CONTENTS

Efficient Maintenance of Distance Labelling for Incremental Updates in Large Dynamic Graphs

Muhammad Farhan
Australian National University
Canberra, Australia
muhammad.farhan@anu.edu.au

Qing Wang
Australian National University
Canberra, Australia
qing.wang@anu.edu.au

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 been successfully attempted on this problem, most of which is limited to static graphs. Due to the dynamic nature of real-world networks, there is a pressing need to address this problem for dynamic networks undergoing changes. In this paper, we propose an *online incremental* method to efficiently answer distance queries over very large dynamic graphs. Our proposed method incorporates incremental update operations, i.e. edge and vertex additions, into a highly scalable framework of answering distance queries. We theoretically prove the correctness of our method and the preservation of labelling minimality. We have also conducted extensive experiments on 12 large real-world networks to empirically verify the efficiency, scalability, and robustness of our method.

1 INTRODUCTION

Given a very large graph with billions of vertices and edges, how efficiently can we find the shortest path distance between any two vertices? If such a graph is dynamically changing over time (e.g. inserting edges or vertices), how can we not only efficiently but also accurately find the shortest path distance between any two vertices? These questions are intimately related to distance queries on dynamic graphs. As one of the most fundamental operations on graphs, distance queries have a wide range of real-world applications that operate on increasingly large dynamic graphs, such as context-aware search in web graphs [19], social network analysis in social networks [5, 20], management of resources in computer networks [6], and so on. Many of these applications use distance queries as a building block to realise more complicated tasks, and require distance queries to be answered instantly, e.g. in the order of milliseconds.

Previous studies have primarily focused on distance queries on static graphs [1–3, 10, 11, 13, 22], with little attention being paid to dynamics on graphs. To speed up query response time, a key technique is to precompute a data structure called *distance labelling* that satisfies certain properties such as 2-hop cover [8], and then use this data structure to answer distance queries efficiently. However, when a graph dynamically changes, its distance labelling needs to be changed accordingly; otherwise, distance queries may yield overestimated distances. Although it is possible to recompute a distance labelling from scratch, this leads to inefficiency. As shown in Figure 1, the percentage of affected vertices by a single change often ranges from $10^{-5}\%$ to 10% in various real-world networks, recomputing distance labelling from scratch for each single change not only wastes
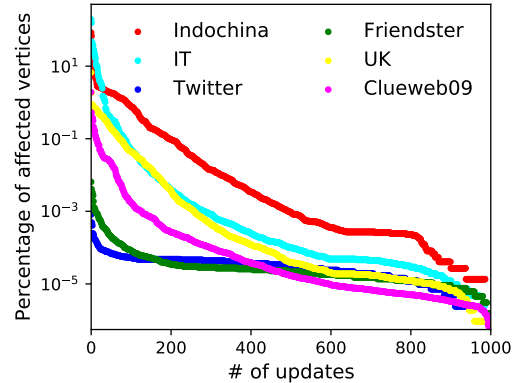
Figure 1: Distribution of affected vertices by a single graph change in various networks, where the results for 1000 graph changes are sorted in the descending order.

computing resources, but also may generate inaccurate query results during recomputing process. The question arising is thus how to efficiently and accurately change distance labelling on dynamic graphs in order to support distance queries?

In this paper, we aim to develop an *online incremental* method that can dynamically maintain distance labelling on graphs being changed by edge and vertex insertions. Typically, real-world dynamic networks are more vulnerable to insertions than removals and a plethora of such real-world networks are large and frequently updated, primarily accommodating insertions [15, 21]. Thus, an online incremental method for dynamic graphs should possess the following desirable characteristics: (1) *time efficiency* - It can answer distance queries and update distance labelling efficiently (in the order of milliseconds); 2) *space efficiency* - It guarantees the minimum size of distance labelling to reduce storage costs; (3) *scalability* - It can scale to very large networks with billions of vertices and edges.

Challenges. Designing online incremental methods for distance queries on dynamic graphs is known to be challenging [4]. When an edge or a vertex is inserted into a graph, outdated and redundant entries of distance labelling may occur. It was reported that removing such entries is a complicated task [4] because affected vertices need to be precisely identified so as to update their labels without violating the original properties of a distance labelling such as minimality. 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 [4, 12] mostly considered 2-hop labelling, which has very high space requirements and index construction time; as a result, their query and update performance

are 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 labeling method (PLL) [4], still yields a distance labelling whose size is 20-30 times larger than the original size of a dataset.

Contributions. Our contributions are summarised as follows:

- Our method overcomes the challenge of eliminating outdated and redundant distance entries. None of the previous studies have addressed this challenge because detecting those entries is too costly [4, 9]. When an edge or a vertex is inserted, previous studies only add new distance entries or modify existing distance entries. This would however lead to an ever increasing size of labelling, particularly when a graph is frequently updated by newly added edges or vertices. Accordingly, both query performance and space efficiency would deteriorate over time.

- We prove the correctness of our proposed method and show that it preserves the desirable property of minimality on our distance labelling. Due to a property called highway cover [10], the minimal size of a distance labelling in this work is much smaller than the size of a 2-hop labelling in previous work [4, 12]. Preserving minimality on a distance labelling thus improves space efficiency and query performance, as well as update performance. We also provide a complexity analysis of our proposed method.

- We conducted experiments using 12 real-world large networks across different domains to show the efficiency, scalability and robustness of our method. Particularly, our method can perform updates under one second, on average, even on billion-scale networks, while still answering queries efficiently in the order of milliseconds and guaranteeing the labelling size of a graph to be much smaller.

2 RELATED WORK

Answering shortest-path distance queries in graphs has been an active research topic for many years. Traditionally, a distance query can be answered using Dijkstra's algorithm [18] on positively weighted graphs or Breadth-First Search (BFS) algorithm on unweighted graphs. However, these traditional algorithms fail to achieve desired response time for distance queries on large graphs. Later, labelling-based methods have emerged as an attractive way of accelerating response time to distance queries [1–3, 8, 10, 11, 13], among which Akiba et al. [3] proposed a pruned landmark labeling (PLL) to precompute a 2-hop cover distance labelling [8]. This method serves as the state-of-the-art for labelling-based distance queries and can handle graphs with hundreds of millions of edges.

So far, only a few attempts have been made to study distance queries over dynamic graphs [4, 12], which are all based on the idea of 2-hop distance labelling or its variants. Akiba et al. [4] studied the problem of updating a pruned landmark labelling for incremental updates (i.e. vertex additions and edge additions). This work however does not remove redundant entries in distance labels because the authors considered that detecting such outdated entries is too costly. This inevitably breaks the minimality of pruned landmark labelling, leading to an ever increase of labelling size and deteriorated query performance over time. 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. [12] proposed

a fully dynamic approach that selects a small set of landmarks R and precompute a shortest-path tree (SPT) rooted at each $r \in R$. Then, an online search is conducted on a sparsified graph under an upper distance bound being computed via the SPTs. Nevertheless, this method still fails to construct labelling on networks with billions of vertices. Following the same line, a recent work by Farhan et al. [10] introduced a highway-cover labelling method (HL), which can provide fast response time (milliseconds) for distance queries even on billion-scale graphs. However, this approach only works for static graphs.

3 PROBLEM FORMULATION

Let $G = (V, E)$ be an undirected graph where V is a set of vertices and E is a set of edges. We denote by $N(v)$ the set of neighbors 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. Given a graph $G = (V, E)$, an *edge insertion* is to add an edge (a, b) into G where $\{a, b\} \subseteq V$ and $(a, b) \notin E$. Accordingly, a *node insertion* is to add a new node into G together with a set of edge insertions that connect v to existing vertices in G. The following fact is critical for designing algorithms for an edge insertion.

FACT 3.1. *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 $s, t \in V$, $d_G(s, t) \geq d_{G'}(s, t)$.*

That is, the distance between any two vertices never increases after inserting edges or vertices in a graph.

Highway cover labelling. Unlike the previous work [4, 9, 12] that uses 2-hop cover labelling [8], we develop our method using a highly scalable labelling approach, called *highway cover labelling* [10]. Let $R \subseteq V$ be a small set of *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 whose *size* is defined as: $size(L) = \sum_{v \in V} |L(v)|$. A *highway* $H = (R, \delta_H)$ 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 3.2. A *highway cover labelling* is a pair $\Gamma = (H, L)$ where H is a highway and L is a distance labelling s.t. for any vertex $v \in V \backslash R$ and $r \in R$, we have:

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

Highway cover labelling enjoys several nice theoretical properties, such as minimality and order independence. A minimal highway cover labelling can be efficiently constructed, independently of the order of applying landmarks [10].

