Dynamic and Historical Shortest-Path Distance Queries on Large Evolving Networks by Pruned Landmark Labeling

Takuya Akiba
The University of Tokyo
Tokyo, 113-0033, Japan
t.akiba@is.s.u-tokyo.ac.jp

Yoichi Iwata
The University of Tokyo
Tokyo, 113-0033, Japan
y.iwata@is.s.u-tokyo.ac.jp

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

ABSTRACT

We propose two dynamic indexing schemes for shortest-path and distance queries on large time-evolving graphs, which are useful in a wide range of important applications such as real-time network-aware search and network evolution analysis. To the best of our knowledge, these methods are the first practical exact indexing methods to efficiently process distance queries and dynamic graph updates.

We first propose a dynamic indexing scheme for queries on the last snapshot. The scalability and efficiency of its offline indexing algorithm and query algorithm are competitive even with previous static methods. Meanwhile, the method is dynamic, that is, it can incrementally update indices as the graph changes over time. Then, we further design another dynamic indexing scheme that can also answer two kinds of *historical queries* with regard to not only the latest snapshot but also previous snapshots.

Through extensive experiments on real and synthetic evolving networks, we show the scalability and efficiency of our methods. Specifically, they can construct indices from large graphs with millions of vertices, answer queries in microseconds, and update indices in milliseconds.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation, Performance

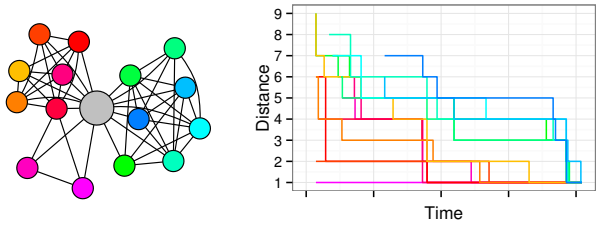
Keywords

Graphs, dynamic graphs, shortest paths, query processing

1. INTRODUCTION

A *shortest-path query* asks the shortest path between two vertices in a graph, and a *distance query* asks the distance between two vertices in a graph. As two of the most fundamental and important operations on graph data, it has a wide range of applications including network-aware search [32, 35, 31, 24], social network analysis [17, 6], bioinformatics [26, 27], computer network management [23, 9], and so on. Consequently, indexing methods for efficiently answering these queries have been intensely studied [11, 24, 34, 5, 25, 16, 4, 15].

However, while many real-world networks are dynamic and growing rapidly, these methods are designed for static networks and none of them can immediately reflect changes on graphs nor answer queries on graphs in the past. To address this issue, in this paper, we study dynamic indexing methods for these queries on large evolving networks. We assume the following two scenarios: one is to ask only *contemporary queries*, and the other is to also ask *historical queries*.



(a) An ego network (b) Distances to the neighbors

Figure 1: An example of social network analysis on a dynamic Facebook subgraph [33] using our method for historical shortest-path distance queries.

1.1 Contemporary Queries

One of two scenarios that we consider in this paper is that we have a real-time dynamic network and we are especially interested in the latest snapshot. In this scenario, we first construct an index from the current graph, then we process *(i)* incremental index updates for graph changes and *(ii)* shortest-path or distance queries on the latest snapshot. In this paper, we refer to queries to the latest snapshot as *contemporary queries*.

Applications in network-aware search: Distance between two users on social networks is considered to indicate the closeness, and used in socially-sensitive search to help users to find more related users or contents [32, 35]. Similarly, distance between two pages in web graphs is one of indicators of relevance, and used in context-aware search to give higher ranks to pages more related to the currently visiting page [31, 24].

WWW'14, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2744-2/14/04.
http://dx.doi.org/10.1145/2566486.2568007.

Real-time index update would definitely improve user experience because these networks are highly dynamic and, more importantly, operations of users are *bursty* with regard to temporal locality [7]. For example, on an online social networking service, when a user begins a friendship with another user, chances are high for the user to keep using the service for a few more minutes. Using our dynamic indexing method, the new friendship can be immediately reflected, which has never been possible with static methods that require periodic index reconstruction (see Section 9 for further discussion).

1.2 Historical Queries

When analyzing historical networks, for which timestamps of vertices and edges are also available, in addition to the latest snapshot, the shortest paths and distances on previous snapshots or transition of them by time are also of interest. In this paper, we call such queries about previous snapshots *historical queries*. In particular, we study two kinds of historical queries. A *snapshot query* asks the shortest path or distance on a specified previous snapshot, and a *change-point query* asks all the moments when the distance between two vertices has changed (see the definition in Section 3.2).

Applications in evolving network analysis: Indexing schemes supporting historical queries would be a powerful back-end for time-evolving network analysis, as it enables many new interesting studies. For example, from the transition of the shortest paths between two vertices, we can grasp the events that shortened the distance or important links that lie in the shortest paths for a long duration, which would provide valuable insights. Moreover, it would also enable distance-based analysis of influential people and communities [17, 6] on dynamic networks. In particular, transition of *closeness centrality* and *distance distribution* can be efficiently computed with historical change-point queries (see Section 8).

Case study: Figure 1 illustrates an example of analysis on a real-world Facebook subgraph [33] based on historical change-point queries. Figure 1a depicts an ego network, i.e., the induced subgraph of a *center vertex* and its neighbors, where the center vertex is the gray one. From the figure, we can observe that there are two clusters on the left and right of the center vertex. Figure 1b shows the transition of the distances between the center vertex and its neighbors, where the colors of the lines correspond to those of the vertices in Figure 1a. We can confirm that the time periods of the appearance of the friendship links are different between the two clusters. Moreover, we find that the two clusters happened quite differently. That is, while the left cluster gradually approached the center vertex, the right cluster became neighbors almost instantly.

1.3 Contributions

In this paper, we propose two dynamic indexing schemes tailored to the two scenarios introduced above. Our methods are exact, that is, answers to queries contain no error. To the best of our knowledge, our method for contemporary queries is the first practical exact indexing method to efficiently process distance queries and dynamic graph updates, and our method for historical queries is the first practical exact indexing method to efficiently handle these historical queries and dynamic graph updates. The two methods are referred to as *dynamic pruned landmark labeling* and *historical pruned landmark labeling*.

Experimental results presented in Section 6 show the efficiency and robustness of our methods. They can construct indices from large networks with millions of vertices, and their query time is very small and around microseconds, which are competitive with previous static methods. Meanwhile, the proposed methods can update their indices for single graph modification in around milliseconds, which is several orders of magnitude faster than reconstructing indices from scratch.

In what follows, we assume undirected unweighted graphs and consider only distance queries for simplicity of exposition, but supporting directed and/or weighted graphs and shortest-path queries is discussed in Section 7.

Supported updates: In this paper, we focus on two kinds of graph updates: vertex additions and edge additions. This is due to the following reasons.

1. As we can see that no previous methods support any incremental updates, supporting only additions is already quite technically challenging.
2. Supporting removals is even much harder and it seems impossible to efficiently support removals without making big compromise on performance such as index size and query time.
3. Removals never happen in certain kinds of real-world dynamic networks such as interaction networks in social media and instant messaging services, co-author networks, co-starring networks, e-mail networks, telephone networks, and so on.
4. In other kinds of real dynamic networks, still, additions are much more frequent than removals [21].
5. For these reasons, it is quite common to ignore removals when analyzing and modeling dynamic networks [8, 19, 21, 33]. As an evidence, widely used public datasets of time-evolving graphs do not contain any removal[1][2].

See Section 9 for further discussion on the practicability of proposed methods.

2. RELATED WORK

In this section, we review previous indexing methods on exact and approximate distance queries. As none of previous methods support incremental updates, all of them are static.

Approximate methods: The major approach to approximate distances is the *landmark-based approach* [29,32,24,13, 25, 30], which precompute and store a number of shortest-path trees rooted at *landmarks*. While these methods easily attain preferable scalability, some of them have critical precision problems for close pairs [25,5], and the other methods with better precision have three orders of magnitude slower query time. Consequently, focus of the research community is shifting toward undermentioned exact methods, leading to recent large improvement on exact methods. In this paper, we also concentrate on exact methods.

