Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks

Takanori Hayashi[†,§] Takuya Akiba[‡*] Ken-ichi Kawarabayashi[#,§]
†The University of Tokyo ‡Preferred Networks, Inc. #National Institute of Informatics
§JST, ERATO, Kawarabayashi Large Graph Project
†flowlight0@gmail.com, ‡akiba@preferred.jp #k_keniti@nii.ac.jp

ABSTRACT

The distance between vertices is one of the most fundamental measures for representing relations between them, and it is the basis of other classic measures of vertices, such as similarity, centrality, and influence. The 2-hop labeling methods are known as the fastest exact point-to-point distance algorithms on million-scale networks. However, they cannot handle billion-scale networks because of the large space requirement and long preprocessing time. In this paper, we present the first algorithm that can process exact distance queries on fully dynamic billion-scale networks besides trivial non-indexing algorithms, which combines an online bidirectional breadth-first search (BFS) and an offline indexing method for handling billion-scale networks in memory. First, we accelerate bidirectional BFSs by using heuristics that exploit the small-world property of complex networks. Then, we construct *bit-parallel shortest-path trees* to maintain sets of shortest paths passing through high-degree vertices of networks in compact form, the information of which enables us to avoid visiting vertices with high degrees during bidirectional BFSs. Thus, the searches achieve considerable speedup. In addition, our index size reduction technique enables us to handle billion-scale networks in memory. Furthermore, we introduce dynamic update procedures of our data structure to handle fully dynamic networks. We evaluated the performance of the proposed method on real-world networks. In particular, on large-scale social networks with over 1B edges, the proposed method enables us to answer distance queries in around 1 ms, on average.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*Graphs and networks*

Keywords

Graphs, shortest-path, dynamic updates, query processing

*Work done mostly at National Institute of Informatics

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA

© 2016 ACM. ISBN 978-1-4503-4073-1/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2983323.2983731

1. INTRODUCTION

The distance between a pair of vertices is one of the most fundamental measures for representing relations between them. The distance itself is used as a meaningful indicator of the strength of relevance [25, 26, 32]. Furthermore, it is the basis of many useful indicators for network analysis, such as centrality [8, 14, 30], similarity [10, 21, 25], and influence [2, 13, 18]. Computation of these indicators generally requires distances between a considerable number of vertex pairs. Therefore, distance computation sometimes accounts for a non-negligible portion of the computational cost for large-scale network analysis. Accordingly, the efficiency of distance computation has a significant impact in terms of scaling up graph data mining.

Indeed, many studies have investigated both approximate and exact algorithms for answering distance queries on large-scale complex networks [3, 9, 12, 16, 19, 22, 27–29, 31]. First, these algorithms typically construct a data structure called an *index* from the given graph, which is then used for efficiently answering distance queries. Landmark-based approximate methods [19, 27–29, 31] and labeling-based exact methods [3,9,12,16,22] are the most widely studied methods for handling large complex networks. However, landmark-based approximate methods tend to fail to accurately estimate distances between close pairs [5, 29]. Labeling-based methods [3, 9, 12, 16, 22] are known as the fastest exact distance querying methods on medium-scale networks with millions of edges. However, they cannot handle billion-scale networks because of the extremely large space requirement and long preprocessing time, as confirmed by our experiments.

Moreover, whereas real-world networks are highly dynamic, among the methods described above, only the algorithms proposed in [4, 31] can handle dynamic updates. The algorithm proposed in [31] gives us only the estimation of distances, while the algorithm proposed in [4] can barely construct indices for medium-scale networks in memory and supports only additions of vertices and edges. Thus, there is no exact algorithm that can process distance queries on fully dynamic billion-scale networks (besides trivial non-indexing algorithms).

1.1 Contributions

In this paper, we present a new exact algorithm that can quickly answer distance queries on billion-scale networks in memory. Unlike previous methods, the proposed method combines an online bidirectional breadth-first search (BFS) and offline indexing, which enables us to avoid full computation of prohibitively large data structures. To the best of

our knowledge, our method is the first exact method that is faster than a trivial bidirectional BFS on billion-scale complex networks. In addition, the proposed method is fully dynamic in the sense that it can process both insertions and deletions of edges.

Our technical contributions can be summarized as follows.

- **Optimized Bidirectional BFS:** We present new search expansion heuristics that consider the extremely small average distance of vertices in complex networks. When we find a shortest path while conducting our bidirectional BFS, we can immediately terminate searches without verifying whether the distance of a found path is truly the shortest distance.
- **Acceleration Based on Bit-Parallel Shortest-Path Trees:** We construct shortest-path trees (SPTs) rooted at vertices with the highest degrees for further speedup of the querying time of the proposed method. We maintain multiple SPTs with bit-parallelized form and reduce the space requirement of each SPT by around four times. Moreover, we present an index size reduction technique to further reduce the data structure size in order to handle billion-scale networks in memory, which discards the information of the bit-parallel SPTs from some of the vertices and quickly recovers this information if we need it for answering distance queries.
- **Fully Dynamic Update Procedure:** We also present fully dynamic update procedures of bit-parallel SPTs. For each bit-parallel SPT, we simulate a partial bit-parallel BFS as with the update procedure for an SPT [15, 31]. These update procedures are carefully designed to correctly propagate changes in the bit-parallelized information of SPTs.

We evaluated the performance of our method and the effectiveness of the proposed techniques on real-world complex networks. We successfully constructed indices for billion-scale networks within 1 h, and the sizes of the constructed indices did not exceed the original graph sizes significantly. This indicates that the proposed method is highly scalable in comparison with the fastest existing algorithms. Moreover, the proposed method enables us to answer distance queries up to 70 times faster than trivial bidirectional BFSs. In particular, the average querying time is around 200 μs on Twitter, which consists of 41M vertices and 1.5B edges. Further, as compared to full recomputation, our dynamic update procedure is faster by several orders of magnitude in terms of reflecting graph changes into the indices.

Organization. Section 2 reviews related studies on distance queries. Section 3 introduces the basic notations and problem definitions. Section 4 describes our method for answering distance queries. Specifically, it discusses our optimized bidirectional BFS for complex networks, search acceleration based on bit-parallel SPTs, and update procedures for handling fully dynamic networks. Section 5 describes experiments conducted to demonstrate the performance of our method. Finally, Section 6 concludes the paper.

2. RELATED WORK

Approximation Methods. First, we describe landmark-based approximate algorithms [19, 27–29, 31]. In landmark-based methods, we choose a small set R of vertices as landmarks and compute single-source shortest paths from each landmark. Then, we output $\min_{r \in R} \{d(s,r) + d(r,t)\}$ as the distance estimation when answering the distance query between vertex s and vertex t. Potamias *et al.* observed that selecting landmarks according to their degrees and centralities provides better estimation accuracy than random landmark selection [27]. Several researchers have proposed algorithms that exploit not only the distance from each root but also the structure of the SPT rooted at each landmark [19, 29, 31]. However, these methods still lack accuracy between close vertex pairs [5, 29]. Tretyakov *et al.* and Qiao *et al.* further improved the estimation accuracy by conducting local searches [29, 31], but their querying time was slower by up to three orders of magnitude. The algorithm proposed in [31] can reflect dynamic graph updates into distance estimation by updating the distances of vertices from each landmark. Qi *et al.* proposed an algorithm for handling large-scale networks with over a billion vertices [28], which combines landmark-based estimation with graph embedding.

Labeling-Based Exact Methods. We explain the exact distance query algorithm based on the 2-hop cover [11]. For each $v \in V$, it maintains a label $L(v)$ consisting of pairs (u, d_{vu}), where d_{vu} is the distance between u and v. We construct 2-hop covers such that, for any vertex pair (s,t), they satisfy $d(s,t) = \min\{d_{su} + d_{tu} \mid (u, d_{su}) \in L(s), (u, d_{tu}) \in L(t)\}$. Many studies have investigated the efficient construction of small-sized 2-hop covers on real-world networks owing to their extremely fast querying time [3, 9, 12, 16, 22]. Akiba *et al.* proposed pruned landmark labeling, which constructs a 2-hop index by conducting pruned BFSs of vertices ordered by their degrees. The algorithm prunes BFSs by means of the information of a partially constructed 2-hop index, thereby reducing the construction cost significantly. It successfully constructs 2-hop cover indices for million-scale networks and answers each distance query in several microseconds, but its memory requirement is still too large to construct indices for billion-scale networks. Akiba *et al.* also proposed a dynamic version of pruned landmark labeling, which updates 2-hop indices after each edge insertion [4]. Delling *et al.* further improved the scalability of pruned landmark labeling by introducing a better vertex ordering method and an index compression scheme [12]. Jiang *et al.* proposed the disk-based 2-hop cover algorithm [22]. However, none of the above-mentioned methods can handle billion-scale networks owing to the prohibitively large space requirement and long index construction time.