Given a highway cover labeling $\Gamma = (H, L)$, an upper bound on the distance between any two vertices $u, v \in V \backslash R$ is computed:

$$d_{uv}^\top = \min\{\delta_L(r_i, u) + \delta_H(r_i, r_j) + \delta_L(r_j, v)|$$
$$(r_i, \delta_L(r_i, u)) \in L(u), (r_j, \delta_L(r_j, v)) \in L(v)\} \quad (2)$$

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 $d_{uv}^\top$ such that:

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

Problem definition. In this work, we study the problem of answering distance queries over a graph that is dynamically changed by edge and vertex insertions over time. Since a vertex insertion can be treated as a set of edge insertions, without loss of generality, below we define the problem based on edge insertions.

Definition 3.3. Let $G \hookrightarrow G'$ denote that a graph G is changed to a graph G' by an edge insertion. The *dynamic distance querying* problem is, given any two vertices u and v in the changed graph G', to efficiently compute the distance $d_{G'}(u, v)$.

4 ONLINE INCREMENTAL ALGORITHM

In this section, we propose an algorithm IncHL$^+$ to incrementally update labelling to reflect graph changes. Algorithm 1 describes the main steps of IncHL$^+$. Below, we discuss them in detail.

4.1 Finding Affected Vertices

When an update operation occurs on a graph $G = (V, E)$, there exists a subset of "affected" vertices in V whose labels need to be updated as a consequence of this update operation on the graph.

Definition 4.1. A vertex $v \in V$ is *affected* by $G \hookrightarrow G'$ iff $P_G(v, r) \neq P_{G'}(v, r)$ for at least one $r \in R$; *unaffected* otherwise.

We use Λ_r to denote the set of all affected vertices w.r.t. a landmark r and $\Lambda = \bigcup_{r \in R} \Lambda_r$ the set of all affected vertices.

Example 4.2. Consider Figure 2(a) in which 0 and 10 are two landmarks. After inserting an edge $(2, 5)$, $\Lambda_0 = \{5, 8, 9, 10, 13, 14\}$ in Figure 2(b) and $\Lambda_{10} = \{0, 1, 2\}$ in Figure 2(d).

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

LEMMA 4.3. *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).*

Following Lemma 4.3, we can reduce the search space of affected vertices by eliminating landmarks r with $d_G(r, a) = d_G(r, b)$ since $\Lambda_r = \emptyset$ in such a case. 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. Further, by the lemma below, we can also reduce the search space by "jumping" from the root of a BFS to vertex b.

LEMMA 4.4. *When $G \hookrightarrow G'$ with an inserted edge (a, b), we have $d_G(v, r) \geq d_G(a, r) + 1$ for any affected vertex $v \in \Lambda_r$.*

PROOF. By Lemma 4.3, there exists a shortest path from any affected vertex v to r going through the 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, $d_G(v, r) \geq d_G(a, r) + 1$ thus holds. □

Algorithm 2 describes our algorithm for finding affected vertices. Given a graph G with an inserted edge (a, b) and a highway cover labelling $\Gamma = (H, L)$ over G, we conduct a *jumped* BFS w.r.t. a landmark r starting from the vertex b with its new depth $\pi = Q(r, a, \Gamma) + 1$ (Lines 3-4). For every $(v, \pi) \in Q$, we enqueue all the neighbors of v that are affected into Q with new distances $\pi + 1$ (Lines 7-8) and add v to Λ_r as affected vertex (Line 9). This process continues until Q is empty.

Example 4.5. Figure 2 illustrates how our algorithm finds affected vertices as a result of inserting an edge $(2, 5)$. The BFS rooted at landmark 0 is depicted in Figure 2(b), which jumps to vertex 5 and finds six affected vertices $\{5, 8, 9, 10, 13, 14\}$. Similarly, the BFS rooted at landmark 10 is depicted in Figure 2(d), which jumps to vertex 2 and finds three affected vertices $\{0, 1, 2\}$.

Algorithm 1: Incremental algorithm (IncHL$^+$).

Input: $G, G', (a, b), \Gamma = (H, L)$
Output: $\Gamma' = (H', L')$
1 **foreach** $r \in R$ **do**
2    $\Lambda_r \leftarrow$ FINDAFFECTED$(G, (a, b), r, \Gamma)$
3   REPAIRAFFECTED$(G', (a, b), \Lambda_r, r, \Gamma)$

Algorithm 2: Finding affected vertices.

1 **Function** FINDAFFECTED$(G, (a, b), r, \Gamma)$
2    $Q \leftarrow \emptyset, \Lambda_r \leftarrow \emptyset$
3    $\pi \leftarrow Q(r, a, \Gamma) + 1$
4   Enqueue (b, π) to Q
5   **while** Q *is not empty* **do**
6   Dequeue (v, π) from Q
7   **foreach** $w \in N(v)$ *s.t.* $Q(r, w, \Gamma) \geq \pi + 1$ **do**
8   Enqueue $(w, \pi + 1)$ to Q
9       $\Lambda_r = \Lambda_r \cup \{v\}$
10   **return** Λ_r

4.2 Repairing Affected Vertices

Now we propose a repair strategy to efficiently update the labels of affected vertices in order to reflect graph changes. The key idea is that, instead of conducting a full BFS on all vertices, we conduct a partial BFS from b only on affected vertices. Further, to avoid unnecessary computations, we distinguish two kinds of affected vertices: (1) affected vertices that are *covered* by other landmarks and can thus be easily repaired by removing an entry from their labels; (2) affected vertices whose labels need to be repaired with accurately calculated distances on a changed graph. The following lemma characterizes the first kind according to the definition of highway cover labelling.

LEMMA 4.6. *An affected vertex $v \in \Lambda_r$ is covered by a landmark $r' \in R \backslash \{r\}$ iff r' exists in $P_{G'}(v, r)$. If an affected vertex $v \in \Lambda_r$ is covered by r', then any affected vertex $v' \in \Lambda_r$ satisfying $d_{G'}(r, v') = d_{G'}(r, v) + d_{G'}(v, v')$ must also be covered by r'.*

By Lemma 4.6, we can efficiently repair affected vertices $v \in \Lambda_r$ as follows. If v is covered by a landmark $r' \in R \backslash \{r\}$ (i.e., one of the unaffected parents of v does not contain r in its label) and is also a landmark, we only update the highway; otherwise, we remove the entry of r from $L(v)$. If v is not covered by any $r' \in R \backslash \{r\}$, we add/modify the entry of r in $L(v)$. If v is a descendant of covered vertices, we simply remove the entry of r from $L(v)$ (if exists).

Algorithm 3 describes our algorithm for repairing affected vertices. Given a graph G with an inserted edge (a, b) and a set of affected vertices Λ_r, we conduct a BFS w.r.t. a landmark r starting from the vertex b with its new distance $\pi = d_G(r, a) + 1$ (Lines 3-4). We use two queues $Q_{uncovered}$ and $Q_{covered}$ to process uncovered and covered vertices, respectively. If b is covered, we enqueue (b, π) to $Q_{covered}$ and remove the entry of r from the labels of affected vertices (Line 25). Otherwise, we enqueue (b, π) to $Q_{uncovered}$ and start processing vertices in $Q_{uncovered}$ (Line 5). For each vertex $v \in Q_{uncovered}$ at depth π, we examine its affected neighbors w at depth $\pi + 1$. If w is covered, then if w is a landmark, we update the highway (Line 10); otherwise we remove the entry of r from $L(w)$ (Line 12) because there must exist another landmark in the shortest path from w to r and add
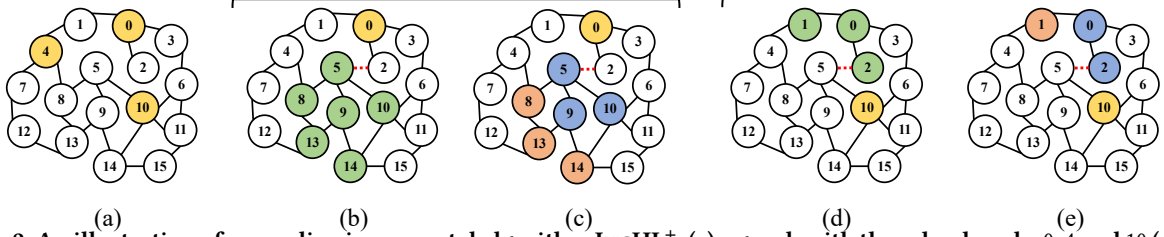

Figure 2: An illustration of our online incremental algorithm IncHL⁺: (a) a graph with three landmarks 0, 4 and 10 (colored in yellow); (b) and (d) the BFSs for finding affected vertices (colored in green) w.r.t. landmarks 0 and 10, respectively; (c) and (e) the BFSs for repairing affected vertices w.r.t. landmarks 0 and 10, respectively, where vertices with added/modified entries are colored in blue, and vertices with removed entries are colored in red.

