World Wide Web manuscript No.
(will be inserted by the editor)

Efficient Computation of Distance Labeling for Decremental Updates in Large Dynamic Graphs

Yongrui Qin · Quan Z. Sheng · Nickolas J.G. Falkner · Lina Yao · Simon Parkinson

Received: date / Accepted: date

Abstract Since today's real-world graphs, such as social network graphs, are evolving all the time, it is of great importance to perform graph computations and analysis in these dynamic graphs. Due to the fact that many applications such as social network link analysis with the existence of inactive users need to handle failed links or nodes, decremental computation and maintenance for graphs is considered a challenging problem. Shortest path computation is one of the most fundamental operations for managing and analyzing large graphs. A number of indexing methods have been proposed to answer distance queries in static graphs. Unfortunately, there is little work on answering such queries for dynamic graphs. In this paper, we focus on the problem of computing the shortest path distance in dynamic graphs, particularly on decremental updates (i.e., edge deletions). We propose maintenance algorithms based on distance labeling, which can handle decremental updates efficiently. By exploiting properties of distance labeling in original graphs, we are able to efficiently maintain distance labeling for new graphs. We experimentally evaluate our algorithms using eleven real-world large graphs and confirm the effectiveness and efficiency of our approach. More specifically, our method can speed up index re-computation by up to an order of magnitude compared with the state-of-the-art method, Pruned Landmark Labeling (PLL).

Keywords Shortest Path · Graph Computation · Distance Labeling · Dynamic Graph

Yongrui Qin and Simon Parkinson are with School of Computing and Engineering, University of Huddersfield, UK
Quan Z. Sheng and Nickolas J.G. Falkner are with School of Computer Science, The University of Adelaide, Australia
Lina Yao is with School of Computer Science and Engineering, The University of New South Wales, Australia
Corresponding Author: Yongrui Qin, E-mail: y.qin2@hud.ac.uk

1 Introduction

Recent years have witnessed the fast emergence of massive graph data in many application domains, such as the World Wide Web, linked data technology, online social networks, and Web of Things. In a graph, one of the most fundamental problems is the computation of the shortest path or distance between any given pair of vertices. For instance, distances or the numbers of links between web pages in a large web graph can be considered a robust measure of web page relevancy, especially in relevance feedback analysis in web search [21]. In RDF graphs of linked data, the shortest path distance from one entity to another is important for ranking entity relationships and keyword querying [18,15]. For online social networks, the shortest path distance can be used to measure the closeness centrality between users [22,23].

A large body of indexing techniques have been recently proposed to process exact shortest path distance queries in graphs [9,24,8,7,2,26,16]. Among them, a significant portion of indexes are based on *2-hop* distance labeling, which is originally proposed by Cohen et al. [11]. The 2-hop distance labeling pre-computes a label for each vertex so that the shortest path distance between any two vertices can be computed by giving only their labels. These labeling indexes, such as [9,7,2,16], prove to be efficient when processing large graphs with edge numbers up to hundreds of millions.

Motivation. The above mentioned approaches generally make the assumption that graphs are static. However, in reality, many graphs are subject to constant changes. For example, it is reported that in the fourth quarter of 2012, Facebook reached 1.056 billion users amounted to a 24.97% increase from the same period in 2011 [14]. Around April 2013, DBpedia, one of the most popular RDF graphs, released its version 3.9. In this new release, an overall increase in the number of concepts in the English edition changed from 3.7 million to 4.0 million things compared with its last release in June 2012[1]. Similarly, the emerging social Web of Things also supports the need for dynamic graph data management because smart things are normally moving and their connectivity could be intermittent, leading to frequent and unpredictable changes in the corresponding graph models [10,25].

We believe that it is imperative to design novel algorithms that can update shortest path indexes efficiently for large dynamic graphs. Existing shortest path indexing techniques based on 2-hop labeling may take up to hundreds of seconds to pre-compute the whole shortest path index for a graph with millions of edges. For larger graphs, it can take up to thousands of seconds [2,16]. Applying indexing techniques designed for static graphs directly to dynamic graphs may lead to inefficiency. This is because that if only a small part of the graph is changed, i.e., only a deletion of an existing edge occurs, a significant proportion of the shortest paths are likely to remain unchanged and the index for the original graph may contain a large amount of correct distance

[1] `http://wiki.dbpedia.org/`

information. In such case, simply recomputing the 2-hop distance index from scratch would unnecessarily waste computing resources.

An alternative is to maintain dynamic all-pairs shortest paths (APSP). Many approaches have been proposed to maintain dynamic APSP data structures. For example, in [12,13], a dynamic algorithm for general directed graphs with non-negative edge weights was proposed with a computational complexity of $O(n^2 \log^3 n)$, where n is the number of vertices. However, this time bound is comparable to recomputing all-pairs shortest paths from scratch, which makes the algorithm inefficient for handling changes in graphs. Recently, an algorithm for maintaining dynamic all-pairs $(1+\epsilon)$ approximate shortest paths for directed graphs with polynomial weights is proposed in [5]. The total update complexity is $\tilde{O}(mn/\epsilon)$, where n is the number of vertices and m is the number of edges. Unfortunately it only applies to dynamic approximate shortest path problems.

Incremental updates (i.e., edge insertions) of 2-hop labeling in large dynamic graphs have been recently investigated in [3]. However, the problem of supporting decremental updates (i.e., edge deletions) of 2-hop labeling still remains unsolved and is considered a challenging problem [3]. Decremental updates are very useful in the presence of many real-world problems such as outdated web links in a web graph or obsolete user profiles in a social network. Clearly, decremental maintenance is a fundamental and important operation on graph data to support efficient web link analysis and social network analysis.

Contributions. To address the deficiency of existing shortest path indexing techniques, this paper proposes a generic framework to update shortest path indexes efficiently for dynamic graphs where edge deletions are allowed. As an initial attempt on this challenging issue, we focus on unweighted, undirected graphs. Similar to other distance labeling based indexing methods [2,16], our method can be extended to weighted and/or directed graphs. We highlight our main contributions in the following:

- We present the concept of well-ordering 2-hop distance labeling and identify its important properties that can be utilized to design update algorithms for shortest path indexes in dynamic graphs.
- We analyze cases of shortest path index maintenance in dynamic graphs with decremental updates. We develop the corresponding theorems as well as novel algorithms to enable efficient updates without reconstruction of distance labeling for the entire graph.
- We conduct extensive experiments on eleven real-world large graphs to verify the efficiency and effectiveness of our method. Compared with the state-of-the-art technique [2] which is designed for static graphs, our method is on average an order of magnitude faster.

The rest of this paper is organized as follows. In Section 2, we review the related work. In Section 3, we present some preliminaries on 2-hop distance labeling. We then present the framework and the details of our approach in

Section 4. In Section 5, we report the results of an extensive experimental study using eleven large graphs from real-world. Finally, we present some concluding remarks in Section 6.

2 Related Work

In this section, we review the major techniques that are most closely related to our work.

Distance labeling has been an active research area in recent years. In [9], Cheng and Yu exploit the strongly connected components property and graph partitioning to pre-compute 2-hop distance cover. However, the graph partitioning process introduces high cost because it has to find vertex separators recursively. Hierarchical hub labeling (HHL) proposed by Abraham et al. [1] is based on the partial order of vertices. Smaller labeling results can be obtained by computing labeling for different partial order of vertices. In [17], Jin et al. propose a highway-centric labeling (HCL) that uses a spanning tree as a highway and based on the highway, a 2-hop labeling is generated for fast distance computation.

Very recently, the Pruned Landmark Labeling (PLL) [2] is proposed by Akiba et al. to pre-compute 2-hop distance labels for vertices by performing a breadth-first search from every vertex. The key is to prune vertices that have obtained correct distance information during breadth-first searches, which helps reduce the search space and sizes of labels. Further, query performance is also improved as the number of label entries per vertex is reduced. IS-Label (or ISL) is developed by Fu et al. in [16] to pre-compute 2-hop distance label for large graphs in memory constrained environments. ISL is based on the idea of independent set of vertices in a large graph. By recursively removing an independent set of vertices from the original graph, and by augmenting edges that preserve distance information after the removal of vertices in the independent set, the remaining graph keeps the distance information for all remaining vertices in the graph. Apart from the 2-hop distance labeling technique, a multi-hop distance labeling approach [7] is also studied, which can reduce the overall size of labels at the cost of increased distance querying time.

Tree decomposition approaches have been recently investigated [24, 4] for answering distance queries in graphs. Wei proposes TEDI [24], which first decomposes a graph into a tree and forms a tree decomposition. A tree decomposition of a graph is a tree with each vertex associated with a set of vertices in the graph, which is also called a *bag*. The shortest paths among vertices in the same bag are pre-computed and stored in bags. For any given source and target vertices, a bottom-up operation along the tree can be executed to find the shortest path. An improved TEDI index is further proposed by Akiba et al. in [4] that exploits a core-fringe structure to improve index performance. However, due to the large size of some bags in the decomposed tree, the construction time for a large graph is costly and thus such indexing approaches cannot scale well.

