vertex set V_τ. The pseudo-code of our updating procedures is given in Algorithm 1.

4.2.2 Correctness

Let $\mathcal{H}_1, \mathcal{H}_2, \ldots$ be the distribution of hypergraph sketches for $G_1, G_2, \ldots$, respectively, where G_τ is obtained from $G_{\tau-1}$ by applying the update method described in the previous section. For any τ and a hypergraph sketch H for G_τ constructed by our algorithm, the value $\widetilde{C}_H(v)$ is an unbiased estimator of $C_\tau(v)$ that is well concentrated:

THEOREM 2. *For any τ, the following hold: We have $\mathbf{E}_{H \sim \mathcal{H}_\tau}[\widetilde{C}_H(v)] = C_\tau(v)$ for any $v \in V$. Moreover for $\epsilon, \delta > 0$, by choosing $M = \Omega(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$, we have $\Pr_{H \sim \mathcal{H}_\tau}[|\widetilde{C}_H(v) - C_\tau(v)| \geq \epsilon n^2] \leq \delta$ for any $v \in V$.*

Algorithm 1 Vertex operations

procedure AddVertex(H, v)
  Let G_τ be obtained from $G_{\tau-1}$ by adding v.
  for each $e_{st} \in E(H)$ **do**
    continue with probability $|V_{\tau-1}|^2/|V_\tau|^2$.
    Sample $(s', t') \in (V_\tau \times V_\tau) \setminus (V_{\tau-1} \times V_{\tau-1})$.
    Replace e_{st} by the hyperedge $e_{s't'}$ made from (s', t').

procedure RemoveVertex(H, v)
  Let G_τ be obtained from $G_{\tau-1}$ by deleting v.
  for each $e_{st} \in E(H)$ **do**
    if $s \neq v$ and $t \neq v$ **then continue**.
    Sample $(s', t') \in V_\tau \times V_\tau$ uniformly at random.
    Replace e_{st} by the hyperedge $e_{s't'}$ made from (s', t').

In words, for any τ and a vertex $v \in V$, the probability that $C_H(v)$ is far apart from $C_\tau(v)$ can be made arbitrarily small by choosing M sufficiently large. In particular, the accuracy of our method does not deteriorate over time.

Let $\mathcal{S}_\tau$ be the distribution of the set S of vertex pairs at time τ, that is, $H \sim \mathcal{H}_\tau$ is obtained by sampling $S \sim \mathcal{S}_\tau$. Then compute a hyperedge for each pair in S. To prove Theorem 2, by Theorem 1, it suffices to show that $\mathcal{S}$ is a uniform distribution over the sets of M vertex pairs in V_τ.

Proof of Theorem 2. We use the induction on τ. When $\tau = 1$, the distribution $\mathcal{S}_\tau$ is clearly uniform from the construction.

Let G_τ be the current graph, and assume that the distribution $\mathcal{S}_{\tau-1}$ is a uniform distribution over sets of M vertex pairs in $V_{\tau-1}$. Suppose that we are to modify G. When adding or deleting an edge, we do not modify the pairs, and hence $\mathcal{S}_\tau$ remains a uniform distribution. When adding or deleting a vertex, $\mathcal{S}_\tau$ remains a uniform distribution from Lemmas 3 and 4, as given below. □

Lemma 3. *Suppose that $G_\tau = (V_\tau, E_\tau)$ is obtained from $G_{\tau-1} = (V_{\tau-1}, E_{\tau-1})$ by adding a vertex v, and $\mathcal{S}_{\tau-1}$ is a uniform distribution over sets of M vertex pairs in $V_{\tau-1}$. Then, the distribution $\mathcal{S}_\tau$ is also a uniform distribution over sets of M vertex pairs in V_τ.*

Proof. Since $\mathcal{S}_{\tau-1}$ is a uniform distribution, the process of sampling $S \sim \mathcal{S}_\tau$ can be regarded as follows. Each time we sample a pair $(s, t) \in V_{\tau-1} \times V_{\tau-1}$ uniformly at random, we keep it as is with probability $|V_{\tau-1}|^2/|V_\tau|^2$, and replace it with a pair sampled from $(V_\tau \times V_\tau) \setminus (V_{\tau-1} \times V_{\tau-1})$ uniformly at random with the remaining probability. Hence, $\mathcal{S}_\tau$ is also a uniform distribution. □

Lemma 4. *Suppose that $G_\tau = (V_\tau, E_\tau)$ is obtained from $G_{\tau-1} = (V_{\tau-1}, E_{\tau-1})$ by deleting a vertex v, and $\mathcal{S}_{\tau-1}$ is a uniform distribution over sets of M vertex pairs in $V_{\tau-1}$. Then, the distribution $\mathcal{S}_\tau$ is also a uniform distribution over sets of M vertex pairs in V_τ.*

Proof. Since $\mathcal{S}_{\tau-1}$ is a uniform distribution, the process of sampling $S \sim \mathcal{S}_\tau$ can be regarded as follows. Each time we sample a pair $(s, t) \in V_{\tau-1} \times V_{\tau-1}$ uniformly at random, we keep it as is if $(s, t) \in V_\tau \times V_\tau$ and replace it with a pair sampled from $V_\tau \times V_\tau$ uniformly at random otherwise. Hence, $\mathcal{S}_\tau$ is also a uniform distribution. □

5. TWO-BALL INDEX

In order to compute approximate betweenness centrality in a dynamic setting, we need to be able to update the hyperedge e_{st} efficiently for each vertex pair $(s, t) \in S$ in the hypergraph sketch. In this section, we describe our *two-ball index* (*TB-index*), which addresses this issue.

5.1 Data Structure

For each sampled vertex pair $(s, t) \in S$, the hyperedge e_{st} must be updated when the set of shortest paths from s to t is changed by dynamic updates. Hence, quickly detecting the change of shortest paths from s to t is important to efficiently update the approximate betweenness centralities.

A straightforward way of detecting the change of shortest paths from s to t is conducting a (bidirectional) BFS from s to t. Obviously, traversing the whole graph is inefficient for computing shortest paths between thousands of vertex pairs. Another approach is to maintain the shortest path tree (SPT) rooted at the vertex s. Several dynamic algorithms [7, 20, 23] are based on variants of this approach because the incremental updates of an SPT can be processed quickly. However, keeping thousands of SPTs on billion-scale networks requires huge amount of space, and it is difficult to handle such networks in memory.

To achieve both high performance and scalability, we construct and store data structures using *balls*. We define a set of pairs $\overrightarrow{B}(v, d_v)$ and $\overleftarrow{B}(v, d_v)$ as follows.

$$\overrightarrow{B}(v, d_v) = \{(w, d(v, w)) \mid d(v, w) \leq d_v\}$$
$$\overleftarrow{B}(v, d_v) = \{(w, d(w, v)) \mid d(w, v) \leq d_v\}$$

These sets have important roles in our method. We describe $\overrightarrow{B}(v, d_v)$ and $\overleftarrow{B}(v, d_v)$ *balls* whose *centers* are v and *radius* are d_v. The set of vertices whose distance from v is less than or equal to d_v is denoted by $V(\overrightarrow{B}(v, d_v))$. Let $\overrightarrow{B}(v, d_v)[w] = d(v, w)$ if $w \in V(\overrightarrow{B}(v, d_v))$. Otherwise, $\overrightarrow{B}(v, d_v)[w] = \infty$. We define $V(\overleftarrow{B}(v, d_v))$ and $\overleftarrow{B}(v, d_v)[w]$ in the same way as $V(\overrightarrow{B}(v, d_v))$ and $\overrightarrow{B}(v, d_v)[w]$.

Then, our data structures are as follows.

- A set $\Delta_{s,t} = \{\delta_s(v) \mid v \in P(s, t)\}$. Here $\delta_s(v)$ is supposed[1] to be $d(s, v)$. This information is not needed to estimate the betweenness centrality, but it is useful to detect the change of shortest paths from s to t when an edge or a vertex is deleted.
- Two sets $\overrightarrow{\beta}_s$ and $\overleftarrow{\beta}_t$ of vertices with distance information, which are supposed to be the balls $\overrightarrow{B}(s, d_s)$ and $\overleftarrow{B}(t, d_t)$, respectively. Here, two non-negative integers d_s and d_t satisfy $d_s + d_t + 1 = d(s, t) - x$. The values of d_s and d_t are determined when the two balls are constructed. The newly formed shortest paths from s to t by edge insertions are detected by using these balls.

