The VLDB Journal (2022) 31:483–506
https://doi.org/10.1007/s00778-021-00707-z

REGULAR PAPER

Fast fully dynamic labelling for distance queries

Muhammad Farhan[1] · Qing Wang[1] · Yu Lin[1] · Brendan McKay[1]

Received: 28 February 2021 / Revised: 6 September 2021 / Accepted: 2 October 2021 / Published online: 19 October 2021

Abstract
Finding the shortest-path distance between an arbitrary pair of vertices is a fundamental problem in graph theory. A tremendous amount of research has explored this problem, most of which is limited to static graphs. Due to the dynamic nature of real-world networks, such as social networks or web graphs in which a link between two entities may fail or become alive at any time, there is a pressing need to address this problem for dynamic networks. Existing work can only accommodate distance queries over moderately large dynamic networks due to high space cost and long pre-processing time required for constructing distance labelling, and even on such moderately large dynamic networks, distance labelling can hardly be updated efficiently. In this article, we propose a fully dynamic labelling method to efficiently update distance labelling so as to answer distance queries over large dynamic graphs. At its core, our proposed method incorporates two building blocks: (i) *incremental algorithm* for handling incremental update operations, i.e. edge insertions, and (ii) *decremental algorithm* for handling decremental update operations, i.e. edge deletions. These building blocks are built in a highly scalable framework of distance query answering. We theoretically prove the correctness of our fully dynamic labelling method and its preservation of the minimality of labelling. We have also evaluated on 13 real-world large complex networks to empirically verify the efficiency, scalability and robustness of our method.

Keywords Graph algorithms · Dynamic graphs · Distance labelling · Query processing

1 Introduction

The problem of answering distance queries has a wide range of real-world applications, such as context-aware search in web graphs [40,47], social network analysis [5,48] and management of resources in computer networks [7]. However, the underlying graphs in these applications are typically dynamic, and their topological structures may change over time. For example, social networks are reported to be highly dynamic [29,37,51], thereby requiring distance information to be dynamically updated in order to perform social network analysis accurately (i.e. find closeness and similarity between users and contents) [48,52]. In computer networks, communication between two points often happens through links between routers, subnets, interfaces and network locations. In such an environment, a single cable damage or fault in devices can affect many links to be unavailable which requires to quickly replace communication channels with the best available links using distance information.

Previous studies have primarily focused on distance queries on static graphs [1,2,4,12,15,22,23,26–28,35,50], with limited attention being paid to dynamics on graphs. Traditionally, to speed up query response time on static graphs, a key technique is to pre-compute a data structure called *distance labelling* and use such an offline data structure to answer distance queries more efficiently. However, when a graph dynamically changes, its distance labelling needs to be changed accordingly; otherwise, distance queries may yield underestimated or overestimated distances. Thus, after a change is applied on a graph, one could naturally consider either (1) recompute an offline distance labelling from scratch or (2) conduct an online search on the changed graph, in order to be able to answer distance queries correctly. However, both of these approaches are very inefficient. Take graphs with a couple of millions of vertices such as

✉ Muhammad Farhan
muhammad.farhan@anu.edu.au

Qing Wang
qing.wang@anu.edu.au

Yu Lin
yu.lin@anu.edu.au

Brendan McKay
brendan.mckay@anu.edu.au

1 Australian National University, Acton, Australia

Livejournal [33] and Hollywood [10] for example, it may require hundreds to thousands of minutes to recompute a distance labelling from scratch as a result of a single change [26]. Furthermore, if only a very small portion of a graph is affected against a change, recomputing distance labelling from scratch would not only waste computing resources, but also prevent the availability of distance queries during recomputing. We observe in our experiments that more than 90% graph changes cause between $10^{-7}\%$ and 1% vertices to be affected in many real-world networks. On the other hand, answering a distance query entirely based on online search is often too slow to be useful in time-sensitive applications; for example, it takes about 30 seconds on average to answer a distance query on a Twitter network with 42 millions of vertices [10].

In this article, we aim to develop a robust solution for distance queries on dynamic graphs with the following characteristics: (1) *fully dynamics*—It can handle both edge/node insertion and deletion on a graph; (2) *time and space efficiency*—It can answer *exact* distance queries in milliseconds; and (3) *scalability*—It can scale to very large networks with billions of vertices and edges. It is worth to note that previous studies generally consider real-world networks as being in two categories [2,49]: (a) complex networks and (b) road networks. It has been discussed that complex networks and road networks often exhibit different properties such as small diameter and local clustering [2,23,24]. Our work in this article focuses on complex networks including web graphs, social networks and computer networks.

Challenges It has been reported [3,19] that designing a fully dynamic method for answering distance queries is very challenging. First, the difficulty of updating a distance labelling lies in two aspects: (1) When adding an edge into a graph, outdated and redundant entries of distance labelling may occur. Although outdated and redundant entries do not affect the correctness of distance query answering, they would deteriorate the query performance over time. However, identifying and removing such entries are known to be a complicated task [3]; (2) when deleting an edge from a graph, outdated distance entries have to be removed; otherwise, distance queries cannot be correctly answered. Hence, entries of distance labelling being affected must be accurately identified and repaired with new distances. However, finding the new distances between affected vertices is computationally expensive and indeed much more challenging than the case of adding an edge into a graph [3,19,41]. Both aspects, in a nutshell, require us to pinpoint affected vertices so as to update their labels efficiently. Further, although query time and update time are both critical for answering distance queries on dynamic graphs, it is not easy (if not impossible) to design a solution that is efficient in both. This requires us to find new insights into dynamic properties of a distance labelling, as well as a good trade-off between query time and update time. Last but not least, scaling distance queries to dynamic graphs with billions of nodes and edges is hard. Previous work [3,19,26,41] has mostly considered 2-hop distance labelling [14], which has quadratic space requirements and unbearable index construction time; as a result, their query and update performance is dramatically degraded on large-scale dynamic graphs. Ideally, the labelling size of a graph should be much smaller than its original size. However, the state-of-the-art distance labelling technique, i.e. pruned landmark labelling method (PLL) [2], still yields a distance labelling whose size is several orders of magnitude larger than the original size of a dataset.

Contributions To address these challenges, we propose a fully dynamic method to efficiently answer distance queries over large dynamic graphs. At its core, our proposed method incorporates two building blocks: (i) *incremental algorithm* for handling incremental update operations, i.e. edge insertion, and (ii) *decremental algorithm* for handling decremental update operations, i.e. edge deletion. These two building blocks are built in a highly scalable framework of distance query answering. To the best of our knowledge, our method is the first fully dynamic method that can scale to graphs with billions of vertices and edges, without compromising performance on query time and labelling size. To summarize, the main contributions of this work are as follows:

– Our incremental algorithm overcomes the challenge of eliminating outdated and redundant distance entries in order to preserve the minimality of labelling. None of the previous studies have addressed this challenge because detecting outdated and redundant distance entries is too costly [3]. When an edge is inserted, previous studies only add new distance entries or modify existing distance entries. This, however, leads to an ever increasing size of labelling. Then, both query performance and space efficiency deteriorate over time.
– Our decremental algorithm can efficiently identify affected vertices and update their labels without compromising on query time and labelling size. We achieve this based on two observations. The first is to characterize a special kind of vertices, called *anchor vertices*, which are critical for updating labelling. The second is to prune unnecessary searches by characterizing *prunable vertices*, thereby improving update efficiency. Previous work [19] has reported that edge deletion requires much longer update time than edge insertion, but no interpretation was provided. We fill in this gap by analysing the fundamental differences between edge insertion and edge deletion on dynamic graphs.
– We theoretically prove the correctness of our fully dynamic method and show that it preserves the minimality of labelling under update operations (edge insertion and edge deletion). Note that, by leveraging the property

Table 1 Flickr, UK and Clueweb12 are the largest networks evaluated by the methods FULPLL [3,19], FULFD [26] and FULHL (this work), respectively, where "-" indicates no result due to scalability issues

Network	Network Size		Update Time			Query Time			Labelling Size		
	$\|V\|$	$\|E\|$	FULPLL	FULFD	FULHL	FULPLL	FULFD	FULHL	FULPLL	FULFD	FULHL
Flickr	1.7M	16M	6810 ms	7.655 ms	**0.053 ms**	0.009 ms	0.012 ms	**0.007 ms**	12.7 GB	152 MB	**34 MB**
UK	106M	3.7B	–	337.6 ms	**1.075 ms**	–	5.858 ms	**3.488 ms**	–	11.8 GB	**1.78 GB**
Clueweb12	~1B	43B	–	–	**1796 ms**	–	–	**9.375 ms**	–	–	**49.1 GB**

of highway cover [22], the minimal size of a distance labelling in this work is much smaller than the size of a 2-hop labelling in previous work [2,26]. Finally, we provide a complexity analysis for our fully dynamic method.

To empirically verify the efficiency and scalability of our fully dynamic method, we have conducted experiments using 13 real-world large networks across different domains. In particular, our methods can perform updates within a couple of seconds even on networks with billions of vertices and edges, while still answering distance queries efficiently in the order of milliseconds and maintaining very small labelling sizes. Table 1 depicts the performance of our fully dynamic method FULHL against with the two state-of-the-art methods FULPLL [3,19] and FULFD [26]. We present the results for the largest network that was previously evaluated by each of these methods. We can see that FULHL significantly outperforms FULPLL and FULFD in all three dimensions, i.e. update time, query time and labelling size, and can scale to billion-scale networks. On the other hand, FULPLL and FULFD fail to scale to networks of size over 16 millions and 3.7 billions of edges, respectively. We will further discuss these results in detail in Sect. 6.

Outline. The rest of the article is organized as follows. In Sect. 2, we review existing work in the literature that is related to our present work. In Sect. 3, we present the preliminary notations and definitions used in this article. Then, we formulate the fully dynamic framework and present our incremental and decremental algorithms in Sect.'4. The formal proofs for showing that our algorithms are correct and preserve the property of minimality are provided in Sect. 5. In Sect. 6, we discuss our experimental results. In Sect. 7, we discuss the extensions of our work on directed and weighted graphs. Then, Sect. 8 discusses the advantages of designing dynamic algorithms using highway cover framework. Finally, we conclude the article and outline future research directions in Sect. 9.

2 Related work

The problem of answering shortest-path distance queries has been an active research topic for many years. Traditionally, a distance query can be answered using Dijkstra's algorithm [45] on positively weighted graphs or breadth-first search (BFS) algorithm on unweighted graphs. However, these traditional algorithms fail to achieve desired response time performance required by many real-world applications that operate on increasingly large graphs.

Labelling-based methods have emerged as an attractive way of accelerating response time to distance queries [1,2, 12,12,14,22,23,28,50]. Most of these methods constructed a labelling based on 2-hop cover labelling [14]. For example, Cheng and Yu [13] proposed a heuristic-based algorithm to construct a 2-hop distance labelling on directed graphs. Their method used the property of strongly connected components to exploit graph partitioning techniques. However, such a graph partitioning process introduces high computational time cost because it has to find vertex separators recursively. Furthermore, their method is limited to handle only directed graphs. Theoretically, it has been shown that computing a minimal 2-hop cover labelling is NP-hard [1,14].

Tree decomposition-based approaches [4,50] have also been studied for answering distance queries on graphs. Wei [50] proposed an index for shortest-path query answering, called TEDI, which heuristically decomposes a graph into a tree through tree decomposition. Given a graph G, a tree decomposition of G yields a tree T in which each vertex is associated with a set of vertices in the graph G (also called a bag [42]). The shortest-path distances between vertices in the same bag are pre-computed and stored in the corresponding bags. Then, given a distance query, a bottom-up operation along the tree T can be carried out to answer the distance query. Further, Akiba et al. [4] proposed an improved TEDI index that exploited a core–fringe structure in a graph. However, due to the presence of core–fringe structure in complex networks [11,38], these methods may produce bags of large sizes in a decomposed tree and computing pairwise distances of vertices in these large bags can require long pre-processing time and huge storage space, making it impractical on large graphs. Moreover, the decomposition time of a large graph is very costly and only small-sized graphs can be processed within a reasonable amount of time.

Hierarchical hub-labelling (HHL) was proposed by Abraham et al. [1], which is based on the partial order of vertices. In their method, they used a top-down approach to compute a partial order of vertices that can produce a smaller HHL. However, their method is not scalable to handle large graphs due to very high storage and computation requirements for finding the partial order of vertices. Another method called Highway Centric Labeling (HCL) was proposed by Jin et al. [28], which exploits highway structure of a graph by finding a spanning tree that can be used as a highway to efficiently compute 2-hop distance labelling for fast distance computation.

In [23], Fu et al. proposed IS-Label which gained a significant scalability in pre-computing a 2-hop distance labelling for large graphs in a memory-constrained environment. IS-Label uses the notion of an independent set of vertices in a graph. It recursively computes independent sets of vertices and augments edges by removing these sets to preserve the distance information. Generally, IS-Label is regarded as a hybrid method that combines distance labelling with graph traversal. Then, Akiba et al. proposed the pruned landmark labelling (PLL) [2] to pre-compute a 2-hop distance labelling by performing a pruned breadth-first search from every vertex. The idea is to prune vertices whose distance information can be obtained using the partially available 2-hop distance labelling constructed via previous breadth-first searches. This work helps to achieve low construction cost and small labelling size due to reduced search space on million-scale networks and serves as the state-of-the-art labelling-based distance queries. A recent method [34] parallelised PLL to further increase its scalability to answer distance queries on large graphs. Apart from these 2-hop distance labelling techniques, a multi-hop distance labelling approach has also been studied in [12], which reduces the size of labelling at the cost of increased response time.

So far, only a few attempts have been made to study distance queries over dynamic graphs [3,19,21,26,39,41], which are mostly based on the idea of 2-hop cover labelling or its variants. Akiba et al. [3] studied the problem of updating pruned landmark labelling for incremental updates (i.e. vertex additions and edge additions). This work, however, does not remove outdated and redundant entries in distance labels because the authors considered that detecting such entries is too costly. This inevitably breaks the minimality of pruned landmark labelling, leading to an ever increase of labelling size and deteriorating query performance over time. Qin et al. [41] and D'angelo et al. [19] studied the problem of updating pruned landmark labelling for decremental updates (i.e. edge deletions). These methods can only scale to graphs with a few millions of nodes due to their high time complexities. Their experiments [19,41] showed that the average update time of an edge deletion on a network with 19 millions of edges is 135 seconds in [41] and on a network with 16 millions of edges is 19 seconds in [19], which are significantly inefficient for dynamic graphs. In the work by D'angelo et al. [19], they combined the algorithm for incremental updates proposed in [3] with their method for decremental updates to form a fully dynamic algorithm. Nevertheless, this fully dynamic algorithm can only be applied to networks with around 20 millions of edges. A recent method by D'Emidio et al. [20] claims an improvement over the method proposed in [19] for decremental updates. However, this method is limited to graphs with few millions of nodes and updates labelling in the order of seconds.