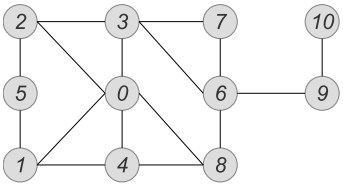


Fig. 1 A graph example

The above studies focus on point to point shortest path distance processing. Some studies also investigate other types of distance queries, such as single-source shortest path (SSSP) distance queries. For example, Cheng et al. propose VC-index [8], a disk-based method for processing SSSP distance queries based on the concept of vertex cover. Highways-on-Disk (HoD) is also proposed in [26] to support SSSP distance queries for directed graphs. HoD reduces I/O and computation costs for query processing by augmenting a set of auxiliary edges or shortcuts in the original graphs.

HOPI (2-HOP-cover-based Index) developed by Schenkel et al. in [20] is designed to speed up connection or reachability tests in XML documents based on the idea of 2-hop cover. HOPI introduces index maintenance for both insertions and deletions of nodes, edges or even XML documents. To the best of our knowledge, HOPI is the first work on maintenance of 2-hop labeling. Recently, maintenance of 2-hop labeling for large graphs has also been studied by Bramandia et al. in [6]. However, all these studies focus on reachability queries and are based on 2-hop labeling but not on 2-hop distance labeling. Hence they cannot be applied to dynamic 2-hop distance labeling.

The most related work is proposed very recently by Akiba et al. in [3]. In that work, incremental updates (i.e., edge insertions) of 2-hop labeling indexes are investigated. To support fast incremental updates, outdated distance labels are kept, which will not affect the distance computation in the updated graphs in the incremental case. However, for the decremental case (i.e., edge deletions), this approach will not work, as outdated distance labels must be removed first and then some necessary labels of the 2-hop labeling index need to be recomputed. Another piece of research work [19] adopts a similar strategy and proposes a supplementary index structure called SIEF to support single-failure on edges. The SIEF index only updates the original distance labeling index with necessary new distance information for each single-edge failure case. To be specific, the outdated distance labels in the original index are kept but additional labels are constructed to provide updated distance information for a specific edge failure. However, the SIEF approach cannot handle multi-edge failures or a sequence of edge failures in the same graph. The support of decremental maintenance on multi-edge failures is the focus of our work in this paper.

3 Preliminaries

3.1 2-Hop Distance Labeling

The technique of 2-hop cover can be used to solve reachability problems (using reachability labels) and shortest path distance querying problems (using distance labels) in graphs. Since our work focuses on the shortest path distance querying problems, we adopt distance labels with the 2-hop cover technique. We specifically refer to it as *2-hop distance labeling* or *2-hop distance cover*.

Assume a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges. For each vertex $v \in V$, there is a pre-computed label $L(v)$ which is a set of vertex and distance pairs (u, δ_{uv}). Here u is a vertex and δ_{uv} is the shortest path distance between u and v. Given such a labeling for all vertices in G, denoted by L, for any pair of vertices s and t in G, we can have

$$\begin{aligned} dist(s,t,L) = & min\{\delta_{vs} + \delta_{vt} | (v, \delta_{vs}) \in L(s) \\ & \text{and } (v, \delta_{vt}) \in L(t)\} \end{aligned} \tag{1}$$

If $L(s)$ and $L(t)$ do not share any vertices, we have $dist(s,t,L)=\infty$. The distance between any given vertices s and t in G is denoted by $d_G(s,t)$. If we have $d_G(s,t) = dist(s,t,L)$ for all s and t in G, we call the labeling result L a 2-hop distance cover.

3.2 Well-Ordering 2-Hop Distance Labeling

Definition 1 Well-Ordering 2-Hop Distance Labeling[2] For a connected graph G, there exists a sequence of vertices $\sigma =< v_0, v_1, v_2, \ldots, v_{n-1} >$. We denote the order of any vertex v_i as $\sigma[v_i]$ and we have $\sigma[v_i] = i$ for the above given vertex sequence. Each vertex v_i has a distance labeling $L(v_i)$, and the labeling result L of all vertices forms a 2-hop distance cover of G. For any pair of vertices v_i and v_j, given that $\sigma[v_i] < \sigma[v_j]$, then v_j is not in $L(v_i)$ and v_i may be in $L(v_j)$. We call such a 2-hop distance cover a *well-ordering 2-hop distance labeling*. Alternatively we say that a 2-hop distance cover has well-ordering property. □

Similar concepts of well-ordering 2-hop distance labeling also appear in recent research efforts such as PLL [2], and ISL [16]. This confirms that well-ordering 2-hop distance labeling is important in the related research area. More importantly, we will show in this paper that the well-ordering property is also a basic concept in the design of update algorithms for distance labeling computation in dynamic graphs.

In a graph containing multiple connected components, suppose its 2-hop labeling is L. For any pair of vertices u and v in different connected components, we can assert that $L(u)$ and $L(v)$ do not share any vertex according to

[2] Note that, this concept is a special case of hierarchical labeling proposed in [1].

Table 1 2-Hop Distance Labeling L

Label	Entries
$L(0)$	(0,0)
$L(1)$	(0,1) (1,0)
$L(2)$	(0,1) (2,0)
$L(3)$	(0,1) (2,1) (3,0)
$L(4)$	(0,1) (1,1) (4,0)
$L(5)$	(0,2) (1,1) (2,1) (5,0)
$L(6)$	(0,2) (2,2) (3,1) (4,2) (6,0)
$L(7)$	(0,2) (2,2) (3,1) (6,1) (7,0)
$L(8)$	(0,1) (4,1) (6,1) (8,0)
$L(9)$	(0,3) (2,3) (3,2) (4,3) (6,1) (9,0)
$L(10)$	(0,4) (2,4) (3,3) (4,4) (6,2) (9,1) (10,0)

the definition of 2-hop cover. Each connected component has its own vertex orders. For such a graph, we will have separate vertex orders for each connected component. We denote a connected component containing vertex u as $C(u)$. If u and v belong to the same connected component, we have $C(u) = C(v)$.

Figure 1 shows an example graph with 11 vertices and Table 1 shows a well-ordering 2-hop distance labeling result L for the graph (L can be constructed by PLL [2] using the same vertex ordering as that specified in the table). In the table, the order of vertices is $< 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 >$. Take $L(5)$ as an example to further explain the idea of well-ordering 2-hop distance labeling. $L(5)$ is the label of vertex 5. By the well-ordering property, label entries in $L(5)$ can only contain vertices $0, 1, 2, 3, 4$ and 5. Since label entries containing vertices 3 and 4 are redundant in $L(5)$ (this will be explained in more details later in this section), label entries in $L(5)$ only contain vertices $0, 1, 2$ and 5.

3.3 Properties of Well-Ordering 2-Hop Distance Labeling

Technically speaking, if we index all-pairs shortest paths using a labeling method, we will obtain an index that occupies $O(n^2)$ disk space. This index can be considered as a special 2-hop distance cover labeling. Obviously, the space complexity of this is too high for large graphs. Constructing a minimal 2-hop distance cover labeling has been proven to be NP-hard [11]. Therefore, an alternative way to obtain labeling results with reduced sizes is by using heuristic methods [9,7,2,16]. Well-ordering 2-hop distance labeling is one of the techniques that can help to design efficient algorithms for constructing shortest path distance labeling indexes and for index maintenance. We list its useful properties in the following. For more details, please refer to [1].

Lemma 1 *(Abraham et al. [1]) Given a well-ordering 2-hop distance labeling L of a connected graph G, suppose $s, t, u \in G$ and u has minimum vertex order $\sigma[u]$ among all shortest paths between s and t. Then we must have $(u, \delta_{us}) \in L(s)$ and $(u, \delta_{ut}) \in L(t)$ and $dist(s, t, L) = \delta_{us} + \delta_{ut}$.*

Take vertices 1 and 6 in Figure 1 as an example. Paths $p_1 =< 1, 0, 8, 6 >$, $p_2 =< 1, 0, 3, 6 >$ and $p_3 =< 1, 4, 8, 6 >$ are all the shortest paths between vertices 1 and 6. Vertex 0 is the one with minimum order along all these paths. From Table 1 we can see that both vertices 1 and 6 contain a label entry $(0, \delta)$. We can also easily check that $dist(1, 6, L) = \delta_{0,1} + \delta_{0,6} = 1 + 2 = 3$.