Algorithm 3: Repairing affected vertices.

1 Function RepairAffected(G', (a, b), Λ_r, r, Γ)

2 $Q_{uncovered} \leftarrow \emptyset$, $Q_{covered} \leftarrow \emptyset$

3 $\pi \leftarrow d_G(r, a) + 1$

4   Enqueue (b, π) to $Q_{covered}$ if covered; otherwise to $Q_{uncovered}$

5   **while** $Q_{uncovered}$ *is not empty* **do**

6   **while** $(v, \pi) \in Q_{uncovered}$ *at depth* π **do**

7   **forall** $w \in N(v)$ *s.t.* $w \in \Lambda_r$ *at depth* $\pi + 1$ **do**

8   **if** *covered*$(w, \pi + 1)$ **then**

9   **if** w *is a landmark* **then**

10 $\delta_H(r, w) \leftarrow \pi + 1$

11   **else**

12   Remove r from $L(w)$

13   Enqueue $(w, \pi + 1)$ to $Q_{covered}$

14   **else**

15   Add/Modify $\{(r, \pi + 1)\}$ in $L(w)$

16   Enqueue $(w, \pi + 1)$ to $Q_{uncovered}$

17   Remove w from Λ_r

18   Dequeue (v, π) from $Q_{uncovered}$

19   **while** $(v, \pi) \in Q_{covered}$ *at depth* π **do**

20   **forall** $w \in N(v)$ *s.t.* $w \in \Lambda_r$ *at depth* $\pi + 1$ **do**

21   Remove r from $L(w)$

22   Remove w from Λ_r

23   Enqueue $(w, \pi + 1)$ to $Q_{covered}$

24   Dequeue (v, π) from $Q_{covered}$

25   Remove entry r from remaining vertices in $Q_{covered}$

$(w, \pi + 1)$ to $Q_{covered}$ (Line 13). Otherwise, we add/modify the entry of r with the new distance $\pi + 1$ in $L(w)$ and enqueue w to $Q_{uncovered}$ (Lines 15-16). After that, we remove w from Λ_r (line 17). Then, for each $(v, \pi) \in Q_{covered}$, we remove r from the labels of affected neighbors of v, remove these affected vertices from Λ_r and enqueue them to $Q_{covered}$ (Lines 19-24). We process these two queues, one after the other, until $Q_{uncovered}$ is empty. Finally, we remove the entry of r from the labels of the remaining vertices in $Q_{covered}$ (Line 25).

Example 4.7. Figure 2 illustrates how our algorithm repairs labels as a result of inserting an edge $(2, 5)$. The BFS for landmark 0 is depicted in Figure 2(c), which jumps to vertex 5 and repairs three affected vertices $\{5, 9, 10\}$. The vertices $\{8, 13, 14\}$ are covered by landmarks 4 and 10. Similarly, the BFS for landmark 10 is depicted in Figure 2(e), in which vertices $\{0, 2\}$ are repaired and vertex 1 is covered by landmarks 0 and 4.

5 THEORETICAL RESULTS

Proof of correctness. For $G \hookrightarrow G'$ where our method IncHL⁺ updates a highway cover labelling Γ over G into a highway cover labelling Γ' over G', we consider IncHL⁺ to be *correct* iff, whenever $Q(u, v, \Gamma) = d_G(u, v)$ holds for any two vertices u and v in G, then $Q(u', v', \Gamma') = d_{G'}(u', v')$ also holds for any two vertices u' and v' in G'. We prove the theorem below for IncHL⁺.

THEOREM 5.1. *IncHL⁺ is correct.*

PROOF. First, we prove that FindAffected returns the set of all affected vertices Λ_r as a result of an edge insertion. IncHL⁺ (Lines 7-8 of Algorithm 2) guarantees that any vertex being added to Q has one shortest path to a landmark r which goes through the inserted edge (a, b). By Lemma 4.3, such vertices are affected vertices, and thus a vertex v is added to Q in Algorithm 2 iff $v \in \Lambda_r$. Then, we prove that RepairAffected repairs $\Gamma = (H, L)$ s.t. (1) $(r, d_{G'}(r, v)) \in L(v)$ for $v \in \Lambda_r$, iff $P_{G'}(r, v)$ contains only one landmark r; (2) $\delta_H(r, r') = d_{G'}(r, r')$ for any $r' \in R\backslash\{r\}$. Starting from b with new distance π, the distances of affected vertices in Λ_r are iteratively inferred on G' and reflected into their labels via $Q_{uncovered}$ if these affected vertices are not covered (Lines 15-16 of Algorithm 3). If an affected vertex v is covered, it is kept in $Q_{covered}$; if v is also a landmark, $\delta_H(r, v)$ in H is updated (Lines 9-10). Thus, the distance entry of r is removed from the labels of affected vertices appearing in $Q_{covered}$, whereas any vertex v appearing in $Q_{uncovered}$ must have $(r, d_{G'}(r, v)) \in L(v)$. □

Preservation of minimality. It has been reported in [10] that, given a graph G, a minimal highway cover labelling $\Gamma = (H, L)$ of G can be constructed using an algorithm proposed in their work, i.e., $size(L') \geq size(L)$ holds for any $\Gamma' = (H, L')$ of G. For $G \hookrightarrow G'$ where IncHL⁺ updates Γ over G into Γ' over G', we prove that IncHL⁺ preserves the minimality of labelling.

THEOREM 5.2. *If Γ is minimal on G, then Γ' is minimal on G'.*

PROOF. By Lemma 4.6, $(r, d_{G'}(r, v)) \in L(v)$ for $v \in \Lambda_r$ iff $P_{G'}(r, v)$ does not contain any other landmark $R\backslash\{r\}$; otherwise we remove the entry of r from the label of v (Line 12, 21 and 25 of Algorithm 3). Thus, the labels of all affected vertices must be minimal after applying IncHL⁺. For unaffected vertices, their labels should remain unchanged. Hence, Γ' must be minimal. □

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 of vertices. For a landmark, Algorithm 2 takes $O(mdl)$ time to find all affected vertices and Algorithm 3 takes $O(md)$ to repair the labels of all affected vertices. We omit l from $O(md)$ for Algorithm 3 because distances for all unaffected neighbors of affected vertices are stored in Algorithm 2. Therefore, IncHL⁺ has time complexity $O(|R| \times mdl)$. In our

Table 1: Comparing the update time, query time and labelling size of our method with the baseline methods.

Dataset	Update Time (ms)			Query Time (ms)			Labelling Size		
	IncHL$^+$	IncFD	IncPLL	IncHL$^+$	IncFD	IncPLL	IncHL$^+$	IncFD	IncPLL
Skitter	0.194	0.444	2.05	0.027	0.019	0.047	42 MB	153 MB	2.44 GB
Flickr	0.006	0.074	1.73	0.007	0.012	0.064	34 MB	152 MB	3.69 GB
Hollywood	0.031	0.101	48	0.027	0.037	0.109	27 MB	263 MB	12.58 GB
Orkut	2.026	2.049	-	0.101	0.103	-	70 MB	711 MB	-
Enwiki	0.134	0.163	5.91	0.054	0.035	0.071	82 MB	608 MB	12.57 GB
Livejournal	0.245	0.268	-	0.044	0.046	-	122 MB	663 MB	-
Indochina	5.443	158	2018	0.737	0.839	0.063	81 MB	838 MB	18.64 GB
IT	95.92	224	-	1.069	1.013	-	854 MB	4.74 GB	-
Twitter	0.027	0.134	-	0.863	0.177	-	1.14 GB	3.83 GB	-
Friendster	0.159	0.419	-	0.814	0.904	-	2.43 GB	9.14 GB	-
UK	11.49	384	-	3.443	5.858	-	1.78 GB	11.8 GB	-
Clueweb09	40.68	-	-	16.93	-	-	163 GB	-	-

Table 2: Summary of datasets.