Alternatively, methods for maintaining all-pair shortest paths (APSP) have also been studied to allow direct look-up of the shortest-path distance at the cost of quadratic space and update time. Theoretically, the update time and space complexities of maintaining all-pair shortest paths (APSP) data structure are prohibitively very high and cannot scale to large graphs, e.g. the dynamic algorithm proposed in [16] takes $\tilde{O}(n^2)$ amortized time per update operation and $O(n^3)$ space. A recent method [25] proposed an improved bound in the form of a deterministic algorithm with $\tilde{O}(n^{2+2/3})$ update time and a Las-Vegas algorithm with $\tilde{O}(n^{2+1/2})$ update time for unweighted graphs having $\tilde{O}(n^2)$ space requirements. With these quadratic time and space requirements, they are not practical to be applied to large graphs having millions of vertices. Approximate methods for dynamically maintaining APSP have also been studied in order to improve the update time of maintaining APSP data structure. The work in [43] maintains dynamic all-pair $(1+\epsilon)$ approximate shortest paths in $\tilde{O}(mn/t)$ update time and $\tilde{O}(n^2)$ space, where t is a parameter that describes the trade-off between the update time and query time. Another $(2+\epsilon)$ approximate algorithm by Bernstein et al. [6] claimed a faster update time of $o(mn^{\epsilon} log R/\epsilon)$ for any fixed ϵ, where R is the ratio between the largest and the smallest edge weights. Unfortunately, their update times suffer the drawback of a super-polynomial dependence on ϵ. These approximate methods are not practical, particularly when exact distances are sought.

To accelerate shortest-path distance queries on large networks, another line of research is to combine a partial distance labelling with online shortest-path searches. Hayashi et al. in [26] proposed a fully dynamic approach that selects a small set of landmarks R and pre-computes bit-parallel shortest-path trees (SPTs) rooted at each landmark $r \in R$. Then, an online search is conducted on a sparsified graph under an upper-distance bound being computed via the bit-parallel SPTs. However, this method still fails to construct labelling on networks with billions of vertices because of very high pre-processing time and space consumption of computing bit-parallel SPTs. Following the same line, Farhan et al. [22] introduced a highway-cover labelling method (HL), which can provide fast response time (i.e. milliseconds) for distance queries even on billion-scale graphs. This approach, however, only works for static graphs. Very recently, Farhan et al. [21] proposed an online incremental method which solves the problem of eliminating outdated and redundant entries caused by edge insertions. However, the update time of this method is still high on large graphs with billions of vertices and edges, which makes it impractical for applications that require updates to be performed in the order of milliseconds.

The present work aims to propose a fully dynamic method that can leverage the advantages of the approach [22] and also overcome the limitations of previous methods for distance queries on fully dynamic graphs. Unlike the incremental algorithm proposed in [21] which can only handle incremental updates (edge insertions) through two stages, i.e. first find affected vertices and then update their labels, this present work unifies the separate processes in these two stages into a single process, i.e. find affected vertices and update their labels simultaneously, while still guaranteeing the minimality of labelling. This improvement is based on new insights we gain about edge insertions (as formulated in Lemma 4 in Sect. 4.2). Further, not only handling edge insertions, the present work has also designed a decremental algorithm to handle edge deletions which has long been recognised as a difficult task in the literature [3,19,41]. Taking the two sides of update operations on graphs (i.e. both edge insertion and edge deletion) into consideration, the fully dynamic method proposed in the present work provides a novel, fast and scalable solution for answering distance queries on large and dynamic graphs.

3 Preliminaries

Let $G = (V, E)$ be an undirected graph where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. We denote by $N(v)$ the set of neighbours of a vertex $v \in V$, i.e. $N(v) = \{u \in V | (u, v) \in E\}$. Given two vertices u and v in G, the *distance* between u and v, denoted as $d_G(u, v)$, is the length of the shortest path from u to v. If there does not exist a path from u to v, then $d_G(u, v) = \infty$. We use $P_G(u, v)$ to denote the set of all shortest paths between u and v in G.

We consider two kinds of update operations: edge insertion and edge deletion. Given a graph $G = (V, E)$, an *edge insertion* is to add an edge (a, b) into G where $\{a, b\} \subseteq V$ and $(a, b) \notin E$. Conversely, an *edge deletion* is to delete an edge (a, b) from G where $(a, b) \in E$. Note that we consider vertex insertion and vertex deletion as a sequence of edge insertions and edge deletions, respectively. When inserting a new vertex, we first create an isolated vertex and then insert edges incident to it, and vice versa for deleting a vertex. Similarly, multiple updates (i.e. edge insertions or deletions) are processed one by one. For clarity, we use $G \hookrightarrow G'$ to indicate that a graph G is changed to a graph G' by an edge insertion or an edge deletion.

The following facts are important for designing dynamic algorithms. They state that an edge insertion may decrease distances between vertices, and conversely, an edge deletion may increase distances between vertices.

Fact 1 *Let $G' = (V, E \cup \{(u, v)\})$ be the graph after inserting an edge (u, v) into $G = (V, E)$. Then for any two vertices $s, t \in V$, $d_G(s, t) \geq d_{G'}(s, t)$.*

Fact 2 *Let $G' = (V, E \cup \{(u, v)\})$ be the graph after deleting an edge (u, v) from $G = (V, E)$. Then for any two vertices $s, t \in V$, $d_G(s, t) \leq d_{G'}(s, t)$.*

3.1 2-hop cover labelling

Various labelling techniques have been previously used for improving efficiency of answering distance queries on static graphs, among which 2-hop cover labelling [14] is the most well known. Let $R \subseteq V$ be a small set of special vertices, namely *landmarks*, in a graph $G = (V, E)$. For each vertex $v \in V$, the *label* of v is a set of *distance entries* $L(v) = \{(r_1, \delta_L(r_1, v)), \ldots, (r_n, \delta_L(r_n, v))\}$, where $r_i \in R$ and $\delta_L(r_i, v) = d_G(r_i, v)$. We call $L = \{L(v)\}_{v \in V}$ a *distance labelling* over G. The *size* of a distance labelling L is defined as $size(L) = \sum_{v \in V} |L(v)|$.

Definition 1 *(2-hop cover labelling)* A distance labelling L over a graph $G = (V, E)$ is a *2-hop cover labelling* if the following holds for any two vertices $u, v \in V$:

$$d_G(u, v) = \min\{\delta_L(r_i, u) + \delta_L(r_i, v) | \\ (r_i, \delta_L(r_i, u)) \in L(u), (r_i, \delta_L(r_i, v)) \in L(v)\} \quad (1)$$

Thus, for any two vertices $u, v \in V$, an exact distance query can be answered by only looking up the labels of u and v in a 2-hop cover labelling. Given a graph G, the complexity of finding a minimal 2-hop cover labelling of G is known to be NP-hard [14].

3.2 Highway cover labelling

Unlike the previous work [3,19,26,41,46] that uses 2-hop cover labelling, we develop our fully dynamic method using a highly scalable labelling approach, called *highway cover labelling* [22].

Definition 2 *(Highway)* A *highway* $H = (R, \delta_H)$ over a graph $G = (V, E)$ consists of a set R of landmarks and a distance decoding function $\delta_H : R \times R \to \mathbb{N}^+$ such that, for any two landmarks $r_1, r_2 \in R$, $\delta_H(r_1, r_2) = d_G(r_1, r_2)$ holds.

Definition 3 *(Highway cover labelling)* A *highway cover labelling* is a pair $\Gamma = (H, L)$ where H is a highway and L is a distance labelling satisfying that, for any vertex $v \in V \backslash R$ and $r \in R$, we have:

$$d_G(r, v) = \min\{\delta_L(r_i, v) + \delta_H(r, r_i) |$$

$$(r_i, \delta_L(r_i, v)) \in L(v)\} \quad (2)$$

Highway cover labelling enjoys several nice theoretical properties, such as minimality and order independence. It is shown in [22] that there exists an algorithm that can construct a minimal highway cover labelling independently of the order of applying landmarks.

Given a highway cover labelling $\Gamma = (H, L)$, an upper bound on the distance between a pair of vertices $u, v \in V$ in a graph $G = (V, E)$ can be computed as follows:

$$\begin{aligned} d^{\top}_{uv} = \mathtt{min}\,\{&\delta_L(r_i, u) + \delta_H(r_i, r_j) + \delta_L(r_j, v) | \\ &(r_i, \delta_L(r_i, u)) \in L(u), \\ &(r_j, \delta_L(r_j, v)) \in L(v)\} \end{aligned} \quad (3)$$

Then, an exact distance query $Q(u, v, \Gamma)$ can be answered by conducting a distance-bounded shortest-path search over a sparsified graph $G[V \backslash R]$ (i.e. removing all landmarks in R from G) under the upper bound $d^{\top}_{uv}$ such that:

$$Q(u, v, \Gamma) = \begin{cases} d_{G[V\backslash R]}(u, v) & \text{if } d_{G[V\backslash R]}(u, v) \le d^{\top}_{uv}, \\ d^{\top}_{uv} & \text{otherwise.} \end{cases}$$

3.3 Problem definition

In this paper, we study the fully dynamic labelling problem for distance queries. Given a graph that is dynamically changed by edge insertions or deletions over time, the fully dynamic labelling problem is concerned about updating a distance labelling to ensure that distance queries can be correctly answered on the dynamic graph. Formally, we define this problem below.

Definition 4 *(Fully dynamic labelling problem)* Let $G = (V, E)$ and $G' = (V, E')$ be two graphs, and G be changed to G' by edge insertions or deletions. The *fully dynamic labelling* problem is, given a distance labelling Γ over G such that $Q(u, v, \Gamma) = d_G(u, v)$ for any two vertices u and v in G, to compute a distance labelling Γ' over G' such that $Q(u, v, \Gamma') = d_{G'}(u, v)$ for any two vertices u and v in G'.

In the following, we will discuss how to tackle the fully dynamic labelling problem based on highway cover labelling.

4 Fully dynamic framework

In this section, we propose a fully dynamic framework for distance queries on large graphs. This framework consists of two novel dynamic algorithms: *incremental algorithm* and *decremental algorithm*, which efficiently update a highway cover labelling after edge insertions or edge deletions, respectively. We first introduce a key search strategy, i.e. *jumped-and-pruned search*, used by both incremental algorithm and decremental algorithm. Then, we present the algorithmic details of these two algorithms.

4.1 Jumped-and-pruned search

To efficiently reflect changes on graphs, we develop a jumped-and-pruned search strategy for updating distance labelling. This strategy requires us to identify two special types of vertices in a fully dynamic graph: *affected vertices* and *anchor vertices*.

4.1.1 Affected vertices

When an update operation occurs on a graph $G = (V, E)$, no matter whether it is an edge insertion or an edge deletion, there always exists a subset of "affected" vertices in V whose labels need to be updated as a consequence of this update operation on the graph. But, *can we identify such vertices efficiently?* To answer this question, we define the notion of affected vertices and analyse their properties.

Definition 5 *(Affected vertex)* Let $G = (V, E)$ and $R \subseteq V$ be a set of landmarks on G. A vertex $v \in V$ is *affected* by $G \hookrightarrow G'$ if $P_G(v, r) \neq P_{G'}(v, r)$ holds for at least one $r \in R$, and *unaffected* otherwise.

For simplicity, we use $\Lambda_r = \{v \in V | P_G(v, r) \neq P_{G'}(v, r)\}$ to denote the set of all affected vertices w.r.t. a landmark r and $\Lambda = \bigcup_{r \in R} \Lambda_r$ refers to the set of all affected vertices. Note that vertices affected by $G \hookrightarrow G'$ are the same as vertices affected by $G' \hookrightarrow G$, i.e. the same set of vertices is affected when inserting an edge (a, b) into a graph G or deleting an edge (a, b) from a graph G'.

Example 1 Consider Fig. 1a in which 0, 10 and 4 are three landmarks. After inserting an edge $(2, 5)$ in Fig. 1b–d, we have $\Lambda = \{0, 1, 2, 5, 8, 9, 10, 13, 14\}$ because $\Lambda_0 = \{5, 8, 9, 10, 13, 14\}$, $\Lambda_{10} = \{0, 1, 2\}$ and $\Lambda_4 = \{2\}$.

The following lemma states how affected vertices relate to an edge being inserted or deleted.

Lemma 1 *When $G \hookrightarrow G'$ for an edge insertion (a, b), a vertex $v \in \Lambda_r$ iff there exists a shortest path between v and r in G' passing through (a, b). Similarly, when $G \hookrightarrow G'$ for an edge deletion (a, b), a vertex $v \in \Lambda_r$ iff there exists a shortest path between v and r in G passing through (a, b).*

Proof For any vertex $v \in V$, if $d_{G'}(r, v) = d_{G'}(r, a) + d_{G'}(a, b) + d_{G'}(b, v)$ after inserting an edge (a, b) and $d_G(r, v) = d_G(r, a) + d_G(a, b) + d_G(b, v)$ after deleting an edge (a, b), then by Definition 5, $P_G(v, r) \neq P_{G'}(v, r)$. Thus, v is an affected vertex. □

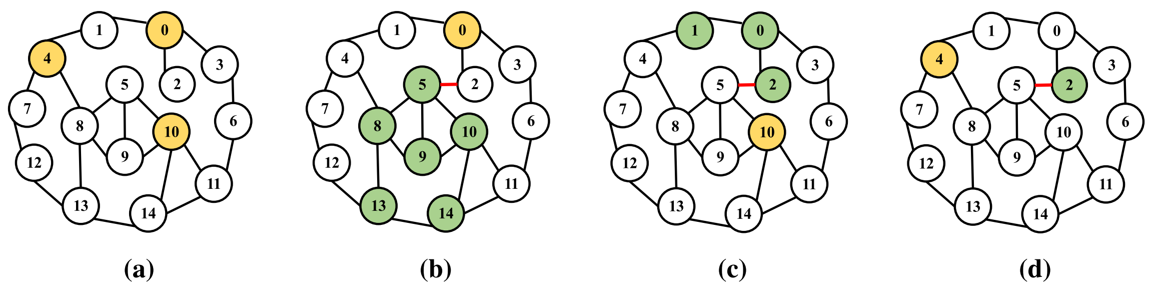

Fig. 1 An illustration of affected vertices by an edge insertion (2, 5) occurring on the graph in (**a**), where three landmarks 0, 10 and 4 are highlighted in yellow. In (**b**)–(**d**), the affected vertices w.r.t. the landmarks 0, 10 and 4 are highlighted in green, respectively. Note that the affected vertices w.r.t. the landmarks 0, 10 and 4 would remain the same if an edge deletion (2, 5) occurs on the graph in (**b**)

Following Lemma 1, we have the following corollary.

Corollary 1 *When $G \hookrightarrow G'$ with an inserted or deleted edge (a, b), if $d_G(r, a) = d_G(r, b)$ holds for a landmark $r \in R$, then we have $\Lambda_r = \emptyset$.*

This corollary allows us to reduce the search space of affected vertices by eliminating landmarks r with $d_G(r, a) = d_G(r, b)$. Thus, we assume that $d_G(r, b) > d_G(r, a)$ w.r.t. a landmark r in the rest of this section w.l.o.g.