Lemma 2 *(Abraham et al. [1]) Given a well-ordering 2-hop distance labeling L of a connected graph G, suppose $\sigma[u] < \sigma[v]$. If there is a label entry $(u, \delta_{uv}) \in L(v)$, we must have for any label entry $(r, \delta_{rv}) \in L(v)$, (1) $\delta_{uv} \leq \delta_{rv} + dist(r, u, L)$; (2) if $\sigma[r] < \sigma[u]$ and $\delta_{uv} = \delta_{rv} + dist(r, u, L)$ then $(u, \delta_{uv}) \in L(v)$ is a redundant label entry.*

Take label entries of vertex 5 in Table 1 as an example. We have $\sigma(3) < \sigma(5)$ and $\sigma(2) < \sigma(3)$. We also have $\delta_{3,5} = 2 = \delta_{2,5} + \delta_{2,3}$. Therefore $(3, 2)$ is a redundant label entry in $L(5)$, which can be removed from $L(5)$.

3.4 Brief Introduction of PLL

The Pruned Landmark Labeling (PLL) [2] performs Breadth-First-Search (BFS) at each vertex in a given graph to generate 2-hop labels. The main idea is that when labeling a vertex v with a BFS rooted at u, if the current labeling information, i.e, the sum of distances d_{rv} and d_{ru} that were calculated during a previous BFS rooted at r, has already provided the correct distance between v and u, then v can be pruned in the current BFS. Take Figure 2 as an example. Suppose vertex order of vertices r, u, v is $\sigma(r) < \sigma(u) < \sigma(v)$. We run BFS rooted at r first and obtain distances d_1 and d_2. At a later stage, we run BFS rooted at u (Note that at the beginning of this BFS, r has been pruned since a BFS rooted at r has been completed) and if $d_3 >= d_1 + d_2$, we prune v from the rest of the current BFS process. As more BFSs proceed, pruning can be seen more and more frequently, which contributes to the high efficiency of labeling using the PLL method.

PLL can also be used together with the bit-parallel technique [2] for further accelerating the indexing process. When bit-parallel is applied in PLL, t times of full BFSs will be performed and all distance information on each vertex will be kept in the labeling process for the first t runs of BFSs. This means that no pruning will happen for the first t roots. To exploit parallel computing during the labeling for these first t roots of BFSs, the bit-parallel technique will be able to label up to a fixed number of neighbors (e.g., up to 32 or 64 neighbors) in a batch mode when processing one vertex. For more details about this bit-parallel technique, please refer to [2].

4 Distance Labeling Maintenance in Dynamic Graphs

In this section, we first provide an overview of our update method. We then analyze different cases of the 2-hop distance labeling maintenance in dynamic

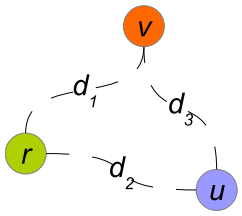


Fig. 2 The Pruned Landmark Labeling (PLL) concept demonstration

graphs and introduce a set of algorithms to handle index updates. Proofs of correctness of our algorithms are also presented.

4.1 Approach Overview

Our approach on decremental updates can be divided into two main stages. In the first stage, called *IDENTIFY*, we identify *possible* and *real* affected vertices, invalid label entries, and missing label entries. In the second stage, called *UPDATE*, we remove all invalid label entries from real affected vertices and relabel all real affected vertices based on the collected information of the invalid label entries and missing label entries.

Before the detailed discussions of our algorithms, suppose that the deleted edge is (u, v) in G, and the resulted graph is G', we define the following necessary concepts for the maintenance:

- *Possible affected vertices* $PA(u)$*:* for any vertex s, if $d_G(s, v) = d_G(s, u)+1$, then $s \in PA(u)$.
- *Possible affected vertices* $PA(v)$*:* for any vertex s, if $d_G(s, u) = d_G(s, v)+1$, then $s \in PA(v)$.
- *Real affected vertices* $RA(u)$*:* for any vertex s, if $s \in PA(u)$ and $d_{G'}(s, v) \neq d_G(s, u) + 1$, then $s \in RA(u)$.
- *Real affected vertices* $RA(v)$*:* for any vertex s, if $s \in PA(v)$ and $d_{G'}(s, u) \neq d_G(s, v) + 1$, then $s \in RA(v)$.
- *Invalid labels* $IL(u)$*:* suppose $t \in RA(u)$, for any vertex $s \in RA(v)$, if there is a label entry $(s, \delta) \in L(t)$, then $s \in IL(u)$.
- *Invalid labels* $IL(v)$*:* suppose $t \in RA(v)$, for any vertex $s \in RA(u)$, if there is a label entry $(s, \delta) \in L(t)$, then $s \in IL(v)$.
- *Missing labels* $ML(u)$*:* suppose $t \in RA(u)$, for any vertex $s \in PA(v)$, if (s, δ) is a redundant label entry in $L(t)$ and there is a label entry $(s', \delta') \in L(t)$ where $s' \in IL(u)$ and (s', δ') makes (s, δ) redundant in $L(t)$, then $s \in ML(u)$.
- *Missing labels* $ML(v)$*:* suppose $t \in RA(v)$, for any vertex $s \in PA(u)$, if (s, δ) is a redundant label entry in $L(t)$ and there is a label entry $(s', \delta') \in L(t)$ where $s' \in IL(v)$ and (s', δ') makes (s, δ) redundant in $L(t)$, then $s \in ML(v)$.

Algorithm 1: Identify affected vertices

Input: G, (u, v), distance vectors d_u, d_v, d'_u, d'_v
Output: $PA(u)$, $PA(v)$, $RA(u)$, $RA(v)$
Initialize flag $m[t] \leftarrow 0$ for any vertex t in G
$m[u] \leftarrow 1$
$Q \leftarrow \emptyset$
Enqueue u into Q
while Q is not empty **do**
    Dequeue t from Q
    for all neighbor vertex r of t **do**
        if $m[r] = 0$ **then**
            if $d_v[r] = d_u[r] + 1$ **then**
                Enqueue r into Q
                $PA(u) \leftarrow PA(u) \cup \{r\}$
                if $d'_v[r]! = d_u[r] + 1$ **then**
                    $RA(u) \leftarrow RA(u) \cup \{r\}$
                $m[r] \leftarrow 1$
Repeat the above steps by mapping $u \leftarrow v$ and $v \leftarrow u$ to identify $PA(v)$ and $RA(v)$

We will show later that the induced sub-graph by $PA(u)$ in G can be considered as a tree rooted at u. We will also show that the induced sub-graph by $PA(v)$ in G can be considered as a tree rooted at v. Furthermore, $PA(u) \bigcap PA(v) = \emptyset$.

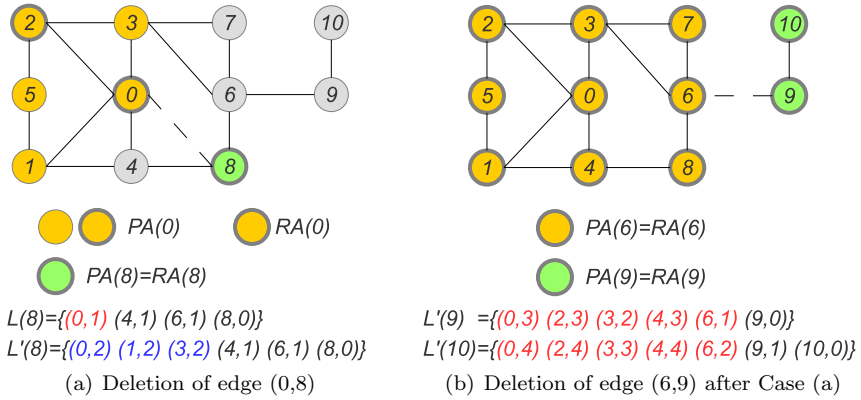
To be specific, $PA(u) \cup PA(v)$ contains all vertices whose distance to some other vertex might have been changed due to the update. In other words, at least one shortest path starting from these vertices contains edge (u, v). $RA(u) \cup RA(v)$ contains all vertices whose distance to some other vertex must have been changed due to the update. In other words, there exists one and only one shortest path from these vertices to some other vertex containing edge (u, v). $IL(u) \cup IL(v)$ contains all vertices that appear in some vertices in $RA(u) \cup RA(v)$ as an outdated label entry due to the update. Finally, $ML(u) \cup ML(v)$ contains all vertices that become a missing label entry in some vertices in $RA(u) \cup RA(v)$ due to the update. Later in this section, we will look into some examples of all these concepts.

4.2 Decremental Maintenance

4.2.1 Algorithms

Consider the deletion of an edge between vertices u and v in graph G, which results in a new graph G'. There are two cases after the deletion, namely Case (1): u and v are still connected in G', and Case (2): u and v becomes disconnected in G'. Figure 3 depicts an example of decremental maintenance process.

For Case (1), at the IDENTIFY stage, we develop Algorithm 1 to identify $PA(u)$, $PA(v)$, $RA(u)$ and $RA(v)$ and Algorithm 2 to identify $IL(u)$, $IL(v)$, $ML(u)$ and $ML(v)$. Note that distance vectors d_u, d_v, d'_u and d'_v can



(a) Deletion of edge (0,8) (b) Deletion of edge (6,9) after Case (a)

Fig. 3 A graph decremental maintenance example (the updated label entries are in blue; outdated label entries are in red)

be obtained using the Breadth-First-Search (BFS) algorithm. At the UPDATE stage, we first remove all invalid label entries for all other vertices in $RA(u)$ and $RA(v)$ using Algorithm 3 and then we use a modified PLL (Pruned Landmark Labeling) technique [2] to relabel affected vertices. The whole process dealing with Case (1) is depicted in Algorithm 4.