Search-Based Exact Methods. Goldberg *et al.* proposed *ALT search* algorithms for road networks [17]. They used landmark-based precomputation to estimate the lower bounds of point-to-point distances, which are used for bidirectional A^* search. However, it is known that this approach does not improve performance on complex networks at all, because the characteristics of such networks are different from those of road networks.

3. PRELIMINARIES

Notations. First, we introduce the basic notations used in the following sections. Let $G = (V, E)$ denote a graph consisting of vertex set V with n vertices and edge set E with m edges. For each $v \in V$, we denote a set of neighbors of v by $N(v)$. Let $d(s,t)$ be the length of the shortest path from vertex s to vertex t. If there is no path from s to t, we consider that $d(s,t) = \infty$.

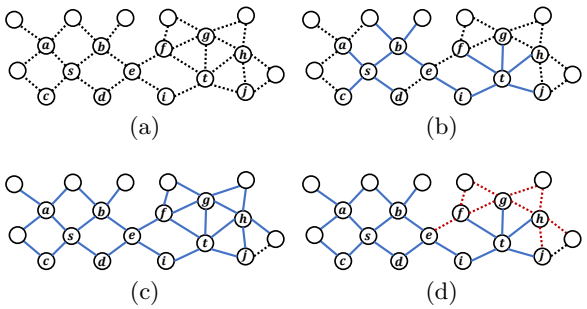


Figure 1: An example of search expansion heuristics. Dashed lines are edges that are not traversed yet in a BFS. Solid lines are edges that have been traversed in a BFS.

Problem Definition. We study the following problem: Given a graph G, we preprocess G and construct the index for efficiently answering the shortest distance query of an arbitrary vertex pair. We also study efficient methods to reflect dynamic updates of a graph, such as edge insertion and edge deletion, into the index for distance queries.

4. PROPOSED METHOD

In this section, we present a new algorithm for distance queries on massive complex networks. First, we introduce a bidirectional BFS optimized for computing distances on unweighted small-world networks. Then, we accelerate our bidirectional BFS by combining it with a landmark-based method, i.e., we construct a small number of SPTs and use the information of these SPTs to reduce the search space. Next, we introduce a *bit-parallel SPT* for further speedup of our bidirectional BFS by compactly maintaining more SPTs with bit-parallelized form. Further, we present a technique that reduces the memory usage of bit-parallel SPTs. Finally, we describe the dynamic update procedures of bit-parallel SPTs for handling fully dynamic networks. For simplicity, the algorithm described here assumes that a given graph is undirected, but it can be easily extended to directed graphs.

4.1 Optimized Bidirectional BFS

4.1.1 Search Expansion Heuristics

In this section, we introduce a bidirectional BFS optimized for complex networks. When we compute the distance from vertex s to vertex t by conducting a bidirectional BFS, we carry out both a forward search from s and a backward search from t. Let P_s and P_t be the sets of vertices visited by the forward search from s and the backward search from t, respectively. Let Q_s and Q_t be the queues used in the forward and backward searches, respectively. Further, let d_s and d_t be the maximum distances such that all vertices whose distances from s and t are less than or equal to d_s and d_t are already visited during the search, respectively.

In a bidirectional BFS in [17, 23], we alternately choose one vertex v from Q_s or Q_t and examine its neighbors until a shortest path is obtained. We maintain the minimum distance $\overline{d}$ of already found paths from s to t, which is initialized to ∞. Here, let us assume that v is popped from Q_s. We examine each neighbor w of v. If $w \in P_t$, we find a path passing through an edge (v, w) and update $\overline{d}$ to $\min\{\overline{d}, d(s,v) + 1 + d(w,t)\}$. If $w \notin P_s$, we add w to P_s and push w into Q_s. We continue this procedure as long as both Q_s and Q_t are not empty and $d_s + 1 + d_t < \overline{d}$, i.e., the lower bound of $d(s,t)$ does not reach the upper bound of $d(s,t)$. The drawback of this algorithm is that we have to continue the search, even if we have already found a shortest path, in order to verify that it truly is a shortest path.

Algorithm 1 Bidirectional BFS from s to t.

```
1: procedure BiBFS(s, t)
2:     Q_s, Q_t ← a FIFO queue containing each of s and t.
3:     P_s ← {s}, P_t ← {t}, d_s ← 0, d_t ← 0.
4:     while not (Q_s.empty() ∨ Q_t.empty()) do
5:         if |P_s| ≤ |P_t| then found ← Search-F(Q_s).
6:         else found ← Search-B(Q_t).
7:         if found = true then return d_s + d_t + 1.
```

Figure 1 shows an example of the drawback of the existing methods. Let us assume that we compute $d(s,t)$ on the graph in Figure 1a. Let us consider that we alternately choose a vertex from both sides and we find a path of length four after examining neighbors of vertices s, t, b, and i (Figure 1b). At this stage, we cannot terminate the search because we have not verified that there is no paths of length three, even though $d(s,t) = 4$. We can only finish the search after we traverse all of edges from a, c, and d and increase d_s to two. As a result, we have to traverse all of blue solid edges before finding $d(s,t)$ (Figure 1c). After finishing the search, it turns out that traversing red dashed edges is not necessary to compute $d(s,t)$ (Figure 1d).

To avoid such unnecessary edge traversal, we introduce search expansion heuristics optimized for complex networks. Our idea is to increase d_s or d_t in each iteration. As described later, we can terminate the search right after we find any shortest path from s to t. Our method especially works well on complex networks because the number of unnecessary traversal edges often significantly decreases owing to their small average distance and large average degree. Our experimental results demonstrate the effectiveness of our search expansion heuristics.

Algorithm 1 shows the pseudo-code of our bidirectional BFS. First, we initialize P_s and P_t to $\{s\}$ and $\{t\}$, respectively. Then, we enqueue s and t into Q_s and Q_t, respectively. We iteratively increase two integers d_s and d_t one by one and expand the two vertex sets P_s and P_t as long as they have no common vertex and Q_s and Q_t are not empty. In each iteration, we expand P_s or P_t according to the sizes of P_s and P_t in order to avoid imbalance between $|P_s|$ and $|P_t|$. If $P_s \leq P_t$, we expand P_s in the iteration by traversing forward edges and updating queue Q_s by calling the procedure Search-F(Q_s). Otherwise, we traverse backward edges by calling Search-B(Q_t). Here, we only describe the procedure Search-F(Q_s) since Search-B(Q_t) is nearly the same as Search-F(Q_s). In the procedure Search-F(Q_s), we first prepare an empty queue Q_s^{next}. At this point, the set of vertices contained in Q_s equals to a vertex set $\{u \in V \mid d(s,u) = d_s\}$. For each vertex $v \in Q_s$, we examine each neighbor w of v. If vertex w is not included in P_s, we add w to P_s and enqueue w into Q_s^{next}. If vertex w is included in vertex set P_t, it means that we have found the vertex t that can be reached from s with $d_s + d_t + 1$ hops, and we immediately return $d_s + d_t + 1$ as the answer for the distance query since we already know $d(s,t) \geq d_s + d_t + 1$.

When we cannot find the shortest distance in the iteration, we replace Q_s with Q_s^{next} and increase d_s by 1. Note that the vertex set contained in Q_s at this point corresponds to a vertex set $\{u \in V \mid d(s,u) = d_s\}$ again.

Let us look at Figure 1 again. If we use our search expansion heuristics, we first traverse edges from vertices s, t, b, a, c, and d before examining edges from neighbors of t. Then, we traverse edges from vertex i, and forward and backward searches meet at vertex e. Since we already have $d_s = 2$ and $d_t = 1$, we can immediately terminate the search and avoid unnecessary edge traversals (Figure 1d).