The data structures are parameterized by a non-negative integer x, which allows the trade-off between the index size and the update time: As the value of x increases, the index size and the update time of edge deletions decrease whereas the update time of edge insertions increases. Two balls $\overrightarrow{B}(s, d_s)$ and $\overleftarrow{B}(t, d_t)$ never overlap. When $x = 0$, two balls are directly connected by an edge on a shortest path between s and t, i.e., the distance between the two balls is 1. By increasing x, two balls become smaller, and the two balls become distant.

[1] We use the expression "supposed to be" here because, while $\delta_s(v)$ usually corresponds to $d(s, v)$, during update procedure, $\delta_s(v)$ may temporarily differ from $d(s, v)$.

The set of triplets

$$\mathcal{I} = \left\{ (\Delta_{s,t}, \overrightarrow{\beta}_s, \overleftarrow{\beta}_t) \mid (s,t) \in S \right\}$$

is the TB-index for the set S of sampled vertex pairs.

The worst-case space complexity of each triplet $(\Delta_{s,t}, \overrightarrow{\beta}_s, \overleftarrow{\beta}_t)$ is $O(n)$. However, the size of $\overrightarrow{B}(s, d_s)$ and $\overleftarrow{B}(t, d_t)$ tends to be much smaller than n on real-world complex networks in practice. This is because these networks have the small-world property, i.e., the average distance between two vertices is often $O(\log n)$ [34]. Hence the radius of two balls d_s and d_t tend to be quite small. Moreover, the size of the hyperedge e_{st} is further smaller than the size of two balls. The compactness of the TB-index on real-world networks is empirically shown in Section 7.

5.2 Index Construction

In this subsection, we explain how to efficiently construct a triplet $(\Delta_{s,t}, \overrightarrow{\beta}_s, \overleftarrow{\beta}_t)$ for a fixed vertex pair (s,t). Then, the construction of $\mathcal{I}$ is simply performed by computing the triplet for each sampled vertex pair $(s,t) \in S$. This procedure is used when we start dealing with a new dynamic graph, as well as when sampled pairs are changed due to vertex addition and removal (see Section 4.2).

First, two balls $\overrightarrow{B}(s, d_s)$ and $\overleftarrow{B}(t, d_t)$ are computed based on bidirectional BFS. We initialize d_s and d_t to be zero. Then, as long as $\overrightarrow{B}(s, d_s)$ and $\overleftarrow{B}(t, d_t)$ share no vertex, we increment d_s if the size of $\overrightarrow{B}(s, d_s)$ is smaller than that of $\overleftarrow{B}(t, d_t)$, and increment d_t otherwise. Then, we set $\overrightarrow{\beta}_s = \overrightarrow{B}(s, d_s)$ and $\overleftarrow{\beta}_t = \overleftarrow{B}(t, d_t)$. If we find $\overrightarrow{B}(s, d_s + 1) = \overrightarrow{B}(s, d_s)$ or $\overleftarrow{B}(t, d_t + 1) = \overleftarrow{B}(t, d_t)$ along the way, then there is no path from s to t. In this case, we simply set $\overrightarrow{\beta}_s = \overrightarrow{\beta}_t = \emptyset$ to save space.

When we find that $\overrightarrow{B}(s, d_s)$ and $\overleftarrow{B}(t, d_t)$ share a vertex, the vertex set $P(s,t)$ is computed by using $\overrightarrow{B}(s, d_s)$ and $\overleftarrow{B}(t, d_t)$. Let $C = V(\overrightarrow{B}(s, d_s)) \cap V(\overleftarrow{B}(t, d_t))$ be the set of shared vertices. Let P_s and P_t be the set of vertices on the shortest paths from s to C and the shortest paths from C to t, respectively. Note that $P(s,t)$ equals $P_s \cup P_t \cup C$, because $C = \{v \in P(s,t) \mid d(s,v) = d_s\}$ and every shortest path from s to t passes through C. Two vertex sets P_s and P_t can be easily obtained by conducting BFSes from C to s and t. After the vertex set $P(s,t)$ is obtained, we decrease the radius d_s and d_t one by one as long as $d_s + d_t + 1 > d(s,t) - x$ holds in order to save space. More specifically, in each iteration, the radius of the larger ball is decreased. Finally, the set $\Delta_{s,t}$ is computed by conducting a BFS on $P(s,t)$ (when $\overrightarrow{\beta}_s$ and $\overleftarrow{\beta}_t$ are empty, we simply set $\Delta_{s,t} = \emptyset$ to save the space). Since the number of vertices in $P(s,t)$ is much smaller than n, the BFS on $P(s,t)$ can be conducted quickly. Similarly, we update the distance information in e_{st} by conducting BFSes on $P(s,t)$.

Since the sizes of $P(s,t)$ and the two balls $\overrightarrow{B}(s, d_s)$ and $\overleftarrow{B}(t, d_t)$ are often much smaller than n on real-world networks, we use hash tables to keep $\sigma(s,v)$ and $\sigma(v,t)$ for each $v \in V(e_{st})$, $\overrightarrow{\beta}_s[v]$ for each $v \in V(\overrightarrow{\beta}_s)$, and $\overleftarrow{\beta}_t[v]$ for each $v \in V(\overleftarrow{\beta}_t)$ in order to save the space. As with balls, $\overrightarrow{\beta}_s[v]$ denotes $d(s,v)$, which is stored in the hash table.

Algorithm 2 Update $\overrightarrow{B}(s, d_s)$ after edge (u,v) is inserted

```
1: procedure InsertEdgeIntoBall(u, v, β→_s)
2:     Q ← An empty FIFO queue.
3:     if β→_s[v] > β→_s[u] + 1 then
4:         β→_s[v] ← β→_s[u] + 1; Q.push(v).
5:     while not Q.empty() do
6:         v ← Q.pop().
7:         if β→_s[v] = d_s then continue.
8:         for each (v, c) ∈ E do
9:             if β→_s[c] > β→_s[v] + 1 then
10:                β→_s[c] ← β→_s[v] + 1; Q.push(c).
```

5.3 Incremental Update

In this subsection, we present how to efficiently update each triplet $(\Delta_{s,t}, \overrightarrow{\beta}_s, \overleftarrow{\beta}_t)$ in the TB-index as well as the hyperedge e_{st} when the graph is dynamically updated. Throughout this section, τ denotes the latest time.

5.3.1 *Edge Insertion*

Suppose that an edge (u,v) is inserted. Two balls $\overrightarrow{\beta}_s$ and $\overleftarrow{\beta}_t$ are updated first; then, we update $\Delta_{s,t}$ and the hyperedge e_{st}. The value of d_s and d_t are not modified although we may have a new shortest path between s and t.

Algorithm 2 shows the pseudo-code for updating each ball after edge insertion. Balls are updated in a similar manner as in the case of previous methods for maintaining SPTs [1, 17, 33]. In the case that $\overrightarrow{\beta}_s[u] + 1 < \overrightarrow{\beta}_s[v]$, the distance from s to v becomes smaller by passing through the new edge (u,v), and $\overrightarrow{\beta}_s[v]$ is updated to $\overrightarrow{\beta}_s[u] + 1$. Then we recursively examine out-neighbors of v. In other words, the ball $\overrightarrow{\beta}_s$ is updated by conducting a BFS from v on vertices whose distance from s is at most d_s. The ball $\overleftarrow{\beta}_t$ is updated in the same manner as above.