Case (a) of Figure 3 shows an example. After deleting edge $(0, 8)$, we can identify according to Algorithm 1 that $PA(0) = \{0, 3, 2, 1, 5\}$ and $PA(8) = \{8\}$[3] since the distances between vertices $0, 3, 2, 1, 5$ and vertex 8 might have been changed due to the edge deletion. We also identify that $RA(0) = \{0, 2\}$ as only these two vertices have changed their distance to vertex 8 and similarly, $RA(8) = \{8\}$.

Based on $RA(0) = \{0, 2\}$ and according to Algorithm 2, we have $IL(8) = \{0\}$ since $(0, 1) \in L(8)$ while $(2, 2) \notin L(8)$ ($(2, 2)$ is a redundant label entry due to the existence of $(0, 1)$ in $L(8)$). Based on $IL(8)$, we can identify that $ML(8) = \{1, 2, 3\}$, as label entries $(1, 2)$, $(2, 2)$ and $(3, 2)$ are redundant in $L(8)$ due to the existence of $(0, 1)$. Now $(0, 1)$ becomes invalid in $L(8)$, and thus distances between vertex 8 and vertices $1, 2, 3$ cannot be computed via the original $L(8)$ anymore. Therefore, according to Steps 7 and 8 in Algorithm 2, we have $ML(8) = \{1, 2, 3\}$. After relabeling, we have the updated label for vertex 8: $L'(8) = \{(0, 2)(1, 2)(3, 2)(4, 1)(6, 1)(8, 0)\}$. Note that, $(2, 3)$ is not in the updated $L'(8)$ as it is a redundant label entry.

It is worth mentioning that we identify $IL(8)$ and $ML(8)$ at vertex 8 only, which is one end of the deleted edge. We will later show that they are exactly $IL(8)$ and $ML(8)$ for any vertex in $RA(8)$ in Lemma 6 and Lemma 7. At the UPDATE stage, we firstly remove all invalid label entries in $RA(0)$ and $RA(8)$ according to $IL(0)$ and $IL(8)$. Then we perform a modified PLL BFS algorithm rooted at each vertex in $IL(u) \cup IL(v) \cup ML(u) \cup ML(v)$. Here, the

[3] Note that, we use a BFS manner to identify $PA(0)$ and $PA(8)$ in Algorithm 1 as we will show later that all vertices in $PA(0)$ will appear consecutively in Lemma 5.

Algorithm 2: Identify invalid labels and missing labels

Input: $G, (u, v)$, L, $PA(u)$, $PA(v)$, $RA(u)$, $RA(v)$
Output: $IL(u)$, $IL(v)$, $ML(u)$, $ML(v)$
1: Initialize $IL(u)$, $IL(v)$, $ML(u)$, $ML(v)$ as empty sets
2: **for all** vertex t in $RA(u)$ **do**
3: **if** $(r, \delta) \in L(t)$ and $r \in RA(v)$ **then**
4: $IL(u) \leftarrow IL(u) \cup \{r\}$
5: **if** $(r', \delta') \in L(t)$ and r' is first vertex in $L(t)$ where $\sigma(r) < \sigma(r')$ **then**
6: **for all** vertex w where $\sigma(r) < \sigma(w) < \sigma(r')$ and $w \in PA(v)$ **do**
7: **if** (r, δ) made w a redundant label entry in $L(t)$ **then**
8: $ML(u) \leftarrow ML(u) \cup \{w\}$
9: Repeat the above steps by mapping $u \leftarrow v$ and $v \leftarrow u$ to identify $IL(v)$ and $ML(v)$

modification of PLL algorithm [2] refers to that if a vertex already contains a label entry of the root vertex, we just move forward and no new label entry will be added. Otherwise, a normal process of PLL algorithm will be applied.

For Case (2), at the IDENTIFY stage, it is obvious that we have $PA(u) = RA(u) = C'(u)$ and $PA(v) = RA(v) = C'(u)$. Here, $C'(u)$ and $C'(v)$ refer to the two connected components of the updated graph. We simply let $IL(u) = RA(v)$ and $IL(v) = RA(u)$. At the UPDATE stage, we just remove all invalid label entries for all real affected vertices using Algorithm 3. The whole process of dealing with Case (2) is depicted in Algorithm 5.

Case (b) of Figure 3 shows an example. When deleting edge $(6, 9)$, we have $IL(9) = RA(6) = C'(6)$ and $IL(6) = RA(9) = C'(9)$. We first delete invalid label entries for vertices in $RA(6)$, where label entries contain vertices in $IL(6)$. In our example, no such label entries have been found. Then we delete invalid label entries for vertices in $RA(9)$, where label entries contain vertices in $IL(9)$. Take vertex 9 as an example, its label $L(9) = \{(0, 3)(2, 3)(3, 2)(4, 3)(6, 1)(9, 0)\}$ contains five invalid label entries: $(0, 3)(2, 3)(3, 2)(4, 3)(6, 1)$. This is because vertices $0, 2, 3, 4, 6$ are in $IL(9)$. Then Algorithm 5 removes all these invalid label entries from vertex 9. Similarly, invalid label entries in $L(10)$ can be identified and removed. After updates are completed, we get two labels updated, which are $L'(9)$ and $L'(10)$, ten invalid label entries in total and no new label entries. These are shown in Case (b) of Figure 3.

4.2.2 Proof of Correctness

Lemma 3 *After the deletion of edge (u, v) from graph G, for any vertex s, t in G', we must have $d_{G'}(s, t) \geq dist(s, t, L)$.*

Proof: In the old graph G, there are only two types of shortest paths: (1) shortest paths containing edge (u, v); and (2) shortest paths not containing edge (u, v). For the former, we have $d_{G'}(s, t) \geq d_G(s, t) = dist(s, t, L)$. For the latter, we have $d_{G'}(s, t) = d_G(s, t) = dist(s, t, L)$. Thus the lemma is proved.

□

Algorithm 3: Remove invalid label entries for real affected vertices

Input: $L', RA(u), RA(v), IL(u), IL(v)$
Output: Updated L'
//Remove invalid label entries for real affected vertices rooted at u
for all vertex $s \in RA(u)$ **do**
  for all label entry $(r, \delta_r) \in L'(s)$ **do**
    if $r \in IL(u)$ **then**
      Remove (r, δ_r) from $L'(s)$
//Remove invalid label entries for affected vertices rooted at v
for all vertex $s \in RA(v)$ **do**
  for all label entry $(r, \delta_r) \in L'(s)$ **do**
    if $r \in IL(v)$ **then**
      Remove (r, δ_r) from $L'(s)$

Algorithm 4: Update L after deleting an edge (u, v) $(C'(u) = C'(v))$

Input: $G', L, (u, v)$
Output: The updated 2-hop labeling L'
$L' \leftarrow L$
Identify $PA(u)$, $PA(v)$, $RA(u)$ and $RA(v)$using Algorithm 1
Identify $IL(u)$, $IL(v)$, $ML(u)$ and $ML(v)$ using Algorithm 2
Remove invalid label entries using Algorithm 3
Update label entries for all vertices in $RA(u) \cup RA(v)$ by performing a modified PLL BFS algorithm [2] on each vertex $r \in IL(u) \cup IL(v) \cup ML(u) \cup ML(v)$ in G' (in the ascending order of $\sigma[r]$)

Algorithm 5: Update L after deleting an edge (u, v) $(C'(u) \neq C'(v))$

Input: $G', L, (u, v)$
Output: The updated 2-hop labeling L'
$L' \leftarrow L$
$IL(u) \leftarrow C'(v)$
$IL(v) \leftarrow C'(u)$
Remove invalid label entries based on $IL(u)$ and $IL(v)$

Lemma 4 *After the deletion of edge (u, v) from graph G, for any vertex s, t in G', if $d_{G'}(s, t) > dist(s, t, L)$, and suppose a shortest path between s and t in G is $\pi_G(s, t)$, then we must have $uv \in \pi_G(s, t)$ or $vu \in \pi_G(s, t)$.*

Proof: This can be proved by contradiction. Suppose we have $d_{G'}(s, t) > dist(s, t, L)$ but $uv \notin \pi_G(s, t)$ and $vu \notin \pi_G(s, t)$, which means edge (u, v) does not appear in $\pi_G(s, t)$. In such case, there must exist a path $P_{G'}(s, t)$ in G' where $\pi_G(s, t) = P_{G'}(s, t)$. This means $d_{G'}(s, t)$ must be at most the length of $P_{G'}(s, t)$, i.e., the length of $\pi_G(s, t)$. Thus, we must have $d_{G'}(s, t) = dist(s, t, L') \leq d_G(s, t)$. This contradicts our assumption $d_{G'}(s, t) > dist(s, t, L)$. □

Lemma 5 *After the deletion of edge (u, v), for any vertex w in G', suppose w is a possible affected vertex, i.e. $w \in PA(u)$ or $w \in PA(v)$. Without loss of generality, we assume $w \in PA(u)$. Then there must exist a certain shortest*

path between w and v containing edge (u, v) in the original graph, where each internal vertex is a possible affected vertex.