Exact methods: Large portion of exact methods can be considered as based on the idea of 2-hop cover [12]. Finding small 2-hop covers efficiently is a challenging and long-standing problem, and thus several algorithms have been

[1] `http://socialnetworks.mpi-sws.org/`
[2] `http://konect.uni-koblenz.de/`

developed [12, 11, 1, 16, 4]. Among them, the most recent method *pruned landmark labeling* [4,36,3] achieves orders of magnitude better scalability than previous methods by exploiting the structures of real networks such as existence of highly central vertices and the core–fringe structure [10,22]. The proposed methods are built on this method.

IS-Label [15] is another recent method that also gained prominent scalability. It partially constructs a 2-hop cover index and combines it with bidirectional Dijkstra's algorithm to answer each query. We compare the proposed methods with this method in our experiments.

An approach based on *tree decompositions* is also reported to be efficient [34,5]. A tree decomposition of a graph G is a tree T with each vertex associated with a set of vertices in G, called a *bag* [28]. Also, the set of bags containing a vertex in G forms a connected component in T. It heuristically computes a tree decompositions and stores shortest-distance matrices for each bag. We also present the results of a tree-decomposition-based method in our experiments.

3. PRELIMINARIES

3.1 Notations and Basic Facts

Static graph (latest snapshot): Let $G = (V, E)$ be a graph with vertex set V and edge set E. We use symbols n and m to denote the number of vertices $|V|$ and the number of edges $|E|$, respectively, when the graph is clear from the context. We suppose $V = \{v_1, v_2, \ldots, v_n\}$ and refer to i as the *ID* of vertex v_i. We also denote the vertex set of G by $V(G)$ and the edge set of G by $E(G)$. Let $d(u, v)$ denote the distance between vertices u, v. If u and v are disconnected in G, we define $d(u, v) = \infty$. We denote the set of neighbors of a vertex $v \in V$ by $N(v)$. That is, $N(v) = \{u \in V \mid (u, v) \in E\}$.

Dynamic graph: When we consider dynamic networks, we use symbol G to denote the latest network, and symbol G_τ to denote the network at time τ. We use these notations for both graphs stored with timestamps for edges and real-time dynamic graphs. For simplicity, we assume time is described by positive integers (i.e., graph snapshots are $G_1, G_2, \ldots$), and we define G_0 as an empty graph. Since we only consider vertex and edge additions, for any $\tau > 0$, $V(G_{\tau-1}) \subseteq V(G_\tau)$ and $E(G_{\tau-1}) \subseteq E(G_\tau)$. We denote the distance between vertices u, v in graph G_τ by $d_\tau(u, v)$. For edge $(u, v) \in G$, we define $t(u, v) = \tau$, where τ is the time when the edge appeared in the graph (i.e., $(u, v) \notin E(G_{\tau-1})$ and $(u, v) \in E(G_\tau)$).

Decreasing distance: As we only consider vertex and edge additions, the following lemma is a key for designing algorithms.

LEMMA 3.1. *Let $G_\tau, G_{\tau'}$ be graphs where $E(G_\tau) \subseteq E(G_{\tau'})$. For any pairs of vertices s and t, $d_\tau(s, t) \geq d_{\tau'}(s, t)$.*

That is, for any pairs of vertices, distance between them never increases by adding vertices or edges. This is actually trivial since the path that was the shortest path in G_τ is also present in $G_{\tau'}$.

Please note that this lemma does not tell that distance on a dynamic graph without removals is uninteresting. For example, diameter or average distance do not necessarily always decrease because of vertex additions (see Figure 3 in Section 8).

3.2 Problem Definition

In this paper, we study indexing methods that, given a graph, construct an index to quickly answer the following queries.

Problem 1 *(Contemporary Distance Query):*
Given: Two query vertices s, t.
Answer: Distance $d_\tau(s, t)$, where τ is the last time when a vertex or an edge is added.

Problem 2 *(Historical Snapshot Distance Query):*
Given: Two query vertices s, t and time τ.
Answer: Distance $d_\tau(s, t)$.

Problem 3 *(Historical Distance Change-point Query):*
Given: Two query vertices s, t.
Answer: Set $C(s, t) = \{(\tau_1, \delta_1), (\tau_2, \delta_2), \ldots\}$ where $(\tau_i, \delta_i) \in C(s, t)$ if and only if $\delta_i = d_{\tau_i}(s, t) \neq d_{\tau_i - 1}(s, t)$.

In addition to answering these queries, we also discuss ways to efficiently update the index as graph changes. As we mentioned before, we focus on the two most important operations, vertex additions and edge additions. Please note that we can assume the newly inserted vertex is isolated since otherwise we can process it by first inserting an isolated vertex and then inserting edges incident to it. Similarly, if multiple edges are inserted simultaneously, we process them one by one.

4. CONTEMPORARY QUERIES

In this section, we propose our dynamic indexing method named *dynamic pruned landmark labeling* for contemporary queries. The method is based on the simple but effective notion of *pruned landmark labeling* [4]. First, we review the index data structure and query algorithm of the *2-hop cover* framework in Section 4.1. Next, we explain our offline indexing algorithm in Section 4.2. Finally, we present the incremental update algorithm in Section 4.3.

4.1 2-Hop Cover Framework

The general framework of *2-hop cover* [12, 11, 1] is as follows. Our method also follows this framework. For each vertex v, we precompute and store a *label* denoted as $L(v)$, which is a set of pairs (u, δ_{uv}), where u is a vertex and $\delta_{uv} = d(u, v)$. We call the set of labels $\{L(v)\}_{v \in V}$ as an *index*, and a pair in a label as a *label entry*. We sometimes abbreviate $(v, \delta_{uv}) \in L(u)$ to $v \in L(u)$.

To answer a distance query between vertices s and t, we compute and answer $\textsc{Query}(s, t, L)$ defined as follows,

$$\textsc{Query}(s, t, L) = \min \{\delta_{vs} + \delta_{vt} \mid (v, \delta_{vs}) \in L(s), (v, \delta_{vt}) \in L(t)\}.$$

We define $\textsc{Query}(s, t, L) = \infty$ if $L(s)$ and $L(t)$ do not share any vertex. Index L is called a *(distance-aware) 2-hop cover* of G if $\textsc{Query}(s, t, L) = d(s, t)$ for any pair of vertices s and t.

For each vertex v, we store the label $L(v)$ so that pairs in it are sorted by their vertices. Then, we can compute $\textsc{Query}(s, t, L)$ in $O(|L(s)| + |L(t)|)$ time using a merge-sort-like algorithm.

4.2 Offline Indexing Algorithm

In this subsection, we explain our labeling algorithm that efficiently computes small labels for the 2-hop cover framework by conducting pruned BFSs from every vertex. We

explain the algorithm in two steps: we first introduce the *naive landmark labeling algorithm*, which computes labels by conducting normal BFSs from every vertex in Section 4.2.1, then introduce pruning to the algorithm in Section 4.2.2.

4.2.1 Naive Landmark Labeling Algorithm

Let $V = \{v_1, v_2, \ldots, v_n\}$. We start with an empty index L_0, where $L_0(u) = \emptyset$ for any $u \in V$. We conduct BFSs from all vertices in the order of $v_1, v_2, \ldots, v_n$. After the k-th BFS from vertex v_k, we update the index as follows. For all the vertices u that are reachable from v_k, we set $L_k(u) = L_{k-1}(u) \cup \{(v_k, d(v_k, u))\}$. For all the unreachable vertices u, we do not change their labels, that is, we just set $L_k(u) = L_{k-1}(u)$.