The set $\Delta_{s,t}$ is updated by using the updated balls $\overrightarrow{\beta}_s = \overrightarrow{B}_\tau(s, d_s)$ and $\overleftarrow{\beta}_t = \overleftarrow{B}_\tau(t, d_t)$. The change of $\Delta_{s,t}$ occurs when new shortest paths are formed by adding (u,v). To quickly detect newly formed shortest paths, two more balls $\overleftarrow{B}_\tau(u, d_u)$ and $\overrightarrow{B}_\tau(v, d_v)$ are used. The radii of two balls d_u and d_v are initialized to zero. The radius of $\overleftarrow{B}_\tau(u, d_u)$ is increased one by one by conducting a BFS from u while $\overrightarrow{B}_\tau(s, d_s)$ and $\overleftarrow{B}_\tau(u, d_u)$ share no vertex, and the radius of $\overrightarrow{B}_\tau(v, d_v)$ is increased one by one as long as $\overrightarrow{B}_\tau(v, d_v)$ and $\overleftarrow{B}_\tau(t, d_t)$ share no vertex. However, if vertices u and v are distant from vertices s and t, a large part of the graph might be visited before the whole $\overleftarrow{B}_\tau(u, d_u)$ and $\overrightarrow{B}_\tau(v, d_v)$ are obtained even when there is no newly formed shortest paths passing through the edge (u,v). To avoid such an unnecessary computation, we carefully introduce upper bounds r_u and r_v on d_u and d_v, respectively, and stop BFSes when the radii reach these upper bounds. We consider the following four cases:

1. The case where $u \in V(\overrightarrow{B}_\tau(s, d_s))$ and $v \in V(\overleftarrow{B}_\tau(t, d_t))$: In this case, we can immediately detect that a new shortest path from s to t is formed.
2. The case where $u \in V(\overrightarrow{B}_\tau(s, d_s))$ and $v \notin V(\overleftarrow{B}_\tau(t, d_t))$: In this case, we already know that $d(s,u) = \overrightarrow{B}_\tau(s, d_s)[u] = \overrightarrow{\beta}_s[u]$. Thus, the following lemma holds.

LEMMA 5. *If there exists a shortest path from s to t passing through the edge (u,v), then there exists $d_v \leq d_{\tau-1}(s,t) - \overrightarrow{B}_\tau(s,d_s)[u] - d_t - 1$ such that $V(\overrightarrow{B}_\tau(v,d_v)) \cap V(\overleftarrow{B}_\tau(t,d_t)) \neq \emptyset$.*

Therefore, we can set $r_v = d_{\tau-1}(s,t) - \overrightarrow{\beta}_s[u] - d_t - 1$. Note that the index has the information of $d_{\tau-1}(s,t)$.

3. The case where $u \notin V(\overrightarrow{B}_\tau(s,d_s))$ and $v \in V(\overleftarrow{B}_\tau(t,d_t))$: As in the previous case, we can set $r_u = d_{\tau-1}(s,t) - \overleftarrow{\beta}_t[v] - d_s - 1$.

4. The case where $u \notin V(\overrightarrow{B}_\tau(s,d_s))$ and $v \notin V(\overleftarrow{B}_\tau(t,d_t))$: In this case, the following lemma holds.

LEMMA 6. *If there exists a shortest path from s to t passing through the edge (u,v), then there exist $d_u, d_v \leq x$ such that $V(\overrightarrow{B}_\tau(s,d_s)) \cap V(\overleftarrow{B}_\tau(u,d_u)) \neq \emptyset$ and $V(\overrightarrow{B}_\tau(v,d_v)) \cap V(\overleftarrow{B}_\tau(t,d_t)) \neq \emptyset$.*

PROOF. Assume that there are no such d_u and d_v. If $V(\overrightarrow{B}_\tau(s,d_s)) \cap V(\overleftarrow{B}_\tau(u,d_u)) = \emptyset$, then we have

$$d_\tau(s,u) + d_\tau(u,v) + d_\tau(v,t) \geq d_\tau(s,u) + 1 + d_t$$
$$> d_s + x + 1 + d_t = d_{\tau-1}(s,t)$$

Hence, there is no newly formed shortest path from s to t, which is a contradiction. Similarly, the case where $V(\overrightarrow{B}_\tau(v,d_v)) \cap V(\overleftarrow{B}_\tau(t,d_t)) = \emptyset$ reaches a contradiction. □

From the lemma above, we can set $r_u = r_v = x$.

If there exist $d_u \leq r_u$ and $d_v \leq r_v$ such that $V(\overleftarrow{B}(u,d_u)) \cap V(\overrightarrow{B}(s,d_s)) \neq \emptyset$ and $V(\overrightarrow{B}(v,d_v)) \cap V(\overleftarrow{B}(t,d_t)) \neq \emptyset$, then there may exist a newly formed shortest path passing through the edge (u,v). In this case, the set of vertex $P_\tau(s,t)$ is contained in the vertex set $P := V(e_{st}) \cup P_\tau(s,u) \cup P_\tau(v,t)$. Therefore, $\Delta_{s,t}$ and e_{st} can be efficiently updated by conducting BFSes on P. We note that P is often small because of the small average distance.

5.3.2 Edge Deletion

Suppose that an edge (u,v) is deleted from the graph. We first present a procedure for updating the ball $\overrightarrow{\beta}_s$. The ball $\overleftarrow{\beta}_t$ can be updated similarly.

Ball $\overrightarrow{B}(s,d_s)$ changes only if $\{u,v\} \subseteq V(\overrightarrow{B}_{\tau-1}(s,d_s))$ and $\overrightarrow{B}_{\tau-1}(s,d_s)[u]+1 = \overrightarrow{B}_{\tau-1}(s,d_s)[v]$ hold. In this case, $\overrightarrow{\beta}_s$ is updated in the same way as the procedure for maintaining SPTs [17]. Algorithm 3 shows the pseudo-code for updating a ball after an edge deletion. This algorithm mainly consists of two steps.

Step 1: The set U of vertices whose distances from s are increased is obtained by conducting a BFS from v. We first push v into a queue Q.

Let w be the vertex popped from Q. Each neighbor c of w is examined and checked whether there still exists a vertex p that satisfies $(p,c) \in E$ and $\overrightarrow{\beta}_s[p] + 1 = \overrightarrow{\beta}_s[c]$. If there is no such vertex, it implies that the distance $d(s,w)$ is increased after the edge deletion. In this case, w is added to U and enqueued to Q, and neighbors of w are examined recursively.

Algorithm 3 Update $\overrightarrow{B}(s,d_s)$ after edge (u,v) is deleted.

```
1: procedure HasParent(w, β⃗_s)
2:     for each (p, w) ∈ E do
3:         if β⃗_s[p] + 1 = β⃗_s[w] then return true.
4:     return false.
5: procedure DeleteEdgeFromBall(u, v, β⃗_s)
6:     if {u, v} ⊈ V(β⃗_s) ∨ β⃗_s[u] + 1 ≠ β⃗_s[v] then return
7:     Step 1: Collect vertices w with d(s, w) increased.
8:     D ← ∅.
9:     Q ← An empty FIFO queue.
10:    if not HasParent(v, B⃗(s, d_s)) then
11:        D ← {v}.
12:        β⃗_s[v] ← ∞; Q.push(v).
13:    while not Q.empty() do
14:        w ← Q.pop().
15:        for each (w, c) ∈ E do
16:            if β⃗_s[c] < ∞∧ not HasParent(c, β⃗_s) then
17:                D ← D ∪ {c}.
18:                β⃗_s[c] ← ∞; Q.push(c).
19:    Step 2: Update β⃗_s[w] for each w ∈ D.
20:    Q' ← An empty min-based priority queue.
21:    for each w ∈ D do
22:        d^w_min ← min_{p∈V} {β⃗_s[p] | (p, w) ∈ E}.
23:        β⃗_s[w] ← d^w_min + 1; Q'.push(w).
24:    while not Q'.empty() do
25:        w ← Q'.pop().
26:        d ← β⃗_s[w].
27:        if d = d_s then continue.
28:        for each (w, c) ∈ E do
29:            if β⃗_s[c] > d + 1 then
30:                β⃗_s[c] ← d + 1; Q'.push(c).
31:    for each w ∈ D do
32:        Remove w from β⃗_s if β⃗_s[w] > d_s.
```

Step 2: In this step, for each $w \in U$, the distance $d_\tau(s,w)$ is computed so as to determine $\overrightarrow{\beta}_s[w]$. For each vertex $w \in U$, $\overrightarrow{\beta}_s[w]$ is initialized to $d^w_{\min} + 1$, where $d^w_{\min} = \min_{p \in V} \left\{ \overrightarrow{\beta}_s[p] \mid (p,v) \in E \right\}$ and pushed into a min-based priority queue Q'.

Suppose that a vertex w with the minimum $\overrightarrow{\beta}_s[w]$ is popped from Q'. The distance $d_\tau(s,w)$ is determined for each $w \in U$ by conducting Dijkstra's algorithm using the priority queue Q'. After the algorithm ends, for each $w \in U$, the vertex w is deleted from $\overrightarrow{\beta}_s$ if $\overrightarrow{\beta}_s[w]$ is larger than d_s.