The following lemma enables us to further reduce the search space of affected vertices by "jumping" from the root of a BFS to the vertex b.

Lemma 2 *When $G \hookrightarrow G'$ with an inserted or deleted edge (a, b), $d_G(r, v) \geq d_G(r, a) + 1$ hold for any affected vertex $v \in \Lambda_r$.*

Proof By Lemma 1, there exists a shortest path from any affected vertex v to r going through the affected edge (a, b) and thus through a. Since a is unaffected and the distance from a to v is equal to or greater than 1, $d_G(r, v) \geq d_G(r, a) + 1$ must hold. □

Remark 1 Algorithmically, by Corollary 1, we may design algorithms for finding affected vertices as follows. Let (a, b) be a graph change (either an edge insertion or an edge deletion) and $d_G(r, b) > d_G(r, a)$. (1) We start from the vertex b with distance $d_G(r, a) + 1$ on the changed graph to find affected vertices w.r.t. each landmark $r \in R$. (2) We identify the neighbours of b satisfying the condition in Lemma 2 as affected whose shortest path to r passes through the change. (3) We then iteratively keep examining the neighbours of already found affected vertices under the condition in Lemma 2 until all the affected vertices are found. This process corresponds to Lines 12–14 of Algorithm 1 for edge insertions and Lines 14–16 of Function `FindAffected` for edge deletion (will be further discussed in detail in Sects. 4.2 and 4.3).

4.1.2 Anchor vertices

Although efficiently identifying affected vertices is critical for dynamic algorithms, it is equally important to efficiently update the labels of affected vertices against changes on a graph. A naive approach is to run a full BFS from each landmark r on the changed graph in order to decide the new distances of affected vertices w.r.t. a landmark r. However, this is inefficient, particularly if only a very small portion of vertices in a graph is affected by an update operation. *Can we pinpoint the differences between the old labels of affected vertices in an original graph and their new labels in the changed graph, so as to change a distance labelling in an efficient way?* To answer this question, we need to identify a special kind of affected vertices, called *anchor vertices*, which have the smallest distance to a landmark r on the changed graph.

Definition 6 *(Anchor vertex)* When $G \hookrightarrow G'$, a vertex $v \in V$ is an *anchor vertex* w.r.t. a landmark r in G' if $v \in \Lambda_r$ and $d_{G'}(r, v) \leq d_{G'}(r, u)$ for any vertex $u \in \Lambda_r$.

The following lemma states that the exact distances of anchor vertices can be inferred from their unaffected neighbours. Note that this does not generally hold for every affected vertex. Let $d^*_{G'}(r, v)$ refer to a *contingent distance* between a landmark r and a vertex $v \in \Lambda_r$ in G', which is the minimum length of paths between v and r going through only unaffected vertices in G'. If a vertex $v \in \Lambda_r$ in G' has no any unaffected neighbours, we consider $d^*_{G'}(r, v) = \infty$.

Lemma 3 *When $G \hookrightarrow G'$, if a vertex $v \in \Lambda_r$ has the smallest contingent distance to a landmark r among all vertices in Λ_r, then v is an anchor vertex w.r.t. landmark r and $d_{G'}(r, v) = d^*_{G'}(r, v)$ holds.*

Proof We prove this by contradiction. Assume that $d_{G'}(r, v) \neq d^*_{G'}(r, v)$ for such a vertex v. Then, $d_{G'}(r, v) < d^*_{G'}(r, v)$ must hold because $d_{G'}(r, v)$ is the shortest path distance. Since $d^*_{G'}(v, r)$ is the minimum length of all paths between v and r that go through only unaffected vertices, one shortest path between v and r must go through at least one affected

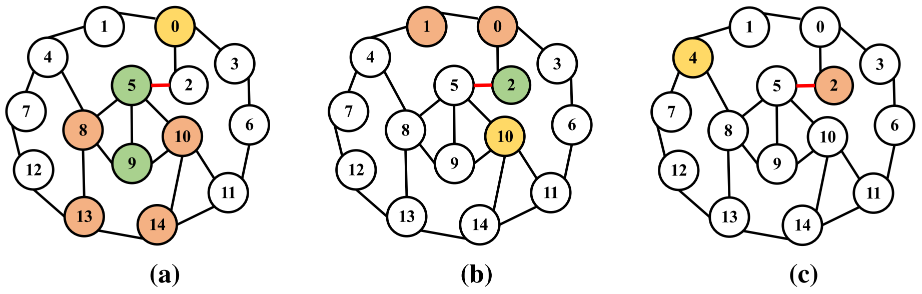

Fig. 2 An illustration of our incremental algorithm IncHL for an edge insertion (2,5): (**a**), (**b**) and (**c**) describe the JP-BFSs that are rooted at the landmarks 0, 10 and 4, respectively, where yellow vertices denote the landmarks (i.e. the roots), green colour denotes affected vertices whose labels are updated and red colour denotes affected vertices that are pruned during the JP-BFSs

vertex $v' \in \Lambda_r$ and $d^*_{G'}(v, r) > d^*_{G'}(v', r)$ must hold. This contradicts with the assumption that v has the minimum contingent distance to r in Λ_r. Hence, $d_{G'}(r, v) = d^*_{G'}(r, v)$. Accordingly, v must be an anchor vertex w.r.t. r. □

The observation here is that once anchor vertices are identified, we can *locally* infer their new distances from their unaffected neighbours. Then, new distances of other affected vertices can be inferred *inductively* by a level-by-level propagation in a BFS tree from r through unaffected neighbours and affected neighbours whose new distances have already been inferred.

Example 2 Firstly, we consider Fig. 2a in which the edge $(2, 5)$ is inserted. This causes the vertices $\{5, 8, 9, 10, 13, 14\}$ to be affected w.r.t. the landmark 0. Among these vertices, the vertex 5 has the smallest contingent distance (i.e. the distance through unaffected vertices) and thus is an anchor vertex. Now, we consider Fig. 3a in which the edge $(2, 5)$ is deleted. This causes the same set of vertices to be affected w.r.t. the landmark 0. However, in this case, the vertex 5 has the contingent distance ∞ because there is no path between vertex 5 and landmark 0 passing through only unaffected vertices. Instead, the vertices $\{8, 10\}$ have the smallest contingent distances and thus are anchor vertices in this case.

4.1.3 Jumped-and-pruned BFS

Our dynamic algorithms, including both incremental and decremental algorithms, use a *jumped-and-pruned* search strategy to efficiently update a distance labelling. The key idea is that, instead of conducting a full BFS from a landmark to all vertices, we conduct a partial BFS (named as JP-BFS) that jumps from the root of a BFS directly to affected vertices, thereby skipping unaffected vertices. Further, a JP-BFS exploits the property of highway cover labelling (i.e. an distance entry of a vertex v w.r.t. a landmark r can be pruned if there is another landmark lying in a shortest path between v and r) to prunes affected vertices as many as possible after its jump.

Definition 7 *(Prunable vertex)* When $G \hookrightarrow G'$, a vertex v is *prunable* w.r.t. a landmark r iff there exists a landmark $r' \in R - \{r\}$ such that all of the following conditions hold:

(1) $d_G(r, v) = d_G(r, r') + d_G(r', v)$;
(2) $d_{G'}(r, v) = d_{G'}(r, r') + d_{G'}(r', v)$;
(3) $d_G(r', v) = d_{G'}(r', v)$.

A vertex v is *weakly prunable* iff it only satisfies the conditions (2) and (3).

Intuitively, the conditions in the above definition state that we can prune a vertex v only if there is another landmark r' lying in a shortest path between this vertex and the landmark r in both G and G', i.e. (1) and (2), and the distance from this vertex to r' also remains the same in both G and G', i.e. (3). A vertex v satisfying these three conditions implies that its label w.r.t. landmark r remains the same in G and G', and a JP-BFS can thus prune v as well as the children of v from search. When a vertex v is weakly prunable, it means that the label of v may contain outdated or redundant entries w.r.t. landmark r, which would affect the correctness of a distance labelling in the case of edge deletion but not edge insertion. In fact, the case of weakly prunable vertices can only occur during edge insertion due to newly added shortest path(s).

The following example illustrates the notions of prunable vertex and weakly prunable vertex. We will discuss further how vertices are pruned during a JP-BFS in our dynamic algorithms in Sects. 4.2 and 4.3.

Example 3 Let us consider the vertex 8 in Fig. 2a which is pruned because it satisfies all three conditions in Definition 7. We can see that the path $\langle 0, 1, 4, 8\rangle$ exists before and after adding the edge $(2, 5)$ and passes through the landmark 4 satisfying all the conditions (1), (2) and (3). For the vertex 14 in Fig. 2a, it is weakly pruned due to the newly added path $\langle 0, 2, 5, 10, 14\rangle$ through landmark 10 that did not exist before adding the edge $(2, 5)$ thus satisfying only the conditions (2) and (3). Now, we consider the pruned vertices

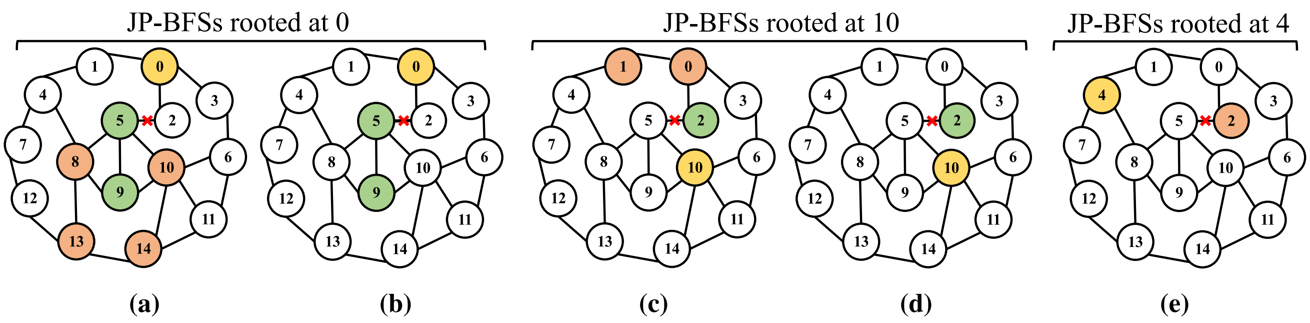

Fig. 3 An illustration of our decremental algorithm DecHL for an edge deletion (2, 5): (**a**)–(**b**), (**c**)–(**d**) and (**e**) describe the JP-BFSs that are rooted at the landmarks 0, 10 and 4, respectively, where yellow vertices denote the landmarks (i.e. the roots), green colour denotes affected vertices whose labels need to be updated, and red colour denotes affected vertices being pruned

highlighted in Fig. 3a. All of these vertices satisfy the three conditions in Definition 7, e.g. we prune from 10 because a path $\langle 0, 3, 6, 10\rangle$ exists between 0 and 10 before and after deleting the edge (2, 5).

In a nutshell, a JP-BFS has the following two features: (1) *jumping* from the root (i.e. a landmark) to affected vertices so as to traverse locally, rather than globally; (2) *pruning* affected vertices that are prunable or weakly prunable whenever possible.

4.1.4 Algorithmic design

Before introducing our dynamic algorithms in detail (as will be shown in Sects. 4.2 and 4.3), we briefly discuss how this jumped-and-pruned search strategy is applied in these algorithms.

In the most general case, two kinds of JP-BFS are needed. One kind of JP-BFS is to *identify affected vertices w.r.t. a landmark* r. By Lemmata 1 and 2, such a JP-BFS jumps from the root r to the vertex b and starts to identify affected vertices iteratively through checking neighbours and their old distances. The other kind of JP-BFS is to *update affected vertices w.r.t. a landmark* r. By Lemma 3, such a JP-BFS jumps from the root r to anchor vertices and starts to update the labels of affected vertices through a level-by-level propagation in order to infer the new distances of these affected vertices.

Nonetheless, we notice the following:

– In the case of edge insertion, there exists exactly one anchor vertex for an inserted edge; further, such an anchor vertex can be easily identified according to the inserted edges. This enables an efficient design for our incremental algorithm which can not only identify affected vertices, but also simultaneously update the labels of affected vertices through a carefully designed propagation on new distances. Hence, instead of conducting two separate JP-BFSs, our incremental algorithm merges these two JP-BFSs into one JP-BFS for improving efficiency. Section 4.2 will discuss how the merged JP-BFS is designed in our incremental algorithm (i.e. Algorithm 1).
– In the case of edge deletion, finding anchor vertices turns out to be challenging. For a deleted edge, there may exist multiple anchor vertices; further, these anchor vertices can be far away from the deleted edge and cannot be identified without knowing a full picture on how vertices are affected by the deleted edge. Hence, our decremental algorithm must first find affected vertices in the first JP-BFS, which leads to identifying anchor vertices, and then update the labels of affected vertices in the second JP-BFS based on the information about anchor vertices and affected vertices obtained from the first JP-BFS. Section 4.3 will discuss further on how these two separate JP-BFS are designed in our decremental algorithm (i.e. Algorithm 2).

4.2 Incremental algorithm

We now present our incremental algorithm, called IncHL. This algorithm can efficiently update a highway cover labelling to reflect changes caused by an edge insertion.

We start with the following lemma that characterises anchor vertices in the case of edge insertion.

Lemma 4 *For* $G \hookrightarrow G'$ *with an edge insertion* (a, b)*, if* $d_G(r, a) < d_G(r, b)$ *holds for a landmark* $r \in R$*, then* b *must be the only anchor vertex w.r.t. the landmark* r*.*

By the above lemma, since b is the only anchor vertex in the case of edge insertion, our incremental algorithm IncHL is carefully designed to merge two JP-BFSs (i.e. one for identifying affected vertices and the other for updating the labels of affected vertices) into one JP-BFS to identify affected vertices and updates their labels simultaneously. This significantly improves update efficiency for edge insertion. Another insight we obtain is that a large amount of weakly prunable

Algorithm 1: IncHL

```
Input: $G = (V, E), (a, b) \notin E, \Gamma = (H, L)$
Output: $\Gamma = (H, L)$
1  foreach $r \in R$ with $d_G(r, b) > d_G(r, a)$ do
2      $\mathcal{Q} \leftarrow \emptyset, V_r^{infer} \leftarrow \emptyset, \pi \leftarrow d_G(r, a) + 1$
3      Enqueue $(b, \pi)$ to $\mathcal{Q}$
4      while $\mathcal{Q}$ is not empty do
5          Dequeue $(v, \pi)$ from $\mathcal{Q}$
6          if $v$ is prunable then
7              if $v$ is a landmark then
8                  $\delta_H(r, v) \leftarrow \pi$ (Updating $H$)
9              end
10         else
11             Update($r, v, \pi, V_r^{infer}, \emptyset$)
12             foreach $w \in N(v)$ and $d_G(r, w) > \pi$ do
13                 Enqueue $(w, \pi + 1)$ to $\mathcal{Q}$
14             end
15         end
16         Add $(v, \pi)$ to $V_r^{infer}$
17     end
18 end
```

```
1  Function Update ($r, v, \pi, V_r^{infer}, V_r^{uninfer}$)
2      $V_{parent}(v) = \{w | w \in N(v)$ and $((w, \pi) \in V_r^{infer}$ or
       $((w, \pi') \notin V_r^{uninfer}$ and $Q(r, w, \Gamma) = \pi - 1))\}$
3      if $\forall w \in V_{parent}(v)$ s.t. $r$ appears in $L(w)$ then
4          Add/Modify $(r, \pi)$ to $L(v)$
5      else
6          Remove $r$ from $L(v)$ (if exists)
7      end
8  end
```

vertices can be pruned away during this merged JP-BFS to further considerably improve update efficiency.