Dataset	Network	$|V|$	$|E|$	avg. deg	avg. dist
Skitter	comp (u)	1.7M	11M	13.081	5.1
Flickr	social (u)	1.7M	16M	18.133	5.3
Hollywood	social (u)	1.1M	114M	98.913	3.9
Orkut	social (u)	3.1M	117M	76.281	4.2
Enwiki	social (d)	4.2M	101M	43.746	3.4
Livejournal	social (d)	4.8M	69M	17.679	5.6
Indochina	web (d)	7.4M	194M	40.725	7.7
IT	web (d)	41M	1.2B	49.768	7.0
Twitter	social (d)	42M	1.5B	57.741	3.6
Friendster	social (u)	66M	1.8B	55.056	5.0
UK	web (d)	106M	3.7B	62.772	6.9
Clueweb09	web (d)	1.7B	7.8B	9.27	7.4

experiments, we notice that m is usually orders of magnitudes smaller than $|V|$ and l is also significantly smaller than $|R|$.

Directed and weighted graphs. For directed graphs, we can store sets of forward and backward labels, namely $L_f(v)$ and $L_b(v)$, for each vertex v which contain pairs $(r_i, \delta_{r_i v})$ from forward and backward BFSs w.r.t. each landmark. Accordingly, we can store forward and backward highways H_f and H_b. Then, we conduct two BFSs to update these labels and highways: one in the forward direction and the other in the backward direction. Our method can also be easily extended to handling weighted graphs by using Dijkstra's algorithm instead of BFSs.

6 EXPERIMENTS

We have evaluated our method to answer the following questions: (Q1) How efficiently can our method perform against state-of-the-art methods? (Q2) How does the number of landmarks affect the performance of our method? (Q3) How does our method scale to perform updates occurring rapidly in large dynamic networks?

Datasets. We used 12 large real-world networks as detailed in Table 2. These networks are accessible at Stanford Network Analysis Project [16], Laboratory for web Algorithmics [7], Koblenz Network Collection [14], and Network Repository [17]. We treated these networks as undirected and unweighted graphs.

Updates and queries. For each network, we randomly sampled 1,000 pairs of vertices as edge insertions, denoted as E_I, where $E_I \cap E = \emptyset$ to evaluate the average update time. Further, we evaluate the average query time with 100,000 randomly sampled pairs of vertices from each network and report the labelling size after reflecting all the updates.

Baseline methods. We compared our method (IncHL$^+$) with the state-of-the-art methods: (1) IncPLL: an online incremental algorithm proposed in [4] which is based on the 2-hop cover labelling to answer distance queries; (2) IncFD: an online incremental algorithm proposed in [12] which combines a 2-hop cover labelling with a graph traversal algorithm to answer distance queries. The codes of these methods were provided by their authors and implemented in C++. We used the same parameter settings for these methods as suggested by their authors unless otherwise stated. For a fair comparison, following [12] we set $|R| = 20$ for IncFD and our methods, except for Clueweb09 which has $|R| = 150$ due to its billion-scale vertices. Our methods were implemented in C++11 and compiled using gcc 5.5.0 with the -O3 option. We performed all the experiments using a single thread on Linux server (Intel Xeon W-2175 with 2.50GHz and 512GB of main memory).

6.1 Performance Comparison

6.1.1 Update Time. Table 1 shows that the average update time of our method IncHL$^+$ outperforms the state-of-the-art methods IncFD and IncPLL on all datasets. This is due to a novel repair strategy utilized by IncHL$^+$. Further, only IncHL$^+$ can scale to very large networks with billions of vertices and edges. IncFD fails to scale to Clueweb09, and IncPLL fails for 7 out of 12 datasets due to very high preprocessing time and memory requirements.

6.1.2 Labelling Size. From Table 1, we see that IncHL$^+$ has significantly smaller labelling sizes than IncFD and IncPLL. When updates occur on a graph, the labelling sizes of IncFD and IncHL$^+$ remain stable because their average label sizes are bounded by the size of landmarks set (i.e. $|R|$). Moreover, IncFD stores complete shortest path trees w.r.t. landmarks; while IncHL$^+$ stores pruned shortest-path trees which lead to labelling of much smaller sizes than IncFD. For IncPLL, the labelling sizes may increase because IncPLL does not remove outdated and redundant entries.

6.1.3 Query Time. In Table 1 the query times of IncHL$^+$ are comparable with IncFD and IncPLL. It has been shown in [9] that query time depends on labelling size. As discussed in Section 6.1.2, the update operations do not considerably affect the labelling sizes of IncFD and IncHL$^+$, and thus their query times remain stable. However, the query times for IncPLL may increase over time because of the presence of outdated and redundant entries, which result in labelling of increasing size.
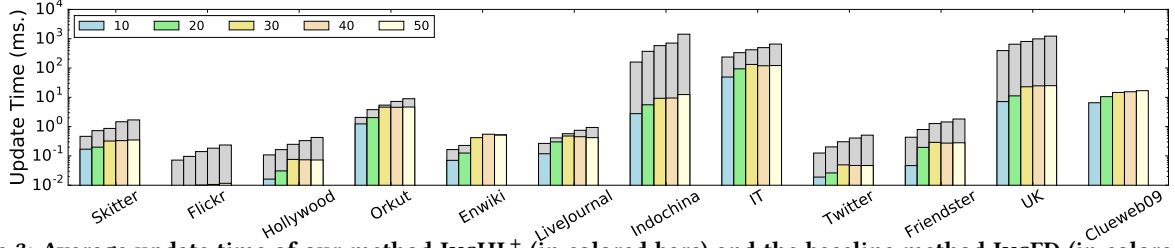
Figure 3: Average update time of our method IncHL⁺ (in colored bars) and the baseline method IncFD (in colored plus grey bars) under 10-50 landmarks. There are no results of IncFD for Clueweb09 due to the scalability issue.
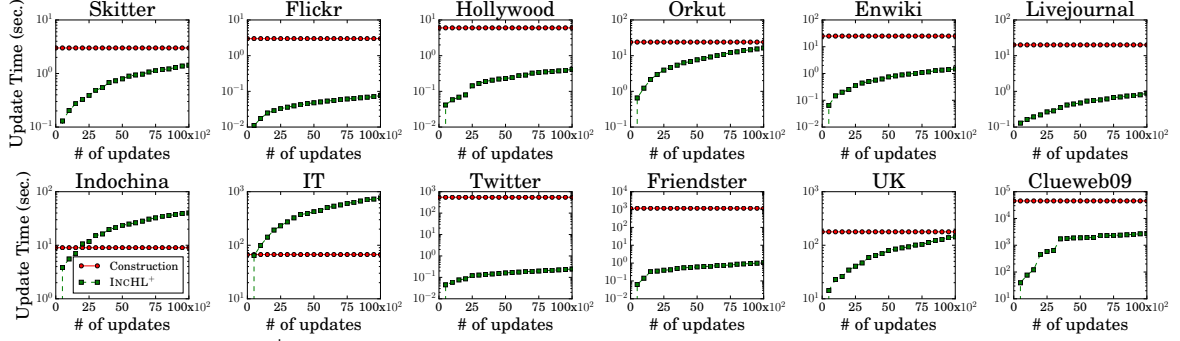

Figure 4: Update time of IncHL⁺ for performing up to 10,000 updates against construction time of labelling from scratch.

6.2 Performance with Varying Landmarks

Figure 3 shows the average update time of our method IncHL⁺ against the baseline method IncFD under varying landmarks, i.e., $|R| \in [10, 20, 30, 40, 50]$. As we can see, IncHL⁺ outperforms IncFD on all the datasets against almost every selection of landmarks. We can also see the performance gap remains stable for most of the datasets when increasing the number of landmarks. This empirically verifies the efficiency of our repair strategy.