After n BFSs, L_n is the final index. Obviously $\textsc{Query}(s, t, L_n) = d(s, t)$ for any pair of vertices s and t, since $(s, 0) \in L(s)$ and $(s, d(s, t)) \in L(t)$ if they are reachable. Therefore, L_n is a correct 2-hop cover for exact distance queries. However, this method is also obviously inefficient, due to $\Theta(mn)$ indexing time and $\Theta(n^2)$ index space.

LEMMA 4.1. $\textsc{Query}(s, t, L_n) = d(s, t)$ *for any pair of vertices* s *and* t.

4.2.2 Pruned Landmark Labeling Algorithm

As with the naive method above, we conduct *pruned* BFSs from all vertices in the order of $v_1, v_2, \ldots, v_n$. We start with an empty index L'_0, and we construct index L'_k from L'_{k-1} and the result of the k-th pruned BFS from v_k.

We prune BFSs as follows. Suppose we are conducting the k-th pruned BFS from v_k to create index L'_k from L'_{k-1}, and visiting vertex u with distance δ. We issue a query between v_k and u to the current incomplete index L'_{k-1}, and compare the answer to δ. If $\textsc{Query}(v_k, u, L'_{k-1}) \leq \delta$, then we prune u. That is, we do not add pair (v_k, δ) to the label of u (i.e. $L'_k(u) = L'_{k-1}(u)$), and we do not traverse any edges incident to u. Otherwise, as usual, we set $L'_k(u) = L'_{k-1}(u) \cup \{(v_k, \delta)\}$ and traverse all the edges incident to u. As with the previous method, we also set $L'_k(u) = L'_{k-1}(u)$ for all vertices $u \in V$ that were not visited in the k-th pruned BFS. This pruned BFS algorithm is described in Algorithm 1. After n pruned BFSs, L'_n is the final index.

THEOREM 4.1. *For any* $s, t \in V$ *and* $1 \leq k \leq n$, *we have* $\textsc{Query}(s, t, L'_k) = \textsc{Query}(s, t, L_k)$.

We give a proof sketch (refer to [4] for details).

PROOF SKETCH. Let $P_k(s, t)$ be a shortest path between s and t passing through one of $v_1, \ldots, v_k$, and let $d_k(s, t)$ be the length of it. Note that $\textsc{Query}(s, t, L_k) = d_k(s, t)$.

Suppose that the theorem holds for $k' < k$, and we show the theorem for k. Fix vertices s and t in V. Let v_i ($1 \leq i \leq k$) be the vertex in $P_k(s, t)$ with the lowest index. It suffices to show that v_i is added to $L'_i(s)$ and $L'_i(t)$ as it implies that $\textsc{Query}(s, t, L'_k) \leq d(s, v_i) + d(v_i, t) = d_k(s, t)$.

Suppose that v_i is not added to $L'_i(s)$ (t can be handled analogously). Then, there exists some vertex u on the path $P_k(v_i, s)$ such that the BFS from v_i is pruned at u. The reason it is pruned is that there exists a path between v_i and u of length at most $d(v_i, u)$ passing through some vertex $v_{i'}$ with $i' < i$. This implies that there exists a shortest path between v_i and s passing through $v_{i'}$ and hence a shortest path between s and t passing through $v_{i'}$. This contradicts the choice of i. □

Algorithm 1 Pruned BFS from $v_k \in V$ to create index L'_k.

```
1: procedure PrunedBFS(G, v_k, L'_{k-1})
2:     Q ← a queue with only one element v_k.
3:     P[v_k] ← 0 and P[v] ← ∞ for all v ∈ V(G) \ {v_k}.
4:     L'_k[v] ← L'_{k-1}[v] for all v ∈ V(G).
5:     while Q is not empty do
6:         Dequeue u from Q.
7:         if Query(v_k, u, L'_{k-1}) ≤ P[u] then
8:             continue
9:         L'_k[u] ← L'_{k-1}[u] ∪ {(v_k, P[v_k])}
10:        for all w ∈ N_G(v) s.t. P[w] = ∞ do
11:            P[w] ← P[u] + 1.
12:            Enqueue w onto Q.
13:    return L'_k
```

The theorem above says that we can answer exact distance with the index L'_n. Furthermore, it says that, during the k-th pruned BFS, we prune vertices whose shortest path to v_k passes through at least one of vertices $v_1, v_2, \ldots, v_{k-1}$.

Time complexity: We present an estimate. Let l be the average size of labels and d be the average degree. We visit $O(nl)$ vertices in total, where we traverse $O(d)$ edges and evaluate a query in $O(l)$ time. Therefore, the time complexity is roughly estimated as $O(nl(d + l))$ time, which corresponds with our experimental results in Section 6 to some extent. In our experiments, l was around hundreds.

4.2.3 Vertex Ordering Strategy

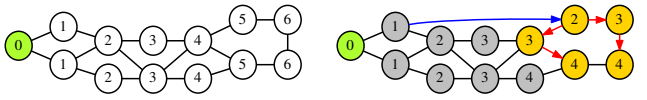
In real-world networks, by properly choosing the order of vertices from which we conduct BFSs, the pruning drastically reduces the search space and label sizes, leading to the high efficiency of the proposed method. In real-world networks, there are highly central vertices (sometimes called *hubs*). By conducting BFSs from these vertices first, label entries to these vertices greatly contribute to pruning since many shortest paths pass through these central vertices. Since it is obviously too costly to compute the optimal order of vertices, as a heuristic vertex ordering strategy, we choose vertices by the decreasing order of degrees. Experimental results in Section 6 show that this simple vertex ordering strategy makes our method sufficiently effective.

4.3 Online Incremental Update Algorithm

Finally, we present our new algorithm that incrementally updates the current index to reflect graph changes. As we noted in Section 3.2, when a new vertex is added, we can assume it is an isolated vertex. Under the 2-hop framework, inserting a new isolated vertex v can be easily done by setting a new empty label $L(v) = \emptyset$. Thus, in what follows, we focus on edge additions. The idea behind our efficient update algorithm is to carefully *resume* and *stop* pruned BFSs.

Suppose that we are maintaining index M for dynamic graph G and we want to update M to reflect a newly inserted edge (a, b) that was previously absent. We refer to the old graph without the new edge as $G_{\tau-1}$ and the new graph with the new edge as G_τ. Similarly, we refer to the old and new index as $M_{\tau-1}$ and M_τ, respectively.

From Lemma 3.1, it is sound to keep outdated distances in the index since we never underestimate distances because of them. Therefore in our method, we do not remove outdated label entries since detecting them is too costly. We only



(a) A graph (b) The graph after inserting a new edge (drawn in blue)

Figure 2: A running example for the update algorithm. The green vertex is the root, and the distance to the root is written in each vertex.

add new label entries or rewrite distances of existing label entries. Under this strategy, the minimality of the index M_τ as a whole is broken after updates, but we will later see that the set of newly added label entries is minimal to answer correct distances in G_τ. Moreover, we will see that the increase of label sizes is satisfyingly small in practice (Section 6.2.3).

4.3.1 Update Algorithm for Naive Labeling

We first describe an update algorithm for the naive landmark labeling method. The obvious way is to conduct full BFSs from every vertex again and update labels, which is not different from the indexing algorithm at all. To update labels more efficiently, let us reduce the search space of each BFS. The two key insights here are the following.

LEMMA 4.2. *If the distance from vertex v_k to vertex u has changed, then all the new shortest paths between them pass through the new edge (a, b).*

LEMMA 4.3. *Suppose the shortest path P between v_k and $u \neq a, b$ has changed. Then the distance between v_k and w has changed, where w is the penultimate vertex in P.*

We can assume $d_{\tau-1}(v_k, a) \leq d_{\tau-1}(v_k, b)$ without loss of generality. Based on these facts, for every v_k, it suffices to *resume* the BFS from b originally rooted at v_k and *stop* at unchanged vertices. That is, instead of inserting $(v_k, 0)$ to the initial queue, we insert $(b, d_{\tau-1}(v_k, a) + 1)$ to the initial queue, which corresponds to the position after passing through the new edge (a, b), and we do not traverse edges from vertex u if $\delta \geq d_{\tau-1}(v_k, u)$, where δ is the tentative distance for u drawn from the queue. Figure 2 illustrates an example.