Algorithm 1 describes the detailed steps of our incremental algorithm. Given a graph G with an inserted edge (a, b) and a highway cover labelling $\Gamma = (H, L)$ over G, we conduct one JP-BFS for each landmark $r \in R$ starting from the vertex b with its new distance $\pi = Q(r, a, \Gamma) + 1$, and enqueue (b, π) into $\mathcal{Q}$ (Lines 1–3, Algorithm 1). To identify all affected vertices and update their labels, this JP-BFS works as follows. For every $(v, \pi) \in \mathcal{Q}$, if v is prunable, then we stop the search from v by simply updating the highway H (Lines 6–9, Algorithm 1). This also eliminates weakly prunable vertices, and we will prove this in Sect. 5. Otherwise, we update the label of v in Function `Update` using the neighbours of v that appear in at least one shortest path between v and r in the changed graph G', i.e. $V_{parent}(v)$ (Line 2, `Update`). Based on $V_{parent}(v)$, we update $L(v)$ as follows. If there exists at least one vertex $w \in V_{parent}(v)$ that does not contain r in its label, then there must exist another landmark in a shortest path between v and r, and we thus remove r from $L(v)$ if exists (Line 6, `Update`); otherwise, we add/modify (r, π) in the label of v (Line 4, `Update`). After updating the label of v, we enqueue all affected neighbours of v into $\mathcal{Q}$ with new distances $\pi + 1$ (Lines 12–14, Algorithm 1) and add (v, π) to V_r^{infer}, where V_r^{infer} contains the set of affected vertices w.r.t. the landmark r whose new distances have been inferred (Line 16, Algorithm 1). This process of identifying affected vertices and updating their labels continues until $\mathcal{Q}$ is empty.

Example 4 Figure 2 illustrates how our incremental algorithm updates affected labels as a result of inserting an edge (2, 5). The JP-BFS starting from the anchor vertex 5 w.r.t. the landmark 0 is depicted in Fig. 2a. The labels of vertices 5 and 9 are updated using the information in the labels of their parents, i.e. $V_{parent}(5) = \{2\}$ and $V_{parent}(9) = \{5\}$. This JP-BFS is pruned from vertices 8 and 10 because the landmark 4 lies in the shortest path from vertex 8 to landmark 0, and vertex 10 is a landmark itself. Accordingly, the highway is updated. Similarly, the JP-BFS w.r.t. the landmark 10 is depicted in Fig. 2b. This JP-BFS starts from vertex 2 which updates the label of 2 with $V_{parent}(2) = \{5\}$ and prunes from the landmark 0 after updating the highway. The JP-BFS w.r.t. the landmark 4 is depicted in Fig. 2c, which works in a similar fashion.

Outdated and redundant entries may exist in a distance labelling due to edge insertions. For clarity, we formally define the notions of outdated and redundant entries below.

Definition 8 *(Outdated entry)* An entry $(r, \delta_L(r, v)) \in L(v)$ is *outdated* on a graph G iff $\delta_L(r, v) \neq d_G(r, v)$.

Definition 9 *(Redundant entry)* An entry $(r, \delta_L(r, v)) \in L(v)$ is *redundant* on a graph G iff $\delta_L(r, v) = d_G(r, v)$ and $Q(u, v, \Gamma) = Q(u, v, \Gamma')$ hold for any vertex u in G, where Γ' is obtained from Γ by only removing $(r, \delta_L(r, v))$ from $L(v)$.

These entries do not affect the correctness of answering distance queries when a graph is changed only by edge insertions [19]. However, they may deteriorate query and update performance over time. Thus, to ensure that neither outdated nor redundant entries exist, we can revise the design of a merged JP-BFS in IncHL by removing its pruning step (Line 6, Algorithm 1), which leads to a distance labelling without any outdated and redundant entries. This variant of IncHL is called IncHL- m. The proof for the preservation of minimality by IncHL- m is provided in Sect. 5.

4.3 Decremental algorithm

We also propose a decremental algorithm, called DecHL, which can efficiently update a highway cover labelling to reflect changes caused by an edge deletion.

Different from edge insertion, by Fact 2, distances between vertices may increase in the case of edge deletion.

Algorithm 2: DecHL

```
Input: $G = (V, E \cup \{(a,b)\}), (a,b) \in E, \Gamma = (H, L)$
Output: $\Gamma = (H, L)$
foreach $r \in R$ with $d_G(r,b) > d_G(r,a)$ do
    $V_r^{infer} \leftarrow \emptyset, V_r^{uninfer} \leftarrow$ FindAffected$(r, b)$
    foreach $(v, \pi) \in V_r^{uninfer}$ with min. $\pi$ do
        if $v$ is already pruned then
            if $v$ is a landmark then
                $\delta_H(r, v) \leftarrow \pi$ (Updating $H$)
            end
        else
            Update$(r, v, \pi, V_r^{infer}, V_r^{uninfer})$
        end
        foreach $w \in N(v)$ and $(w, \pi') \in V_r^{uninfer}$ and $\pi' < \pi + 1$ do
            Modify $(w, \pi')$ with $(w, \pi + 1)$ in $V_r^{uninfer}$
        end
        Remove $(v, \pi)$ from $V_r^{uninfer}$ to $V_r^{infer}$
    end
end
```

```
Function FindAffected$(r, b)$
    $\mathcal{Q} \leftarrow \emptyset, V_{aff} \leftarrow \emptyset, \pi' \leftarrow Q(r, b, \Gamma)$
    Enqueue $(b, \pi')$ to $\mathcal{Q}$
    while $\mathcal{Q}$ is not empty do
        Dequeue $(v, \pi')$ from $\mathcal{Q}$
        $\pi \leftarrow \infty$
        if $\exists w$ s.t. $w \in N(v)$ and $d_G(r, w) \leq \pi'$ and $w \notin V_{aff}$ then
            $\pi \leftarrow \min_w\{Q(r, w, \Gamma)\} + 1$
        end
        Add $(v, \pi)$ to $V_{aff}$
        if $v$ is prunable then
            continue
        else
            foreach $w \in N(v)$ and $d_G(r, w) > \pi'$ do
                Enqueue $(w, \pi' + 1)$ to $\mathcal{Q}$
            end
        end
    end
    return $V_{aff}$
end
```

This thus poses the following new challenges. First, outdated and redundant entries do affect the correctness of answering distance queries in the case of edge deletion. Second, identifying anchor vertices becomes much harder for edge deletion due to two reasons: (1) More than one anchor vertex may exist w.r.t. a landmark for edge deletion, in contrast to edge insertion which has exactly one anchor vertex w.r.t. a landmark; (2) anchor vertices can be far away from a deleted edge and are thus difficult to identify, whereas for an inserted edge there exists exactly one anchor vertex that must be incident to the inserted edge, as stated in Lemma 4.

Example 5 Consider Fig. 3a, after deleting the edge (2, 5), the set of affected vertices w.r.t. the landmark 0 is {5, 8, 9, 10, 13, 14}. Among these vertices, the vertices {8, 10, 13, 14} are pruned. In Fig. 3b, there are two anchor vertices {8, 10} w.r.t. the landmark 0. None of these vertices 8 and 10 are incident to the deleted edge (2, 5).

Since anchor vertices for a deleted edge (a, b) may be different from the vertex b (recall that $d_G(r, b) > d_G(r, a)$ is assumed), we need to conduct two JP-BFSs w.r.t. a landmark r in the case of edge deletion. The first JP-BFS starts from b to identify affected vertices and their contingent distances through local neighbourhoods iteratively. The second JP-BFS starts from anchor vertices to infer new distances of affected vertices and update their labels via a level-by-level propagation. Prunable vertices are identified and pruned away in the first JP-BFS, which helps improve update efficiency in the second JP-BFS significantly.

Algorithm 2 describes the detailed steps of our decremental algorithm. Given a graph G with a deleted edge (a, b) and a highway cover labelling $\Gamma = (H, L)$ over G, we conduct the first JP-BFS w.r.t. a landmark $r \in R$ in Function FindAffected starting from vertex b with $\pi = Q(r, b, \Gamma)$, and enqueue (b, π) to $\mathcal{Q}$ (Line FindAffected). Then, for every $(v, \pi') \in \mathcal{Q}$, if a neighbour w of v is unaffected and has a depth less than or equal to π', we compute the contingent distance π of v w.r.t. r based on the distance between w and r (Lines 7–9, FindAffected) and add (v, π) into V_{aff} (Line 10, FindAffected). If v is prunable, we stop the search from v; otherwise, we continue to traverse all the children of v and enqueue them to $\mathcal{Q}$ as affected vertices (Lines 11–16, FindAffected). This process continues iteratively until $\mathcal{Q}$ is empty. Thus, the first JP-BFS identifies affected vertices to be updated, as well as their contingent distances to r in V_{aff}. If contingent distances of all vertices in V_{aff} are ∞, we remove the distance entry of r from the labels of these vertices because the deleted edge must cut all these vertices off from other vertices as being disconnected. Otherwise, we return V_{aff} and perform the second JP-BFS.

The second JP-BFS starts from the anchor vertices which are the vertices in V_{aff} with the minimum contingent distance (Line 3, Algorithm 2) and infers new distances of affected vertices iteratively. At each iteration, for each $(v, \pi) \in V_r^{uninfer}$ with the minimum contingent distance π, if v is already pruned, then if it is a landmark, we update the highway H; otherwise, we update the label of v using Function Update (Lines 4–10, Algorithm 2). After that, we update the contingent distances of the affected neighbours of v (Lines 11–13, Algorithm 2). Next, we remove v from $V_r^{uninfer}$ to V_r^{infer} meaning that the new distance of v w.r.t. r has been inferred so that $V_r^{uninfer}$ contains only affected vertices whose new distances have not been inferred. This process continues until the new distances of all affected vertices in V_{aff} are inferred and updated.

Example 6 Figure 3 illustrates how our decremental algorithm updates affected labels as a result of deleting an edge (2, 5). Figure 3a depicts the first JP-BFS w.r.t. the landmark 0 for finding affected vertices. This JP-BFS identifies affected vertices {5, 8, 9, 10, 13, 14}, among which vertices {8, 10, 13, 14} are pruned. Then, the second JP-BFS w.r.t. the landmark 0 for updating the labels is depicted in Fig. 3b. This JP-BFS starts from anchor vertices {8, 10} with the minimum contingent distances. Then, it moves to the affected neighbours {5, 9} and updates their labels using the information in the labels of their parents, i.e. $V_{parent}(5) = \{8, 10\}$ and $V_{parent}(9) = \{8, 10\}$. Similarly, the first JP-BFS w.r.t. the landmark 10 is depicted in Fig. 3c which identifies affected vertices {0, 1, 2}, among which {0, 1} are pruned. The second JP-BFS w.r.t. the landmark 10 is depicted in Fig. 3(d) which starts from anchor vertex {0} with the minimum contingent distance and moves to the affected neighbour {2} and update its label using the information in $V_{parent}(2) = \{0\}$. Next, the JP-BFSs w.r.t. the landmarks 4 are depicted in Fig. 2(e), respectively, which work in the same manner.

5 Theoretical results

In this section, we prove the proposed fully dynamic method to be (1) correct, i.e. after each update operation, queries on the updated labelling return exact distances, and (2) able to preserve minimality of the labelling, a desirable property that has impacts on both query time and space efficiency. Then, we briefly analyse the complexity of the proposed algorithms.

5.1 Proof of correctness

Let $G_1 \hookrightarrow G_2 \ldots, \hookrightarrow G_n$ by a sequence of update operations (edge insertions or edge deletions). Our fully dynamic method, denoted as FulHL, is to update a highway cover labelling Γ_1 over G_1 into a highway cover labelling Γ_n over G_n such that IncHL and DecHL are applied for edge insertions and edge deletions, respectively. We consider FulHL to be *correct* iff, whenever $Q(u, v, \Gamma_1) = d_{G_1}(u, v)$ holds for any two vertices u and v in G_1, $Q(u, v, \Gamma_n) = d_{G_n}(u, v)$ also holds for any two vertices u and v in G_n.

Below, we first prove that Lemmata 5- 7 hold for both algorithms IncHL and DecHL. For simplicity, let G' refer to the changed graph after applying an edge insertion or deletion on a graph G in the input of IncHL and DecHL.

Lemma 5 *A pair (v, π) appears in $\mathcal{Q}$ iff $v \in \Lambda_r$.*

Proof IncHL (Lines 12–14, Algorithm 1) guarantees that an inserted edge (a, b) is in one shortest path between any vertex added to $\mathcal{Q}$ and a landmark r in G'. Similarly, DecHL (Lines 14–16, `FindAffected`) guarantees that a deleted edge (a, b) is in one shortest path between any vertex added to $\mathcal{Q}$ and a landmark r in G. From Lemma 1, we thus have that a vertex is added to $\mathcal{Q}$ iff $v \in \Lambda_r$. □

Lemma 6 *A pair (v, π) is in V_r^{infer} iff $\pi = d_{G'}(r, v)$.*

Proof In both IncHL (Lines 5 and 16, Algorithm 1) and DecHL (Lines 3 and 14, Algorithm 2), (v, π) is added into V_r^{infer} iff v has its new distance being inferred from unaffected and affected vertices whose new distances have already been inferred w.r.t. a landmark r in G'. Following the proof for Lemma 3, we can thus prove $\pi = d_{G'}(r, v)$. □

Lemma 7 *In Function `Update`, V_{parent} is sufficient and necessary to update $L(v)$.*

Proof In both IncHL and DecHL, V_r^{infer} contains all inferred vertices which lie in the shortest path(s) between r and v, and whose new distances to r have been correctly inferred in G' (according to Lemma 6). Therefore, Line 2 in Function `Update` ensures that V_{parent} consists of both unaffected and affected vertices that are parents of v w.r.t. r in G', which is sufficient and necessary to update $L(v)$. □

Based on Lemmata 5-7, the definitions of prunable and weakly prunable vertices, and the fact that a highway H is updated by Algorithm 1 (Line 8) and Algorithm 2 (Line 6), respectively, the following theorem can be proven.