Proof: Since $w \in PA(u)$, we must have that $d_G(r, v) = d_G(r, u) + 1$, which means that any shortest path between w and u, denoted as p_{wu}, plus edge (u, v) in the original graph must also be a shortest path between w and v. Hence, there must exist a certain shortest path between w and v containing edge (u, v) in the original graph and we can denote it as $p_{wv} = p_{wu} + (u, v)$.

Note that the internal vertices of p_{wv} must be on path p_{wu}. And since these internal vertices must also have a certain shortest path to vertex v containing edge (u, v), they must also be possible affected vertices in $PA(u)$ like w. □

Note that, according to Lemma 5, $PA(u)$ and $PA(v)$ can be considered as trees rooted at u and v, respectively. Moreover, we must have $PA(u) \bigcap PA(v) = \emptyset$. This is because otherwise, any vertex r in $PA(u) \bigcap PA(v)$ must have $d_G(r, v) = d_G(r, u) + 1$ and $d_G(r, u) = d_G(r, v) + 1$, which is impossible. Similarly, $RA(u)$ and $RA(v)$ are also trees rooted at u and v, respectively. This is because by definition, vertices in $RA(u)$ or in $RA(v)$ must be on the only shortest path to some other vertex via edge (u, v). Hence, vertices in $RA(u)$ or in $RA(v)$ must be trees rooted at u and v, respectively. Lemma 5 forms the basis of Algorithm 1.

Lemma 6 *After the deletion of edge (u, v), for any vertex w in $RA(u)$ (or $RA(v)$), if $L(w)$ contains an outdated label entry (r, δ), then $r \in IL(u)$ (or $IL(v)$) and $L(u)$ (or $L(v)$) must also contain an outdated label entry (r, δ').*

Proof: Let us take vertex u as an example (a similar proof applies to v). To prove that $r \in IL(u)$, by definition, we need to show that $r \in RA(v)$. Since (r, δ) is outdated in $L(w)$, we must have that edge (u, v) must appear in some shortest path between r and w according to Lemma 4. Then we have $d_G(r, u) = d_G(r, v) + 1$, and hence $r \in PA(u)$. Since (r, δ) is outdated, there must be only one shortest path between r and w. Then after the deletion of edge (u, v), we must also have $d_{G'}(r, u)! = d_G(r, v)+1$, which means $r \in RA(v)$. Therefore, we have $r \in IL(u)$.

Next we need to prove $L(u)$ must also contain an outdated label entry (r, δ'). According to Lemma 1, r must be the vertex with minimum order along the only single shortest path between r and w, where u is also on the path. Hence $L(u)$ must also contain a label entry (r, δ'). Since $r \in RA(v)$, it must be an outdated label entry in $L(u)$. Hence, the lemma is proved. □

Lemma 7 *After the deletion of edge (u, v), for any vertex w in $RA(u)$ (or $RA(v)$), if $L(w)$ contains a missing label entry (r, δ) because some label entry in $L(w)$ becomes invalid, then $r \in ML(u)$ (or $ML(v)$) and $L(u)$ (or $L(v)$) must also contain a missing label entry (r, δ').*

Proof: We take vertex u as an example (a similar proof applies to v). To prove that $r \in ML(u)$, by definition, we need to show that $r \in PA(v)$. Since (r, δ) is a missing label entry due to some outdated label entry in $L(w)$, we have that

edge (u, v) must appear in some shortest path between r and w according to Lemma 4. Then after the deletion of edge (u, v), we have $d_G(r, u) = d_G(r, v)+1$, and hence $r \in PA(u)$. Therefore, we have $r \in ML(u)$.

Next we need to prove $L(u)$ must also contain a missing label entry (r, δ'). Suppose (r', δ') is the outdated label entry in $L(w)$ that makes (t, δ_t) a missing label. Then according to Lemma 1, t must be the vertex with minimum order along the only single shortest path between r and w, where u is also on the path. Hence $L(u)$ must also contain a label entry (t, δ'_t). Since $r \in PA(v)$, edge (u, v) must appear in some shortest path between r and w, which means (r, δ') will become redundant due to the existence of (t, δ'_t) in $L(u)$. Therefore, (r, δ') must be an missing label entry in $L(u)$. Hence, the lemma is proved. □

Acoording to Lemma 6 and Lemma 7, in order to identify all invalid label entries and missing label entries, we only need to check vertices u and v and then we can apply the results to other real affected vertices. Hence, Lemma 6 and Lemma 7 both form the basis of Algorithm 2.

Theorem 1 *Algorithm 4 correctly updates the original well-ordering 2-hop distance labeling.*

Proof: Firstly, we apply Algorithm 1 to identify the possible and real affected vertices rooted at u and v, respectively. According to our analysis of Lemma 5, Algorithm 1 correctly identifies all possible and real affected vertices. Secondly, we apply Algorithm 2 to identify invalid labels $IL(u)$ in $RA(u)$ and invalid labels $IL(v)$ in $RA(v)$, as well as missing labels $ML(u)$ for $RA(u)$ and $ML(v)$ for $RA(v)$. According to Lemma 6 and Lemma 7, Algorithm 2 correctly identifies $IL(u)$, $IL(v)$, $ML(u)$ and $ML(v)$. Thirdly, we remove invalid label entries based on the results of $RA(u)$, $RA(v)$, $IL(u)$ and $IL(v)$. According to Lemma 4 and Lemma 5, all invalid label entries are removed correctly from L'. Finally, PLL BFS algorithm [2] helps to perform PLL BFS from each vertex in $IL(u)$, $IL(v)$, $ML(u)$ and $ML(v)$ by ascending order of their vertex orders and to relabel all the real affected vertices. Its correctness has been already proved in [2]. After these steps, all invalid labels and missing labels have been recovered, which results in an updated index for the new graph. Therefore, the theorem is proved. □

Theorem 2 *Algorithm 5 correctly updates the original well-ordering 2-hop distance labeling.*

Proof: Suppose s, t are both in $C'(u)$. Then any shortest path between s and t must not contain edge (u, v) which is the bridge between $C'(u)$ and $C'(v)$. In other words, the deletion of (u, v) will not affect shortest paths between s and t. So we have $d_{G'}(s, t) = d_G(s, t) = L(s, t, L)$.

We also need to prove that for the same pair of vertices s, t, when we calculate $dist(s, t, L')$, if we have $(r, \delta_{rs}) \in L'(s)$, $(r, \delta_{rt}) \in L'(t)$ and $\delta_{rs}+\delta_{rt} = dist(s, t, L')$, we must have $r \in C'(u)$. In such case, r must be an internal vertex of some shortest path between s and t [1]. So we have $r \in C'(u)$. Hence, only label entries with vertices in the same connected component are needed to calculate $dist(s, t, L')$.

Similar conclusions can be obtained if $s, t \in C'(v)$. Algorithm 5 only removes label entries containing vertices in another connected component which are all outdated label entries. Hence, the theorem has been proved. □

4.3 Remarks on Update Algorithms

Identifying Different Cases. We need to identify different cases before we can apply our algorithms. This could be achieved by performing a traversal of the whole graph G at the beginning and record all connected components. Then we incrementally maintain all the connected components as the graph changes.

Initial Index Construction. Pruned Landmark Labeling (PLL) technique presented in [2] is the state-of-the-art indexing technique for large static graphs. Indexes constructed by PLL [2] already have well-ordering property defined in Section 3. Therefore we use indexes constructed by PLL as the initial indexes in our experiments.

Index Quality. Our update algorithms preserve the index size very well. As to be shown in Section 5, the index size decreases gradually after deletions of edges. This further confirms the scalability of our method.

Complexity. Our algorithms can be directly applied on indexes constructed by PLL. Let w be the tree width [2] of G, n be the number of vertices and m be the number of edges in G. Also let (u, v) be the updated edge. According to analysis of PLL in [2], the number of label entries per vertex is $O(w \log n)$. If let $p' = (|IL(u) \cup IL(v) \cup ML(u) \cup ML(v)|)/n$ the time complexity of our decremental update algorithm is: (1) $O(n + m + p'wm \log n + p'w^2 n \log^2 n)$ for Algorithm 4, which is almost proportional to the complexity of PLL [2]; (2) $O(p'wn \log n)$ for Algorithm 5 as it removes invalid label entries only.