4.1.2 Landmark-Based Acceleration

Now, we accelerate our bidirectional BFS by combining it with a landmark-based method. The landmark-based method is adopted for speedup of our bidirectional BFS, which is achieved by avoiding scanning of neighbors of vertices having an extremely large number of neighbors. For example, more than 1,000 Twitter users have more than 10 million followers [20]. Scanning all the followers of these users during bidirectional BFSs entails a prohibitively high cost. Avoiding such expensive operations is highly beneficial in terms of the efficiency of query processing. Hence, we avoid the high scanning costs explained above by choosing vertices of high degrees as landmarks and precomputing the distance from s to t when we pass through these high-degree vertices. Our method is described below.

Construction. Given a positive integer K, we choose a small set R of vertices of the highest degrees as landmarks, and construct an SPT rooted at each $r \in R$ by conducting a BFS. Each SPT rooted at r is represented as an array δ_r such that $\delta_r[v] = d(r,v)$ for each $v \in V$. These SPTs $\{\delta_r\}_{r \in R}$ are obtained by conducting K BFSs in $O(Km)$ time. The SPTs are represented with Kn bytes in memory, whereby each element is represented with 8 bits. We note that representing distances with 8 bits is reasonable in social networks and web graphs because the diameters of these networks are small.

Distance Querying. Given a distance query that asks for the distance between vertices s and t, we first compute the upper bound $\overline{d}(s,t) = \min_{r \in R} \{\delta_r[s] + \delta_r[t]\}$ of the distance from s to t, which corresponds to the distance of shortest paths from s to t passing through R. Then, we obtain $d_{G \setminus R}(s,t)$ by carrying out our bidirectional BFS on the graph $G \setminus R$ and output the minimum value of $\overline{d}(s,t)$ and $d_{G \setminus R}(s,t)$ as the distance from s to t.

We can return $\overline{d}(s,t)$ immediately when $d_s + d_t + 1$ in Algorithm 1 reaches $\overline{d}(s,t)$ since $d_s + d_t + 1 \leq d(s,t) \leq \overline{d}(s,t)$. This pruning technique further reduces the search space, as shown by our experimental results (see Section 5). Although such pruning occurs only when $\overline{d}(s,t) = d(s,t)$, the upper bound $\overline{d}(s,t)$ often matches $d(s,t)$ in real-world complex networks [27, 31].

LEMMA 1. *Given a vertex pair (s,t), the above algorithm returns $d(s,t)$.*

PROOF. Let p^* be a shortest path from s to t. If p^* passes through R, the upper bound obtained from $\{\delta_r\}_{r \in R}$ matches the length of p^*. Otherwise, p^* is also a shortest path from s to t on the graph $G \setminus R$. □

4.2 Bit-Parallel SPTs

In the previous section, we showed that our bidirectional BFS is accelerated by constructing SPTs rooted at landmarks. We can prune searches more effectively if we have more SPTs. However, it is difficult to construct thousands of SPTs in memory on billion-scale networks owing to the prohibitively large memory consumption and long construction time. In this section, we introduce a *bit-parallel SPT* to address the above-mentioned issue, which is inspired by the bit-parallel BFS presented in [3]. A bit-parallel SPT implicitly represents multiple SPTs in a bit-parallel manner. It requires only $0.27n$ bytes per SPT, whereas an SPT requires n bytes.

4.2.1 Construction

Given a positive integer K, we first choose a vertex set R of K vertices. Moreover, for each vertex $r \in R$, we choose a subset of its neighbors $N_r \subseteq N(r)$. The set N_r is called the *neighbor set* of r. We describe how to select R and its neighbor sets $\{N_r\}_{r \in R}$ later. For each $r \in R$, we maintain not only an SPT rooted at r but also SPTs rooted at N_r. We can efficiently obtain such multiple SPTs in linear time and store them in compact bit-parallelized form by conducting a bit-parallel BFS. The following data structures are maintained for each bit-parallel SPT rooted at r and N_r:

- The array δ_r such that $\delta_r[v] = d(r,v)$ for each $v \in V$.
- Two arrays μ_r^- and $\mu_r^=$ of subsets of N_r such that

$$\begin{aligned} \mu_r^-[v] &= \{p \in N_r \mid d(p,v) = d(v,r) - 1\} \\ \mu_r^=[v] &= \{p \in N_r \mid d(p,v) = d(v,r)\}. \end{aligned}$$

  We represent $\mu_r^-[v]$ and $\mu_r^=[v]$ with $|N_r|$ bits each, where these $|N_r|$ bits indicate the membership of each element of N_r in $\mu_r^-[v]$ and $\mu_r^=[v]$. Thus, each set operation on these sets is conducted by bitwise OR and bitwise AND in constant time if $|N_r|$ is bounded by some constant.

We can find the distance from $v \in V$ to any vertex c in the neighbor set N_r by using two sets $\mu_r^-[v]$ and $\mu_r^=[v]$:

$$d(c,v) = d(r,v) + \begin{cases} -1 & c \in \mu_r^-[v] \\ 0 & c \in \mu_r^=[v] \\ +1 & \text{otherwise.} \end{cases}$$

We can see that $(\delta_r, \mu_r^-, \mu_r^=)$ implicitly represents multiple SPTs whose roots are $\{r\} \cup N_r$. We refer to the set of tuples $\{(\delta_r, \mu_r^-, \mu_r^=)\}_{r \in R}$ as bit-parallel SPTs. We choose N_r such that $|N_r| \leq 64$ holds, and represent $\mu_r^-[v]$ and $\mu_r^=[v]$ with 64-bit integers for each $v \in V$. Thus, we can represent up to 65 SPTs with only $(1+8+8) \times n = 17n$ bytes, which means that bit-parallel SPTs require only $0.27n$ bytes per SPT.

Algorithm 2 shows the pseudo-code for constructing a bit-parallel SPT rooted at r and N_r by improving the bit-parallel BFS construction presented in [3]. Three arrays $(\delta_r, \mu_r^-, \mu_r^=)$ are initialized such that $(\delta_r[r], \mu_r^-[r], \mu_r^=[r])$ equals $(0, \emptyset, \emptyset)$, $(\delta_r[p], \mu_r^-[p], \mu_r^=[p])$ equals $(1, \{p\}, \emptyset)$ for each $p \in N_r$, and $(\delta_r[v], \mu_r^-[v], \mu_r^=[v])$ equals $(\infty, \emptyset, \emptyset)$ for all $v \in V \setminus (\{r\} \cup N_r)$. We enqueue all vertices in N_r into a FIFO queue Q. We determine each value of $(\delta_r, \mu_r^-, \mu_r^=)$ from a vertex close to the root r in a manner similar to BFS. Let v be a vertex popped from Q. The distance from r to v has already been obtained. First, we determine the value of $\mu_r^-[v]$ by looking at each neighbor u of v whose distance $\delta_r[u]$ from

Algorithm 2 Construct a bit-parallel SPT by conducting a bit-parallel BFS

procedure BP-BFS(r, N_r)
  $(\delta_r[v], \mu_r^-[v], \mu_r^=[v]) \leftarrow (\infty, \emptyset, \emptyset)$ for all $v \in V$.
  $(\delta_r[r], \mu_r^-[r], \mu_r^=[r]) \leftarrow (0, \emptyset, \emptyset)$.
  $(\delta_r[v], \mu_r^-[v], \mu_r^=[v]) \leftarrow (1, \{v\}, \emptyset)$ for all $v \in N_r$.
  $Q \leftarrow$ An FIFO queue containing all vertices in N_r.
  while not Q.empty() **do**
    $v = Q$.pop().
    for each $u \in N(v)$ **do**
      if $\delta_r[v] = \delta_r[u] + 1$ **then**
        $\mu_r^-[v] \leftarrow \mu_r^-[v] \cup \mu_r^-[u]$; $\mu_r^=[v] \leftarrow \mu_r^=[v] \cup \mu_r^=[u]$.
    for each $w \in N(v)$ **do**
      if $\delta_r[w] = \infty$ **then** $\delta_r[w] \leftarrow \delta_r[v] + 1$; Q.push(w).
      if $\delta_r[w] = \delta_r[v]$ **then** $\mu_r^=(w) \leftarrow \mu_r^=(w) \cup \mu_r^-(v)$.
  return $(\delta_r, \mu_r^-, \mu_r^=)$.