Theorem 1 FulHL *is correct.*

5.2 Preservation of minimality

It has been reported in [22] that, given a graph G, a highway cover labelling $\Gamma = (H, L)$ over G can be constructed using an algorithm proposed in their work and such a highway cover labelling Γ is also guaranteed to be minimal in terms of the labelling size, i.e. $size(L') \geq size(L)$ holds for any $\Gamma' = (H, L')$ over G. Following Lemmata 5-7 and Fact 2, we can prove the following theorem.

Theorem 2 *When $G_1 \hookrightarrow G_2 \ldots, \hookrightarrow G_n$, let* IncHL- m *and* DecHL *update a highway cover labelling Γ_1 over G_1 into a highway cover labelling Γ_n over G_n for edge insertions and edge deletions, respectively. If Γ_1 is minimal over G_1, then Γ_n is also minimal over G_n.*

We use FulHL- m to refer to our fully dynamic method that preserves the minimality of labelling. More specifically, for $G_1 \hookrightarrow G_2 \ldots, \hookrightarrow G_n$ by a sequence of update operations (edge insertions or edge deletions), FulHL- m updates a highway cover labelling Γ_1 over G_1 into a highway cover labelling Γ_n over G_n such that IncHL- m and DecHL are applied for edge insertions and edge deletions, respectively.

5.3 Complexity analysis

Let m be the total number of affected vertices, l be the average size of labels (i.e. $l = size(L)/|V|$) and d be the average degree. For each landmark, IncHL- m and IncHL visit $O(m)$ affected vertices in the worst case, update each affected label by checking d neighbours in $O(dl)$ time. Thus, the time complexity of IncHL- m and IncHL is $O(|R| \times mdl)$. On the other hand, DecHL takes $O(mdl)$ time to find all affected vertices with their contingent distances (in Function `FindAffected`) and takes $O(md)$ to fix the labels of all affected vertices. We omit l from $O(md)$ for Algorithm 2 because distances for all unaffected neighbours of affected vertices can be stored during the first JP-BFS to avoid query cost while updating the labels during the second JP-BFS. Thus, the time complexity of DecHL is $O(|R| \times md(l+1))$. In our experiments, we notice that m is usually orders of magnitudes smaller than the total number of vertices $|V|$, while l is significantly smaller than $|R|$.

6 Experiments

In this section, we evaluate our dynamic algorithms, including the incremental and decremental ones, aiming to answer the following questions:

- How efficiently can our dynamic algorithms deal with updates in very large dynamic networks, in comparison with the state-of-the-art methods?
- How does the number of landmarks affect the performance of our dynamic algorithms?
- How do affected vertices correlate to update efficiency in our dynamic algorithms?
- How well can our dynamic algorithms scale to deal with updates in very large dynamic networks?

6.1 Experimental setup

We first present the datasets, then discuss how updates and queries are generated and introduce the baseline methods considered in our experiments.

6.1.1 Datasets

We used 13 real-world large networks in our experiments, in order to empirically verify the efficiency, scalability and robustness of our algorithms. These networks are accessible at Stanford Network Analysis Project [33], Laboratory for web Algorithmics [8,10], Koblenz Network Collection [30] and Network Repository [44]. We treated these networks as undirected and unweighted graphs. The types of networks, number of vertices and edges and statistical information of these network datasets are summarized in Table 2, while a brief description of each dataset is given below:

- **Skitter:** This is an Internet topology network, obtained by running daily traceroute in 2005 [31], in which nodes represent routers and edges represent communication links.
- **Flickr:** This is a social network of users and their connections in a photo sharing website Flickr (www.flickr.com) [36].
- **Hollywood:** This is a social network of movie actors, where vertices represent actors and two actors are joined by an edge whenever they appeared in a movie together in 2009 [8,10].
- **Orkut:** This is a social network of users and their connections in a social networking website, Orkut (www.orkut.com) [36].
- **Enwiki:** This is a network of hyperlinks from a snapshot of English Wikipedia, obtained in 2013, where vertices represent pages and edges indicate hyperlinks between pages [8,10].
- **Livejournal:** This is a social network which allows its members to manage their journals and blogs, and to declare which other members and their friends they belong to, in an online social website (www.livejournal.com) [5,32].
- **Indochina:** This is a web graph of web pages, obtained by performing a large crawl of the country domains of Indochina in 2004 for the Nagaoka University of Technology [8,10].
- **IT:** This is a web graph, obtained by performing a fairly large crawl of the .it domain in 2004 [8,10].
- **Twitter:** This is a social network with information about who follows whom on Twitter, where vertices represent users and edges represent follow relationships between users, in an online social website (www.twitter.com) [8,10].
- **Friendster:** This is a social gaming network, where users are connected with friendship relationships, in an online website (www.friendster.com) [53].
- **UK:** This is a web graph which is part of a time-aware network, obtained by collecting monthly snapshots of the .uk domain for twelve months in 2006 and 2007 [9].
- **Clueweb09:** This is a web graph of web pages in ten languages collected in January and February 2009, where nodes are unique URLs (pages) and edges represent links between pages [44].
- **Clueweb12:** This dataset is a successor to the Clueweb09 dataset. It was obtained by crawling the web for about 1 billion English web pages between February 10, 2012, and May 10, 2012 [8,10].

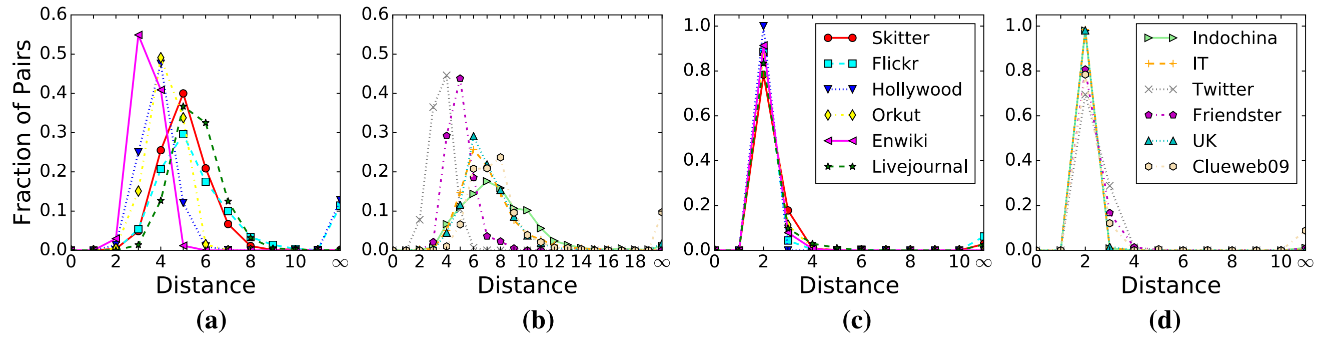

Fig. 4 (**a**)–(**b**) show the distance distribution of 1000 pairs of vertices in E_I on all the datasets, where the distance for each pair is recorded before insertion, and (**c**)–(**d**) show the distance distribution of 1000 pairs of vertices in E_D on all the datasets, where the distance for each pair is recorded after deletion

6.1.2 Generation of updates and queries

For each network $G = (V, E)$, we randomly sampled 1,000 pairs of vertices as edge insertions, denoted as E_I, where $E_I \cap E = \emptyset$, and 1,000 pairs of vertices as edge deletions, denoted as E_D, where $E_D \subseteq E$. We use E_I and E_D to evaluate incremental and decremental algorithms, respectively. Then, we randomly selected 1,000 pairs of vertices $E_F \subseteq E_I \cup E_D$ with 50% from E_I (edge insertions) and 50% from E_D (edge deletions) to evaluate fully dynamic methods. The distance distribution before applying updates in E_I is shown in Fig. 4a, b, and the distance distribution after applying the updates in E_D is shown in Fig. 4c, d. We can see that most of the pairs have a small distance ranging from 1 to 10 in E_I and from 1 to 4 in E_D for most of the datasets and only a few of them are disconnected (i.e. have distance ∞).

Furthermore, we randomly sampled 100,000 pairs of vertices in each network as queries, to evaluate the query performance on graphs being changed by updates in E_F. Table 2 shows the average distance in each network using these 100,000 randomly sampled pairs of vertices. We can see that most of these networks have a small average distance. We also report the labelling size produced by fully dynamic methods after performing updates in E_F.

6.1.3 Baseline methods

We compared our fully dynamic methods (i.e. FulHL- m and FulHL), as well as the incremental and decremental algorithms (IncHL, IncHL- m and DecHL), with the following state-of-the-art methods:

(1) The dynamic algorithms IncFD, DecFD and FulFD proposed in [26], which combine a distance labelling with a graph traversal algorithm for answering distance queries;
(2) The dynamic algorithms IncPLL [3], DecPLL [19] and FulPLL proposed in [3,19], which are based on the pruned landmark labelling (PLL) [2] to answer distance queries;

Table 2 Datasets, where $|G|$ denotes the size of a graph G with each edge appearing in the forward and reverse adjacency lists and being represented by 8 bytes

Dataset	Network	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	avg. deg.	max. deg.	avg. dist.	$\|G\|$
Skitter	comp (u)	1.7M	11M	6.54	13.081	35455	5.1	85 MB
Flickr	social (u)	1.7M	16M	9.07	18.133	27224	5.3	119 MB
Hollywood	social (u)	1.1M	114M	49.5	98.913	11467	3.9	430 MB
Orkut	social (u)	3.1M	117M	38.1	76.281	33313	4.2	894 MB
Enwiki	social (d)	4.2M	101M	21.9	43.746	432260	3.4	701 MB
Livejournal	social (d)	4.8M	69M	8.84	17.679	20333	5.6	327 MB
Indochina	web (d)	7.4M	194M	20.4	40.725	256425	7.7	1.1 GB
IT	web (d)	41M	1.2B	24.9	49.768	1326744	7.0	7.7 GB
Twitter	social (d)	42M	1.5B	28.9	57.741	2997487	3.6	9.0 GB
Friendster	social (u)	66M	1.8B	27.4	55.056	5214	5.0	13 GB
UK	web (d)	106M	3.7B	31.4	62.772	979738	6.9	25 GB
Clueweb09	web (d)	1.7B	7.8B	4.64	9.27	6444720	7.4	58.2 GB
Clueweb12	web (d)	~1B	43B	39.1	78.249	75611696	5.2	279 GB

Table 3 Comparison of the update time, labelling size and query time of the proposed methods with the baseline methods, where "-" denotes that a method did not finish to construct labelling in one day (24 hours) or ran out of memory (512 GB)

Dataset	Update Time (ms)				Labelling Size			Query Time (ms)		
	FULHL- M	FULHL	FULFD	FULPLL	FULHL	FULFD	FULPLL	FULHL	FULFD	FULPLL
Skitter	1.096	1.019	10.67	20400	42 MB	153 MB	11.6 GB	0.027	0.019	0.006
Flickr	0.055	0.053	7.655	6810	34 MB	152 MB	12.7 GB	0.007	0.012	0.009
Hollywood	0.223	0.212	10.54	–	27 MB	262 MB	–	0.027	0.037	–
Orkut	1.234	1.075	40.12	–	70 MB	711 MB	–	0.101	0.103	–
Enwiki	1.488	1.459	88.54	–	82 MB	608 MB	–	0.054	0.035	–
Livejournal	0.275	0.179	2.564	–	122 MB	662 MB	–	0.044	0.046	–
Indochina	1.414	0.598	107.2	–	87 MB	840 MB	–	0.737	0.839	–
IT	22.96	10.62	160.3	–	862 MB	4.74 GB	–	1.069	1.013	–
Twitter	73.37	72.76	2512	–	1.14 GB	3.83 GB	–	0.863	0.177	–
Friendster	2.131	2.097	21.64	–	2.43 GB	9.14 GB	–	0.814	0.904	–
UK	2.755	1.075	337.6	–	1.78 GB	11.8 GB	–	3.443	5.858	–
Clueweb09	103.1	56.25	–	–	163 GB	–	–	16.93	–	–
Clueweb12	15950	1796	–	–	49.1 GB	–	–	9.375	–	–

(3) The online incremental algorithm INCHL$^+$ proposed in [21], which combines a highway cover labelling with a graph traversal algorithm for answering distance queries;
(4) The parallel pruned landmark labelling methods PSL, PSL$^+$ and PSL* for static graphs proposed in [34], which are also based on the pruned landmark labelling (PLL) [2] to answer distance queries;
(5) The optimized online bidirectional BFS algorithm which answers distance queries by applying an optimized strategy to expand search from the direction with less vertices [26], and we name this algorithm OPT- BIBFS in our experiments.

The implementations of these baseline methods were provided by their authors and are in C++. We used the same parameter settings for these methods as suggested by their authors unless otherwise stated. The initial distance labellings were constructed using their original static methods, i.e. HL for FULHL [22], FD for FULFD [26] and PLL for FULPLL [2,3,19]. For a fair comparison, we set the number of landmarks to 20 for our methods, following the same setting of FULFD [26] except for the largest datasets, i.e. Clueweb09 and Clueweb12. We set the number of landmarks to 150 for Clueweb09 and Clueweb12 because a small number of landmarks on such large networks do not help much in pruning the search space. For parallel PLL methods PSL, PSL$^+$ and PSL* [34], we set the number of threads to the total number of available cores in our server, i.e. 28. We developed our software in C++11 using STL libraries, compiled with gcc 5.5.0 with the -O3 option. We executed all of our experiments using a single thread on a Linux server (Intel Xeon W-2175 with 2.50GHz (CPU) and 512GB of main memory).

6.2 Performance comparison

We first compare our methods against the labelling-based methods in terms of update time, labelling size and query time. After that, we compare our methods against online search methods.

6.2.1 Labelling-based dynamic methods

Update Time. Table 3 shows that the average update times taken by our methods FULHL- M and FULHL are significantly less than the average update times taken by the baseline methods FULFD and FULPLL. As we can see, only our methods can scale to very large networks with billions of vertices and edges. Specifically, FULFD failed to have results for Clueweb09 and Clueweb12, and FULPLL failed for 11 out of 13 networks. There are several reasons why FULPLL cannot scale to large networks. Firstly, FULPLL is based on the pruned landmark labelling algorithm [2] which has very high space requirements and construction time of labelling on large networks. Secondly, FULPLL has a very high updating cost in restoring the 2-hop cover property [19] for the decremental case. Overall, our methods are more than 30 times faster as compared to FULFD and several orders of magnitude faster than FULPLL.