Now we describe how to update the hyperedge e_{st}. We only use the information of $\Delta_{s,t}$ to update e_{st}. Note that e_{st} should be updated only if the set of shortest paths from s to v is changed, i.e., $\{u,v\} \subseteq V(e_{st}) \wedge \Delta_{s,t}(u) + 1 = \Delta_{s,t}(v)$ hold. There are two cases to consider:

1. $d_\tau(s,t) \neq d_{\tau-1}(s,t)$: This case happens if and only if $\sigma_{\tau-1}(s,u) = \sigma_{\tau-1}(v,t) = 1$ holds. This condition can be easily checked by looking at variables $\sigma(s,u)$ and $\sigma(v,t)$ in e_{st}. If this happens, the triplet $(\Delta_{s,t}, \overrightarrow{\beta}_s, \overleftarrow{\beta}_t)$ as well as the hyperedge e_{st} are constructed from scratch again.

2. Otherwise: In this case, although $d_\tau(s,t) = d_{\tau-1}(s,t)$ holds, the number of shortest paths may have changed. Since the vertex set $P_\tau(s,t)$ is contained in $V(e_{st})$, we

can obtain e_{st} after the edge deletion by conducting a BFS from s to t on $V(e_{st})$.

5.3.3 Vertex Operations

First, we update the vertex pair as described in Section 4. If the vertex pair is replaced, then we construct the hyperedge as well as the triplet from scratch.

When adding a vertex, we have nothing to do. When removing a vertex, the update procedure is similar to the update procedure when an edge is deleted. We first update two balls $\overrightarrow{\beta}_s$ and $\overleftarrow{\beta}_t$. Then, we detect the set U of vertices such that the distances from the centers of balls are increased, and then, the distances from the centers are determined using Dijkstra's algorithm from U. We also update the hyperedge e_{st} if v is contained in $V(e_{st})$. If the distance from s to t is changed after the vertex removal, the triplet $(\Delta_{s,t}, \overrightarrow{\beta}_s, \overleftarrow{\beta}_t)$ is constructed from scratch. Otherwise, the updated hyperedge e_{st} is obtained by conducting BFS from s and t on $V(e_{st})$.

6. SPECIAL-PURPOSE REACHABILITY INDEX

The algorithm explained in the previous section works well on social networks. However, on web graphs, the performance of our method becomes much worse than that on social networks. This is because most web graphs are directed and there are many pairs of vertices that are unreachable from each other. Since we only keep a triplet of three empty sets for each unreachable vertex pair $(s,t) \in S$ in the TB-index in order to save space, we have to check whether there exists a path from s to t after each edge insertion. Checking the reachability of thousands of vertex pairs is not acceptable for handling dynamic updates efficiently.

Therefore, to efficiently process the update after edge insertions, we need to construct an index that quickly answers and maintains reachability of each vertex pair $(s,t) \in S$. In this section, we present an index for this purpose. By exploiting the structure of web graphs [12], we construct a highly scalable index. Our index outperforms in terms of the performance and scalability compared to state-of-the-art methods that are designed to answer the reachability of any vertex pair [35, 38].

6.1 Data Structure

To quickly answer the reachability of vertex pairs, we want to manage a vertex set R_s for each vertex pair (s,t), where R_s is the set of all vertices that can be reached from s. The reachability from s to t can be easily answered by checking whether $t \in R_s$ or not. Obviously, keeping all the reachable vertices spoils the scalability of our method owing to the large number of reachable vertices. To reduce the number of vertices stored in the index, we exploit the structure of web graphs. On web graphs, the size of vertex set R_s tends to be large when the *largest* strongly connected component of the graph, called the SCC [12], can be reached from s.

We introduce another data structure with which we can reduce the size of the index size for a vertex pair (s,t) such that there is a path from s to t passing through the SCC.

To maintain the reachability of such vertex pairs, we use SPTs rooted at vertices in the SCC. As distance information is not necessary here, any data structure that can maintain

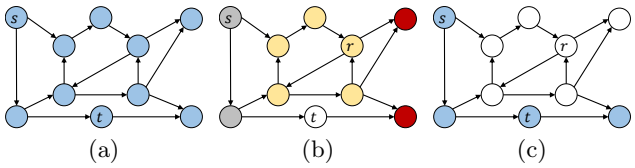


Figure 1: An example of index size reduction. Vertex color indicates its status: Blue is a vertex that should be maintained for vertex pair (s,t), Red is a vertex that can be reached from a vertex r, Gray is a vertex that can reach r, and Yellow is a vertex contained in the same SCC with r.

reachable vertices from a single vertex suffices. However, SPTs can be updated more easily than other alternatives.

Let r be a vertex contained in the SCC. Let $\overrightarrow{T}_r$ and $\overleftarrow{T}_r$ be the SPT rooted at r on the original graph and the graph with each edge reversed, respectively.

We define the reduced reachable vertices $R^{T_r}_{st}$ from s as follows.

$$R^{T_r}_{st} = \begin{cases} \emptyset & \text{if } s \in V(\overrightarrow{T}_r) \wedge t \in V(\overleftarrow{T}_r), \\ R_s \setminus V(\overrightarrow{T}_r) & \text{if } t \notin V(\overrightarrow{T}_r), \\ R_s & \text{otherwise.} \end{cases}$$

Using $\overrightarrow{T}_r$, $\overleftarrow{T}_r$ and $R^{T_r}_{st}$, the reachability from s to t can be easily answered.

LEMMA 7. *There exists a path from s to t if and only if $s \in V(\overrightarrow{T}_r) \wedge t \in V(\overleftarrow{T}_r)$ or $t \in R^{T_r}_{st}$ holds.*

Figure 1 shows an example of index size reduction using our technique. Let us consider a graph and a vertex pair (s,t) as shown in Figure 1a. In this graph, all the vertices in the graph can be reached from a vertex s. Hence, without pruning by using $\overrightarrow{T}_r$ and $\overleftarrow{T}_r$, all the vertices in the graph should be maintained for (s,t). Let us choose a vertex r in the graph as the root of two SPTs $\overleftarrow{T}_r$ and $\overrightarrow{T}_r$. As shown in Figure 1b, the vertex t is not contained in $V(\overrightarrow{T}_r)$, and reduced reachable vertices $R^{T_r}_{st}$ becomes $R_s \setminus V(\overrightarrow{T}_r)$. Thus, we only have to maintain four vertices included in $R^{T_r}_{st}$ instead of ten vertices that can be reached from s (Figure 1c).

Since the SCC will change over time, it is difficult for us to quickly check which vertex is contained in the SCC. Hence, we randomly choose a set of K vertices $\{r_1, r_2, \ldots, r_K\}$ as roots of SPTs. If we set the value K to a moderate value, at least one of the chosen vertices will be contained in the SCC with high probability because of a large number of the vertices belong to the SCC [12]. We define the set $\mathcal{T}$ of pairs of SPTs as follows.

$$\mathcal{T} = \left\{ (\overrightarrow{T}_{r_1}, \overleftarrow{T}_{r_1}), (\overrightarrow{T}_{r_2}, \overleftarrow{T}_{r_2}), \ldots, (\overrightarrow{T}_{r_K}, \overleftarrow{T}_{r_K}) \right\}$$

We denote the set of roots that can be reached from a vertex v by $T_{\text{in}}(v) = \left\{ r_i \mid (\overrightarrow{T}_{r_i}, \overleftarrow{T}_{r_i}) \in \mathcal{T}, v \in V(\overleftarrow{T}_{r_i}) \right\}$. We denote the set of roots that can reach a vertex v by $T_{\text{out}}(v) = \left\{ r_i \mid (\overrightarrow{T}_{r_i}, \overleftarrow{T}_{r_i}) \in \mathcal{T}, v \in V(\overrightarrow{T}_{r_i}) \right\}$. The following lemma holds from the definition of T_{in} and T_{out}.