r equals $\delta_r[v] - 1$. Since there is a path from r to v passing through u, the neighbor sets $\mu_r^-[u]$ and $\mu_r^=[u]$ are subsets of $\mu_r^-[v]$ and $\mu_r^=[v]$, respectively, and we update $\mu_r^-[v]$ and $\mu_r^=[v]$ to $\mu_r^-[v] \cup \mu_r^-[u]$ and $\mu_r^=[v] \cup \mu_r^=[u]$, respectively. Then, we scan each neighbor w of v and propagate the information of v to w. If $\delta_r[w] = \infty$, we enqueue w into Q and update $\delta_r[w]$ to $\delta_r[v] + 1$. If $\delta_r[v] = \delta_r[w]$, we update $\mu_r^=[w]$ to $\mu_r^=[w] \cup \mu_r^-[v]$. After processing all the vertices whose distances from r equal $\delta_r[v]$, $\mu_r^-[v']$ and $\mu_r^=[v']$ are obtained for all $v' \in V$ such that $\delta_r[v'] = \delta_r[v]$, and $\delta_r[w]$ is obtained for all $w \in V$ such that $\delta_r[w] = \delta_r[v] + 1$. We stop this procedure when Q becomes empty after processing some vertex, and we obtain $(\delta_r, \mu_r^-, \mu_r^=)$. This algorithm runs in $O(m)$ time since each set operation can be performed in constant time.

Our bit-parallel BFS is an improved version of the original bit-parallel BFS described in [3]. We use only $O(n)$ space in addition to the graph G and a bit-parallel SPT $(\delta_r, \mu_r^-, \mu_r^=)$, while the original version might require $\Omega(m)$ space to store temporal variables while conducting a bit-parallel BFS. This might lead to unexpectedly large memory consumption when we want to handle billion-scale networks because the size of temporal variables in the original version might be of the order of tens of gigabytes.

Finally, we explain how to select roots and their neighbor sets for a positive integer K. We choose them iteratively. First, we choose a vertex r and its neighbor set N_r such that the value of $\sum_{u \in \{r\} \cup N_r} |N(u)|$ is maximized. For each $v \in V$, the maximum value of $\sum_{u \in \{v\} \cup N_v} |N(u)|$ is obtained by choosing a set $N_u \subseteq N(u)$ of vertices of the highest degrees. Thus, we can compute the maximum value of $\sum_{u \in \{v\} \cup N_v} |N(u)|$ for all $v \in V$ in $O(m)$ time, and choose r with the maximum $\sum_{u \in \{r\} \cup N_r} |N(u)|$. Then, we remove r and N_r from the graph, and choose the next root vertex and its neighbor sets from the graph in the same manner as described above. We repeat this procedure until we obtain K root vertices. To choose root vertices more efficiently, we use lazy greedy heuristics. Its running time becomes $O(K(m + n \log n))$, but it runs much faster than selecting root vertices without this heuristics. Thereafter, we remove those vertices from N_r whose degrees are less than 5% of the maximum degree of $\{r\} \cup N_r$. Although vertices of small degrees do not make a significant contribution in terms of speedup of distance queries, the SPTs rooted at them require the same updating cost as SPTs rooted at other vertices.

Algorithm 3 Recover $\mu_r^-[v]$ and $\mu_r^=[v]$. Two vertex sets X and Y are initialized to empty sets and are used to store vertices visited in RECOVER-R and RECOVER, respectively.

procedure RECOVER(v, r)
  if $v \in U \wedge v \notin X$ **then**
    $(\mu_r^-[v], \mu_r^=[v]) \leftarrow (\emptyset, \emptyset)$; $X \leftarrow X \cup \{v\}$.
    if $v \in N_r$ **then** $\mu_r^-[v] \leftarrow \{v\}$.
    for each $u \in N(v)$ **do**
      if $\delta_r[v] = \delta_r[u] + 1$ **then**
        RECOVER(u, r).
        $\mu_r^-[v] \leftarrow \mu_r^-[v] \cup \mu_r^-[u]$; $\mu_r^=[v] \leftarrow \mu_r^=[v] \cup \mu_r^=[u]$.
    for each $u \in N(v)$ **do if** $\delta_r[v] = \delta_r[u]$ **then**
      RECOVER-R(u, r); $\mu_r^=[v] \leftarrow \mu_r^=[v] \cup \mu_r^-[u]$.

4.2.2 Distance Querying

When answering the distance query $d(s, t)$, the algorithm with bit-parallel SPTs differs from the algorithm described in the previous section in two aspects.

First, we compute the distance upper bound $\overline{d}(s, t)$ as described in [3]. For each $r \in R$, the shortest length of paths from s to t passing through $\{r\} \cup N_r$ is obtained in constant time by using the information of the bit-parallel SPT. The upper bound $\overline{d_r}(s, t)$ from a root r is obtained as follows.

- When $v \in \mu_r^-[s] \cap \mu_r^-[t]$:
  In this case, we have $\overline{d_r}(s, t) = \delta_r[s] + \delta_r[t] - 2$ because, if a vertex $v \in N_r$ satisfies $v \in \mu_r^-[s] \cap \mu_r^-[t]$, it means that $d(s, v) = \delta_r[s] - 1$ and $d(v, t) = \delta_r[t] - 1$.
- When $\mu_r^=[s] \cap \mu_r^-[t] \neq \emptyset$ or $\mu_r^-[s] \cap \mu_r^=[t] \neq \emptyset$:
  In this case, we have $\overline{d_r}(s, t) = \delta_r[s] + \delta_r[t] - 1$ because, if a vertex $v \in N_r$ satisfies $\mu_r^=[s] \cap \mu_r^-[t] \neq \emptyset$ or $\mu_r^-[s] \cap \mu_r^=[t] \neq \emptyset$, we have $d(s, v) = \delta_r[s] \wedge d(v, t) = \delta_r[t] - 1$ or $d(s, v) = \delta_r[s] - 1 \wedge d(v, t) = \delta_r[t]$.
- All other cases: we have $\overline{d_r}(s, t) = \delta_r[s] + \delta_r[t]$.

Hence, we can obtain $\overline{d_r}(s, t)$ in constant time for each $r \in R$ and hence compute $\overline{d}(s, t) = \min_{r \in R} \overline{d_r}(s, t)$ in $O(K)$ time.

Second, we conduct our bidirectional BFS on $G \backslash (\bigcup_{r \in R} \{r\} \cup N_r)$ instead of $G \setminus R$ since we already know the shortest length of paths from s to t passing through $\bigcup_{r \in R} \{r\} \cup N_r$.

4.2.3 Size Reduction

In this subsection, we discuss how to further reduce the space requirement of each SPT in order to handle billion-scale networks. By using our index size reduction technique, the total size of bit-parallel SPTs is reduced by three times on billion-scale networks.

The main idea of our index size reduction is to discard the information of μ_r^- and $\mu_r^=$ from $U \subseteq V$ for each $r \in R$. Thus, we only maintain $\{\delta_r[v]\}_{v \in V}$ and $\{(\mu_r^-[v], \mu_r^=[v])\}_{v \in V - U}$ for each $r \in R$, and recover the discarded information online when we answer distance queries. First, we describe how the discarded information is recovered, and then, we describe how vertex set U is selected. We empirically show the effectiveness of this technique in Section 5.

Recovery of μ_r^- and $\mu_r^=$. Given vertices s and t, we must know $(\delta_r[s], \mu_r^-[s], \mu_r^=[s])$ and $(\delta_r[t], \mu_r^-[t], \mu_r^=[t])$ to compute the upper bound $\overline{d}(s, t)$. Hence, when $s \in U$ or $t \in U$, we need to recompute the discarded information.

A recursive procedure RECOVER in Algorithm 3 shows the pseudo-code of our recovery method, which partially simulates a bit-parallel BFS. We use two vertex sets X and Y

to memorize the vertices visited during the recovery procedure and their information. If necessary, we call RECOVER(s, r) and RECOVER(t, r) to recover $(\mu_r^-[s], \mu_r^=[s])$ and $(\mu_r^-[t], \mu_r^=[t])$, respectively.