5 Experiments

We evaluated the performance of our decremental maintenance approach and compared with the state-of-the-art in memory indexing method Pruned Landmark Labelling (PLL) [2]. All experiments were performed under Linux (Ubuntu 16.04) on a server with Intel Xeon 8-Core E5-2690 CPU at 2.90 GHz, 64 GB main memory and 5 TB disk. All methods were implemented in C++ (the code of PLL was obtained from the first author's code repository on GitHub[4]), using the same GCC compiler (version 5.4.0) with the optimizer option O3.

5.1 Datasets and Performance Baseline

Table 2 lists the eleven real-world datasets used in our experiments, consisting of three computer networks, three web graphs, and five social networks. It

[4] https://github.com/iwiwi/pruned-landmark-labeling

should be noted that $|V|$ refers to the number of vertices and $|E|$ refers to the number of edges. More details on the first nine datasets can be found at the Stanford Network Analysis Project website[5] and more details on the last two datasets can be found at KONECT[6]. Similar to [3,2], we treat all graphs as undirected, unweighted graphs.

In addition, in Table 2, IT (or IT-bp, if bit-parallel is applied) denotes the indexing time or index construction time (in seconds) and LN (or LN-bp, if bit-parallel is adopted and only normal label entries are considered to calculate LN-bp, resulting in smaller numbers of LN-bp) denotes the average number of label entries of each vertex. In our experiments, we used IT and IT-bp in Table 2 as the baselines. We obtained all these results by using the Pruned Landmark Labeling (PLL) technique presented in [2], with or without bit-parallel BFSs. The number of times we conducted bit-parallel BFSs was set to 16 (the same setting was used in [3]). As mentioned, we applied our update algorithms directly on the indexes constructed by PLL in our experiments.

Table 2 Real-world Datasets

Dataset	**Type**	**|V|**	**|E|**	**IT (s)**	**IT-bp (s)**	**LN**	**LN-bp**
Gnutella	Computer	63 K	148 K	60.3	51.0	780.9	643.3
Epinions	Social	76 K	509 K	5.2	1.4	124.0	32.6
Slashdot	Social	82 K	948 K	15.4	4.7	216.0	68.3
Gowalla	Computer	197 K	950 K	14.5	9.5	100.2	57.3
NotreDame	Web	326 K	1.5 M	7.1	4.7	59.6	33.5
Youtube	Social	1.1 M	3 M	148.4	96.9	163.2	100.0
WikiTalk	Social	2.4 M	5 M	180.0	62.0	118.2	34.0
BerkStan	Web	685 K	7.5 M	19.7	19.1	58.0	41.1
Skitter	Computer	1.7 M	11.1 M	1,302.3	562.3	456.4	273.6
Flickr-links	Social	1.7 M	15.6 M	1,137.8	712.1	492.8	370.0
Hudong-pages	Web	2.5 M	18.9 M	7,066.0	5,813.0	960.2	829.2

5.2 Maintenance Performance

We have conducted extensive experiments to validate our proposed approach. We particularly focus on presenting the performance of our decremental update algorithm. In the experiments, we performed 1,000 deletions randomly on each real-world dataset and recorded the total update time (including IDENTIFY time and UPDATE time), the number of affected vertices, invalid label entries, and label number changes for each deletion.

[5] http://snap.stanford.edu/

[6] http://konect.uni-koblenz.de/networks/

Table 3 Decremental Maintenance Times

Dataset	AUT	MUT	SU	AUT-bp	MUT-bp	SU-bp
Gnutella	2.46 s	10.55 s	24.51	1.51 s	10.16 s	33.77
Epinions	0.11 s	1.98 s	47.27	0.08 s	0.85 s	17.50
Slashdot	0.26 s	3.22 s	59.23	0.12 s	1.57 s	39.17
Gowalla	0.49 s	7.43 s	29.59	0.46 s	7.07 s	20.65
NotreDame	0.55 s	4.07 s	12.91	0.46 s	3.11 s	10.22
Youtube	5.48 s	88.41 s	27.08	5.33 s	83.45 s	18.18
WikiTalk	10.90 s	103.52 s	16.51	6.74 s	49.32 s	9.20
BerkStan	1.47 s	11.2 s	13.40	1.46 s	10.78 s	13.08
Skitter	50.82 s	697.81 s	25.62	50.10 s	592.77 s	11.22
Flickr-links	12.48 s	339.03 s	91.17	9.60 s	292.95 s	74.18
Hudong-page	135.45 s	1,616.81 s	52.17	99.18 s	1549.33 s	58.61

5.2.1 Maintenance Times

Table 3 shows the detailed performance of our decremental update method. In the table, `AUT` (or `AUT-bp`, with bit-parallel) represents the average update time for each update. `MUT` (or `MUT-bp`, with bit-parallel) represents the maximum update time of all updates. Finally, `SU` (or `SU-bp`, with bit-parallel) represents the speedup ratio between baseline IT (indexing time) in Table 2 and `AUT` in the current table.

From the table, we can observe that for small graphs, without applying the bit-parallel technique, the average update time (AUT) is within a few seconds. For instance, the majority for small graphs is actually within one second, e.g., 0.55 seconds for `NotreDame`) while for large graphs, the average update time keeps low at a few seconds (e.g., 5.48 seconds for `Youtube`. Regarding the maximum update times, most of them are far less than the construction time of the whole index. Nevertheless, when compared with the baseline (i.e., IT or IT-bp in Table 2), regardless of small graphs or large graphs, our approach can speed up the maintenance process by an order of magnitude. This confirms the efficiency and effectiveness of our approach. Meanwhile, when bit-parallel is applied, the average update times (AUT-bp) are even smaller, though the average speedup ratio is not as large as the instances without bit-parallel, which could be due to the faster indexing processes with bit-parallel and the fact that less room is available for speeding up the maintenance processes.

It should be noted that we consider such comparisons are reasonable for two reasons: (1) to the best of our knowledge, our approach is the first work that can handle decremental updates in large graphs using distance labeling and it is impossible to compare our approach with other approaches; and (2) decremental update has been considered a challenging problem [3], which indicates the maintenance process is comparable to the reconstruction process of the whole index.

Figure 4(a) and Figure 4(b) further show the breakdown times spent on the IDENTIFY and UPDATE processes of each deletion, without applying bit-

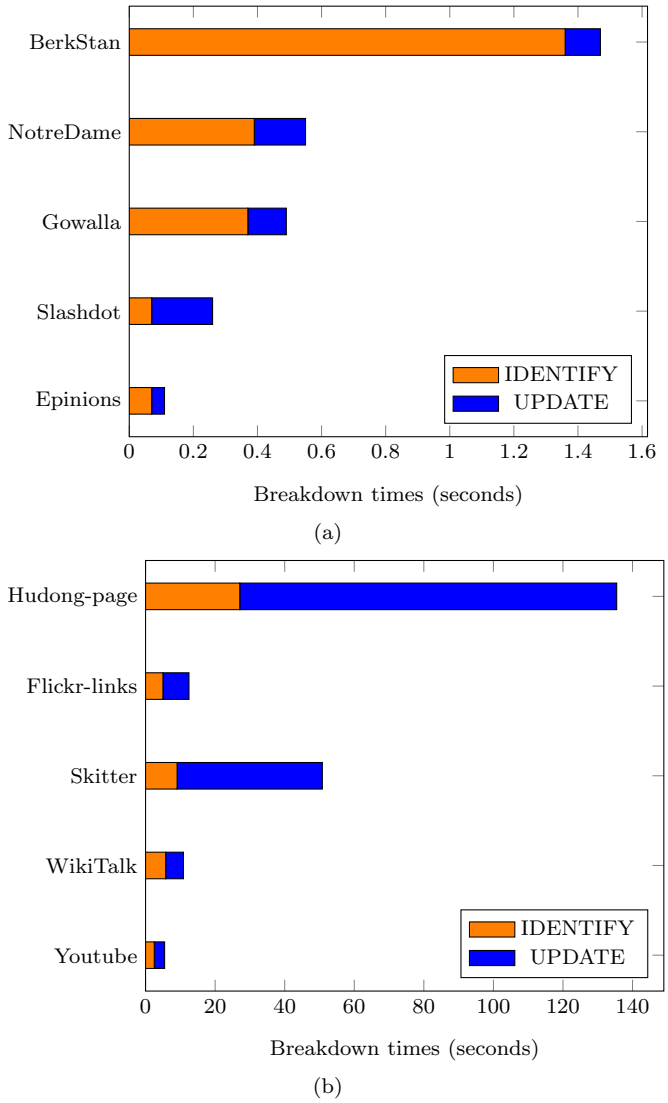


Fig. 4 Breakdown of maintenance times without bit-parallel

parallel, while Figure 5(a) and Figure 5(b) show breakdown times for cases with bit-parallel. From these figures, we can observe that IDENTIFY time and UPDATE times are comparable with each other, which indicates that we have no obvious bottleneck of the whole update process. The main reason for this is that the more we need to IDENTIFY, the higher possibility that we need to relabel or UPDATE more vertices.