4.3.2 Update Algorithm for Pruned Labeling

Now we explain our update algorithm for the pruned landmark labeling. We introduce pruning to the previous update algorithm. Let s, t be vertices and k be an integer. We define a new function PREFIXALQUERY as follows,

$$\text{PREFIXALQUERY}(s, t, M, k) = \min\{\delta_s + \delta_t \mid (v_i, \delta_s) \in M(s), (v_i, \delta_t) \in M(t), i \leq k\}.$$

That is, PREFIXALQUERY(s, t, M, k) is the answer to the query between vertices s and t computed from the index M only using distances to vertices whose IDs are at most k. We define PREFIXALQUERY$(s, t, M, k) = \infty$ if $M(s)$ and $M(t)$ do not share any vertex whose ID is at most k. Using this function, suppose we are conducting a resumed BFS originally rooted at v_k and visiting vertex u with distance δ, we prune u if PREFIXALQUERY$(v_k, u, M, k) \leq \delta$.

However, one crucial question is left: for which roots do we need to resume BFSs? The obvious solution is resuming

Algorithm 2 Update index M for newly added edge (a, b)

```
1: procedure InsertEdge(G, a, b, M)
2:     for all v_k ∈ M(a) ∪ M(b) from lower k do
3:         ResumePBFS(G, v_k, b, d(v_k, a) + 1, M) if v_k ∈ M[a]
4:         ResumePBFS(G, v_k, a, d(v_k, b) + 1, M) if v_k ∈ M[b]
5: procedure ResumePBFS(G, v_k, u, δ_ru, M)
6:     Q ← a queue with only one element (u, δ_ru).
7:     while Q is not empty do
8:         Dequeue (v, δ) from Q.
9:         if PrefixalQuery(v_k, v, M, k) ≤ δ then
10:            continue
11:        M[v] ← M[v] ∪ {(v_k, δ)}
12:        for all w ∈ N(v) do
13:            Enqueue (w, δ + 1) onto Q.
```

BFSs no matter what their roots are as with the previous algorithm, but it is too inefficient since it takes at least $\Omega(|V|)$ time. Interestingly, the answer is exactly what we have in $M(a)$ and $M(b)$. That is, it suffices to conduct resumed BFSs originally rooted at v_k if $v_k \in M(a) \cup M(b)$. This is because, if $v_k \notin M(a) \cup M(b)$, both a and b are pruned or unreached during previous (resumed) BFSs rooted at v_k, and since the shortest path between v_k and them has not changed from the last snapshot, the situation does not change at all. The total algorithm is described as Algorithm 2.

Time complexity: To roughly estimate the time complexity, we assume $|L(v)| = O(l)$ and the number of vertices visited during each resumed BFS is $O(s)$, where l and s are some integers. Then, since we conduct resumed BFSs $O(l)$ times and each pruning test takes $O(l)$ time, in total, each update can be done in $O(l^2 s)$ time. In our experiments, l was around hundreds and s was around tens on average (see Section 6.2.3).

4.3.3 Proof of Correctness

We prove the correctness of the proposed incremental update algorithm. For vertices s, t and $0 \leq k \leq n$, we define the *restricted distance* between s and t with respect to k as $d'(s, t, k) = \min_{i \leq k}\{d(s, v_i) + d(v_i, t)\}$. We define $d'(s, t, k) = \infty$ if $k = 0$ or v_i is unreachable from s or t for all $i \leq k$. As with d_τ, we denote the restricted distance at time τ by $d'_\tau(s, t, k)$. The following notion of correctness is important.

DEFINITION 4.1 (PREFIXAL CORRECTNESS). *Let M be a 2-hop cover index for graph G. Index M is* prefixally correct *if* PREFIXALQUERY$(s, t, M, k) = d'(s, t, k)$ *for any $s, t \in V$ and $0 \leq k \leq n$.*

Note that prefixal correctness is stronger than normal correctness. Due to Theorem 4.1, the initial index constructed by the *full* pruned landmark labeling algorithm satisfies prefixal correctness. In what follows, we prove that prefixal correctness of an index is maintained by the incremental update algorithm.

In what follows, let $M_{\tau-1}$ be a prefixally correct index for graph $G_{\tau-1}$, let G_τ be the graph created by inserting edge $(a, b) \notin E(G_{\tau-1})$ to $G_{\tau-1}$, and M_τ be the index updated by the Algorithm 2 from $M_{\tau-1}$ for the edge addition.

LEMMA 4.4. *For any pair of vertices s and t and $0 < k \leq n$, we have* PREFIXALQUERY$(s, t, M_\tau, k-1) = d'_\tau(s, t, k-1)$. *For any vertex u and $0 < k \leq n$, if $d_\tau(v_k, u) < d_{\tau-1}(v_k, u)$ and $d_\tau(v_k, u) < d'_\tau(v_k, u, k-1)$, then we have $(v_k, d_\tau(v_k, u)) \in M_\tau(u)$.*

PROOF. Since $d_\tau(v_k, u) < d_{\tau-1}(v_k, u)$, all the shortest paths from v_k to u in the new snapshot G_τ pass through the new edge (a, b). We assume $d_\tau(v_k, a) < d_\tau(v_k, b)$ with out loss of generality, and suppose that one of the shortest paths is of the form $(v_k, \ldots, a, b = w_0, w_1, w_2, \ldots, u = w_l)$. We can observe that, not only u but also for any w_i, $d_\tau(v_k, w_i) < d_{\tau-1}(v_k, w_i)$ holds. Moreover, since $d_\tau(v_k, u) < d'_\tau(v_k, u, k-1)$, none of the shortest paths between v_k and u in G_τ goes through the vertex v_j for any $j < k$. This also holds for any w_i, that is, none of the shortest paths between v_k and w_i in G_τ passes through v_j for any $j < k$, and thus $d_\tau(v_k, w_i) < d'_\tau(v_k, w_i, k-1)$.

During the resumed BFS originally rooted at v_k, we have PREFIXALQUERY$(v_k, w_i, M_\tau, k) \geq \min\{d'_\tau(v_k, w_i, k-1), d_{\tau-1}(v_k, w_i)\}$ for any w_i. Therefore, PREFIXALQUERY$(v_k, w_i, M_\tau, k) \geq d_\tau(v_k, w_i)$, and w_i is not pruned. Thus, the BFS reaches vertex u with the correct distance $d_\tau(v_k, u)$, and pair $(v_k, d_\tau(v_k, u))$ is newly added to the label $M_\tau(u)$. □

THEOREM 4.2. *M_τ is a prefixally correct index for G_τ.*

PROOF. We prove the prefixal correctness of M_τ by mathematical induction on k. For $k = 0$, it is true as, for any pair of vertices s and t with $s \neq t$, PREFIXALQUERY$(s, t, M_\tau, 0) = d'(s, t, 0) = \infty$. Now we assume $k > 0$ and PREFIXALQUERY$(s, t, M_\tau, k-1) = d'(s, t, k-1)$ for any pairs of vertices s and t, and prove PREFIXALQUERY$(s, t, M_\tau, k) = d'(s, t, k)$ for any pairs of vertices s, t.

Let $\delta' = d'_\tau(s, t, k)$. If $\delta' = d'_\tau(s, t, k-1)$, then we have nothing to show due to the assumption of the mathematical induction. Otherwise, since $\delta' < d'_\tau(s, t, k-1)$, $\delta' = d_\tau(s, v_k) + d_\tau(v_k, t)$. Therefore, it suffices to show that $(v_k, d_\tau(v_k, s)) \in M_\tau(s)$ and $(v_k, d_\tau(v_k, t)) \in M_\tau(t)$. Due to the symmetry between s and t, we only show $(v_k, d_\tau(v_k, s)) \in M_\tau(s)$.