6.3 Scalability Test

We conducted a scalability test on the update time of our method IncHL⁺, by starting with 500 updates and then iteratively adding 500 updates each time until 10,000 updates. Figure 4 shows the results. We observe that the update time of IncHL⁺ on almost all the datasets is considerably below the construction time of labelling. On Indochina and IT, IncHL⁺ performs relatively worse because these networks have large average distances as depicted in Table 2, which lead to high percentages of affected vertices as shown in Figure 1. In contrast, IncHL⁺ performs well on graphs with small average distances such as Twitter. Overall, IncHL⁺ can scale to perform a large number of updates efficiently.

7 CONCLUSION

This paper has studied the problem of answering distance queries on large dynamic networks. Our proposed algorithm exploits properties of a recent labelling technique called highway cover labelling [10] to efficiently process incremental graph updates, and can preserve the minimality property of labelling after each update operation. We have empirically evaluated the efficiency and scalability of the proposed algorithm. The results show that our proposed algorithm outperforms the state-of-the-art methods. In future, we plan to further investigate the effects of decremental updates on graphs since they are also commonly used in practice.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*. 230–241.
[2] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2012. Hierarchical hub labelings for shortest paths. In *ESA*. 24–35.
[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *ACM SIGMOD*. 349–360.
[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2014. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*. 237–248.
[5] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *ACM SIGKDD*. 44–54.
[6] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. 2006. Complex networks: Structure and dynamics. *Physics reports* 424, 4-5 (2006), 175–308.
[7] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW*. 595–601.
[8] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.
[9] Gianlorenzo D'angelo, Mattia D'emidio, and Daniele Frigioni. 2019. Fully Dynamic 2-Hop Cover Labeling. *JEA* 24, 1 (2019), 1–6.
[10] Muhammad Farhan, Qing Wang, Yu Lin, and Brendan McKay. 2019. A Highly Scalable Labelling Approach for Exact Distance Queries in Complex Networks. In *EDBT*. 13–24.
[11] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. 2013. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *VLDB* 6, 6 (2013), 457–468.
[12] Takanori Hayashi, Takuya Akiba, and Ken-ichi Kawarabayashi. 2016. Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks. In *CIKM*. 1533–1542.
[13] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. 2012. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *ACM SIGMOD*. 445–456.
[14] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *WWW*. 1343–1350.
[15] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM TKDD* 1, 1 (2007), 2–es.
[16] Jure Leskovec and Andrej Krevl. 2015. SNAP Datasets:Stanford Large Network Dataset Collection. (2015).
[17] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *AAAI*.
[18] Robert Endre Tarjan. 1983. *Data structures and network algorithms*. Vol. 44. Siam.
[19] Antti Ukkonen, Carlos Castillo, Debora Donato, and Aristides Gionis. 2008. Searching the wikipedia with contextual information. In *CIKM*. 1351–1352.
[20] Monique V Vieira, Bruno M Fonseca, Rodrigo Damazio, Paulo B Golgher, Davi de Castro Reis, and Berthier Ribeiro-Neto. 2007. Efficient search ranking in social networks. In *CIKM*. 563–572.
[21] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P Gummadi. 2009. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*. 37–42.
[22] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *ACM SIGMOD*. 99–110.

CONTENTS

Efficient Maintenance of Distance Labelling for Incremental Updates in Large Dynamic Graphs

Muhammad Farhan
Australian National University
Canberra, Australia
muhammad.farhan@anu.edu.au

Qing Wang
Australian National University
Canberra, Australia
qing.wang@anu.edu.au

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 been successfully attempted on this problem, most of which is limited to static graphs. Due to the dynamic nature of real-world networks, there is a pressing need to address this problem for dynamic networks undergoing changes. In this paper, we propose an *online incremental* method to efficiently answer distance queries over very large dynamic graphs. Our proposed method incorporates incremental update operations, i.e. edge and vertex additions, into a highly scalable framework of answering distance queries. We theoretically prove the correctness of our method and the preservation of labelling minimality. We have also conducted extensive experiments on 12 large real-world networks to empirically verify the efficiency, scalability, and robustness of our method.

1 INTRODUCTION

Given a very large graph with billions of vertices and edges, how efficiently can we find the shortest path distance between any two vertices? If such a graph is dynamically changing over time (e.g. inserting edges or vertices), how can we not only efficiently but also accurately find the shortest path distance between any two vertices? These questions are intimately related to distance queries on dynamic graphs. As one of the most fundamental operations on graphs, distance queries have a wide range of real-world applications that operate on increasingly large dynamic graphs, such as context-aware search in web graphs [19], social network analysis in social networks [5, 20], management of resources in computer networks [6], and so on. Many of these applications use distance queries as a building block to realise more complicated tasks, and require distance queries to be answered instantly, e.g. in the order of milliseconds.

Previous studies have primarily focused on distance queries on static graphs [1–3, 10, 11, 13, 22], with little attention being paid to dynamics on graphs. To speed up query response time, a key technique is to precompute a data structure called *distance labelling* that satisfies certain properties such as 2-hop cover [8], and then use this data structure to answer distance queries efficiently. However, when a graph dynamically changes, its distance labelling needs to be changed accordingly; otherwise, distance queries may yield overestimated distances. Although it is possible to recompute a distance labelling from scratch, this leads to inefficiency. As shown in Figure 1, the percentage of affected vertices by a single change often ranges from $10^{-5}\%$ to 10% in various real-world networks, recomputing distance labelling from scratch for each single change not only wastes
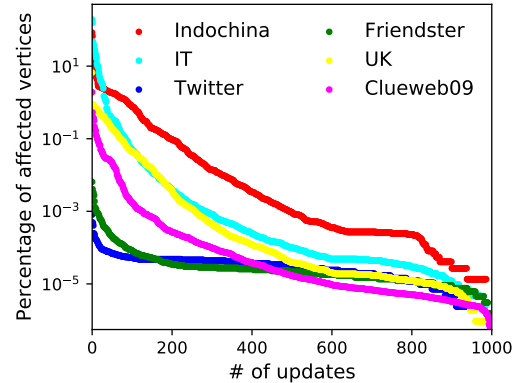
Figure 1: Distribution of affected vertices by a single graph change in various networks, where the results for 1000 graph changes are sorted in the descending order.

computing resources, but also may generate inaccurate query results during recomputing process. The question arising is thus how to efficiently and accurately change distance labelling on dynamic graphs in order to support distance queries?

In this paper, we aim to develop an *online incremental* method that can dynamically maintain distance labelling on graphs being changed by edge and vertex insertions. Typically, real-world dynamic networks are more vulnerable to insertions than removals and a plethora of such real-world networks are large and frequently updated, primarily accommodating insertions [15, 21]. Thus, an online incremental method for dynamic graphs should possess the following desirable characteristics: (1) *time efficiency* - It can answer distance queries and update distance labelling efficiently (in the order of milliseconds); 2) *space efficiency* - It guarantees the minimum size of distance labelling to reduce storage costs; (3) *scalability* - It can scale to very large networks with billions of vertices and edges.

Challenges. Designing online incremental methods for distance queries on dynamic graphs is known to be challenging [4]. When an edge or a vertex is inserted into a graph, outdated and redundant entries of distance labelling may occur. It was reported that removing such entries is a complicated task [4] because affected vertices need to be precisely identified so as to update their labels without violating the original properties of a distance labelling such as minimality. 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 [4, 12] mostly considered 2-hop labelling, which has very high space requirements and index construction time; as a result, their query and update performance

are 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 labeling method (PLL) [4], still yields a distance labelling whose size is 20-30 times larger than the original size of a dataset.

Contributions. Our contributions are summarised as follows:

- Our method overcomes the challenge of eliminating outdated and redundant distance entries. None of the previous studies have addressed this challenge because detecting those entries is too costly [4, 9]. When an edge or a vertex is inserted, previous studies only add new distance entries or modify existing distance entries. This would however lead to an ever increasing size of labelling, particularly when a graph is frequently updated by newly added edges or vertices. Accordingly, both query performance and space efficiency would deteriorate over time.
- We prove the correctness of our proposed method and show that it preserves the desirable property of minimality on our distance labelling. Due to a property called highway cover [10], the minimal size of a distance labelling in this work is much smaller than the size of a 2-hop labelling in previous work [4, 12]. Preserving minimality on a distance labelling thus improves space efficiency and query performance, as well as update performance. We also provide a complexity analysis of our proposed method.
- We conducted experiments using 12 real-world large networks across different domains to show the efficiency, scalability and robustness of our method. Particularly, our method can perform updates under one second, on average, even on billion-scale networks, while still answering queries efficiently in the order of milliseconds and guaranteeing the labelling size of a graph to be much smaller.