LEMMA 8. *There exists a path from s to t passing through at least one of the vertices in $\{r_1, r_2, \ldots, r_K\}$ if and only if $T_{\text{in}}(s) \cap T_{\text{out}}(t) \neq \emptyset$ holds.*

Now we define, by generalizing the notion of $R^{T_r}_{st}$, the set of reachable vertices $R^{\mathcal{T}}_{st}$ reduced by $\mathcal{T}$ for each vertex pair (s,t) as follows:

$$R^{\mathcal{T}}_{st} = \begin{cases} \emptyset & \text{if } T_{\text{in}}(s) \cap T_{\text{out}}(t) \neq \emptyset, \\ R_s \setminus \bigcup_{1 \leq i \leq K : t \notin V(\overrightarrow{T}_{r_i})} V(\overrightarrow{T}_{r_i}) & \text{otherwise.} \end{cases}$$

We note that $R^{\mathcal{T}}_{st}$ is represented as an SPT to support incremental updates, i.e., not only vertices in $R^{\mathcal{T}}_{st}$ but also their distances from s are stored in memory.

In the case where $T_{\text{in}}(s) \cap T_{\text{out}}(t) = \emptyset$ holds, the reduced reachable vertices $R^{\mathcal{T}}_{st}$ is computed by conducting a *pruned BFS* from s using the information of $\mathcal{T}$. Most procedures in pruned BFS are the same as standard BFS. The difference between them is that, when we examine a neighbor w of a vertex v, we skip processing w if $T_{\text{out}}(w) \cap \overline{T}_{\text{out}}(t) \neq \emptyset$ holds, where $\overline{T}_{\text{out}}(t) = \{r_1, r_2, \ldots, r_K\} \setminus T_{\text{out}}(t)$. We note that $T_{\text{out}}(w) \cap \overline{T}_{\text{out}}(t) \neq \emptyset$ implies there is no path from w to t since otherwise there exists a path from $r_i \in T_{\text{out}}(w) \cap \overline{T}_{\text{out}}(t)$ to t via w, which contradicts $r_i \in \overline{T}_{\text{out}}(t)$.

By means of $\mathcal{T}$ and $R^{\mathcal{T}}_{st}$ we can easily check the reachability from s to t.

LEMMA 9. *There exists a path from s to t if and only if $T_{\text{in}}(s) \cap T_{\text{out}}(t) \neq \emptyset$ or $t \in R^{\mathcal{T}}_{st}$.*

Let $\mathcal{R} = \left\{ R^{\mathcal{T}}_{s_i t_i} \mid 1 \leq i \leq M \right\}$. Then, the index for reachability query is the pair of two sets $(\mathcal{T}, \mathcal{R})$. The set $\mathcal{T}$ is obtained by conducting BFSes $2K$ times, and the set $\mathcal{R}$ is obtained by conducting pruned BFSes M times. Since the value K does not have to be large, say less than 30, the condition $T_{\text{out}}(w) \cap \overline{T}_{\text{out}}(t) = \emptyset$ can be checked in $O(1)$ time by representing sets of roots $T_{\text{out}}(w)$ and $T_{\text{out}}(t)$ with a bit vector. Thus, the time complexity of pruned BFS is also $O(m)$, and the overall time complexity of our reachability index is $O((K+M)m)$. Our index consumes $O((K+M)n)$ space, but the actual memory usage is much smaller in practice.

6.2 Incremental Update

When an edge is inserted or deleted, all the SPTs in $\mathcal{T}$ are updated first. Each SPT is updated in the same way as the update of balls except that the distance from the root is not limited. Then, for each vertex pair (s,t), the vertex set $R^{\mathcal{T}}_{st}$ is re-computed. If the set of roots $T_{\text{out}}(t)$ is changed after the update, the vertex set $R^{\mathcal{T}}_{st}$ is also significantly changed. In this case, reduced reachable vertices $R^{\mathcal{T}}_{st}$ are computed from scratch by conducting pruned BFSes again. Otherwise, we update $R^{\mathcal{T}}_{st}$ in a way described below.

Edge Insertion. Assume that an edge (u,v) is inserted into the graph. Some vertices will be newly added to $R^{\mathcal{T}}_{st}$ because of the appearance of paths passing through the edge (u,v). These vertices is detected and added to $R^{\mathcal{T}}_{st}$ by conducting a pruned BFS from v in the same manner as the update of balls. In some cases, there exists a vertex $w \in R^{\mathcal{T}}_{st}$ satisfying $T_{\text{out}}(w) \cap \overline{T}_{\text{out}}(t) \neq \emptyset$ owing to the change in $T_{\text{out}}(w)$, i.e., the vertices included in $R^{\mathcal{T}}_{st}$ becomes a super set of the original definition of $R^{\mathcal{T}}_{st}$. However, our reachability index still answers the reachability between each vertex pair correctly.

Edge Deletion. Assume that an edge (u,v) is deleted from the graph. A vertex $w \in V$ may be reached from s with pruned BFS if $T_{\text{out}}(w) \cap \overline{T}_{\text{out}}(t) = \emptyset$ holds after the update. Let U_1 be the set of such vertices. Let U_2 be the set of vertices whose distance from s increased after the edge deletion. U_2 is obtained in the same way as Step 1 of Algorithm 3. Let $U = U_1 \cup U_2$. For each $w \in U$, it is pushed into a min-based priority queue in the same way as Step 2 of Algorithm 3. The distance from s to each vertex $w \in U$ is determined through Dijkstra's algorithm by using the pruning method in pruned BFS. For each vertex $v \in U$, v is added to $R^{\mathcal{T}}_{st}$ if $d(s,v) < \infty$.

Vertex Operations. When adding a vertex, we have nothing to do. When removing a vertex, we first remove it from each SPT in $\mathcal{T}$. For each sampled vertex pair $(s,t) \in S$, if $T_{\text{out}}(t)$ is changed after vertex removal, we compute $R^{\mathcal{T}}_{st}$ from scratch. Otherwise, we collect the vertex set U whose distances from s are changed, and we update $R^{\mathcal{T}}_{st}$ as is done when edges are removed. When some root vertex r_i is removed from a graph, we choose a new root vertex uniformly at random.

7. EXPERIMENTS

We conducted experiments on many real-world complex networks. The index size, the performance, and the accuracy of our method were compared with other state-of-the-art algorithms. In this section, we refer to the algorithm proposed by Bergamini *et al.* [7] as BMS. Since exact dynamic methods [20,22,23,25] require $\Omega(n^2)$ space and it was difficult to examine them on large networks, only BMS was used as our competitor for performance comparison. Unless otherwise mentioned, our indices are constructed with 1,000 chosen vertex pairs and ten roots of SPTs, that is, $M = 1000$ and $K = 10$.

Environment. All the algorithms were implemented with C++11 and Google Sparse Hash and compiled with gcc 4.8.3 with -O3 option, which will be available from our websites. All experiments were conducted with a single thread on a Linux server with Intel Xeon E5-2670 2.60GHz CPU and 512GB memory.

Datasets. All experiments were conducted on real-world social networks and web graphs that are publicly available. The information of datasets is shown in Table 1. The average distance is estimated by 10,000 random vertex pairs. Five networks twitter-2010, in-2004, indochina-2004, it-2004 and uk-2007 were downloaded from Laboratory for Web Algorithmics [8, 9]. The other networks used in the experiments were downloaded from Stanford Network Analysis Project [26].

Scenario. We first randomly deleted 200 edges from the graph. Second, we constructed indices to measure the construction time and data structure size. Then, we inserted the 200 edges into the graph one by one, and measured the average edge insertion time. Next, we deleted these edges from the graph one by one and measured the average edge deletion time. After that, we added 200 new vertices to the index constructed from each dataset and measured the average vertex insertion time. Finally, we deleted 200 randomly chosen vertices from the graph and measured the average vertex deletion time. Unless otherwise mentioned, we repeated this procedure five times and the average values are reported.

7.1 Construction Time

Table 2 shows that our index can be efficiently constructed on all datasets. Since the index construction time is not strongly affected by the value of the trade-off parameter x,

Table 1: Information of datasets. The average degree and average distance are denoted by $\overline{d}$ and $\overline{D}$, respectively.

Dataset	Type	n	m	$\overline{d}$	$\overline{D}$
HepPh	social(d)	35K	422K	12.2	11.7
Enron	social(u)	37K	368K	10.0	4.0
Slashdot0811	social(d)	77K	905K	11.7	4.1
Pokec	social(d)	1.6M	31M	18.8	5.3
LiveJournal	social(d)	4.8M	69M	14.2	5.9
Orkut	social(u)	3.1M	117M	38.1	4.2
twitter-2010	social(d)	42M	1.5B	35.3	4.5
Friendster	social(u)	66M	1.8B	27.5	5.0
NotreDame	web(d)	326K	1.5M	4.6	11.4
Google	web(d)	876K	5.1M	5.8	11.7
BerkStan	web(d)	685K	7.6M	11.1	13.7
in-2004	web(d)	1.4M	17M	12.2	15.3
indochina-2004	web(d)	7.4M	194M	26.2	15.7
it-2004	web(d)	41M	1.2B	27.9	15.0
uk-2007	web(d)	106M	3.7B	35.3	15.4

the maximum average construction time among $x = 0, 1, 2$ is reported.