First, we consider the case $d_\tau(v_k, s) = d_{\tau-1}(v_k, s)$. From $d'_\tau(s, t, k) = d_\tau(s, v_k) + d_\tau(v_k, t) < d'_\tau(s, t, k-1)$, there is no vertex v_i with $i \leq k-1$ on any shortest path between s and v_k in G_τ. This is the case in $G_{\tau-1}$ since $d_\tau(v_k, s) = d_{\tau-1}(v_k, s)$. It follows that $d'_{\tau-1}(v_k, s, k-1) > d'_{\tau-1}(v_k, s, k)$. Thus if $(v_k, d_\tau(v_k, s)) \notin M_{\tau-1}(s)$, then PREFIXALQUERY$(v_k, s, M_{\tau-1}, k) = d'_{\tau-1}(v_k, s, k-1) > d'_{\tau-1}(v_k, s, k)$, which contradicts to the prefixal correctness of $M_{\tau-1}$. Therefore, $(v_k, d_\tau(v_k, s)) \in M_{\tau-1}(s) \subseteq M_\tau(s)$ holds.

Otherwise, we can assume $d_\tau(v_k, s) < d_{\tau-1}(v_k, s)$. As $\delta' < d'_\tau(s, t, k-1)$, none of the shortest paths from v_k to s goes through the vertex v_i for any $i < k$, and thus $d_\tau(v_k, s) < d'_\tau(v_k, s, k-1)$. From Lemma 4.4, $(v_k, d_\tau(v_k, s)) \in M_\tau(s)$. □

COROLLARY 4.1. *Let M_1 be the index constructed by the offline indexing algorithm for graph G_1. For $2 \leq i \leq \tau$, let M_i be the index updated by Algorithm 2 from M_{i-1} for the edge addition to make G_i from G_{i-1}. Then, the index M_τ is a correct 2-hop index for G_τ.*

Moreover, the sufficient condition for a pair to be added shown in Lemma 4.4 is actually also a necessary condition. Therefore, the minimality of newly added pairs is derived.

THEOREM 4.3. *The label entries added by the update algorithm are minimal. That is, for any vertex u and any pair in $M_\tau(u) \setminus M_{\tau-1}(u)$, if we remove the pair from $M_\tau(u)$, then M_τ becomes an incorrect 2-hop index for G_τ.*

Bit-parallel labeling.

As with static pruned landmark labeling, the dynamic method can be combined with *bit-parallel labeling* [4] to further improve the performance. Due to the space limit, we omit the details, but bit-parallel labels can be also updated incrementally. For each root r, we resume the BFS from one of the endpoints of the new edge. The main difference is that, for any visited vertex v, even if $d(r, v)$ has not changed, we push v to the queue if the associated bitsets are changed.

5. HISTORICAL QUERIES

In this section, we propose a new indexing scheme referred to as *historical pruned landmark labeling* to efficiently process historical queries defined in Section 3. Unlike our first method for contemporary queries, as there is no previous work on these queries, we start from designing a new index framework (i.e., data structure and query algorithms), named *historical 2-hop cover*, in Section 5.1. Then, we propose an offline indexing algorithm that constructs an index from a stored historical graph in Section 5.2. Since it is more involved than that for contemporary queries, we explain our indexing algorithm with three steps: we start from an algorithm based on dynamic programming, next we turn the algorithm into a BFS-like algorithm, and then introduce pruning to the algorithm. Finally, we present an online incremental update algorithm for online graph changes in Section 5.3.

5.1 Historical 2-Hop Cover Framework

First, we propose a new indexing framework (i.e., data structure and query algorithms) referred to as the *historical 2-hop cover framework*. Since there is no previous work on these queries, it is the first framework for historical distance queries. The main technical challenge here is to design (almost) linear-time query algorithms for both kinds of historical queries.

5.1.1 Data Structure

For each vertex v, we store a label $L(v)$. Label $L(v)$ is a set of triples (u, τ, δ_{uv}), where u is a vertex, τ describes time, and $\delta_{uv} = d_\tau(u, v)$. Due to Lemma 3.1, $(u, \tau, \delta_{uv}) \in L(v)$ also indicates $d_{\tau'}(u, v) \leq \delta_{uv}$ for $\tau' \geq \tau$.

As with the normal 2-hop cover framework, we store triples in a label in the ascending order of the IDs of destination vertices. In addition, we sort triples that share the same destination vertex in ascending order of distance (i.e. descending order of time).

5.1.2 Answering Snapshot Queries

A snapshot query between a pair of vertices s and t at time τ can be answered in $O(|L(s)| + |L(t)|)$ time. Though we basically conduct a merge-sort-like algorithm as with normal 2-hop cover, there are several differences. First, we need to ignore label entries with time later than τ. In addition, to handle triples in a label that share the same destination vertex, among them we only see the newest label entry with time earlier than or equal to τ. That is, if (u_i, τ_i, δ_i) and $(u_{i+1}, \tau_{i+1}, \delta_{i+1})$ are consecutive labels in $L(s)$ where $u_i = u_{i+1}$ and $\tau \geq \tau_i > \tau_{i+1}$, then we ignore the second label since $\delta_i \leq \delta_{i+1}$ from Lemma 3.1.

5.1.3 Answering Change-point Queries

Answering a change-point query between vertices s and t is a little more involved, but can be done in $O(l \log l)$ time, where $l = |L(s)| + |L(t)|$. First, we conduct a merge-sort-like algorithm to enumerate candidates of distance change-points. From pairs of triples $(u, \tau_s, \delta_s) \in L(s)$ and $(u, \tau_t, \delta_t) \in L(t)$, we enumerate pairs $(\tau, \delta) = (\max\{\tau_s, \tau_t\}, \delta_s + \delta_t)$, which indicates $d_\tau(s,t) \leq \delta$. Then, we sort these pairs by time τ. Finally, we remove unnecessary pairs. That is, if (τ_i, δ_i) and $(\tau_{i+1}, \delta_{i+1})$ are consecutive pairs, where $\tau_i \leq \tau_{i+1}$ and $\delta_i \leq \delta_{i+1}$, then we remove the second pair.

Again, the remaining issue is to handle triples in a label that share the same destination vertex. If we check every pair of these triples, in the worst case, it would take quadratic time. However, for these triples, we can also apply a merge-sort-like scan algorithm by considering time of these triples. In total, the first step and the final step can be done in linear time, and the time complexity is dominated by sorting.

5.2 Offline Indexing Algorithm

For presenting the indexing algorithm for contemporary queries, we first described a naive labeling algorithm without pruning, then we introduced pruning to present the indexing algorithm. However, designing an algorithm for historical queries is more challenging since, while the naive labeling algorithm was obvious for contemporary queries, this time, even the naive labeling algorithm without pruning is not trivial for historical 2-hop cover. Therefore, we explain our indexing algorithm with three steps: we start from an algorithm based on dynamic programming, next we turn the algorithm into a BFS-like algorithm, and finally introduce pruning to the algorithm to obtain our indexing algorithm.

5.2.1 Dynamic Programming

As usual, we start from an empty index L_0 and construct index L_k from L_{k-1} by adding triples whose destination vertex is v_k. Let D be the maximum distance to a connected vertex from v_k regarding all the snapshots. Let T be a $(D+1) \times |V|$ table. We conduct dynamic programming on the table T so that each cell $T[\delta][u]$ denotes the earliest time τ with $d_\tau(v_k, u) \leq \delta$. First, we fill the cells with distance zero as $T[0][v_k] = 0$ and $T[0][u] = \infty$ for any $u \neq v_k$. Then, we compute the values of cells with distance $\delta > 0$ from smaller δ by the following recurrence relation:

$$T[\delta][u] = \min_{w \in N(u)} \{\max\{T[\delta-1][w], t(w,u)\}\},$$

for any $u \neq v_k$ and $T[\delta][v_k] = 0$.