2 RELATED WORK

Answering shortest-path distance queries in graphs has been an active research topic for many years. Traditionally, a distance query can be answered using Dijkstra's algorithm [18] on positively weighted graphs or Breadth-First Search (BFS) algorithm on unweighted graphs. However, these traditional algorithms fail to achieve desired response time for distance queries on large graphs. Later, labelling-based methods have emerged as an attractive way of accelerating response time to distance queries [1–3, 8, 10, 11, 13], among which Akiba et al. [3] proposed a pruned landmark labeling (PLL) to precompute a 2-hop cover distance labelling [8]. This method serves as the state-of-the-art for labelling-based distance queries and can handle graphs with hundreds of millions of edges.

So far, only a few attempts have been made to study distance queries over dynamic graphs [4, 12], which are all based on the idea of 2-hop distance labelling or its variants. Akiba et al. [4] studied the problem of updating a pruned landmark labelling for incremental updates (i.e. vertex additions and edge additions). This work however does not remove redundant entries in distance labels because the authors considered that detecting such outdated entries is too costly. This inevitably breaks the minimality of pruned landmark labelling, leading to an ever increase of labelling size and deteriorated query performance over time. 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. [12] proposed

a fully dynamic approach that selects a small set of landmarks R and precompute a shortest-path tree (SPT) rooted at each $r \in R$. Then, an online search is conducted on a sparsified graph under an upper distance bound being computed via the SPTs. Nevertheless, this method still fails to construct labelling on networks with billions of vertices. Following the same line, a recent work by Farhan et al. [10] introduced a highway-cover labelling method (HL), which can provide fast response time (milliseconds) for distance queries even on billion-scale graphs. However, this approach only works for static graphs.

3 PROBLEM FORMULATION

Let $G = (V, E)$ be an undirected graph where V is a set of vertices and E is a set of edges. We denote by $N(v)$ the set of neighbors 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. Given a graph $G = (V, E)$, an *edge insertion* is to add an edge (a, b) into G where $\{a, b\} \subseteq V$ and $(a, b) \notin E$. Accordingly, a *node insertion* is to add a new node into G together with a set of edge insertions that connect v to existing vertices in G. The following fact is critical for designing algorithms for an edge insertion.

FACT 3.1. *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 $s, t \in V$, $d_G(s, t) \geq d_{G'}(s, t)$.*

That is, the distance between any two vertices never increases after inserting edges or vertices in a graph.

Highway cover labelling. Unlike the previous work [4, 9, 12] that uses 2-hop cover labelling [8], we develop our method using a highly scalable labelling approach, called *highway cover labelling* [10]. Let $R \subseteq V$ be a small set of *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 whose *size* is defined as: $size(L) = \sum_{v \in V} |L(v)|$. A *highway* $H = (R, \delta_H)$ 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 3.2. A *highway cover labelling* is a pair $\Gamma = (H, L)$ where H is a highway and L is a distance labelling s.t. for any vertex $v \in V \backslash R$ and $r \in R$, we have:

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

Highway cover labelling enjoys several nice theoretical properties, such as minimality and order independence. A minimal highway cover labelling can be efficiently constructed, independently of the order of applying landmarks [10].

Given a highway cover labeling $\Gamma = (H, L)$, an upper bound on the distance between any two vertices $u, v \in V \backslash R$ is computed:

$$d_{uv}^\top = \min\{\delta_L(r_i, u) + \delta_H(r_i, r_j) + \delta_L(r_j, v)|$$
$$(r_i, \delta_L(r_i, u)) \in L(u), (r_j, \delta_L(r_j, v)) \in L(v)\} \quad (2)$$

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 $d_{uv}^\top$ such that:

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

Problem definition. In this work, we study the problem of answering distance queries over a graph that is dynamically changed by edge and vertex insertions over time. Since a vertex insertion can be treated as a set of edge insertions, without loss of generality, below we define the problem based on edge insertions.

Definition 3.3. Let $G \hookrightarrow G'$ denote that a graph G is changed to a graph G' by an edge insertion. The *dynamic distance querying* problem is, given any two vertices u and v in the changed graph G', to efficiently compute the distance $d_{G'}(u, v)$.

4 ONLINE INCREMENTAL ALGORITHM

In this section, we propose an algorithm IncHL$^+$ to incrementally update labelling to reflect graph changes. Algorithm 1 describes the main steps of IncHL$^+$. Below, we discuss them in detail.

4.1 Finding Affected Vertices

When an update operation occurs on a graph $G = (V, E)$, there exists a subset of "affected" vertices in V whose labels need to be updated as a consequence of this update operation on the graph.

Definition 4.1. A vertex $v \in V$ is *affected* by $G \hookrightarrow G'$ iff $P_G(v, r) \neq P_{G'}(v, r)$ for at least one $r \in R$; *unaffected* otherwise.

We use Λ_r to denote the set of all affected vertices w.r.t. a landmark r and $\Lambda = \bigcup_{r \in R} \Lambda_r$ the set of all affected vertices.

Example 4.2. Consider Figure 2(a) in which 0 and 10 are two landmarks. After inserting an edge $(2, 5)$, $\Lambda_0 = \{5, 8, 9, 10, 13, 14\}$ in Figure 2(b) and $\Lambda_{10} = \{0, 1, 2\}$ in Figure 2(d).

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

Lemma 4.3. 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).

Following Lemma 4.3, we can reduce the search space of affected vertices by eliminating landmarks r with $d_G(r, a) = d_G(r, b)$ since $\Lambda_r = \emptyset$ in such a case. 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. Further, by the lemma below, we can also reduce the search space by "jumping" from the root of a BFS to vertex b.

Lemma 4.4. When $G \hookrightarrow G'$ with an inserted edge (a, b), we have $d_G(v, r) \geq d_G(a, r) + 1$ for any affected vertex $v \in \Lambda_r$.

PROOF. By Lemma 4.3, there exists a shortest path from any affected vertex v to r going through the 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, $d_G(v, r) \geq d_G(a, r) + 1$ thus holds. □

Algorithm 2 describes our algorithm for finding affected vertices. Given a graph G with an inserted edge (a, b) and a highway cover labelling $\Gamma = (H, L)$ over G, we conduct a *jumped* BFS w.r.t. a landmark r starting from the vertex b with its new depth $\pi = Q(r, a, \Gamma) + 1$ (Lines 3-4). For every $(v, \pi) \in Q$, we enqueue all the neighbors of v that are affected into Q with new distances $\pi + 1$ (Lines 7-8) and add v to Λ_r as affected vertex (Line 9). This process continues until Q is empty.

Example 4.5. Figure 2 illustrates how our algorithm finds affected vertices as a result of inserting an edge $(2, 5)$. The BFS rooted at landmark 0 is depicted in Figure 2(b), which jumps to vertex 5 and finds six affected vertices $\{5, 8, 9, 10, 13, 14\}$. Similarly, the BFS rooted at landmark 10 is depicted in Figure 2(d), which jumps to vertex 2 and finds three affected vertices $\{0, 1, 2\}$.

Algorithm 1: Incremental algorithm (IncHL$^+$).

Input: $G, G', (a, b), \Gamma = (H, L)$
Output: $\Gamma' = (H', L')$

1 **foreach** $r \in R$ **do**
2     $\Lambda_r \leftarrow$ FINDAFFECTED$(G, (a, b), r, \Gamma)$
3   REPAIRAFFECTED$(G', (a, b), \Lambda_r, r, \Gamma)$

Algorithm 2: Finding affected vertices.