The index can be constructed in about 40 minutes for Friendster, which is the largest social networks used in our experiments. On other networks, our index can be constructed in 15 minutes. Thus, the construction time of our index is highly efficient in comparison with the computation time of exact betweenness centrality.

7.2 Data Structure Size

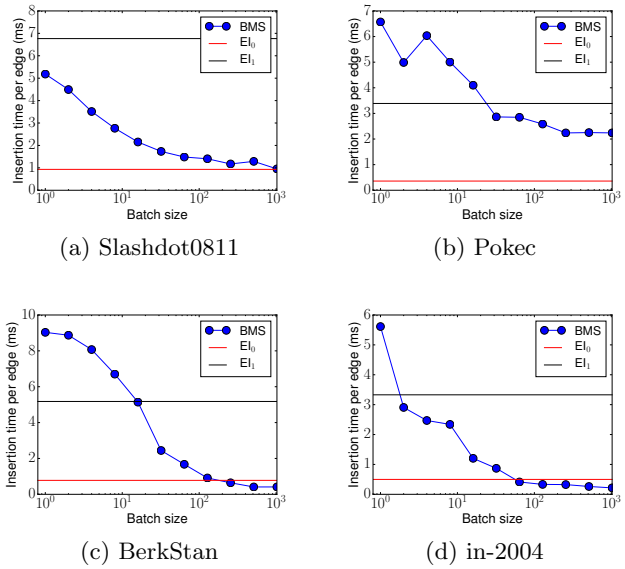
The total index size of our method and the average size of data structures stored to support efficient updates of each hyperedge are listed in Table 3. We also report the memory usage for naturally representing each graph, i.e., each edge appears twice in forward and backward adjacency lists, and represented by four-byte integers.

As shown in Table 3, we successfully constructed indices on both billion-scale social networks and web graphs. Especially on uk-2007, which is the largest network used in our experiments, the index size becomes smaller than the original graph size. In comparison with the index of BMS, our index is 20 times smaller on all datasets even when $x = 0$.

We can see that the size of data structures for each sampled vertex pair in our method is several orders of magnitude smaller than the size of each SPT stored in BMS. This indicates the high scalability of our method with regard to hyperedges. For example, even if we increase the number of hyperedges M from 1,000 to 10,000, the index size of our method on uk-2007 would be smaller than 60 GB. As shown in Table 3, we can further reduce the data size for each hyperedge by increasing the parameter x. Especially on social networks, it becomes 10 times smaller by increasing x from zero to two, which have a great impact when M is large.

7.3 Update Time

Average update times for insertions and deletions of edges and vertices are also listed in Table 2. As shown in the table, each edge insertion is processed in two milliseconds on almost all datasets when $x = 0$. This processing time is three orders of magnitude faster than full re-computation



(a) Slashdot0811 (b) Pokec

(c) BerkStan (d) in-2004

Figure 2: The processing time of batch updates.

time of the index. It can also be seen from Table 2 that the edge insertion time increases as the parameter x gets larger. This bears the trade-off of the index size and the edge insertion time.

The average edge deletion time is less than one millisecond on all datasets when $x = 0$. Moreover, when we increase the parameter x, the edge deletion time becomes even faster. This is because deleted edges affect balls less often when the balls are small.

Vertex insertion/deletion times are also shown in Table 2. Since the update times for vertex insertions and deletions are not strongly affected by the value of the trade-off parameter x, the maximum average update time among $x = 0, 1, 2$ is reported.

Vertex insertion time ranges from 0.1 milliseconds to 35.0 milliseconds. Vertex deletion time ranges from 0.2 milliseconds to 7.3 milliseconds. Although these update times are slower than edge insertion/deletion times, this performance is acceptable because the number of vertex operations tends to be fewer than the number of edge operations.

Batch Update. As BMS can gain update speed by handling multiple edge insertions as a batch, we also evaluated the insertion time when multiple edges are inserted at once and handled as batches. In this experiment, our method still processes each edge insertion one by one. Figure 2 shows the insertion time per edge for different sizes of batches of multiple edge insertion. The results of our method with $x = 2$ are not shown because the processing time is slower than the single edge update of BMS. The processing time for a single edge update of our method with $x = 0$ is at least five times faster than that of BMS. Even when the size of a batch is 1,024, on social networks, the processing time per edge of our method is still faster than that of BMS.

7.4 Parameter M and Accuracy

We compared the accuracy of our method and BMS under various number of samples M. We used BMS only since

Table 2: The timing results of our index on real-world datasets. IT denotes the index construction time. EI_x and ED_x denote the average edge insertion/deletion time, respectively, where x is the trade-off parameter. VI and VD denote the average vertex insertion/deletion time. For IT, VI and VD of our method, maximum values among $x = 0, 1, 2$ are reported. DNF means it did not finish in one hour or ran out memory.

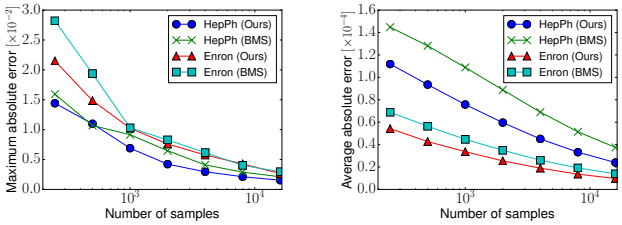
Dataset	Ours									BMS [7]	
	IT	EI_0	EI_1	EI_2	ED_0	ED_1	ED_2	VI	VD	IT	EI
Slashdot0811	626.4 ms	0.9 ms	6.8 ms	205.5 ms	0.2 ms	0.2 ms	0.1 ms	0.1 ms	0.3 ms	21.8 s	6.1 ms
Pokec	18.8 s	0.4 ms	3.4 ms	82.3 ms	0.2 ms	0.2 ms	0.2 ms	0.7 ms	0.2 ms	810.1 s	7.3 ms
LiveJournal	27.5 s	0.3 ms	3.3 ms	164.4 ms	0.2 ms	0.1 ms	0.1 ms	1.5 ms	0.2 ms	2491.1 s	8.0 ms
Orkut	87.1 s	1.1 ms	22.4 ms	3.3 s	0.5 ms	0.4 ms	0.4 ms	0.9 ms	1.0 ms	DNF	—
twitter-2010	931.1 s	219.6 ms	1.0 s	DNF	0.3 ms	0.2 ms	1.2 ms	14.7 ms	7.3 ms	DNF	—
Friendster	2211.4 s	0.8 ms	17.6 ms	14.2 s	0.4 ms	0.3 ms	0.3 ms	24.4 ms	1.3 ms	DNF	—
NotreDame	564.3 ms	0.3 ms	0.9 ms	3.5 ms	0.2 ms	0.2 ms	0.1 ms	0.1 ms	0.5 ms	4.9 s	3.2 ms
Google	3.1 s	0.4 ms	1.4 ms	3.4 ms	0.2 ms	0.2 ms	0.2 ms	0.5 ms	0.2 ms	102.2 s	8.3 ms
BerkStan	4.3 s	0.8 ms	5.2 ms	16.5 ms	0.4 ms	0.3 ms	0.3 ms	0.4 ms	0.4 ms	71.1 s	14.3 ms
in-2004	3.1 s	0.5 ms	3.3 ms	35.7 ms	0.3 ms	0.3 ms	0.2 ms	0.5 ms	0.2 ms	85.8 s	3.9 ms
indochina-2004	23.5 s	0.7 ms	51.7 ms	12.0 s	0.4 ms	0.3 ms	0.3 ms	2.4 ms	0.6 ms	777.2 s	6.3 ms
it-2004	135.6 s	0.9 ms	16.0 ms	346.2 ms	0.5 ms	0.4 ms	0.4 ms	12.9 ms	0.5 ms	DNF	—
uk-2007	419.6 s	1.7 ms	31.8 ms	1.5 s	0.9 ms	0.8 ms	0.7 ms	35.0 ms	0.6 ms	DNF	—

Table 3: The index size on real-world datasets. IS_x denote the total index size, and HS_x denote the data structure size for each vertex pair, where x is the trade-off parameter. PS denotes the size of each SPT in BMS [7]. The memory usage for representing each graph is denoted by G.