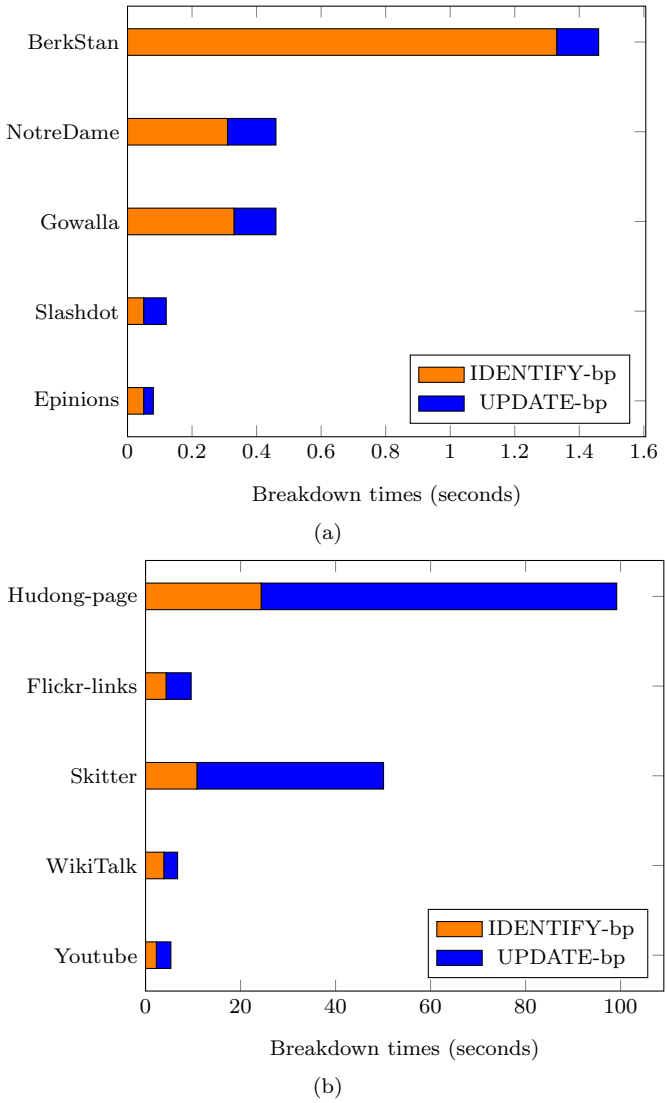


Fig. 5 Breakdown of maintenance times (with bit-parallel)

5.2.2 Affected Vertices

Figure 6 presents the percentage of affected vertices by the updates, showing the impact of decremental changes in a graph. It should be noted that the results are the same for cases with or without applying the bit-parallel technique. In the figure, `PAV` denotes possible affected vertices (i.e., $PA(u) \cup PA(v)$) and `RAV` denotes real affected vertices (i.e., $RA(u) \cup RA(v)$), as defined in Section

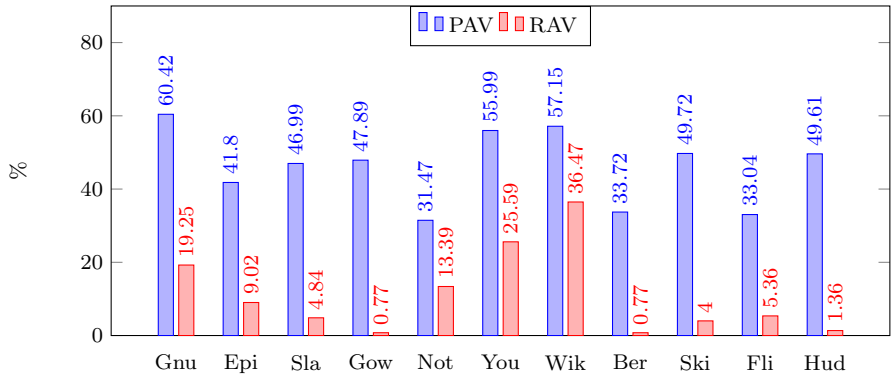


Fig. 6 Percentage of PAV and RAV

4. In the figure, we depict the average proportions of PAV and RAV out of all vertices of the original graphs. It is worthing mentioning that we use the first three letters in the names of each dataset (e.g., `Gnu` for `Gnutella`) for better illustration in the figure.

From the figure, we can see that normally around half of the vertices in the original graphs are possibly affected by an update. This means that around half of the vertices contain shortest paths going through the deleted edge. However, the proportion of RAV (real affected vertices) is much smaller than that of PAV, which means most of the vertices in PAV have other paths that have the same length of the invalidated shortest paths in an update.

5.2.3 Label Changes

Table 4 shows label changes caused by a decremental update. In the table, $|ILE|$ (or $|ILE|$-bp, with bit-parallel) represents the average of the total number of invalid or outdated label entries in an update. LC (or $|LC|$-bp, with bit-parallel) represents the average changes of the label entries in an update, and ΔLN (or ΔLN-bp, with bit-parallel) represents the changes in the average number of label entries per vertex after each update, reflecting the resulted index quality.

From the table, we can see that the total number of outdated label entries is quite large. However, the updated index still contains similar but a bit smaller number of label entries, which indicates that the number of label entries decreases gradually. This is normal as we do not expect dramatic changes of the index size after minor changes in the original graph. Furthermore, according to the column ΔLN, we can see that the average number of label entries per vertex decreases slowly. This indicates that although the number of outdated label entries is large, the index size remains quite stable. This shows the scalability of our update approach.

Further Table 5 and Table 6 show the index size comparisons. IS denotes the original index size, $IS1k$ denotes the index size after 1,000 edge removals

Table 4 Label Changes

Dataset	$\|ILE\|$	LC	ΔLN	$\|ILE\|$**-bp**	LC**-bp**	ΔLN**-bp**
Gnu	3.2×10^5	-79.08	-1.3×10^{-3}	2.5×10^4	-57.05	-9.1×10^{-4}
Epi	6.6×10^2	-5.32	-7.0×10^{-5}	2.2×10^1	-4.41	-5.8×10^{-5}
Sla	4.8×10^3	-6.36	-7.8×10^{-5}	2.9×10^1	-5.38	-6.6×10^{-5}
Gow	1.5×10^3	-2.84	-1.4×10^{-5}	1.4×10^2	-5.60	-2.8×10^{-5}
Not	1.8×10^3	-18.70	-5.7×10^{-5}	4.3×10^2	-5.28	-1.6×10^{-5}
You	1.4×10^4	-4.84	-4.4×10^{-6}	5.4×10^2	-18.62	-1.7×10^{-5}
Wik	9.7×10^3	-2.16	-9.0×10^{-7}	7.2×10^1	-11.01	-4.6×10^{-6}
Ber	9.6×10^2	-0.63	-9.2×10^{-7}	1.1×10^3	-5.55	-8.1×10^{-6}
Ski	3.8×10^4	-14.82	-8.7×10^{-6}	3.1×10^3	-7.94	-4.7×10^{-6}
Fli	5.81×10^3	-32.85	-1.9×10^{-5}	3.5×10^2	-24.97	-1.5×10^{-5}
Hud	6.59×10^5	-36.47	-1.5×10^{-5}	1.9×10^5	-26.38	-1.0×10^{-5}

Table 5 Index Size Comparisons (all in KB)

Dataset	IS	$IS1k$	ISR	$IS1k - IS$	$ISR - IS$
Gnu	239,177.80	238,791.65	237,127.27	-386.15	-2,050.53
Epi	46,591.08	46,565.12	46,542.64	-25.96	-48.44
Sla	87,396.74	87,365.68	87,288.46	-31.06	-108.28
Gow	97,923.36	97,909.51	97,893.62	-13.84	-29.74
Not	97,683.41	97,592.09	97,588.75	-91.33	-94.66
You	932,975.52	932,951.88	932,694.32	-23.64	-281.20
Wik	1,402,788.59	1,402,778.03	1,402,382.76	-10.56	-405.83
Ber	200,231.37	200,228.30	200,104.49	-3.07	-126.88
Ski	3,795,053.34	3,794,980.96	3,794,043.93	-72.38	-1,009.41
Fli	4,142,419.27	4,142,258.85	4,142,270.35	-160.42	-148.92
Hud	11,521,373.34	11,521,195.27	11,521,013.45	-178.07	-359.89

Table 6 Index Size Comparisons (with bit-parallel, all in KB)

Dataset	IS	$IS1k$	ISR	$IS1k - IS$	$ISR - IS$
Gnu	213,766.56	213,488.01	212,198.46	-278.55	-1,568.11
Epi	32,898.54	32,877.01	32,858.31	-21.53	-40.23
Sla	49,951.64	49,925.39	49,878.22	-26.25	-73.42
Gow	108,997.92	108,970.57	108,889.20	-27.34	-108.72
Not	142,610.62	142,584.82	142,511.52	-25.80	-99.10
You	883,205.66	883,114.74	883,086.15	-90.92	-119.51
Wik	1,054,379.36	1,054,325.60	1,054,305.60	-53.76	-73.76
Ber	325,494.86	325,467.78	325,452.97	-27.08	-41.88
Ski	2,731,903.81	2,731,865.02	2,731,270.50	-38.79	-633.31
Fli	3,569,262.95	3,569,141.04	3,569,099.72	-121.92	-163.23
Hud	10,603,276.68	10,603,147.90	10,603,053.55	-128.78	-223.13

using our decremental approach, while ISR denotes the index size from reconstruction of the whole index after 1,000 edge removals. In terms of IS, it is worth mentioning that applying bit-parallel technique does not always yield smaller indexes. For example, `BerkStan` will have a much bigger index if bit-parallel is applied. The main reason for this is that in the bit-parallel

labeling process, a number of full BFSs will be performed at the beginning of the labeling process and no pruning will take place, resulting in bigger indexes in some cases. Now consider the index size differences after 1,000 edge removals. From columns $IS1k - IS$ and $ISR - IS$ we can see that, though our approach does not reduce the index size as much as the reconstruction approach does, the index size (or index quality) is very comparable to the reconstruction approach.