Let us assume that RECOVER(v, r) is called. First, we examine whether $v \in U \wedge v \notin X$. If $v \notin U \vee v \in X$, we did not discard the value of $(\mu_r^-[v], \mu_r^=[v])$ or we already obtained $(\mu_r^-[v], \mu_r^=[v])$, and thus, we can immediately return. Otherwise, we set $(\mu_r^-[v], \mu_r^=[v]) = (\emptyset, \emptyset)$ and update $\mu_r^-[v]$ to $\{v\}$ if $v \in N_r$. Then, we recover $(\mu_r^-[v], \mu_r^=[v])$ by examining each neighbor $u \in N(v)$. If $\delta_r[v] = \delta_r[u] + 1$, we obtain the value of $(\mu_r^-[u], \mu_r^=[u])$ by calling RECOVER(u, r) and update $(\mu_r^-[v], \mu_r^=[v])$ as we do in a bit-parallel BFS. If $\delta_r[v] = \delta_r[u]$, we obtain the value of $\mu_r^-[u]$ by calling RECOVER-R(u, r) and update $\mu_r^=[v]$ to $\mu_r^=[v] \cup \mu_r^-[u]$, which is nearly the same procedure as RECOVER except that it only computes the value of μ_r^- to avoid unnecessary computation. After processing all neighbors of v, $(\mu_r^-[v], \mu_r^=[v])$ is obtained in the same manner as a bit-parallel BFS.

Vertex Set Selection. Now, we consider how to select a vertex set U. We need to choose an appropriate vertex set such that we can achieve a good trade-off between the memory requirement and the time required for recovering the discarded information. For example, let us assume that we choose U, which is an independent set of the graph. For any $u \in U$, we can easily recover $(\mu_r^-[u], \mu_r^=[u])$ from the information of its neighbors without recursive computation since their information was not discarded owing to the property of an independent set.

However, the size of the maximum independent set on the graph tends to not be sufficiently large to reduce the index size significantly. For a parameter N, we choose a larger vertex set that is similar to an independent set as follows.

Let $v_1, v_2, \ldots, v_n$ be vertices sorted in non-decreasing order of their degrees. First, we initialize $U = \emptyset$. For each $i \leq n$, we update U to $U \cup \{v_i\}$ if $\left|U \cap N(v_i)\right| < N$ holds; otherwise, we keep U as it is. Thus, we obtain the vertex set U in $O(m)$ time. We note that U becomes larger as we increase N, and U is an independent set on G when $N = 1$.

4.3 Incremental Update

In this section, we describe methods for efficiently updating bit-parallel SPTs. Since our distance querying method depends on only the information in bit-parallel SPTs, it is sufficient for us to update bit-parallel SPTs. Our algorithm is fully dynamic, i.e., we can update bit-parallel SPTs after both insertions and deletions of edges. Since each bit-parallel SPT is updated independently, we only describe how to update a bit-parallel SPT rooted at $\{r\} \cup N_r$. In subsequent sections, we present methods for updating a bit-parallel SPT whose information is not discarded, but we can extend these methods to situations where we discard the information from a vertex set U by properly recovering the discarded information.

4.3.1 Edge Insertion

Let us assume that an edge (s, t) is inserted into the graph. We update a bit-parallel SPT by partially simulating a bit-parallel BFS from the inserted edge. This procedure is similar to the incremental update methods of SPTs [4, 15, 31], but we must carefully design the update procedure to correctly propagate the information in $\mu_r^=$ and μ_r^-.

Let us consider the condition for deciding whether we need to update the tuple $(\delta_r[t], \mu_r^-[t], \mu_r^=[t])$ after inserting the new edge (s, t). We can see that $(\delta_r[t], \mu_r^-[t], \mu_r^=[t])$ must be updated if any of the following conditions holds: (1) $\delta_r[s]+1 < \delta_r[t]$, (2) $\delta_r[s]+1 = \delta_r[t] \wedge (\mu_r^-[s] \not\subseteq \mu_r^-[t] \vee \mu_r^=[s] \not\subseteq \mu_r^-[t] \cup \mu_r^=[t])$, or (3) $\delta_r[s] = \delta_r[t] \wedge \mu_r^-[s] \not\subseteq \mu_r^-[t] \cup \mu_r^=[t]$. Case (1) means that a shorter path from r to t passing through s is formed by the edge insertion. Cases (2) and (3) mean that at least one of neighbor sets $\mu_r^-[t]$ and $\mu_r^=[t]$ has changed although the distance from r to t does not decrease. We define the procedure HASUPDATE$_\text{I}$ in Algorithm 4 such that it returns *true* when at least one of the above conditions holds. If HASUPDATE$_\text{I}(s, t)$ returns *true*, we update tuple $(\delta_r[t], \mu_r^-[t], \mu_r^=[t])$ as follows:

1. When $\delta_r[t] > \delta_r[s] + 1$
   $(\delta_r[t], \mu_r^-[t], \mu_r^=[t]) \leftarrow (\delta_r[s] + 1, \emptyset, \emptyset)$.
2. When $\delta_r[t] \leq \delta_r[s] + 1$

$$\mu_r^-[t] \leftarrow \begin{cases} \mu_r^-[t] \cup \mu_r^-[s] & \text{if } \delta_r[t] = \delta_r[s] + 1 \\ \mu_r^-[t] & \text{if } \delta_r[t] = \delta_r[s] \end{cases}$$

$$\mu_r^=[t] \leftarrow \begin{cases} \mu_r^=[t] \cup \mu_r^-[s] & \text{if } \delta_r[t] = \delta_r[s] + 1 \\ \mu_r^=[t] \cup \mu_r^-[s] & \text{if } \delta_r[t] = \delta_r[s]. \end{cases}$$

After updating tuple $(\delta_r[t], \mu_r^-[t], \mu_r^=[t])$, we examine each neighbor c of t and recursively update $(\delta_r[c], \mu_r^-[c], \mu_r^=[c])$. An overview of our update procedure is shown in Algorithm 4. We maintain a set U of vertices whose tuples have changed. We add t to U when we update $(\delta_r[t], \mu_r^-[t], \mu_r^=[t])$. Let d be the distance from r to vertices in U, and U_{next} be a set of vertices that we process after we determine the tuples of vertices in U. For each $u \in U$, we examine each neighbor $v \in N(u)$ and call HASUPDATE$_\text{I}(u, v)$ to check whether $(\delta_r[v], \mu_r^-[v], \mu_r^=[v])$ has changed. If this tuple has changed, we add v to U if $\delta_r[v] = \delta_r[u]$ and v to U_{next} if $\delta_r[v] = \delta_r[u] + 1$. Then, we examine each neighbor $v \in N(u)$; if UPDATE$_\text{I}(v, u)$ returns *true*, we update $(\delta_r[u], \mu_r^-[u], \mu_r^=[u])$ according to $(\delta_r[v], \mu_r^-[v], \mu_r^=[v])$.

After processing all vertices in U once, we need to process each vertex in U again in the same way as described above. This is because the tuple of each vertex $u \in U$ may change during the first iteration by the information from its neighbors whose distance from r is $\delta_r[u]$. Hence, we first obtain the correctly updated tuple $(\delta_r[u], \mu_r^-[u], \mu_r^=[u])$ for each $u \in U$ in the first iteration. Then, we propagate this information to vertices whose distances from r are $\delta_r[u]+1$. Finally, we replace U with U_{next}. At this point, U is a set of all vertices whose tuples have changed and whose distances from r are $d+1$. We repeat this process as long as $U \neq \emptyset$.

4.3.2 Edge Deletion

Assume that an edge (s, t) is deleted from the graph. First, we remove s or t from the neighbor set N_r if $s \in N_r \wedge t = r$ or $s = r \wedge t \in N_r$ holds, respectively. Since we want to avoid updating the value of $\mu_r^-, \mu_r^=$ for all vertices $v \in V$, we maintain a bit mask that indicates which vertices are removed from N_r and use the mask for filtering out the information that corresponds to vertices already removed from N_r. Thus, we can obtain the information about current N_r when we answer distance queries and update a bit-parallel SPT without updating μ_r^- and $\mu_r^=$.