Dataset	Ours						BMS [7]		G
	IS_0	IS_1	IS_2	HS_0	HS_1	HS_2	IS	PS	
Slashdot0811	23.0 MB	20.1 MB	19.9 MB	6.0 KB	3.1 KB	2.9 KB	542.7 MB	541.6 KB	7.2 MB
Pokec	394.6 MB	368.7 MB	362.8 MB	36.0 KB	10.1 KB	4.3 KB	11.4 GB	11.4 MB	245.0 MB
LiveJournal	1.1 GB	1.1 GB	1.1 GB	55.5 KB	14.2 KB	5.6 KB	34.0 GB	33.9 MB	552.0 MB
Orkut	867.3 MB	704.2 MB	683.3 MB	192.6 KB	29.5 KB	8.6 KB	—	—	937.5 MB
twitter-2010	9.3 GB	9.2 GB	9.2 GB	108.4 KB	22.9 KB	23.5 KB	—	—	11.7 GB
Friendster	14.7 GB	14.4 GB	14.4 GB	313.5 KB	34.8 KB	14.0 KB	—	—	14.4 GB
NotreDame	90.4 MB	89.3 MB	88.0 MB	18.9 KB	17.8 KB	16.4 KB	2.3 GB	2.3 MB	12.0 MB
Google	248.8 MB	230.1 MB	213.1 MB	56.5 KB	37.8 KB	20.8 KB	6.1 GB	6.1 MB	40.8 MB
BerkStan	290.9 MB	259.8 MB	229.1 MB	140.5 KB	109.3 KB	78.6 KB	4.8 GB	4.8 MB	60.8 MB
in-2004	514.9 MB	483.5 MB	451.2 MB	211.2 KB	179.9 KB	147.5 KB	9.7 GB	9.7 MB	135.3 MB
indochina-2004	2.2 GB	2.1 GB	1.9 GB	609.6 KB	453.3 KB	312.5 KB	52.0 GB	51.9 MB	1.6 GB
it-2004	10.3 GB	9.8 GB	9.5 GB	1.2 MB	704.4 KB	406.8 KB	—	—	9.2 GB
uk-2007	26.4 GB	25.5 GB	24.7 GB	3.3 MB	2.4 MB	1.6 MB	—	—	29.9 GB

other estimation methods are developed for the static setting.

For each number of samples $M = 250, 500, 1,000, 2,000, 4,000, 8,000, 16,000$, we compared the approximate centralities of vertices with the exact values obtained by Brandes' exact algorithm [10]. Due to the high cost of exact betweenness centrality computation, the comparisons were conducted on small networks HepPh and Enron. The maximum and average absolute errors of estimated centrality values were evaluated, where we define the error of an estimated centrality of a vertex v as $\frac{1}{n^2}\left|C(v) - \widetilde{C}(v)\right|$ right after the index construction in order to compare the accuracy on networks of varying size. In Figure 3a and Figure 3b, the averages of these values for ten trials are shown. As shown in Figure 3a, both methods show better accuracy by increasing the number of samples, as suggested by their theoretical guarantees. Figure 3a and Figure 3b show that our method estimates centralities more accurately than BMS does in almost all settings, although BMS has better theoretical guarantee. It may be because hypergraph sketch uses more information for each sampled vertex pair (s, t), i.e., the number of shortest paths $\sigma(s, v)$ and $\sigma(v, t)$ for each $v \in P(s, t)$, to estimate betweenness centrality.

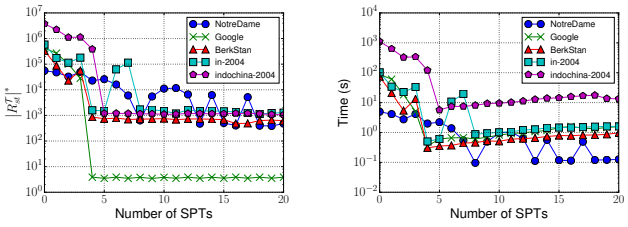


(a) Maximum error (b) Average error

Figure 3: Betweenness centrality estimation error $\frac{1}{n^2}\left|C(v) - \widetilde{C}(v)\right|$ on small networks.

7.5 Parameter *K* and Reachability Index

We now show the index size and construction time of our special-purpose reachability index under various number of root vertices K. The results show the effectiveness of our technique in reducing the index size and how many SPTs



(a) Average size of $R_{st}^{\mathcal{T}}$. (b) Construction time.

Figure 4: The statistical information of the index for maintaining the reachability of 1,000 vertex pairs.

are required to sufficiently reduce the index size.

Index Size. The sizes of $R_{st}^{\mathcal{T}}$ over 1,000 vertex pairs were evaluated under various numbers of SPTs up to 20. The averages over ten trials on directed graphs are shown in Figure 4a. Sampling ten root vertices seems to be enough for reducing the number of vertices stored in $R_{st}^{\mathcal{T}}$ except for NotreDame. By sampling ten root vertices, on average, $\left|R_{st}^{\mathcal{T}}\right|$ becomes at least 3,000 times smaller on indochina-2004 and at least 100,000 times smaller on Google.

Construction Time. The construction time of the index for maintaining the reachability of 1,000 vertex pairs were also evaluated. The average construction time over ten trials are shown in Figure 4b. As the number of SPTs increases, the construction time at first quickly drops due to the decreasing size of $R_{st}^{\mathcal{T}}$, but it gradually increases owing to the increasing construction time of SPTs.

8. COMPLEXITY ANALYSIS

In this section, we compute the construction time and space requirement of our data structure. We also roughly analyze the running time of each update operation. For the sake of simplicity, we assume that a hash table can be accessed in a constant time and the size of a hash table is linear in the number of items stored in it. Although the analysis below lacks some rigidity for simplicity, it would explain why our method is efficient on real-world datasets.

8.1 Construction Time and Space Requirement

Before starting analysis, we introduce several notations. For an integer x, let $\left|B_x\right|$ be the average size of two balls for a vertex pair, i.e., $\sum_{(s,t)\in S}(\left|\overrightarrow{B}(s,d_s)\right| + \left|\overleftarrow{B}(t,d_t)\right|)/\left|S\right|$ under the parameter x that is given when the index is constructed, and $\|B_x\|$ be the average number of edges incident to a vertex in $\overrightarrow{B}(s,d_s)$ or $\overleftarrow{B}(t,d_t)$. Let $\left|R_K\right|$ be the average size of reduced reachable vertices $R_{st}^{\mathcal{T}}$ and $\|R_K\|$ be the average number of edges incident to $R_{st}^{\mathcal{T}}$ when the size of the set $\mathcal{T}$ of SPT pairs is K.

When constructing a TB-index, the bottleneck is performing M bidirectional BFSes to compute the distance between each vertex pair and obtaining two balls, and its running time is $O(M\|B_0\|)$. When constructing our special-purpose reachability index, we need to perform $2K$ BFSes to obtain the set of K pairs of SPTs and M pruned BFSes to compute the reduced reachable vertices for each vertex pair, and its running time is $O(Km+M\|R_K\|)$. Thus, the overall construction time is $O(M(\|B_0\| + \|R_K\|) + Km)$. We note that $\|B_x\|$ and $\|R_K\|$ were at most $0.02m$ on million-node networks in our experiments.

Since we need $O(M\left|B_x\right|)$, $O(Kn)$, $O(M\left|R_K\right|)$, and $O(n)$ space to store balls, SPTs, the set of reduced reachable vertices, and the centrality values of vertices and other variables, respectively, the overall space requirement of our data structure is $O(M(\left|B_x\right| + \left|R_K\right|) + Kn)$. $\left|B_x\right|$ and $\left|R_K\right|$ are much smaller than n in practice; They were at most $0.01n$ on million-node networks in our experiments. The space complexity of our data structure becomes significantly smaller in comparison with the algorithm proposed by Bergamini *et al.* [7], which stores M SPTs.

Table 4: Running time of each update operation.