LEMMA 5.1. *Each cell $T[\delta][u]$ denotes the earliest time τ with $d_\tau(v_k, u) \leq \delta$.*

This lemma can be proved by mathematical induction on δ. After computing the table, we add triple (v_k, δ, τ) to label $L_k(u)$ where $\tau = T[\delta][u]$ if $T[\delta][u] \neq \infty$ and $\delta = 0$ or $T[\delta][u] < T[\delta-1][u]$.

5.2.2 Historical Naive Landmark Labeling

While the algorithm above computes the correct index, it takes $\Theta(D|E|)$ time and $\Theta(D|V|)$ space. In this subsection, we reduce the time and space complexity by skipping unnecessary computations. Again, we suppose we are to construct index L_k from L_{k-1} by adding triples whose destination vertex is v_k.

Algorithm 3 Pruned BFS from $v_k \in V$ to create index L'_k for historical queries.

```
1: procedure PrunedBFS(G, v_k, L'_{k-1})
2:     L'_k[v] ← L'_{k-1}[v] for all v ∈ V(G).
3:     Q ← a queue with only one element v_k.
4:     T[v_k] ← 0 and T[v] ← ∞ for all v ∈ V(G) \ {v_k}.
5:     T'[v] ← ∞ for all v ∈ V(G).
6:     for all δ = 0, 1, ... until Q gets empty do
7:         Q' ← an empty queue.
8:         for all u ∈ Q do
9:             if QuerySnapshot(v_k, u, T[u], L'_{k-1}) ≤ δ then
10:                continue
11:            L'_k[u] ← L'_k[u] ∪ {(v_k, T[u], δ)}
12:            for all w ∈ N_G(v) do
13:                τ' = max{T[u], t(u,w)}
14:                if τ' < T'[w] and τ' < T[w] then
15:                    if T'[w] = ∞ then
16:                        Enqueue w onto Q'.
17:                    T'[w] ← τ'.
18:        T[u] ← T'[u], T'[u] ← ∞ for all u ∈ Q'.
19:        Q ← Q'.
20:    return L'_k
```

The key insight here is the following simple fact. For simplicity, we define $T[-1][u] = \infty$ for any vertex u in the following. For any vertex u and $\delta \geq 0$, if $T[\delta-1][w] = T[\delta][w]$ for all $w \in N(u)$, then $T[\delta+1][u] = T[\delta][u]$. Therefore, we avoid vainly computing values of such cells as follows. For each distance $\delta \geq 0$ and vertex u, we initially set $T[\delta+1][u] = T[\delta][u]$. Then, we only check edges (w,u) incident to vertex w with $T[\delta-1][w] \neq T[\delta][w]$, and update $T[\delta+1][u]$ by $\max\{T[\delta][w], t(w,u)\}$ if it is smaller than the current value. This can be efficiently achieved by managing vertices with queues.

However, even using queues, it still takes $\Omega(D|V|)$ time and $\Theta(D|V|)$ space due to the two-dimensional table. Thus, instead of straightforwardly using a two-dimensional table, we use two one-dimensional arrays with length $O(|V|)$, and avoid full initialization for each distance δ. Consequently, conducting queue-based dynamic programming and avoiding $\Theta(|V|)$ time initialization for each step, the total time complexity becomes $O(m')$, where m' is the number of traversed edges including duplications. Also note that, by using queues, we do not need to obtain the maximum distance D beforehand, as it suffices to stop when the queues get empty.

5.2.3 Historical Pruned Landmark Labeling

We finally introduce pruning to the previous algorithm to obtain our indexing algorithm. Suppose we have started with an empty index L'_0 and we are constructing index L'_k from L'_{k-1} and the result of the k-th pruned BFS from v_k. Along with the labeling algorithm for contemporary queries, after drawing vertex u from the queue Q, we issue a query between v_k and u, and if the distance is at most δ, we prune vertex u. The difference from the algorithm for contemporary queries here is that we issue a snapshot query with regard to time $T_0[u]$. The total algorithm is described as Algorithm 3.

The correctness of this algorithm is not obvious, but can be proved as the almost same way as the correctness of the pruned landmark labeling algorithm for contemporary queries.

Table 2: Comparing our dynamic method (Dynamic PLL) with other static state-of-the-art methods with regard to indexing time (IT), index size (IS) and query time (QT). LN denotes average label size for each vertex. DNF means it did not finish in one day or ran out of memory.

Dataset	Dynamic PLL				IS-Label [15]			Tree Decomposition [5]			BFS
	IT	IS	QT	LN	IT	IS	QT	IT	IS	QT	
Epinions	4.0 s	55 MB	0.6 μs	40.1	49.4 s	14.2 MB	0.4 ms	333.5 s	75.0 MB	8.5 μs	9.8 ms
Enron	1.9 s	29 MB	0.4 μs	15.0	62.5 s	306.5 MB	0.1 ms	43.2 s	23.1 MB	8.1 μs	7.8 ms
P2P	334.6 s	3.3 GB	6.4 μs	769.9	1997.8 s	230.9 MB	1.0 ms	DNF	-	-	140.3 ms
YouTube	163.5 s	2.1 GB	1.2 μs	104.8	1217.5 s	230.6 MB	16.6 ms	DNF	-	-	338.4 ms
Wikipedia	774.3 s	2.7 GB	3.0 μs	317.6	14010.9 s	727.4 MB	8.9 ms	DNF	-	-	551.5 ms
DMS	204.5 s	1.1 GB	1.9 μs	218.4	3912.1 s	571.3 MB	1.8 ms	DNF	-	-	176.8 ms
ForestFire	158.7 s	1.6 GB	2.9 μs	345.7	282.7 s	131.8 MB	26.9 ms	DNF	-	-	159.7 ms

Table 1: Datasets

Dataset	$\|V\|$	$\|E\|$	Snapshots	Avg. deg	Max. deg
Epinions	132 K	831 K	421 K	12.8	3.6 K
Enron	87 K	1.1 M	574 K	26.3	38.8 K
P2P	1.1 M	6.3 M	3.1 M	11.9	4.3 K
YouTube	3.2 M	9.4 M	4.7 M	5.8	91.8 K
Wikipedia	1.9 M	36.5 M	18.3 M	38.6	226.1 K
DMS	1.0 M	10.5 M	5.2 M	20.0	63.3 K
ForestFire	1.0 M	7.6 M	3.8 M	14.6	26.8 K

THEOREM 5.1. *For any pair of vertices s, t and $i, \tau \geq 0$,* $\text{QUERYSNAPSHOT}(s, t, \tau, L_i) = \text{QUERYSNAPSHOT}(s, t, \tau, L'_i)$.

COROLLARY 5.1. *For any pair of vertices s and t and $\tau \geq 0$,* $\text{QUERYSNAPSHOT}(s, t, \tau, L'_n) = d_\tau(s, t)$.

COROLLARY 5.2. *For any pair of vertices s and t,* $(\tau, \delta) \in \text{QUERYCHANGEPOINTS}(s, t, L'_n)$ *if and only if* $d_{\tau_i - 1}(s, t) \neq d_{\tau_i}(s, t) = \delta$.

By the same discussion as Section 4.2.2 the time complexity is estimated as $O(nl(d+l))$ time. As with the method for contemporary queries, to exploit central vertices, we order vertices from those with higher degree in the final snapshot.

Note on weighted graphs: For handling weighted graphs, the algorithm can be applied by simply using a priority queue instead of a normal queue. We push triple (v, δ, τ) to the priority queue if v is reachable by distance δ at time τ. We pop the triple with the smallest distance to compute labels and traverse edges.

5.3 Online Incremental Update Algorithm

Incrementally updating the index to reflect graph changes can be done in the almost same way as Section 4.3. For a newly added vertex, we just prepare a new empty label, and for a newly added edge, we resume pruned BFSs from the endpoints. The time complexity is also the same.