1 **Function** FindAffected$(G, (a, b), r, \Gamma)$
2     $Q \leftarrow \emptyset, \Lambda_r \leftarrow \emptyset$
3     $\pi \leftarrow Q(r, a, \Gamma) + 1$
4   Enqueue (b, π) to Q
5   **while** Q *is not empty* **do**
6   Dequeue (v, π) from Q
7   **foreach** $w \in N(v)$ *s.t.* $Q(r, w, \Gamma) \geq \pi + 1$ **do**
8   Enqueue $(w, \pi + 1)$ to Q
9        $\Lambda_r = \Lambda_r \cup \{v\}$
10   **return** Λ_r

4.2 Repairing Affected Vertices

Now we propose a repair strategy to efficiently update the labels of affected vertices in order to reflect graph changes. The key idea is that, instead of conducting a full BFS on all vertices, we conduct a partial BFS from b only on affected vertices. Further, to avoid unnecessary computations, we distinguish two kinds of affected vertices: (1) affected vertices that are *covered* by other landmarks and can thus be easily repaired by removing an entry from their labels; (2) affected vertices whose labels need to be repaired with accurately calculated distances on a changed graph. The following lemma characterizes the first kind according to the definition of highway cover labelling.

Lemma 4.6. An affected vertex $v \in \Lambda_r$ is covered *by a landmark* $r' \in R\backslash\{r\}$ iff r' exists in $P_{G'}(v, r)$. If an affected vertex $v \in \Lambda_r$ is covered by r', then any affected vertex $v' \in \Lambda_r$ satisfying $d_{G'}(r, v') = d_{G'}(r, v) + d_{G'}(v, v')$ must also be covered by r'.

By Lemma 4.6, we can efficiently repair affected vertices $v \in \Lambda_r$ as follows. If v is covered by a landmark $r' \in R\backslash\{r\}$ (i.e., one of the unaffected parents of v does not contain r in its label) and is also a landmark, we only update the highway; otherwise, we remove the entry of r from $L(v)$. If v is not covered by any $r' \in R\backslash\{r\}$, we add/modify the entry of r in $L(v)$. If v is a descendant of covered vertices, we simply remove the entry of r from $L(v)$ (if exists).

Algorithm 3 describes our algorithm for repairing affected vertices. Given a graph G with an inserted edge (a, b) and a set of affected vertices Λ_r, we conduct a BFS w.r.t. a landmark r starting from the vertex b with its new distance $\pi = d_G(r, a) + 1$ (Lines 3-4). We use two queues $Q_{uncovered}$ and $Q_{covered}$ to process uncovered and covered vertices, respectively. If b is covered, we enqueue (b, π) to $Q_{covered}$ and remove the entry of r from the labels of affected vertices (Line 25). Otherwise, we enqueue (b, π) to $Q_{uncovered}$ and start processing vertices in $Q_{uncovered}$ (Line 5). For each vertex $v \in Q_{uncovered}$ at depth π, we examine its affected neighbors w at depth $\pi + 1$. If w is covered, then if w is a landmark, we update the highway (Line 10); otherwise we remove the entry of r from $L(w)$ (Line 12) because there must exist another landmark in the shortest path from w to r and add
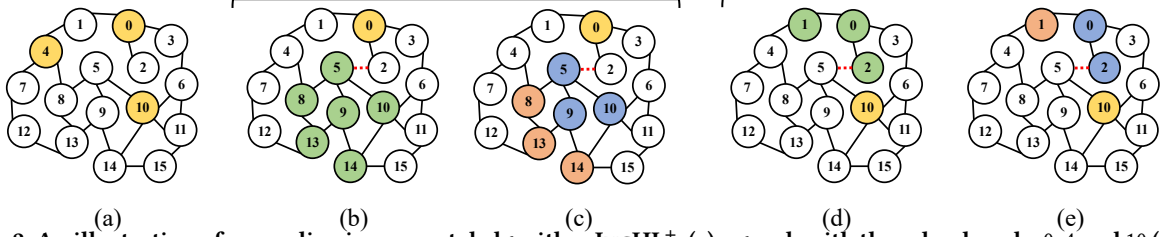
Figure 2: An illustration of our online incremental algorithm IncHL$^+$: (a) a graph with three landmarks 0, 4 and 10 (colored in yellow); (b) and (d) the BFSs for finding affected vertices (colored in green) w.r.t. landmarks 0 and 10, respectively; (c) and (e) the BFSs for repairing affected vertices w.r.t. landmarks 0 and 10, respectively, where vertices with added/modified entries are colored in blue, and vertices with removed entries are colored in red.

Algorithm 3: Repairing affected vertices.

1 **Function** RepairAffected(G', (a, b), Λ_r, r, Γ)
2    $Q_{uncovered} \leftarrow \emptyset$, $Q_{covered} \leftarrow \emptyset$
3    $\pi \leftarrow d_G(r, a) + 1$
4   Enqueue (b, π) to $Q_{covered}$ if covered; otherwise to $Q_{uncovered}$
5   **while** $Q_{uncovered}$ is not empty **do**
6   **while** $(v, \pi) \in Q_{uncovered}$ at depth π **do**
7   **forall** $w \in N(v)$ s.t. $w \in \Lambda_r$ at depth $\pi + 1$ **do**
8   **if** $covered(w, \pi + 1)$ **then**
9   **if** w is a landmark **then**
10              $\delta_H(r, w) \leftarrow \pi + 1$
11   **else**
12   Remove r from $L(w)$
13   Enqueue $(w, \pi + 1)$ to $Q_{covered}$
14   **else**
15   Add/Modify $\{(r, \pi + 1)\}$ in $L(w)$
16   Enqueue $(w, \pi + 1)$ to $Q_{uncovered}$
17   Remove w from Λ_r
18   Dequeue (v, π) from $Q_{uncovered}$
19   **while** $(v, \pi) \in Q_{covered}$ at depth π **do**
20   **forall** $w \in N(v)$ s.t. $w \in \Lambda_r$ at depth $\pi + 1$ **do**
21   Remove r from $L(w)$
22   Remove w from Λ_r
23   Enqueue $(w, \pi + 1)$ to $Q_{covered}$
24   Dequeue (v, π) from $Q_{covered}$
25   Remove entry r from remaining vertices in $Q_{covered}$

$(w, \pi + 1)$ to $Q_{covered}$ (Line 13). Otherwise, we add/modify the entry of r with the new distance $\pi + 1$ in $L(w)$ and enqueue w to $Q_{uncovered}$ (Lines 15-16). After that, we remove w from Λ_r (line 17). Then, for each $(v, \pi) \in Q_{covered}$, we remove r from the labels of affected neighbors of v, remove these affected vertices from Λ_r and enqueue them to $Q_{covered}$ (Lines 19-24). We process these two queues, one after the other, until $Q_{uncovered}$ is empty. Finally, we remove the entry of r from the labels of the remaining vertices in $Q_{covered}$ (Line 25).

Example 4.7. Figure 2 illustrates how our algorithm repairs labels as a result of inserting an edge $(2, 5)$. The BFS for landmark 0 is depicted in Figure 2(c), which jumps to vertex 5 and repairs three affected vertices $\{5, 9, 10\}$. The vertices $\{8, 13, 14\}$ are covered by landmarks 4 and 10. Similarly, the BFS for landmark 10 is depicted in Figure 2(e), in which vertices $\{0, 2\}$ are repaired and vertex 1 is covered by landmarks 0 and 4.

5 THEORETICAL RESULTS

Proof of correctness. For $G \hookrightarrow G'$ where our method IncHL$^+$ updates a highway cover labelling Γ over G into a highway cover labelling Γ' over G', we consider IncHL$^+$ to be *correct* iff, whenever $Q(u, v, \Gamma) = d_G(u, v)$ holds for any two vertices u and v in G, then $Q(u', v', \Gamma') = d_{G'}(u', v')$ also holds for any two vertices u' and v' in G'. We prove the theorem below for IncHL$^+$.