Vertex	Addition	$O(M+K)$
	Removal	$O((M+K)T^{\mathrm{D}})$
Edge	Insertion	$O(ME_x+(M+K)T^{\mathrm{EI}})$
	Deletion	$O((M+K)T^{\mathrm{D}})$

8.2 Updating Time

In our data structure, we store $2M$ SPTs in the TB-index, $2K$ SPTs in the reachability index, and M SPTs for reduced reachable vertices. Updating these SPTs is the bottleneck of our update procedures. Hence, we first consider the time complexity of updating SPTs.

Let C be the set of vertices whose distances from the root of an SPT are modified by the update operation, $\|C\|$ be the number of edges incident to C, and D be the average number of vertices whose distance from $v \in C$ is less than or equal to two, i.e., $\sum_{v\in C}\left|\{w \mid d(v,w) \le 2\}\right|/\left|C\right|$. When adding a vertex, we only have to allocate a new space for the newly added vertex. When inserting an edge, we only need to look at neighbors of vertices whose distances from the root decreased. When removing a vertex or an edge, we need $O(D\left|C\right|)$ time to compute the vertex set C, and $O(\|C\|\log\|C\|)$ time to update the distance from the root to each vertex in C. Hence, the time complexity of vertex addition, edge insertion, and vertex/edge deletion are $O(1)$, $T^{\mathrm{EI}} := O(\|C\|)$, and $T^{\mathrm{D}} := O(D\left|C\right| + \|C\|\log\|C\|)$, respectively. We remark that C is much smaller than V in practice; The average of $\left|C\right|$ on million-node networks was at most 40 when removing a vertex from an SPT on the whole graph and at most four in other cases.

Now we show the running time of each update operation in Table 4. Except vertex addition, the bottleneck of the update procedure is updating SPTs, and the time complexity of each operation is $O((M+K)T^{\mathrm{D}})$. Precisely speaking, we may replace several vertex pairs when processing vertex operations, but we ignore its time complexity in our analysis because the probability of occurring such a replacement is only $O(\frac{1}{n})$. When inserting an edge, we have to check whether new shortest paths are formed. Due to the small size of balls, the probability that a randomly inserted edge has an endpoint in a ball is quite small, and we only consider the case that the inserted edge has no endpoint in a ball. Thus, when an edge is inserted, $O(ME_x)$ additional time is required to traverse a graph from this edge, where E_x is the number of vertices whose distances from endpoints of the inserted edge are less than or equals to x.

9. CONCLUSION

In this paper, we present the first approximate betweenness centrality algorithm for fully dynamic networks. Our TB-index efficiently detects the change of shortest paths between vertex pairs and updates approximate betweenness centralities of vertices after each update. The size of the TB-index is maintained small enough to handle large scale complex networks. We also proposed a fully dynamic special-purpose reachability index for maintaining the reachability of some fixed vertex pairs in order to further improve the performance of the TB-index on directed graphs. The set of SPTs rooted at vertices in the core substantially reduces the number of vertices maintained in our reachability index. Experimental results showed the TB-index is compact enough to handle dynamic updates on very large complex networks with tens of thousands of vertices and over one billion edges in memory. Moreover, each update after edge insertion and edge deletion can be processed in several milliseconds.

Acknowledgments. T. H., T. A., and Y. Y. are supported by JST, ERATO, Kawarabayashi Large Graph Project. T. A. is supported by JSPS Grant-in-Aid for Research Activity Start-up (No. 15H06828) and PRESTO, JST. Y. Y. is supported by JSPS Grant-in-Aid for Young Scientists (B) (No. 26730009) and MEXT Grant-in-Aid for Scientific Research on Innovative Areas (No. 24106003).

10. REFERENCES

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*, pages 237–248, 2014.

[2] J. M. Anthonisse. The Rush In A Directed Graph. Technical Report BN 9/71, Stichting Mathematisch Centrum, 1971.

[3] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating Betweenness Centrality. In *WAW*, pages 124–137, 2007.

[4] M. Baglioni, F. Geraci, M. Pellegrini, and E. Lastres. Fast Exact Computation of Betweenness Centrality in Social Networks. In *ASONAM*, pages 450–456, 2012.

[5] A.-L. Barabasi. The origin of bursts and heavy tails in human dynamics. *Nature*, 435:207–211, 2005.

[6] M. Barthélemy. Betweenness centrality in large complex networks. *The European Physical Journal B - Condensed Matter and Complex Systems*, 38(2):163–168, 2004.

[7] E. Bergamini, H. Meyerhenke, and C. L. Staudt. Approximating Betweenness Centrality in Large Evolving Networks. In *ALENEX*, pages 133–146, 2015.

[8] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW*, 2011.

[9] P. Boldi and S. Vigna. The WebGraph Framework I: Compression Techniques. In *WWW*, pages 595–601, 2004.

[10] U. Brandes. A Faster Algorithm for Betweenness Centrality. *J. Math. Sociol.*, 25(2):163–177, 2001.

[11] U. Brandes and C. Pich. Centrality Estimation in Large Networks. *Int. J. Bifurcat. Chaos*, 17(07):2303–2318, 2007.

[12] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph Structure in the Web. *Comput. Netw.*, 33(1-6):309–320, 2000.

[13] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47(3):45–47, 2004.

[14] A. Del Sol, H. Fujihashi, and P. O'Meara. Topology of small-world networks of protein–protein complex structures. *Bioinformatics*, 21(8):1311–1315, 2005.

[15] D. Erdös, V. Ishakian, A. Bestavros, and E. Terzi. A divide-and-conquer algorithm for betweenness centrality. In *SDM*, 2015. to appear.

[16] L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, 1977.

[17] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully Dynamic Algorithms for Maintaining Shortest Paths Trees. *J. Algorithms*, 34(2):251–281, 2000.

[18] R. Geisberger, P. Sanders, and D. Schultes. Better Approximation of Betweenness Centrality. In *ALENEX*, pages 90–100, 2008.

[19] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *P. Natl. Acad. Sci.*, 99(12):7821–6, 2002.

[20] O. Green, R. McColl, and D. A. Bader. A fast algorithm for streaming betweenness centrality. In *SocialCom/PASSAT*, pages 11–20, 2012.

[21] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, 2001.

[22] M. Kas, M. Wachs, K. M. Carley, and L. R. Carley. Incremental Algorithm for Updating Betweenness Centrality in Dynamically Growing Networks. In *ASONAM*, pages 33–40, 2013.

[23] N. Kourtellis, G. D. F. Morales, and F. Bonchi. Scalable Online Betweenness Centrality in Evolving Graphs. *Transactions on Knowledge and Data Engineering*, 2014. to appear.

[24] V. E. Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002.

[25] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung. QUBE: a Quick algorithm for Updating BEtweenness centrality. In *WWW*, pages 351–360, 2012.

[26] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection.

[27] S. Milgram. The Small World Problem. *Psychology today*, 2(1):60–67, 1967.

[28] M. E. J. Newman. *Networks: An Introduction*. Oxford University Press, 2010.

[29] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69:026113, 2004.

[30] R. Puzis, P. Zilberman, Y. Elovici, S. Dolev, and U. Brandes. Heuristics for Speeding Up Betweenness Centrality Computation. In *SocialCom/PASSAT*, pages 302–311, 2012.

[31] G. Ramalingam and T. W. Reps. *On the Computational Complexity of Incremental Algorithms*. University of Wisconsin-Madison, Computer Sciences Department, 1991.

[32] M. Riondato and E. M. Kornaropoulos. Fast Approximation of Betweenness Centrality Through Sampling. In *WSDM*, pages 413–422, 2014.

[33] K. Tretyakov, A. Armas-cervantes, L. García-ba nuelos, and M. Dumas. Fast Fully Dynamic Landmark-based Estimation of Shortest Path Distances in Very Large Graphs. In *CIKM*, pages 1785–1794, 2011.

[34] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[35] H. Yildirim, V. Chaoji, and M. J. Zaki. DAGGER: A Scalable Index for Reachability Queries in Large Dynamic Graphs. *CoRR*, abs/1301.0:11, 2013.

[36] Y. Yoshida. Almost Linear-Time Algorithms for Adaptive Betweenness Centrality using Hypergraph Sketches. In *KDD*, pages 1416–1425, 2014.

[37] R. Zafarani, M. A. Abbasi, and H. Liu. *Social Media Mining: An Introduction*. Cambridge University Press, 2014.

[38] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability Queries on Large Dynamic Graphs: A Total Order Approach. In *SIGMOD*, pages 1323–1334, 2014.