In Table 4, the average update times taken by incremental and decremental algorithms are compared separately. For incremental algorithms, our methods INCHL- M and INCHL significantly outperform the baseline methods INCHL$^+$, INCFD and INCPLL on all the datasets. Further, INCHL is faster than INCHL- M which strictly preserves the minimality of labelling. This performance difference between IncHL and

Table 4 Comparison of the update time for edge insertion and edge deletion of the proposed methods IncHL- m, IncHL and DecHL with the baseline methods

Dataset	Incremental Algorithms					Decremental Algorithms		
	IncHL- m (ms)	IncHL (ms)	IncHL$^+$ (ms)	IncFD (ms)	IncPLL (ms)	DecHL (ms)	DecFD (ms)	DecPLL (sec.)
Skitter	0.133	0.075	0.194	0.447	2.189	1.443	19.48	21.3
Flickr	0.005	0.005	0.006	0.046	1.869	0.152	17.71	11.7
Hollywood	0.027	0.026	0.031	0.078	48.97	0.265	21.03	–
Orkut	1.687	1.423	2.026	2.039	–	0.418	48.12	–
Enwiki	0.119	0.105	0.134	0.129	6.596	2.969	163.8	–
Livejournal	0.201	0.122	0.245	0.225	–	0.300	7.406	–
Indochina	2.587	1.187	5.443	167.7	2021	0.233	60.60	–
IT	49.77	21.34	95.92	241.8	–	5.843	210.5	–
Twitter	0.017	0.015	0.027	0.106	–	192.6	5126	–
Friendster	0.119	0.119	0.159	0.396	–	2.409	42.92	–
UK	4.071	2.132	11.49	397.7	–	0.267	151.5	–
Clueweb09	27.04	9.205	40.68	–	–	131.8	–	–
Clueweb12	26365	2061	61661	–	–	2129	–	–

IncHL- m provides us good insights on the additional cost required by guaranteeing the minimality property of labelling on dynamic graphs. For the decremental algorithms, we can also see that the average update time taken by DecHL is significantly less than DecFD and DecPLL on all the datasets. DecPLL took time in seconds to update the labellings and failed to update the labelling for Hollywood, Enwiki and Indochina due to very high update time complexity, which is cubic in the worst case in terms of the number of vertices [19].

Labelling Size. Table 3 shows that the labelling sizes of our method FulHL after applying updates in E_F are significantly (ranges from 30% to 90%) smaller than the labelling sizes of FulFD and FulPLL. When updates occur on a graph, the labelling sizes of FulFD and FulHL remain stable because their average label size is bounded by the constant (i.e. the size of landmarks set $|R|$). Specifically, FulFD stores complete shortest-path trees w.r.t. the landmarks, while FulHL stores pruned shortest-path trees, thereby leading to a labelling of much smaller sizes than FulFD. However, the labelling sizes of FulPLL increase because its incremental algorithm does not remove outdated and redundant entries. In Table 6, we present the difference Δ_{IncPLL} and Δ_{IncPLL} in the labelling sizes before and after updating the labelling by our method IncHL and the baseline method IncPLL, respectively. This also shows how much the labelling sizes increase after applying the updates in E_I in several datasets. It confirms that the increase in the labelling size by our method is negligibly small, while IncPLL has considerably large increase in the labelling sizes. Particularly, IncPLL has a huge increase (several gigabytes) in the labelling size of Indochina for just 1000 updates. Thus, IncPLL may cause FulPLL to produce an ever increasing labelling sizes, particularly when graphs are updated frequently.

Query Time. Table 3 shows that the average query times after applying updates in E_F. FulHL performs comparably with FulFD and FulPLL. More specifically, as compared to FulFD, the query time of FulHL outperforms on 7 out of 11 datasets where FulFD can construct labelling. For the two largest datasets, FulFD fails to construct labelling and thus cannot answer queries. Notice that FulHL considerably underperforms FulFD on Twitter when ignoring updates on graphs. This is because the maximum degree of Twitter is very high (Table 2) and FulFD maintains shortest-path trees for landmarks along with their neighbours which may cause a large fraction of pairs to be covered by very high degree landmarks. However, if we consider the overall query time on dynamic graphs as the sum of the total update time plus the query time after the update operation, then our method FulHL would indeed significantly outperform FulFD on Twitter. It has been reported in [19] that the average query time is dependent on the labelling size. As discussed in Sect. 6.2.1, the update operations do not considerably affect the

Table 5 Construction time, labelling size and query time of the state-of-the-art methods PSL, PSL$^+$ and PSL* where "-" denotes that a method did not finish to construct labelling in one day (24 hours) or ran out of memory (512 GB)

Dataset	Construction Time (sec.)			Labelling Size (GB)			Query Time (ms)		
	PSL	PSL$^+$	PSL*	PSL	PSL$^+$	PSL*	PSL	PSL$^+$	PSL*
Skitter	28	24	17	2.16	1.72	1.01	0.003	0.005	0.007
Flickr	36	26	16	2.80	1.78	0.98	0.004	0.004	0.005
Hollywood	577	325	261	11.2	6.17	4.15	0.020	0.020	0.146
Orkut	22755	22983	18971	147	146	121	0.086	0.086	0.192
Enwiki	363	368	302	10.0	9.84	7.04	0.005	0.005	0.021
Livejournal	6149	5754	3179	80.7	73.8	40.4	0.035	0.035	0.047
Indochina	336	79	71	17.3	5.05	3.39	0.004	0.003	0.007
IT	–	15599	10377	–	227	130	–	0.016	0.059
Twitter	–	–	–	–	–	–	–	–	–
Friendster	–	–		–	–	–	–	–	–
UK	–	–	–	–	–	–	–	–	–
Clueweb09	–	–	–	–	–	–	–	–	–
Clueweb12	–	–	–	–	–	–	–	–	–

Table 6 Comparison of the difference in the labelling sizes of the proposed method IncHL and the baseline method IncPLL after applying 1,000 updates

$size(L_{\text{After}}) - size(L_{\text{Before}})$	Datasets				
	Skitter	Flickr	Hollywood	Enwiki	Indochina
Δ_{IncPLL}	5 MB	9 MB	22 MB	2 MB	7,334 MB
Δ_{IncHL}	7 KB	5 KB	1 KB	0 KB	1,833 KB

labelling sizes of FulFD and FulHL; thus, their query times remain stable. The query time of FulPLL increases because it allows the existence of outdated and redundant entries in the labels of affected vertices which deteriorates query performance over time, particularly when graphs are updated frequently.

6.2.2 Labelling-based static methods

To understand how our dynamic algorithms perform against the state-of-the-art methods for static graphs, we compare the performance of our proposed methods against the parallelised pruned landmark labelling methods PSL, PSL$^+$ and PSL*, which have been shown to achieve the state-of-the-art performance for answering distance queries on static graphs [34]. The results for PSL, PSL$^+$ and PSL* are presented in Table 5. It is worth to note that PSL, PSL$^+$ and PSL* are not dynamic methods, thereby requiring us to reconstruct labelling from scratch after each update in a graph. As we can see in Table 5 that the construction time of distance labelling is by far greater than the update time of our methods FulHL- m and FulHL in Table 3. For example, our methods FulHL- m and FulHL take 1 millisecond on average to update labelling on Skitter, whereas PSL takes 28 seconds on average to construct labelling. Furthermore, PSL, PSL$^+$ and PSL* all failed to scale to large graphs. Specifically, PSL failed for 6 out of 13 datasets, and PSL$^+$ and PSL* failed for 5 out of 13 datasets and have a very high construction cost on datasets with large average degrees such as Orkut and Livejournal. Although the construction times of PSL$^+$ and PSL* are reduced using index reduction techniques, these index reduction techniques affect the query performance. For query performance, PSL* is comparable to our method FulHL on Flickr, Hollywood, Orkut, Enwiki and Livejournal.

We also notice that the labelling sizes of PSL, PSL$^+$ and PSL* (as presented in Table 5) are much larger than FulHL (as presented in Table 3). As we can see in Table 5, PSL* produces the labelling of size almost 99% larger than the labelling of FulHL for Orkut and IT. The query times of PSL are the fastest among all baseline methods, but unfortunately, unbearably long construction times and large labelling sizes make these methods hardly scale to very large graphs. This situation becomes even worse when the underlying graphs are dynamic. Considering the overall performance w.r.t. three main factors, i.e. query time, labelling size and construction time, FulHL stands out in claiming the best trade-offs between query time, labelling size and construction time among all other baseline methods for large and dynamic graphs.

6.2.3 Online search methods

To understand the impact of labelling on distance queries, we also compare the query performance of our fully dynamic method FulHL with an online search method Opt- BiBFS. The results are shown in Fig. 5. To make a fair comparison,

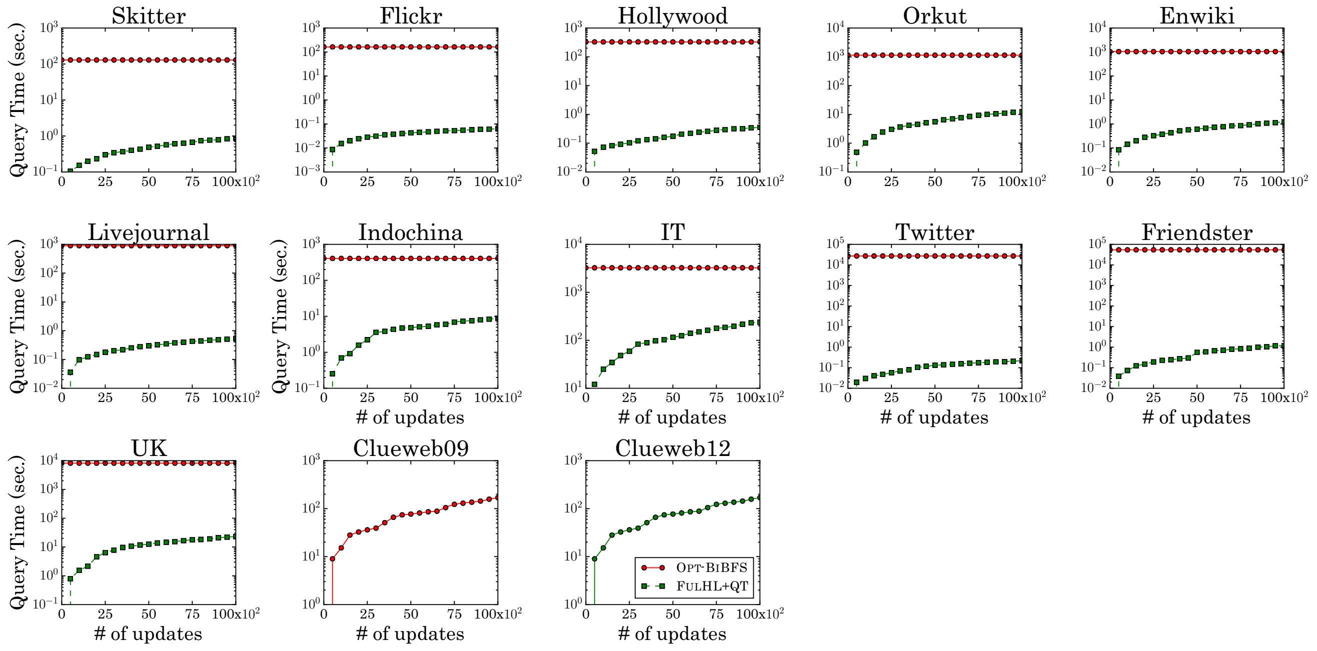

Fig. 5 Comparison of query time of the proposed method FULHL against online search method OPT- BIBFS. OPT- BIBFS has no results for Clueweb09 and Clueweb12 because it did not finish within 24 hours

we consider the overall query time of our method as the sum of the total update time on labelling for randomly sampled updates of varying sizes (i.e. 1 to 10,000) plus the query time of 1,000 queries after applying the updates, denoted as FULHL+QT. For the baseline method OPT- BIBFS, we take only the query time of 1,000 queries after applying updates. We see that the overall performance of our methods is significantly better than OPT- BIBFS on all the datasets. In particular, our methods show promising performance on large networks even when the number of updates is 10,000. Only FULHL is able to have results within 24 hours for the two largest datasets Clueweb09 and Clueweb12. These confirm that our method is efficient and can scale to very large networks.

6.3 Performance with varying landmarks

We also evaluate the performance of our method FULHL under different numbers of landmarks. The results are presented in Fig. 6. For the largest two datasets Clueweb09 and Clueweb12, the baseline method FULFD failed to construct labelling. Thus, Fig. 6 (as well as Fig. 7) does not include any results for FULFD on these two datasets.

Figure 6a shows the labelling sizes produced by FULHL and FULFD after applying the updates in E_F under different numbers of landmarks on all the datasets. As we can see, when the number of landmarks increases, the labelling sizes of FULHL and FULFD also increase. The labelling sizes of our method FULHL increase sublinearly when increasing the number of landmarks. This is due to the pruning during JP-BFSs. In contrast, the labelling sizes of FULFD increase linearly with increasing the number of landmarks since it does not have the minimality of labelling. Thus, the labelling sizes of FULHL are always by far smaller than the labelling sizes of FULFD.

Figure 6(**b**)–(**c**) shows the average update times of our methods INCHL and DECHL against the baseline methods INCFD and DECFD after applying the updates in E_I and E_D, respectively. As we can see in Fig. 6b, INCHL outperforms INCFD on all the datasets against every selection of landmarks, except Orkut and Enwiki for which INCHL and INCFD have comparable results. This is because the average distances on these networks are small, and only a small fraction of vertices are affected to update their labels. In such cases, the performance of our method is comparable with INCFD. When a large fraction of vertices is affected against a graph change, our method better leverages the pruning power to perform than INCFD. For instance, our method significantly outperforms INCFD on the datasets Indochina and UK because a large fraction of affected vertices are caused by graph changes, as can be seen from Fig. 8d.