Algorithm 4 Partially update a bit-parallel SPT after inserting the edge (s, t).

```
1: procedure InsertEdge(s, t)
2:     U ← ∅.
3:     if HasUpdate_I(s, t) then
4:         Update (δ_r[t], μ_r^-[t], μ_r^=[t]); U ← U ∪ {t}.
5:     while not U ≠ ∅. do
6:         U_next ← ∅.
7:         for each u ∈ U do
8:             for each v ∈ N(u) do
9:                 if HasUpdate_I(u, v) then
10:                    Update (δ_r[v], μ_r^-[v], μ_r^=[v]).
11:                    if δ_r[v] = δ_r[u] + 1 then
12:                        U_next ← U_next ∪ {v}.
13:                    else if δ_r[v] = δ_r[u] then
14:                        U ← U ∪ {v}.
15:            for each v ∈ N(u) do
16:                if HasUpdate_I(v, u) then
17:                    Update (δ_r[u], μ_r^-[u], μ_r^=[u]).
18:        Repeat Line 7-17 again.
19:        U ← U_next.
```

Algorithm 5 Partially update a bit-parallel SPT after deleting the edge (s, t).

```
1: procedure DeleteEdge(s, t)
2:     C ← ∅; Q ← An empty FIFO queue.
3:     if (δ_r[t], μ_r^-[t], μ_r^=[t]) must be updated then
4:         Update (δ_r[t], μ_r^-[t], μ_r^=[t]); C ← C ∪ {t}; Q.push(t).
5:     while not Q.empty() do
6:         v ← Q.pop().
7:         for each c ∈ N(v) do
8:             if (δ_r[c], μ_r^-[c], μ_r^=[c]) must be updated then
9:                 Update (δ_r[c], μ_r^-[c], μ_r^=[c]).
10:                C ← C ∪ {c}; Q.push(c).
11:    for each v ∈ C do
12:        if min_{u∈N(v)} δ_r[u] + 1 < ∞ then
13:            Update (δ_r[v], μ_r^-[v], μ_r^=[v]); Add v to U_{δ_r[v]}.
14:    for i = 1 do D do
15:        for each u ∈ U_i do
16:            for each v ∈ N(u) do
17:                if HasUpdate_I(u, v) then
18:                    U_{δ_r[v]} ← U_{δ_r[v]} \ {v}.
19:                    Update (δ_r[v], μ_r^-[v], μ_r^=[v]).
20:                    U_{δ_r[v]} ← U_{δ_r[v]} ∪ {v}.
21:        Repeat Line 15-20 again.
```

After updating N_r, we partially update a bit-parallel SPT in a manner similar to update procedures for an SPT [15,31]. An overview of the procedure is shown in Algorithm 5. Our update procedure consists of two steps.

Step 1: First, we collect a set C of vertices that are updated by the edge deletion. We check whether the distance from r to t has changed after the edge deletion. If $\delta_r[t] > \min_{v \in N(t)} \{\delta_r[v] + 1\}$, the distance from r to t has increased by the edge deletion. Then, we update $(\delta_r[t], \mu_r^-[t], \mu_r^=[t])$ to $(\infty, \emptyset, \emptyset)$, add t to C, and enqueue t into a FIFO queue Q. If the distance from r to t has not changed, we check whether $(\mu_r^-[t], \mu_r^=[t])$ has changed by comparing it with $(\nu_r^-[t], \nu_r^=[t])$, where $\nu_r^-[t] = \bigcup_{c \in N(t): \delta_r[c]+1=\delta_r[t]} \mu_r^-[c]$ and $\nu_r^=[t] = (\bigcup_{c \in N(t): \delta_r[c]+1=\delta_r[t]} \mu_r^=[c]) \cup (\bigcup_{c \in N(t): \delta_r[c]=\delta_r[t]} \mu_r^-[c])$. They correspond to the information of t after we reflect the changes caused by the edge deletion. If $(\mu_r^-[t], \mu_r^=[t]) \neq (\nu_r^-[t], \nu_r^=[t])$, we replace $(\mu_r^-[t], \mu_r^=[t])$ with $(\nu_r^-[t], \nu_r^=[t])$, add t to C, and enqueue t into Q. Then, we recursively examine the neighbors of t. Let c be a vertex popped from Q. We check whether $(\delta_r[c], \mu_r^-[c], \mu_r^=[c])$ is affected by the edge deletion as explained above. If $(\delta_r[c], \mu_r^-[c], \mu_r^=[c])$ changes, we update it correspondingly, and then, we add c to C and push c into Q. We repeat this procedure as long as Q is not empty and obtain C that contains all the vertices affected by the edge deletion.

Step 2: Now, we determine $(\delta_r[v], \mu_r^-[v], \mu_r^=[v])$ for each vertex $v \in C$. For each vertex $v \in C$, we update it to $(\min_{u \in N(v)} \delta_r[u] + 1, \nu_r^-[v], \emptyset)$ if $v \notin N_r$; otherwise, we update it to $(\min_{u \in N(v)} \delta_r[u] + 1, \nu_r^-[v] \cup \{v\}, \emptyset)$. We note that $\nu_r^-[v]$ is obtained from current values of μ_r^-. Then, we prepare vertex sets $U_1, U_2, \ldots, U_D$, where $D = \max_{v \in C: \delta_r[v] < \infty} \delta_r[v]$. For each vertex $v \in C$, we add v to $U_{\delta_r[v]}$ if $\delta_r[v] < \infty$ holds.

Then, we iteratively determine the value of the tuple of each vertex in $U_1, U_2, \ldots, U_D$. We maintain these sets such that $\delta_r[u] = d$ for each $u \in U_d$. Let U_d be the set processed in this iteration. We also assume that we already have the correct value of $(\delta_r[u], \mu_r^-[u], \mu_r^=[u])$ for each $u \in U_{d-1}$ and the correct value of $\mu_r^-[u]$ for each $u \in U_d$. Now, we want to determine the value of $\mu_r^=[u]$ for each $u \in U_d$ and propagate this information to vertices in U_{d+1}. For each $u \in U_d$, we examine each neighbor $v \in N(u)$ such that $v \in C$, and check whether $(\delta_r[v], \mu_r^-[v], \mu_r^=[v])$ must be updated in the same way as the procedure for edge insertions. If HasUpdate$_I(u, v)$ returns *true*, we first remove v from $U_{\delta_r[v]}$, and then, we update $(\delta_r[v], \mu_r^-[v], \mu_r^=[v])$ correspondingly and add v to $U_{\delta_r[v]}$ again in order to reflect this change into the neighbors of v. After processing all vertices $u \in U_d$ once, we repeat this procedure again for correctly reflecting the changes on vertices in $u \in U_d$ caused during the first iteration into the next vertex set U_{d+1}. Finally, we obtain the correct value of $(\delta_r[u], \mu_r^-[u], \mu_r^=[u])$ for each $u \in U_d$, and we also obtain $\delta_r[u] = d + 1$ and the correct value of $\mu_r^-[u]$ for each $u \in U_{d+1}$.

5. EXPERIMENTS

In this section, we describe experiments conducted to evaluate the performance of the proposed method on real-world networks. First, we compared the index size, indexing time, querying time, and update time of the proposed method with those of state-of-the-algorithms on many real-world networks. Then, we analyzed how the number K of bit-parallel SPTs and the parameter N used in index size reduction affect the performance of the proposed method.

5.1 Experimental Settings

All the algorithms were implemented using C++11 and Google Sparse Hash and compiled using gcc 5.2.0 with the -O3 option. All the experiments were conducted using a single thread on a Linux server with Intel Xeon E5-2670 (2.60 GHz) CPU and 512 GB memory.

Datasets. Table 1 summarizes the statistical information of networks used in our experiments. We treated all the graphs as undirected graphs. Datasets used in our experiments are available from Laboratory for Web Algorithmics [6, 7], Koblenz Network Collection [1], and Stanford Network Analysis Project [24].

Benchmark Generation. We sampled 100,000 vertex pairs from $V \times V$ uniformly at random to evaluate the querying time. We reported the average time for answering distance queries on these 100,000 vertex pairs. To evaluate the up-

Table 1: Information of datasets. The average distance when we treat graphs as undirected graphs is denoted by $\overline{D}$. (u) and (d) mean undirected and directed, respectively. $|G|$ denotes the size of a graph when representing each undirected graph with 8 bytes, i.e., each edge appears in the forward and backward adjacency lists, and they are represented by 4-byte integers.