6. EXPERIMENTAL EVALUATION

6.1 Setup

We conducted experiments on a Linux server with Intel Xeon X5670 and 48GB of main memory. The proposed methods were implemented in C++. Only indexing was parallelized to use the six cores, and all the other timing results are sequential. We use 32 bits for each pair in an index for contemporary queries (8 bits for distance and 24 bits for vertex IDs) and 64 bits for each triple in that for historical queries (additional 32 bits represent time).

To show the efficiency and robustness of our method, we conducted experiments on large-scale real-world and synthetic dynamic networks. For some of the datasets with coarse-grind timestamps, we randomly determined the order of insertions of edges with the same timestamps. Moreover, to avoid startup effects, we assume the first half of the edges exist from the beginning. We treated all the graphs as undirected graphs. Statistics of the datasets are given in Table 1. The detailed description of each dataset is as follows.

Real time-evolving networks: `Epinions` [20] is the online social network in Epinions (`www.epinions.com`); `Enron` [18] is an e-mail network among employees of *Enron* between 1999 and 2003; `P2P` [2] is a graph constructed from a log of an *eDonkey* server where vertices describe users and a link between two users appears when one provided a file to the other; `YouTube` [21] is the online social network among users of YouTube (`www.youtube.com`) crawled daily in 2007; `Wikipedia` [21] is a web graph between English Wikipedia pages (`en.wikipedia.org`) constructed from edit history.

Synthetic time-evolving networks: `DMS` is a synthetic graph constructed under the Dorogovtsev-Mendes-Samukhin model [14], which is a simple growth model based on preferential attachment that exhibits power-law degree distribution; `ForestFire` is a graph generated by the forest fire model [19], which exhibits not only standard static properties but also common properties on dynamic networks like densification power laws and shrinking diameter. For the `DMS` network, we set the power-law exponent as around 2.3. We generated the `ForestFire` network by Stanford Network Analysis Platform with the default parameters.

In both experiments for contemporary queries and historical queries, we conducted the experiments as follows. *(1)* We first construct an index from a graph with all the edges except last 10,000 edges by the offline indexing algorithm. *(2)* Then, we measure average update time by inserting the last 10,000 edges. *(3)* Finally, we measure the average query time with 1,000,000 random queries after reflecting all the dynamic updates. As a baseline, the average time of BFSs for 1,000 random pairs is also reported. Justification of this configuration is discussed in Section 9.

6.2 Contemporary Queries

In this subsection, we evaluate our first method for contemporary queries (Table 2). As there are no previous dynamic exact methods for contemporary queries, we compare the proposed method with two state-of-the-art static exact methods: a tree-decomposition-based method [5] and IS-Label [15] (introduced in Section 2).

Table 4: Experimental results of our method for historical queries against real-world and synthetic networks.

Dataset	Historical Pruned Landmark Labeling							BFS	
	Indexing time	Index size	Snapshot query time	Change-point query time	Update time	Label size	Label increase	Snapshot query time	Change-point query time
Epinions	23.1 s	236 MB	2.7 μs	4.6 μs	0.3 ms	234.2	2.6×10^{-4}	6.2 ms	2593 s
Enron	5.4 s	86 MB	1.6 μs	3.0 μs	0.2 ms	128.5	4.0×10^{-4}	5.3 ms	3041 s
P2P	2596.4 s	9.7 GB	12.4 μs	22.0 μs	10.7 ms	1227.9	2.7×10^{-4}	79.5 ms	> 1day
YouTube	1281.8 s	9.1 GB	4.5 μs	8.1 μs	2.1 ms	374.7	4.0×10^{-5}	177.5 ms	> 1day
Wikipedia	5165.9 s	13.0 GB	9.8 μs	12.7 μs	8.3 ms	919.5	3.0×10^{-5}	413.3 ms	> 1day
DMS	920.0 s	3.8 GB	4.0 μs	5.6 μs	1.0 ms	481.1	8.6×10^{-5}	126.4 ms	> 1day
Forestfire	1056.3 s	6.5 GB	8.2 μs	12.8 μs	3.9 ms	835.9	3.6×10^{-4}	91.4 ms	> 1day

Table 3: Experimental results of our online update algorithm for contemporary queries.

Dataset	Update time	Label increase	Visited vertices
Epinions	0.2 ms	1.0×10^{-4}	5.7
Enron	0.5 ms	3.2×10^{-4}	4.6
P2P	4.0 ms	1.5×10^{-4}	2.9
YouTube	4.3 ms	1.6×10^{-4}	11.4
Wikipedia	6.1 ms	4.0×10^{-5}	5.4
DMS	1.9 ms	1.8×10^{-4}	8.2
ForestFire	2.6 ms	3.7×10^{-4}	9.8

6.2.1 Indexing Time, Index Size and Label Size

We first confirm the high scalability of our method from the results on indexing time, index size and average label size (Table 2). We set the number of times we conduct bit-parallel BFSs as 16. The results indicate the competitive scalability of the proposed method in comparison even with static state-of-the-art methods.

6.2.2 Query Time

Then, we study the average query time reported in Table 2. It shows that the query time of our method is generally microseconds. It is several orders of magnitude faster than BFSs and competitive with other static methods, indicating applicability to query-intensive applications such as network-sensitive search. In particular, the query time of the proposed method is thousands times faster than IS-Label. This is because the query algorithm visits only two consecutive regions (i.e., two labels) in the main memory for each query.

6.2.3 Update Time and Label Increase

We observe that the average update time is generally milliseconds (Table 3), which is four or five orders of magnitude faster than full index reconstruction and sufficiently fast for real-time processing. Average label increase in Table 3 describes the average difference of the average label size before and after inserting an edge. As the average label increase is also small, we confirm that the label size grows slowly. Table 3 also lists the average number of vertices visited during each BFS for dynamic index updates. More precisely, we define that a vertex is *visited* if it is inserted in the queue. It shows the numbers of visited vertices are very small, that is, the dynamic update is certainly done quite locally.

6.3 Historical Queries

Second, we present experimental results on our second method for historical queries (Table 4). This time, since there are no previous methods for historical distance queries, the proposed method is not compared with other methods.

6.3.1 Indexing Time, Index Size and Label Size

Indexing time, index size and average label size are shown in Table 4. Although indexing time and index size are a little larger than those for contemporary queries, they are still acceptable even for large dynamic networks. For example, it took only two hours for constructing an index from the Wikipedia dataset. Also note that, since our method can incrementally update an index, we do not need to reconstruct an index frequently. The index size was 13 GB, which adequately fits in the main memory of commodity computers of the day.

6.3.2 Query Time

Average query time is also reported in Table 4. As a baseline for change-point queries, we suppose a naive method that conducts a BFS for each snapshot. Since this baseline method takes too long query time, we estimated the average query time by the product of the average snapshot-query time and the number of snapshots. For both snapshot queries and change-point queries, the query time of our method for historical queries is also generally microseconds and orders of magnitude faster than the baselines.

6.3.3 Update Time and Label Increase

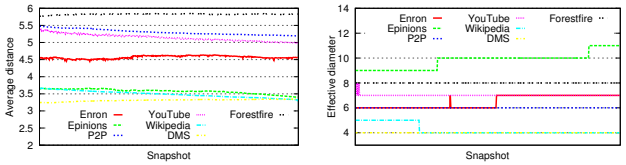
Average update time is also listed in Table 4. As with the method for contemporary queries, this method for historical queries also handles each update in milliseconds. Moreover, as average label increase in Table 4 is small, we can confirm that the label size grows slowly.

7. EXTENSIONS

Shortest-path queries: To efficiently answer not only distance but also the shortest path, we store parents of label entries in BFS trees, and ascend the trees from the two endpoints to restore the path. If the index size is more critical than the query time, then another way to answer the shortest path is to repeatedly move to a vertex that has smaller distance from one endpoint to the other endpoint by using distance queries.