From Fig. 6c, we can also confirm that DECHL outperforms DECFD on all the datasets under every selection of landmarks. Further, we observe that the average update times of our methods INCHL and DECHL either remain low or increase very slowly when we increase the number of landmarks. This is because a larger number of landmarks can contribute more to leverage the pruning power of our

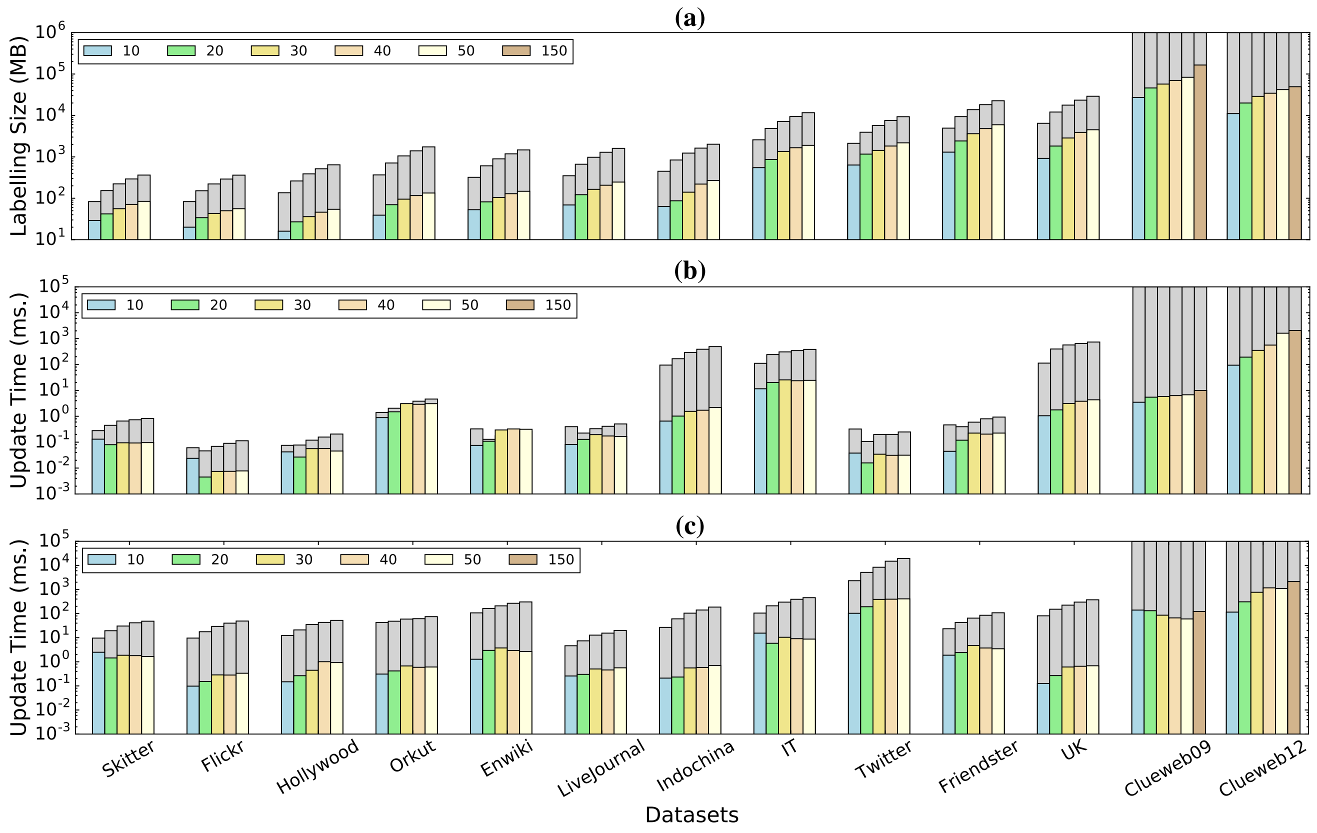

Fig. 6 Comparison of the proposed method FULHL (in coloured bars) and the baseline method FULFD (in coloured plus grey bars) under 10-150 landmarks: (**a**) labelling size, (**b**) average update time for incremental updates and (**c**) average update time for decremental updates. Note that there are no results of FULFD for Clueweb09 and Clueweb12 as FULFD failed to construct labelling

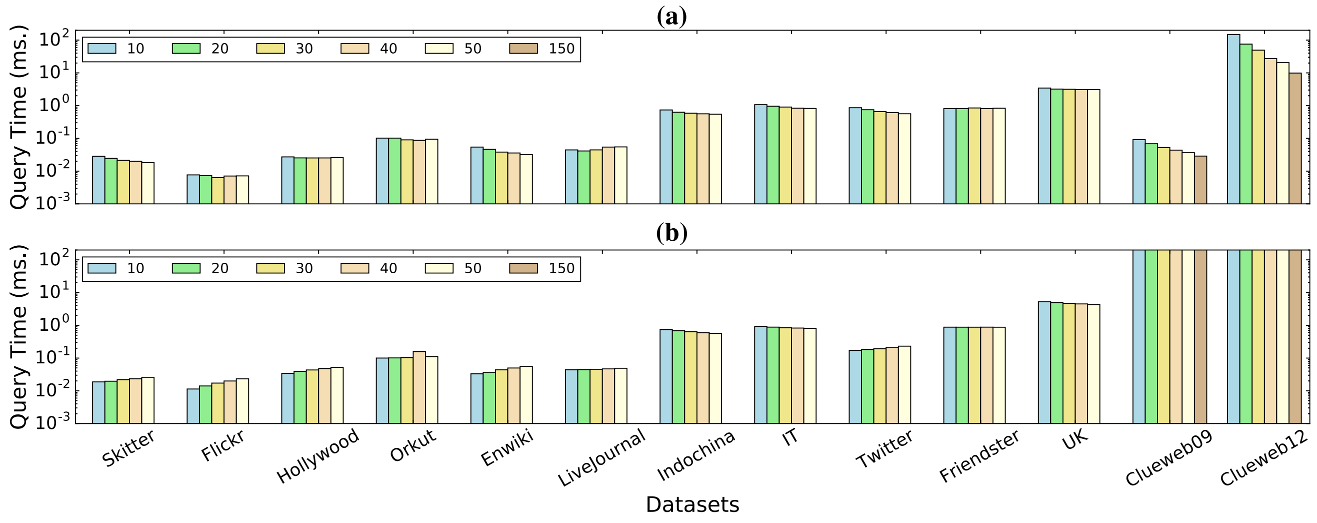

Fig. 7 Comparison of the proposed method FULHL and the baseline method FULFD: (**a**) average query time for FULHL, and (**b**) average query time for FULFD. Note that there are no results of FULFD for Clueweb09 and Clueweb12 as FULFD failed to construct labelling

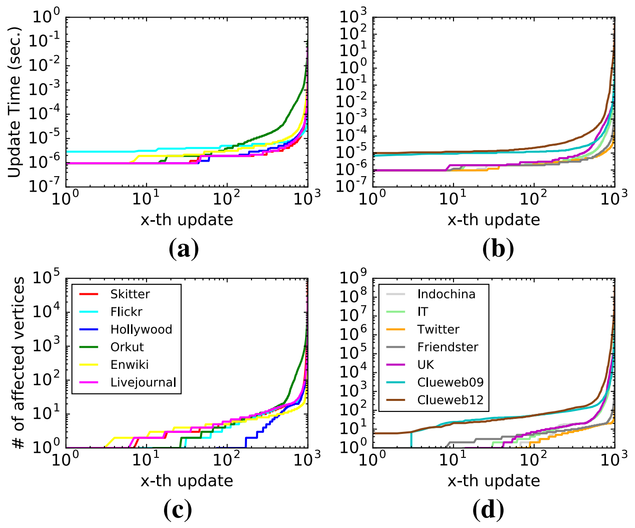

Fig. 8 1000 edge insertions from E_I: (**a**)–(**b**) show the distribution of update times, and (**c**)–(**d**) show the distribution of the numbers of affected vertices

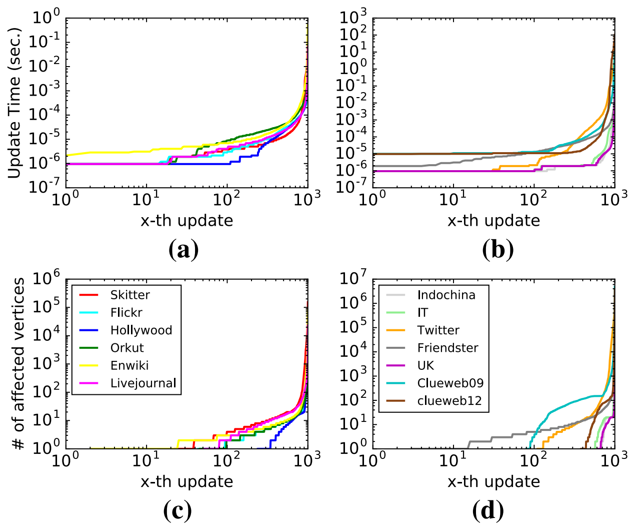

Fig. 9 1000 edge deletions from E_D: (**a**)–(**b**) show the distribution of update times, and (**c**)–(**d**) show the distribution of the numbers of affected vertices

methods, thereby performing much better than the baseline methods.

In Fig. 7, we also show the trend in the average query times of our method FulHL in comparison with the baseline method FulFD under varying landmarks {10, 20, 30, 40, 50} for all datasets and {150} for the two largest datasets after applying the updates in E_F. As we can see in Fig. 7a, generally the trend in the average query times of FulHL is decreasing or remains the same, whereas the trend in Fig. 7b is increasing for FulFD with the increased number of landmarks. Furthermore, we notice that the trend in the average query times of Indochina, IT and UK in Figs. 7a and 7b is all decreasing. This is because they have large average distances (in Table 2), due to which an increased number of landmarks might cover a large fraction of shortest paths and yield the tighter upper-distance bounds to help efficient querying. Overall, the increased number of landmarks help improve query time performance.

6.4 Analysis of affected vertices

To understand how affected vertices correlate with update times against different types of updates: edge insertion and deletion, we analyse the distributions of the numbers of affected vertices and their update times. The results are presented in Figs. 8 and 9, in which 1000 edge insertions and edge deletions are taken from E_I and E_D, respectively, and their numbers of affected vertices and update times are sorted in ascending order.

From the figures, we can see that there is a correlation between the number of affected vertices and update times of these updates. We observe that the difference in the number of affected vertices is not significant among these updates and only a few updates correspond to a large number of affected vertices for which the update times are also high in most of the datasets.

6.5 Scalability of updates

We analyse the performance of our methods with increasing the number of updates. We start with 500 updates and then iteratively increase 500 updates until 10,000 updates. Figures 10–11 show the average update times after constructing the labelling from scratch, and updating the labelling using our incremental and decremental algorithms after each increase. We observe from Fig. 10 that the update time of IncHL on all the datasets is almost always below the construction time of labelling. On IT and Twitter, the update time reaches the construction time after performing 5,000 updates. This is because the average distance of IT is large as depicted in Table 2, which may lead to high percentages of affected vertices to be updated; although the average distance of Twitter is small, the density of Twitter is high and fewer updates can still cause a large fraction of vertices to be affected as can be observed from Fig. 8(d). In Fig. 11, we can also see that DecHL generally performs well on all the datasets. Compared to the other datasets, DecHL performs relatively worse on Skitter, Enwiki, Twitter and Clueweb12. This is because the updates in these networks can have larger distances after removal, as can be observed from the distance distribution in Fig. 4, which may cause more vertices to be affected and require more update time as depicted in Fig. 9. Overall, the performance of the proposed algorithms is dependent on the fraction of affected vertices and our methods can scale to perform large batches of updates efficiently.

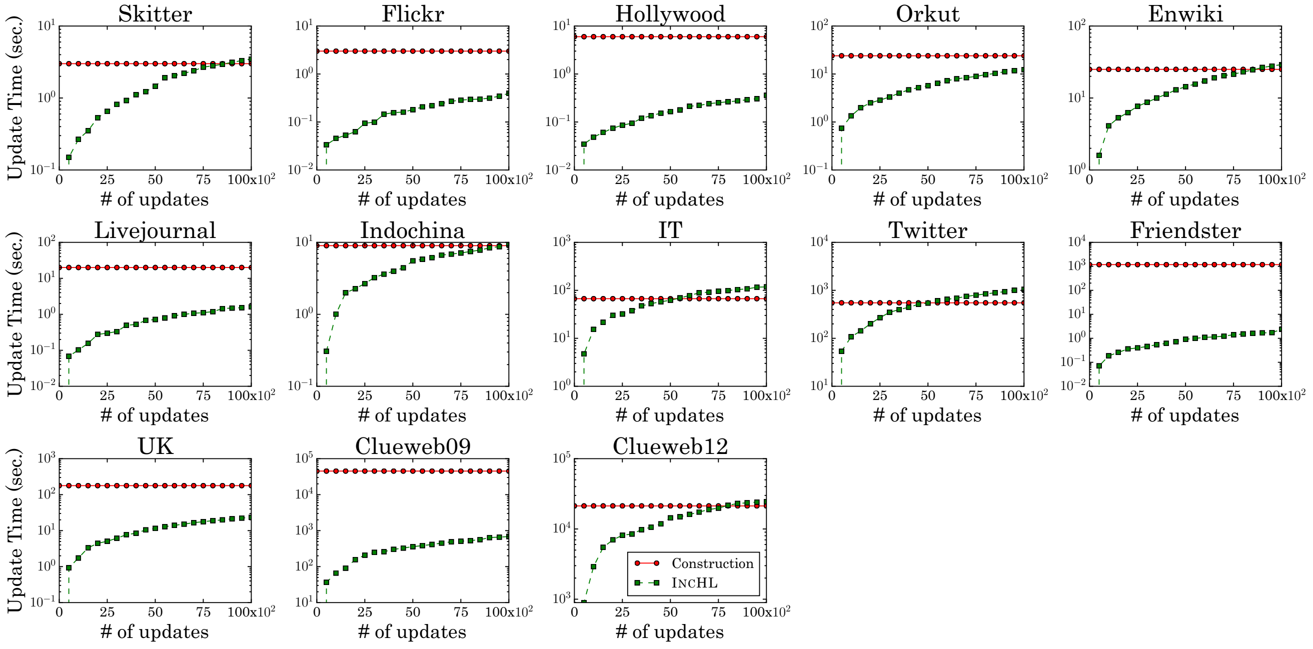

Fig. 10 Comparison of average update time of the proposed method IncHL for performing up to 10,000 updates against the construction time

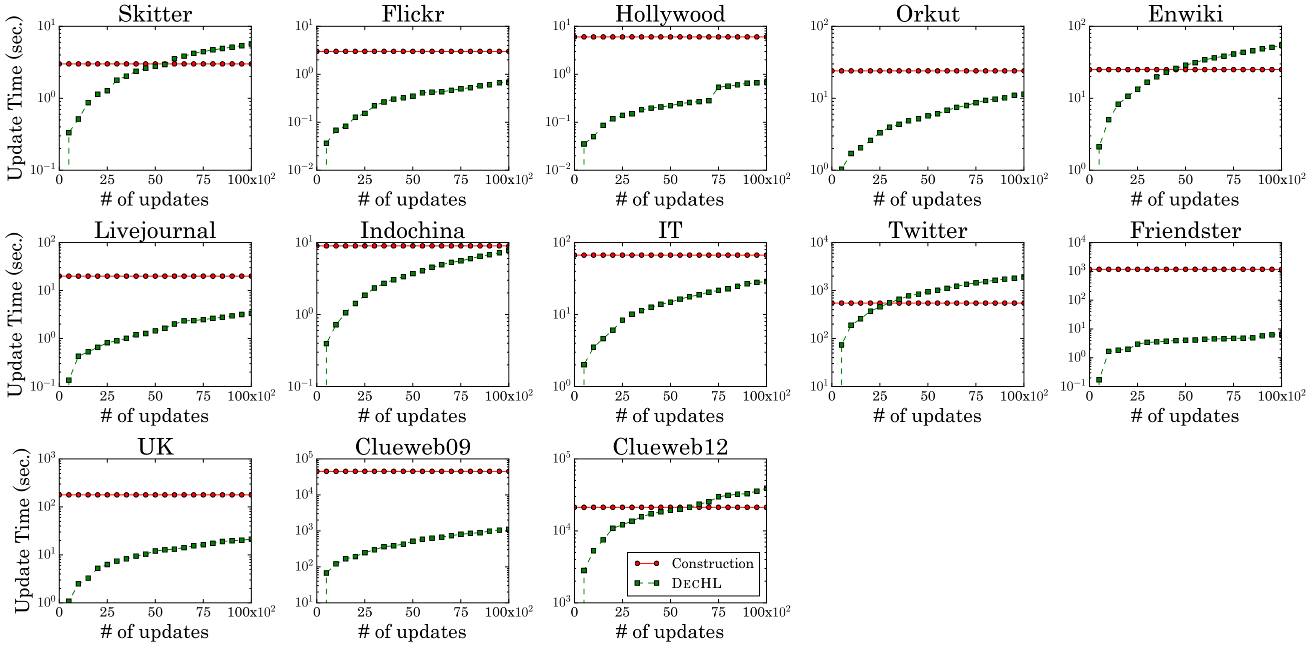

Fig. 11 Comparison of average update time of the proposed method DecHL for performing up to 10,000 updates against the construction time

7 Extensions

7.1 Directed graphs

Our methods can be extended to handle directed graphs. We can store two sets of labels, namely *forward label* $L_f(v)$ and *backward label* $L_b(v)$ for each vertex v, which contain pairs $(r_i, \delta_{r_i v})$ after performing forward and backward pruned BFSs w.r.t. each landmark $r_i \in R$, respectively. Further, we can store a *forward highway* $H_f = (R, \delta_{H_f})$ and a *backward highway* $H_b = (R, \delta_{H_b})$, where for any two landmarks $r_i, r_j \in R$, $\delta_{H_f}(r_i, r_j) = d_G(r_i, r_j)$ and $\delta_{H_b}(r_j, r_i) = d_G(r_j, r_i)$. Then, to repair the labels and highways affected by a update, we conduct pruned BFSs twice: once in the forward direction and once in the backward direction. To answer a distance query (s, t), we can use $L_f(s)$ and

$L_b(t)$ to compute the upper bound distance from s to t in the same way as described in Equation 3.