Dataset	Type	n	m	m/n	$\overline{D}$	$\|G\|$
NotreDame	web (d)	326K	1.5M	4.6	7.2	8.3 MB
Hollywood	social (u)	1.1M	114M	99.9	3.9	430 MB
Skitter	comp (u)	1.7M	11M	6.5	5.1	85 MB
Flickr	social (u)	1.7M	16M	9.1	5.3	119 MB
Orkut	social (u)	3.1M	117M	38.1	4.2	894 MB
enwiki2013	social (d)	4.2M	101M	24.1	3.4	701 MB
LiveJournal	social (d)	4.8M	69M	14.2	5.6	327 MB
Indochina	web (d)	7.4M	194M	26.2	7.7	1.1 GB
it2004	web (d)	41M	1.2B	27.9	7.0	7.7 GB
Twitter	social (d)	42M	1.5B	35.3	3.6	9.0 GB
Friendster	social (u)	66M	1.8B	27.5	5.0	13 GB
uk2007	web (d)	106M	3,7B	35.3	6.9	25 GB

date time, we first selected 200 edges randomly from E. Then, we deleted these edges one by one and measured the average deletion time. Next, we inserted these edges one by one and measured the average insertion time. We repeated these update procedures five times and reported the average values. Further, we reported the index size as the total size of all data structures stored in memory except the graph itself. Unless otherwise specified, we ran our algorithm with $K = 20$ and $N = 2$.

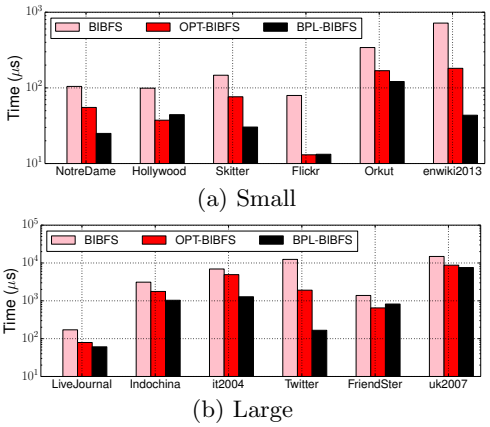
5.2 Performance

5.2.1 *Comparison with Existing Methods*

First, we compared the index size, index construction time, and querying time with those of state-of-the-art exact methods [3, 12, 22]. We also described the average update times. The implementations of (i) PLL with a dynamic update procedure [4] and (ii) HDB [22] were provided by their authors. Both of them were implemented using C++ and compiled using gcc 5.2.0 with the -O3 option. Unfortunately, we could not obtain the implementation of RXL and CRXL [12]; we reported only the experimental results presented in [12]. Delling *et al.* implemented RXL and CRXL in C++ using Visual Studio 2013 with full optimization. Their experiments were conducted on a machine with Intel Xeon E5-2690 (2.90 GHz) CPU and 384 GB memory.

Table 2 lists the construction times, average querying times, and average edge insertion/deletion times. We successfully constructed the indices in less than 1 h on all the datasets used in our experiments. As shown in Table 2, PLL and HDB failed to construct their indices on datasets larger than enwiki2013 owing to their high preprocessing costs. Thus, our method is much more scalable than existing exact methods. Although we did not have information regarding the construction times of RXL and CRXL on all the datasets, we can see that the index construction of our method is several orders of magnitude faster on Flickr and Hollywood.

The average querying times of our method were less than 0.1 ms on networks smaller than Orkut. In particular, the average querying time on Hollywood was at most three times slower than that of the other methods. These timing results



(a) Small

(b) Large

Figure 2: Speedup over the existing method.

seem to be acceptable for some real-world applications. The average querying times of the proposed method reached 1 ms on several large web graphs. This may be because the average distances on these networks are relatively large, which reduces the effectiveness of our techniques. We could handle each edge insertion and deletion in 1.3 ms and 8.1 s, on average, respectively. They are faster than full recomputation of the indices by several orders of magnitude.

Table 3 lists the index sizes. As can be seen, the index sizes of the proposed method are not much larger than the original sizes of the graphs. Hence, our method is highly scalable. Our index sizes for the graphs Skitter, Flickr, and Hollywood were 10 times smaller than the index sizes of PLL and HDB. The index sizes of CRXL become smaller than those of our method on several networks; however, CRXL tends to require larger space during index construction.

5.2.2 *Comparison with bidirectional BFS*

We also compared the average querying times of the BFS, the ordinary bidirectional BFS (BIBFS) that alternately expands searches from two vertices, our optimized bidirectional BFS (OPT-BIBFS), and OPT-BIBFS using bit-parallel SPTs to further prune searches (BPL-BIBFS).

Figure 2 shows their average querying times on small and large networks. As can be seen, our search expansion heuristics consistently offered speedup over BIBFS on all the datasets used in our experiments. The average querying times of OPT-BIBFS were up to six times faster than those of BIBFS. Further, as shown in Figure 2, BPL-BIBFS achieved faster query processing than OPT-BIBFS on almost all the networks. In particular, BPL-BIBFS was more than 70 times faster than BIBFS on Twitter. It could answer distance queries in less than 100 μs on almost all the small networks.

5.3 Analysis

In this section, we evaluate how parameters K and N affect the performance of our method, where K is the number of bit-parallel SPTs and N is the parameter used to reduce the index size.

Figure 3 shows the speedup due to different numbers of bit-parallel SPTs. The average querying times decrease on almost all the datasets when we increase K from 0 to 20 or 30. In particular, the querying times are accelerated by about three times on Skitter and enwiki2013. The querying times with $K = 20$ are not significantly different from those with $K = 40$, and constructing 20 bit-parallel SPTs seems to

Table 2: Timing results. CT, QT, EI, and ED denote the index construction time, average querying time, average edge insertion time, and average edge deletion time, respectively. DNF indicates that the construction was not completed in 10 h or memory usage reached 128 GB.

Dataset	CT [s]					QT [ms]					EI [ms]		ED [ms]
	Ours	PLL	HDB	RXL	CRXL	Ours	PLL	HDB	RXL	CRXL	Ours	PLL	Ours
NotreDame	0.7	13	27	18	22	0.025	0.001	0.001	0.000	0.001	0.49	0.10	19
Skitter	11	901	2,915	2,862	3,511	0.030	0.003	0.004	0.002	0.021	0.38	1.81	39
Flickr	14	1,197	3,038	5,888	7,110	0.013	0.004	0.005	0.003	0.020	0.34	0.96	34
Indochina	43	9,193	DNF	8,390	8,973	1.034	0.005	—	0.001	0.004	0.18	7.22	96
Hollywood	35	18,996	DNF	61,736	75,539	0.044	0.023	—	0.014	0.204	0.24	6.80	41
Orkut	100	DNF	DNF	—	—	0.121	—	—	—	—	0.39	—	25
enwiki2013	109	18,032	DNF	—	—	0.044	0.006	—	—	—	0.39	10.8	148
LiveJournal	56	DNF	DNF	—	—	0.061	—	—	—	—	0.36	—	9
it2004	302	DNF	DNF	—	—	1.278	—	—	—	—	0.11	—	427
Twitter	1,865	DNF	DNF	—	—	0.166	—	—	—	—	0.51	—	6,069
Friendster	3,067	DNF	DNF	—	—	0.821	—	—	—	—	0.55	—	67
uk2007	881	DNF	DNF	—	—	7.571	—	—	—	—	0.53	—	247

Table 3: Data structure sizes. They do not include the sizes of the original graphs

Dataset	Ours	PLL	HDB	RXL	CRXL
NotreDame	29 MB	134 MB	87 MB	23 MB	12 MB
Skitter	169 MB	2.6 GB	3.7 GB	1.0 GB	317 MB
Flickr	168 MB	3.4 GB	4.0 GB	1.8 GB	346 MB
Indochina	910 MB	21 GB	—	1.9 GB	877 MB
Hollywood	273 MB	14 GB	—	5.8 GB	2.0 GB
Orkut	740 MB	—	—	—	—
enwiki2013	648 MB	12 GB	—	—	—
LiveJournal	709 MB	—	—	—	—
it2004	5.1 GB	—	—	—	—
Twitter	4.2 GB	—	—	—	—
Friendster	9.8 GB	—	—	—	—
uk2007	13 GB	—	—	—	—

be sufficient to accelerate bidirectional BFSs. We note that the construction times were nearly linear in the number of SPTs in our experiments.