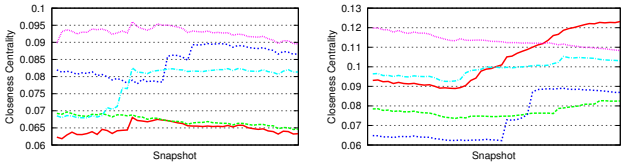
Directed graphs: If the input graph is a directed graph, we store and maintain two labels $L_{\mathrm{IN}}(v)$ and $L_{\mathrm{OUT}}(v)$ for each vertex v, where $L_{\mathrm{IN}}(v)$ contains distances from v and $L_{\mathrm{OUT}}(v)$ contains distances to v. To construct or update an index, we conduct pruned BFSs twice from each vertex in the forward and reverse directions.

Weighted graphs: To handle weighted graphs, for both indexing and updating, we conduct or resume pruned Dijkstra's algorithm instead of pruned BFSs. As for edge weight update, our methods can only handle weight decrease.



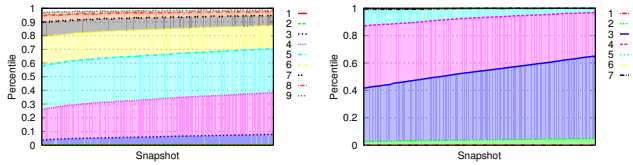
(a) Average distance (b) Effective diameter

Figure 3: Transition of average distance and effective diameter.



(a) Enron (b) Epinions

Figure 4: Transition of the closeness centrality of some popular vertices.



(a) YouTube (b) Wikipedia

Figure 5: Temporal hop plot.

8. APPLICATION TO EVOLVING NETWORK ANALYSIS

In this section, we demonstrate the usefulness of our historical indexing method for evolving network analysis. The proposed method enables quick and fine-grind temporal analysis of large-scale dynamic networks. For example, with our index, users can instantly and interactively check the transition of various features related to distances, which has never been available at all without our index.

As shown by the case study in Section 1.2, transitions of distance and shortest-path themselves are useful and of interest. Furthermore, based on our method, we can also efficiently compute the transition of the following kinds of network features.

Average distance and effective diameter: Distance distribution is one of the most important features of networks, and the transition of distance distribution of dynamic networks is of strong interest to the data mining and social network analysis community [19]. Though calculating distance distribution of one graph is already too costly, to obtain the transition of it, we need to do so for many snapshots of graphs, which would be impossible for large historical networks. Using our historical indexing method, however, we can estimate the transition just by evaluating random change-point queries with regard to a set of randomly sampled pairs of vertices. To demonstrate the effectiveness of our method, we computed the transition of the average distance between pairs and the *effective diameter* (the 90th percentile distance) of various networks (Figure 3). We can observe that average distance decreases over time, which confirms the claim of [19], but the effective diameter sometimes increases.

Closeness centrality: *Closeness centrality* is one of the most popular network centralities defined on vertices. There are several different definitions, but all of them are based on distances to other vertices, and thus they can be efficiently estimated by random change-point queries. Here, we adopt the definition that defines the closeness centrality of vertex v as $\frac{1}{|V|}\sum_{u\in V} 2^{-d(v,u)}$. We picked up several vertices of high closeness centralities from `Enron` and `Epinions` and computed the transition of their closeness centralities by the proposed method (Figure 4). We can see that the closeness centrality sometimes drastically increases as some moment.

Hop Plot: To study distance distribution in depth, the *(temporal) hop plot*, which is the transition of the fraction of pairs within a fixed distance, is also used. We can approximate the hop plot by evaluating change-point queries with regard to a set of randomly sampled pairs of vertices. In Figure 5, we illustrate the hop plot of `YouTube` and `Wikipedia` for various distances. We observe that it tends to increase over time, as expected from the fact that average distance decreases over time.

9. DISCUSSION ON PRACTICABILITY FOR NETWORK-AWARE SERVICES

In this section, we argue that supporting no removals does not limit applicability of the proposed methods, and rather, it is a *more promising approach* than seeking for fully dynamic methods.

First of all, we emphasize again that all the existing methods are static. Thus, we were only able to handle edge updates by reconstructing the index from scratch. Since it can be only done periodically (say, once per day), we have to wait for a long time to reflect edge updates. With our index, however, we can instantly reflect edge additions.

Having said that, when handling edge updates, it is inevitable to fully reconstruct the index periodically since we often want to exploit the global information of the current graph such as centrality or clustering. Thus, a practical way to handle edge updates is a combination of periodic index reconstruction and incremental index update, and our method enables this combination for the first time. We note that edge removals are reflected when reconstructing the index while edge additions are reflected immediately.

We also mention that fully dynamic methods that supporting edge removals would be too complicated and impractical at all. As a realistic compromise, we have focused only on edge additions and have established a method that has a competitive performance.

10. ACKNOWLEDGEMENTS

This work is supported by Grant-in-Aid for JSPS Fellows (256563 and 256487), JSPS Grant-in-Aid for Research Activity Start-up (24800082), MEXT Grant-in-Aid for Scientific Research on Innovative Areas (24106003), and JST, ERATO, Kawarabayashi Large Graph Project.

11. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35. 2012.

[2] F. Aidouni, M. Latapy, and C. Magnien. Ten weeks in the life of an edonkey server. In *IPDPS*, pages 1–5, 2009.

[3] T. Akiba, Y. Iwata, K. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *ALENEX*, pages 147–154, 2014.

[4] 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.

[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] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, pages 44–54, 2006.

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

[8] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[9] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D. Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4-5):175–308, 2006.

[10] D. S. Callaway, M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Network robustness and fragility: Percolation on random graphs. *Physical Review Letters*, 85:5468–5471, 2000.

[11] J. Cheng and J. X. Yu. On-line exact shortest distance query processing. In *EDBT*, pages 481–492, 2009.

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

[13] A. Das Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A sketch-based distance oracle for web-scale graphs. In *WSDM*, 2010.

[14] S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin. Structure of growing networks with preferential linking. *Phys. Rev. Lett.*, 85:4633–4636, 2000.

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

[16] R. Jin, N. Ruan, Y. Xiang, and V. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *SIGMOD*, pages 445–456, 2012.

[17] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, 2003.

[18] B. Klimt and Y. Yang. The enron corpus: A new dataset for email classification research. In *ECML*, volume 3201 of *LNCS*, pages 217–226. 2004.

[19] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1), 2007.

[20] P. Massa and P. Avesani. Controversial users demand local trust metrics: an experimental study on epinions.com community. In *AAAI*, pages 121–126, 2005.

[21] A. Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, 2009.

[22] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E*, 64(2):026118 1–17, 2001.

[23] R. Pastor-Satorras and A. Vespignani. *Evolution and structure of the Internet: A statistical physics approach*. Cambridge University Press, 2004.

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

[25] 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.

[26] S. A. Rahman, P. Advani, R. Schunk, R. Schrader, and D. Schomburg. Metabolic pathway analysis web service (pathway hunter tool at cubic). *Bioinformatics*, 21(7):1189–1193, 2005.

[27] S. A. Rahman and D. Schomburg. Observing local and global properties of metabolic pathways: 'load points' and 'choke points' in the metabolic networks. *Bioinformatics*, 22(14):1767–1774, 2006.

[28] N. Robertson and P. D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[29] L. Tang and M. Crovella. Virtual landmarks for the internet. In *SIGCOMM*, pages 143–152, 2003.

[30] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *CIKM*, pages 1785–1794, 2011.

[31] A. Ukkonen, C. Castillo, D. Donato, and A. Gionis. Searching the wikipedia with contextual information. In *CIKM*, pages 1351–1352, 2008.

[32] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. d. C. Reis, and B. Ribeiro-Neto. Efficient search ranking in social networks. In *CIKM*, pages 563–572, 2007.

[33] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *WOSN*, pages 37–42, 2009.

[34] F. Wei. Tedi: efficient shortest path query answering on graphs. In *SIGMOD*, pages 99–110, 2010.

[35] 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.

[36] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, pages 1601–1606, 2013.