5.3 Discussions

From our experiments, decremental maintenance normally requires substantial efforts to dynamically update its 2-hop distance labeling compared with incremental maintenance [3]. The root cause for this is that, decremental updates can make a large amount of labels become invalid, which have to be relabeled. Further, the 2-hop distance labeling itself does not provide alternative shortest paths information but only distance information to any other vertices. However, during the decremental maintenance, alternative shortest paths are critical for re-constructing the whole index. Due to lack of alternative shortest paths information, we have to perform a large number of BFSs to discover alternative shortest paths in order to maintain the index.

A possible way to further improve performance on decremental maintenance would be to introduce auxiliary information on the labeling or even redundant label entries in the labeling index. We leave this as one aspect of our future work.

6 Conclusions

This paper has studied the problem of computing the shortest path distance in large dynamic graphs. The concept of well-ordering 2-hop distance labeling and its properties have been defined and analyzed. Its properties that are useful for index maintenance have been identified. We have particularly focused on the decremental update operation (i.e., edge deletions), a challenging problem that remains open, to the best of our knowledge. Several algorithms have been designed to compute distance labeling in large dynamic graphs, which can handle decremental updates efficiently. Based on the most recent technique Pruned Landmark Labeling (PLL) [2] that handles only static graphs, we have implemented an extended version using the techniques developed in this paper. The extended PLL is able to support index updates in large dynamic graphs efficiently for decremental maintenance. Extensive experiments have also been performed on eleven real-world graphs to confirm its effectiveness and efficiency. Specifically, the index update process can be accelerated by up to an order of magnitude faster compared with the original PLL algorithm. The resulted indexes preserve the index size well, which further demonstrates the index quality and scalability of our techniques.

Our future work will further investigate several aspects of maintaining distance labeling indexes for large dynamic graphs. The first one centers on how to further speed up the decremental maintenance. We will investigate possible ways to maintain auxiliary information and redundant label entries that could be useful to reduce the relabeling efforts when an update occurs. We also plan to extend our work to efficiently update distance labeling in memory and computing resource constrained environments. This is an important direction because 2-hop distance labeling could be possibly first precomputed by a super computer and then used on resource constrained devices such as GPS units, smart phones and so on.

References

1. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.F.: Hierarchical Hub Labelings for Shortest Paths. In: Proc. of the 20th Annual European Symposium on Algorithms (ESA 2012), pp. 24–35. Ljubljana, Slovenia (2012)
2. Akiba, T., Iwata, Y., Yoshida, Y.: Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In: Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2013), pp. 349–360. New York, NY, USA (2013)
3. Akiba, T., Iwata, Y., Yoshida, Y.: Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In: Proc. of the 23rd International World Wide Web Conference (WWW 2014), pp. 237–248. Seoul, Republic of Korea (2014)
4. Akiba, T., Sommer, C., Kawarabayashi, K.: Shortest-Path Queries for Complex Networks: Exploiting Low Tree-Width Outside the Core. In: Proc. of the 15th International Conference on Extending Database Technology, (EDBT 2012), pp. 144–155. Berlin, Germany (2012)
5. Bernstein, A.: Maintaining Shortest Paths Under Deletions in Weighted Directed Graphs: [Extended Abstract]. In: Proc. of the forty-fifth annual ACM symposium on Theory of computing (STOC 2013), pp. 725–734. Palo Alto, CA, USA (2013)
6. Bramandia, R., Choi, B., Ng, W.K.: Incremental Maintenance of 2-Hop Labeling of Large Graphs. IEEE Trans. Knowl. Data Eng. **22**(5), 682–698 (2010)
7. Chang, L., Yu, J.X., Qin, L., Cheng, H., Qiao, M.: The exact distance to destination in undirected world. VLDB J. **21**(6), 869–888 (2012)
8. Cheng, J., Ke, Y., Chu, S., Cheng, C.: Efficient Processing of Distance Queries in Large Graphs: A Vertex Cover Approach. In: Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2012), pp. 457–468. Scottsdale, AZ, USA (2012)
9. Cheng, J., Yu, J.X.: On-line exact shortest distance query processing. In: EDBT, pp. 481–492 (2009)
10. Ciortea, A., Boissier, O., Zimmermann, A., Florea, A.M.: Reconsidering the social web of things: position paper. In: Proc. the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp 2013) (Adjunct Publication), pp. 1535–1544. Zurich, Switzerland (2013)
11. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. In: Proc. of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002), pp. 937–946. San Francisco, CA, USA (2002)
12. Demetrescu, C., Italiano, G.F.: A New Approach to Dynamic All Pairs Shortest Paths. J. ACM **51**(6), 968–992 (2004)
13. Demetrescu, C., Italiano, G.F.: Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms. ACM Transactions on Algorithms **2**(4), 578–601 (2006)
14. Espinosa, J.: Facebook's Global Growth in Q4: 1.06B MAU, Mobile Surpasses Web. `http://www.insidefacebook.com/2013/01/30/`

15. Farsani, H.K., Nematbakhsh, M.A., Lausen, G.: SRank: Shortest paths as distance between nodes of a graph with application to RDF clustering. J. Information Science **39**(2), 198–210 (2013)
16. Fu, A.W.C., Wu, H., Cheng, J., Wong, R.C.W.: IS-LABEL: an Independent-Set based Labeling Scheme for Point-to-Point Distance Querying. Proc.of the VLDB Endowment **6**(6), 457–468 (2013)
17. Jin, R., Ruan, N., Xiang, Y., Lee, V.E.: A highway-centric labeling approach for answering distance queries on large sparse graphs. In: Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2012), pp. 445–456. Scottsdale, AZ, USA (2012)
18. K., P., Kumar, S.P., Damien, D.: Ranked answer graph construction for keyword queries on RDF graphs without distance neighbourhood restriction. In: Proc. of the 20th International Conference on World Wide Web (WWW 2011, Companion Volume), pp. 361–366. Hyderabad, India (2011)
19. Qin, Y., Sheng, Q.Z., Zhang, W.E.: SIEF: Efficiently Answering Distance Queries for Failure Prone Graphs. In: Proc. of the 18th International Conference on Extending Database Technology (EDBT 2015), pp. 145–156. Brussels, Belgium (2015)
20. Schenkel, R., Theobald, A., Weikum, G.: Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. In: Proc. of the 21st International Conference on Data Engineering (ICDE 2005), pp. 360–371. Tokyo, Japan (2005)
21. Vassilvitskii, S., Brill, E.: Using Web-Graph Distance for Relevance Feedback in Web Search. In: Proc. of the 29th Annual International Conference on Research and Development in Information Retrieval (SIGIR 2006), pp. 147–153. Seattle, Washington, USA (2006)
22. Vieira, M.V., Fonseca, B.M., Damazio, R., Golgher, P.B., de Castro Reis, D., Ribeiro-Neto, B.A.: Efficient search ranking in social networks. In: Proc. of the Sixteenth ACM Conference on Information and Knowledge Management (CIKM 2007), pp. 563–572. Lisbon, Portugal (2007)
23. Wehmuth, K., Ziviani, A.: DACCER: Distributed Assessment of the Closeness CEntrality Ranking in complex networks. Computer Networks **57**(13), 2536–2548 (2013)
24. Wei, F.: TEDI: efficient shortest path query answering on graphs. In: Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2010), pp. 99–110. Indianapolis, Indiana, USA (2010)
25. Yao, L., Sheng, Q.Z.: Exploiting Latent Relevance for Relational Learning of Ubiquitous Things. In: Proc. of the 21st ACM International Conference on Information and Knowledge Management (CIKM 2012). Maui, Hawaii, USA (2012)
26. Zhu, A.D., Xiao, X., Wang, S., Lin, W.: Efficient Single-Source Shortest Path and Distance Queries on Large Graphs. In: Proc. of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2013), pp. 998–1006. Chicago, IL, USA (2013)