Table 4 shows how the index sizes decrease as the value of N increases. In Table 4, we only list the results for three web graphs and social networks; the results for other networks are similar to these results. The index sizes are reduced to half of the original index sizes when we increase N from 0 to 2 on all the networks except Orkut and Hollywood.

Table 4 also lists the timing results. As can be seen, the average querying times gradually increase as the value of N increases, but the difference from the querying times with $N = 0$ is not significantly large on any of the datasets used in our experiments. Table 4 shows that the edge insertion times and deletion times increase by up to 13 times when we increase the value of N from zero to one. Basically, the average update times tend to increase as the value of N increases, but the average update times with $N = 1$ and those with $N = 3$ are not significantly different.

Finally, we would like to investigate why the effectiveness of our method differs among the datasets used in our experiments. Figure 4a and 4b show the relationship between distances between queried vertex pairs and the average numbers of edges searched when applying our method to Twitter and Friendster, respectively. In Twitter, owing to the existence of vertices of high degrees, we can see that the number of visited edges rapidly increases as the distances between vertex pairs increase. Hence, we can achieve significant speedup by choosing high-degree vertices as landmarks and avoiding searches from these vertices. In addition, our pruning technique is more effective when distance between a queried vertex pair is small as shown in Figure 4a. These facts indicate that our method works particularly well on networks that have vertices of significantly high degrees and smaller average distance.

6. CONCLUSION

In this paper, we proposed the first non-trivial algorithm that can process exact distance queries on fully dynamic billion-scale networks. The proposed method is based on bidirectional BFSs optimized for small-world networks. To further accelerate our method, we constructed bit-parallel SPTs rooted at high-degree vertices and maintained them after each dynamic update. The experimental results showed that our method can construct indices on billion-scale networks in 1 h and answer distance queries in less than 1 ms, on average, on almost all social networks. The results also showed that our method can process each edge insertion and edge deletion in 1.3 ms and 8.1 s, on average, respectively.

Acknowledgments

T. Hayashi was supported by JST, ERATO, Kawarabayashi Large Graph Project. T. Akiba was supported by JSPS Grant-in-Aid for Research Activity Startup (No. 15H06828) and JST, PRESTO.

7. REFERENCES

[1] konect network dataset - KONECT, May 2015.
[2] B. Abrahao, F. Chierichetti, R. Kleinberg, and A. Panconesi. Trace complexity of network inference. In *KDD*, pages 491–499, 2013.
[3] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pages 349–360, 2013.
[4] 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.
[5] T. Akiba, C. Sommer, and K. Kawarabayashi. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *EDBT*, pages 144–155, 2012.
[6] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for

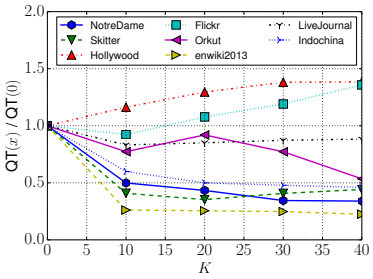
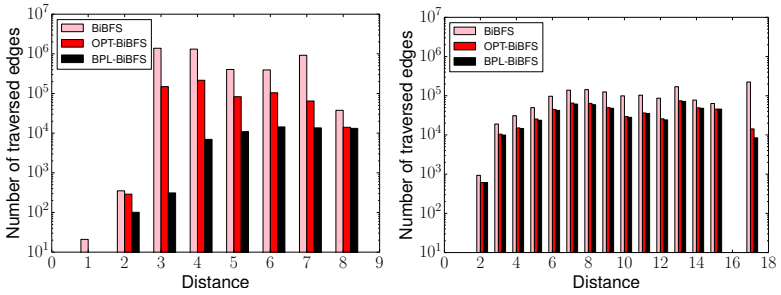


Figure 3: Speedup by bit-parallel SPTs. QT(x) is the average querying time under $K = x$.



(a) Twitter (b) Friendster

Figure 4: Distance vs. number of traversed edges. We used the information obtained when answering 100,000 random queries.

Table 4: Index size reduction and timing results with varying N. IS(x) denotes the index size under $N = x$. QT, EI, and ED denote the average querying time, edge insertion time, and edge deletion time, respectively.

Dataset	IS(N)/IS(0)			QT [ms]				EI [ms]				ED [ms]			
	$N=1$	$N=2$	$N=3$	$N=0$	$N=1$	$N=2$	$N=3$	$N=0$	$N=1$	$N=2$	$N=3$	$N=0$	$N=1$	$N=2$	$N=3$
Indochina	0.44	0.36	0.32	1.76	1.62	1.54	1.49	0.08	0.17	0.18	0.19	23	87	96	103
it2004	0.46	0.37	0.33	1.90	1.81	1.80	1.89	0.07	0.10	0.11	0.13	80	306	427	487
uk2007	0.45	0.36	0.32	10.24	10.29	11.93	11.07	0.06	0.53	0.53	0.54	16	207	247	248
Orkut	0.77	0.71	0.66	0.11	0.12	0.13	0.13	0.24	0.32	0.39	0.50	15	20	25	31
Twitter	0.40	0.30	0.26	0.11	0.15	0.13	0.13	0.30	0.46	0.51	0.57	930	5,395	6,069	6,557
FriendSter	0.51	0.45	0.41	0.85	0.68	0.70	0.88	0.34	0.43	0.55	0.82	52	57	67	105

compressing social networks. In *WWW*, pages 587–596, 2011.

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

[8] P. Boldi and S. Vigna. Axioms for centrality. *Internet Mathematics*, 10(3-4):222–262, 2014.

[9] L. Chang, J. X. Yu, L. Qin, H. Cheng, and M. Qiao. The exact distance to destination in undirected world. *VLDB J.*, 21(6):869–888, 2012.

[10] E. Cohen, D. Delling, F. Fuchs, A. V. Goldberg, M. Goldszmidt, and R. F. Werneck. Scalable similarity estimation in social networks: Closeness, node labels, and random edge lengths. In *COSN*, pages 131–142, 2013.

[11] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[12] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Robust distance queries on massive networks. In *ESA*, pages 321–333. 2014.

[13] N. Du, L. Song, M. Gomez-Rodriguez, and H. Zha. Scalable influence estimation in continuous-time diffusion networks. In *NIPS*, pages 3147–3155, 2013.

[14] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[15] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *J. Algorithms*, 34(2):251–281, 2000.

[16] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. Is-label: An independent-set based labeling scheme for point-to-point distance querying. *PVLDB*, 6(6):457–468, 2013.

[17] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, pages 156–165, 2005.

[18] M. Gomez-Rodriguez, D. Balduzzi, and B. Schölkopf. Uncovering the temporal dynamics of diffusion networks. In *ICML*, pages 561–568, 2011.

[19] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM*, pages 499–508, 2010.

[20] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. WTF: The who to follow service at twitter. In *WWW*, pages 505–514, 2013.

[21] M. Haghir Chehreghani. Effective co-betweenness centrality computation. In *WSDM*, pages 423–432, 2014.

[22] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *PVLDB*, 7(12):1203–1214, 2014.

[23] R. Jin, N. Ruan, B. You, and H. Wang. Hub-accelerator: Fast and exact shortest path computation in large social networks. *CoRR*, abs/1305.0507, 2013.

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

[25] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

[26] S. Maniu and B. Cautis. Network-aware search in social tagging applications: Instance optimality versus efficiency. In *CIKM*, pages 939–948, 2013.

[27] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876, 2009.

[28] Z. Qi, Y. Xiao, B. Shao, and H. Wang. Toward a distance oracle for billion-node graphs. *PVLDB*, 7(1):61–72, 2013.

[29] M. Qiao, H. Cheng, L. Chang, and J. X. Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. In *ICDE*, pages 462–473, 2012.

[30] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

[31] 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.

[32] S. A. Yahia, M. Benedikt, L. V. S. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. *PVLDB*, 1(1):710–721, 2008.