7.2 Weighted graphs

Our methods can also be extended to handle weighted graphs. We may conduct pruned Dijkstra's algorithm in place of pruned BFSs for constructing the labelling and conduct a similar jumped-and-pruned Dijkstra's algorithm in place of JP-BFSs for updating the labelling.

8 Discussion

Now, we discuss the advantages of designing dynamic algorithms upon the highway cover labelling [22] over other state-of-the distance querying methods.

Generally, current research for answering distance queries using pre-computed distance labelling has delved into two directions: (1) *full labelling-based methods*, which target to answer distance queries for any pairs of vertices only using distance labelling, and (2) *partial labelling-based methods*, which aim to answer distance queries using a combination of online searching and partial distance labelling. Although full labelling-based methods often provide promising query response performance, they have some significant limitations in handling distance queries on large and dynamic graphs:

(i) Full labelling-based methods cannot scale to very large networks due to the quadratic growth of labelling sizes, leading to very high space requirements and unbearably long construction time. As shown in our experiments, PLL [2] has failed to construct distance labelling on networks with over hundreds of millions of edges, and the recent parallel PLL methods [34] have failed to construct distance labelling on networks with over 1.2B edges.

(ii) Full labelling-based methods do not perform well for dynamic graphs because they require much more complicated analyses to reflect graph changes into a full distance labelling that captures distance information of all pairs of vertices in a graph, than a partial distance labelling that only captures distance information of some essential pairs of vertices (e.g. the distance between a vertex and a landmark or between two landmarks) in the graph.

Hence, in this work, we choose to design our dynamic algorithms based on the highway cover labelling [22] which is a partial distance labelling, rather than PLL which provides a full distance labelling [2].

Notice that the work in [26] has also proposed a fully dynamic algorithm by combining a partial distance labelling with a graph traversal algorithm for answering distance queries. However, the partial distance labelling in this work was designed naively—simply storing the distance information between any vertex to landmarks. Unlike this work, the highway cover labelling has pruned vertices whose distance information can be obtained from the highway and the labels of other vertices. Therefore, our proposed methods are able to answer distance queries on large dynamic graphs using a more compact data structure for improving the scalability as compared to the work in [26]. It is evident from experiments that our methods provide best trade-offs between query time, update time and labelling size among all the state-of-the-art methods for handling distance queries on large and dynamic graphs.

9 Conclusion and future work

In this article, we have studied the problem of answering distance queries on very large (billion-scale) dynamic networks. We have considered two fundamental update operations on dynamic graphs, i.e. edge insertions and edge deletions. Our proposed algorithms exploit properties of a recent novel technique called highway cover labelling [22] for dynamic graphs in order to efficiently maintain labelling after a graph change. We have shown that our proposed fully dynamic algorithms are correct and can preserve the minimality property of labelling [22] after update operations. We have also conducted extensive experiments to empirically verify the efficiency and scalability of our proposed algorithms. The results show that our proposed algorithms significantly outperform the state-of-the-art methods.

For future work, we plan to investigate the following research directions: 1) It would be interesting to extend the proposed algorithms DecHL or IncHL to handle batches of updates, i.e. sets of updates occurring on a network simultaneously [17,18]. One possible way to explore in this direction could be that of studying how the updates in a batch are related and how they affect the structure of the labelling when occurring simultaneously and comparing this effect with operations taken individually, 2) re-positioning of landmarks in a dynamic setting in order to reduce the size of the labelling and improve the query performance and 3) exploring the possibility of applying the proposed algorithms to road networks.

References

1. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Hierarchical hub labelings for shortest paths. In: Proceedings of the 20th Annual European Conference on Algorithms, pp. 24–35 (2012)
2. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 349–360 (2013)

3. Akiba, T., Iwata, Y., Yoshida, Y.: Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In: Proceedings of the 23rd International Conference on World Wide Web, pp. 237–248 (2014)
4. Akiba, T., Sommer, C., Kawarabayashi, K.: Shortest-path queries for complex networks: exploiting low tree-width outside the core. In: Proceedings of the 15th International Conference on Extending Database Technology, pp. 144–155 (2012)
5. Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group formation in large social networks: Membership, growth, and evolution. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD, pp. 44–54 (2006). https://doi.org/10.1145/1150402.1150412
6. Bernstein, A.: Fully dynamic (2 + epsilon) approximate all-pairs shortest paths with fast query and close to linear update time. In: Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science, FOCS, pp. 693–702 (2009)
7. Boccaletti, S., Latora, V., Moreno, Y., Chavez, M., Hwang, D.U.: Complex networks: structure and dynamics. Phys. Rep. **424**(4–5), 175–308 (2006). https://doi.org/10.1016/j.physrep.2005.10.009
8. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In: Proceedings of the 20th International Conference on World Wide Web, WWW, pp. 587–596 (2011). https://doi.org/10.1145/1963405.1963488
9. Boldi, P., Santini, M., Vigna, S.: A large time-aware graph. SIGIR Forum **42**(2), 33–38 (2008)
10. Boldi, P., Vigna, S.: The webgraph framework i: compression techniques. In: Proceedings of the 13th International Conference on World Wide Web, WWW, pp. 595–602 (2004). https://doi.org/10.1145/988672.988752
11. Callaway, D.S., Newman, M.E.J., Strogatz, S.H., Watts, D.J.: Network robustness and fragility: percolation on random graphs. Phys. Rev. Lett. **85**, 5468–5471 (2000). https://doi.org/10.1103/PhysRevLett.85.5468
12. 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). https://doi.org/10.1007/s00778-012-0274-x
13. Cheng, J., Yu, J.X.: On-line exact shortest distance query processing. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT, pp. 481–492 (2009). https://doi.org/10.1145/1516360.1516417
14. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 937–946 (2002)
15. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Robust distance queries on massive networks. In: European Symposium on Algorithms, pp. 321–333. Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44777-2_27
16. Demetrescu, C., Italiano, G.F.: A new approach to dynamic all pairs shortest paths. J. ACM **51**(6), 968–992 (2004). https://doi.org/10.1145/1039488.1039492
17. D'Andrea, A., D'Emidio, M., Frigioni, D., Leucci, S., Proietti, G.: Dynamically maintaining shortest path trees under batches of updates. In: Revised Selected Papers of the 20th International Colloquium on Structural Information and Communication Complexity - Volume 8179, SIROCCO, pp. 286–297 (2013)
18. D'Andrea, A., D'Emidio, M., Frigioni, D., Leucci, S., Proietti, G.: Experimental evaluation of dynamic shortest path tree algorithms on homogeneous batches. In: Proceedings of the 13th International Symposium on Experimental Algorithms - Volume 8504, pp. 283–294 (2014). https://doi.org/10.1007/978-3-319-07959-2_24
19. D'angelo, G., D'emidio, M., Frigioni, D.: Fully dynamic 2-hop cover labeling. J. Exp. Algorithmics **24**(1) (2019). https://doi.org/10.1145/3299901
20. D'Emidio, M.: Faster algorithms for mining shortest-path distances from massive time-evolving graphs. Algorithms **13**(8), 191 (2020)
21. Farhan, M., Wang, Q.: Efficient maintenance of distance labelling for incremental updates in large dynamic graphs. In: 24th International Conference on Extending Database Technology EDBT (2021)
22. Farhan, M., Wang, Q., Lin, Y., McKay, B.D.: A highly scalable labelling approach for exact distance queries in complex networks. In: 22nd International Conference on Extending Database Technology EDBT, pp. 13–24 (2019)
23. 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. VLDB Endow. **6**(6), 457–468 (2013)
24. Goldberg, A.V., Harrelson, C.: Computing the shortest path: a search meets graph theory. In: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 156–165 (2005)
25. Gutenberg, M.P., Wulff-Nilsen, C.: Fully-dynamic all-pairs shortest paths: Improved worst-case time and space bounds. In: Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, pp. 2562–2574 (2020)
26. Hayashi, T., Akiba, T., Kawarabayashi, K.: Fully dynamic shortest-path distance query acceleration on massive networks. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, pp. 1533–1542 (2016)
27. Jiang, M., Fu, A.W.C., Wong, R.C.W., Xu, Y.: Hop doubling label indexing for point-to-point distance querying on scale-free networks. Proc. VLDB Endow. **7**(12), 1203–1214 (2014). https://doi.org/10.14778/2732977.2732993
28. Jin, R., Ruan, N., Xiang, Y., Lee, V.: A highway-centric labeling approach for answering distance queries on large sparse graphs. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 445–456 (2012)
29. Kumar, R., Novak, J., Tomkins, A.: Structure and evolution of online social networks. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD, pp. 611–617 (2006). https://doi.org/10.1145/1150402.1150476
30. Kunegis, J.: Konect: The koblenz network collection. In: Proceedings of the 22nd International Conference on World Wide Web, WWW, pp. 1343–1350 (2013). https://doi.org/10.1145/2487788.2488173
31. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graphs over time: densification laws, shrinking diameters and possible explanations. KDD (2005). https://doi.org/10.1145/1081870.1081893
32. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters. Internet Math. **6**(1), 29–123 (2009)
33. Leskovec, J., Sosič, R.: Snap: A general-purpose network analysis and graph-mining library. ACM Trans. Intell. Syst. Technol. **8**(1), 2898361 (2016). https://doi.org/10.1145/2898361
34. Li, W., Qiao, M., Qin, L., Zhang, Y., Chang, L., Lin, X.: Scaling distance labeling on small-world networks. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1060–1077 (2019)
35. Li, Y., U, L.H., Yiu, M.L., Kou, N.M.: An experimental study on hub labeling based shortest path algorithms. Proc. VLDB Endow. **11**(4), 445–457 (2017)
36. Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and analysis of online social networks. In: IMC, pp. 29–42 (2007)

37. Myers, S.A., Leskovec, J.: The bursty dynamics of the twitter information network. In: Proceedings of the 23rd International Conference on World Wide Web, pp. 913–924 (2014)
38. Newman, M.E.J., Strogatz, S.H., Watts, D.J.: Random graphs with arbitrary degree distributions and their applications. Phys. Rev. E **64**, 026118 (2001). https://doi.org/10.1103/PhysRevE.64.026118
39. Ouyang, D., Yuan, L., Qin, L., Chang, L., Zhang, Y., Lin, X.: Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. Proc. VLDB Endow. **13**(5), 602–615 (2020)
40. Potamias, M., Bonchi, F., Castillo, C., Gionis, A.: Fast shortest path distance estimation in large networks. In: Proceedings of the 18th ACM Conference on Information and Knowledge Management, pp. 867–876 (2009)
41. Qin, Y., Sheng, Q.Z., Falkner, N.J., Yao, L., Parkinson, S.: Efficient computation of distance labeling for decremental updates in large dynamic graphs. World Wide Web **20**(5), 915–937 (2017). https://doi.org/10.1007/s11280-016-0421-1
42. Robertson, N., Seymour, P.D.: Graph minors. iii. planar tree-width. J. Comb. Theory Ser. B **36**(1), 49–64 (1984). https://doi.org/10.1016/0095-8956(84)90013-3
43. Roditty, L., Zwick, U.: On dynamic shortest paths problems. In: Albers, S., Radzik, T. (eds.) European Symposium on Algorithms, pp. 580–591. Springer, Berlin (2004)
44. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI, pp. 4292–4293 (2015)
45. Tarjan, R.E.: Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, 3600 University City Science Center Philadelphia, PA, United States (1983). https://doi.org/10.1137/1.9781611970265
46. Tretyakov, K., Armas-Cervantes, A., García-Bañuelos, L., Vilo, J., Dumas, M.: Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, CIKM, pp. 1785–1794 (2011). https://doi.org/10.1145/2063576.2063834
47. Ukkonen, A., Castillo, C., Donato, D., Gionis, A.: Searching the wikipedia with contextual information. In: Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM, pp. 1351–1352 (2008). https://doi.org/10.1145/1458082.1458274
48. Vieira, M.V., Fonseca, B.M., Damazio, R., Golgher, P.B., Reis, D.d.C., Ribeiro-Neto, B.: Efficient search ranking in social networks. In: Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management, CIKM, pp. 563–572 (2007). https://doi.org/10.1145/1321440.1321520
49. Wang, Y., Wang, Q., Koehler, H., Lin, Y.: Query-by-sketch: Scaling shortest path graph queries on very large networks. In: Proceedings of the 2021 International Conference on Management of Data, pp. 1946–1958 (2021)
50. Wei, F.: Tedi: efficient shortest path query answering on graphs. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 99–110 (2010)
51. Xu, B., Huang, Y., Kwak, H., Contractor, N.: Structures of broken ties: exploring unfollow behavior on twitter. In: Proceedings of the 2013 Conference on Computer Supported Cooperative Work, CSCW, pp. 871–876 (2013). https://doi.org/10.1145/2441776.2441875
52. Yahia, S.A., Benedikt, M., Lakshmanan, L.V.S., Stoyanovich, J.: Efficient network aware search in collaborative tagging sites. Proc. VLDB Endow. **1**(1), 710–721 (2008). https://doi.org/10.14778/1453856.1453934
53. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. Knowl. Inf. Syst. **42**(1), 181–213 (2015). https://doi.org/10.1007/s10115-013-0693-z