THEOREM 5.1. *IncHL$^+$ is correct.*

PROOF. First, we prove that FindAffected returns the set of all affected vertices Λ_r as a result of an edge insertion. IncHL$^+$ (Lines 7-8 of Algorithm 2) guarantees that any vertex being added to Q has one shortest path to a landmark r which goes through the inserted edge (a, b). By Lemma 4.3, such vertices are affected vertices, and thus a vertex v is added to Q in Algorithm 2 iff $v \in \Lambda_r$. Then, we prove that RepairAffected repairs $\Gamma = (H, L)$ s.t. (1) $(r, d_{G'}(r, v)) \in L(v)$ for $v \in \Lambda_r$, iff $P_{G'}(r, v)$ contains only one landmark r; (2) $\delta_H(r, r') = d_{G'}(r, r')$ for any $r' \in R \backslash \{r\}$. Starting from b with new distance π, the distances of affected vertices in Λ_r are iteratively inferred on G' and reflected into their labels via $Q_{uncovered}$ if these affected vertices are not covered (Lines 15-16 of Algorithm 3). If an affected vertex v is covered, it is kept in $Q_{covered}$; if v is also a landmark, $\delta_H(r, v)$ in H is updated (Lines 9-10). Thus, the distance entry of r is removed from the labels of affected vertices appearing in $Q_{covered}$, whereas any vertex v appearing in $Q_{uncovered}$ must have $(r, d_{G'}(r, v)) \in L(v)$. □

Preservation of minimality. It has been reported in [10] that, given a graph G, a minimal highway cover labelling $\Gamma = (H, L)$ of G can be constructed using an algorithm proposed in their work, i.e., $size(L') \geq size(L)$ holds for any $\Gamma' = (H, L')$ of G. For $G \hookrightarrow G'$ where IncHL$^+$ updates Γ over G into Γ' over G', we prove that IncHL$^+$ preserves the minimality of labelling.

THEOREM 5.2. *If Γ is minimal on G, then Γ' is minimal on G'.*

PROOF. By Lemma 4.6, $(r, d_{G'}(r, v)) \in L(v)$ for $v \in \Lambda_r$ iff $P_{G'}(r, v)$ does not contain any other landmark $R \backslash \{r\}$; otherwise we remove the entry of r from the label of v (Line 12, 21 and 25 of Algorithm 3). Thus, the labels of all affected vertices must be minimal after applying IncHL$^+$. For unaffected vertices, their labels should remain unchanged. Hence, Γ' must be minimal. □

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 of vertices. For a landmark, Algorithm 2 takes $O(mdl)$ time to find all affected vertices and Algorithm 3 takes $O(md)$ to repair the labels of all affected vertices. We omit l from $O(md)$ for Algorithm 3 because distances for all unaffected neighbors of affected vertices are stored in Algorithm 2. Therefore, IncHL$^+$ has time complexity $O(|R| \times mdl)$. In our

Table 1: Comparing the update time, query time and labelling size of our method with the baseline methods.

Dataset	Update Time (ms)			Query Time (ms)			Labelling Size		
	IncHL$^+$	IncFD	IncPLL	IncHL$^+$	IncFD	IncPLL	IncHL$^+$	IncFD	IncPLL
Skitter	0.194	0.444	2.05	0.027	0.019	0.047	42 MB	153 MB	2.44 GB
Flickr	0.006	0.074	1.73	0.007	0.012	0.064	34 MB	152 MB	3.69 GB
Hollywood	0.031	0.101	48	0.027	0.037	0.109	27 MB	263 MB	12.58 GB
Orkut	2.026	2.049	-	0.101	0.103	-	70 MB	711 MB	-
Enwiki	0.134	0.163	5.91	0.054	0.035	0.071	82 MB	608 MB	12.57 GB
Livejournal	0.245	0.268	-	0.044	0.046	-	122 MB	663 MB	-
Indochina	5.443	158	2018	0.737	0.839	0.063	81 MB	838 MB	18.64 GB
IT	95.92	224	-	1.069	1.013	-	854 MB	4.74 GB	-
Twitter	0.027	0.134	-	0.863	0.177	-	1.14 GB	3.83 GB	-
Friendster	0.159	0.419	-	0.814	0.904	-	2.43 GB	9.14 GB	-
UK	11.49	384	-	3.443	5.858	-	1.78 GB	11.8 GB	-
Clueweb09	40.68	-	-	16.93	-	-	163 GB	-	-

Table 2: Summary of datasets.

Dataset	Network	$|V|$	$|E|$	avg. deg	avg. dist
Skitter	comp (u)	1.7M	11M	13.081	5.1
Flickr	social (u)	1.7M	16M	18.133	5.3
Hollywood	social (u)	1.1M	114M	98.913	3.9
Orkut	social (u)	3.1M	117M	76.281	4.2
Enwiki	social (d)	4.2M	101M	43.746	3.4
Livejournal	social (d)	4.8M	69M	17.679	5.6
Indochina	web (d)	7.4M	194M	40.725	7.7
IT	web (d)	41M	1.2B	49.768	7.0
Twitter	social (d)	42M	1.5B	57.741	3.6
Friendster	social (u)	66M	1.8B	55.056	5.0
UK	web (d)	106M	3.7B	62.772	6.9
Clueweb09	web (d)	1.7B	7.8B	9.27	7.4

experiments, we notice that m is usually orders of magnitudes smaller than $|V|$ and l is also significantly smaller than $|R|$.

Directed and weighted graphs. For directed graphs, we can store sets of forward and backward labels, namely $L_f(v)$ and $L_b(v)$, for each vertex v which contain pairs $(r_i, \delta_{r_i v})$ from forward and backward BFSs w.r.t. each landmark. Accordingly, we can store forward and backward highways H_f and H_b. Then, we conduct two BFSs to update these labels and highways: one in the forward direction and the other in the backward direction. Our method can also be easily extended to handling weighted graphs by using Dijkstra's algorithm instead of BFSs.

6 EXPERIMENTS

We have evaluated our method to answer the following questions: (Q1) How efficiently can our method perform against state-of-the-art methods? (Q2) How does the number of landmarks affect the performance of our method? (Q3) How does our method scale to perform updates occurring rapidly in large dynamic networks?

Datasets. We used 12 large real-world networks as detailed in Table 2. These networks are accessible at Stanford Network Analysis Project [16], Laboratory for web Algorithmics [7], Koblenz Network Collection [14], and Network Repository [17]. We treated these networks as undirected and unweighted graphs.

Updates and queries. For each network, we randomly sampled 1,000 pairs of vertices as edge insertions, denoted as E_I, where $E_I \cap E = \emptyset$ to evaluate the average update time. Further, we evaluate the average query time with 100,000 randomly sampled pairs of vertices from each network and report the labelling size after reflecting all the updates.

Baseline methods. We compared our method (IncHL$^+$) with the state-of-the-art methods: (1) IncPLL: an online incremental algorithm proposed in [4] which is based on the 2-hop cover labelling to answer distance queries; (2) IncFD: an online incremental algorithm proposed in [12] which combines a 2-hop cover labelling with a graph traversal algorithm to answer distance queries. The codes of these methods were provided by their authors and implemented in C++. We used the same parameter settings for these methods as suggested by their authors unless otherwise stated. For a fair comparison, following [12] we set $|R| = 20$ for IncFD and our methods, except for Clueweb09 which has $|R| = 150$ due to its billion-scale vertices. Our methods were implemented in C++11 and compiled using gcc 5.5.0 with the -O3 option. We performed all the experiments using a single thread on Linux server (Intel Xeon W-2175 with 2.50GHz and 512GB of main memory).

6.1 Performance Comparison

6.1.1 Update Time. Table 1 shows that the average update time of our method IncHL$^+$ outperforms the state-of-the-art methods IncFD and IncPLL on all datasets. This is due to a novel repair strategy utilized by IncHL$^+$. Further, only IncHL$^+$ can scale to very large networks with billions of vertices and edges. IncFD fails to scale to Clueweb09, and IncPLL fails for 7 out of 12 datasets due to very high preprocessing time and memory requirements.

6.1.2 Labelling Size. From Table 1, we see that IncHL$^+$ has significantly smaller labelling sizes than IncFD and IncPLL. When updates occur on a graph, the labelling sizes of IncFD and IncHL$^+$ remain stable because their average label sizes are bounded by the size of landmarks set (i.e. $|R|$). Moreover, IncFD stores complete shortest path trees w.r.t. landmarks; while IncHL$^+$ stores pruned shortest-path trees which lead to labelling of much smaller sizes than IncFD. For IncPLL, the labelling sizes may increase because IncPLL does not remove outdated and redundant entries.

6.1.3 Query Time. In Table 1 the query times of IncHL$^+$ are comparable with IncFD and IncPLL. It has been shown in [9] that query time depends on labelling size. As discussed in Section 6.1.2, the update operations do not considerably affect the labelling sizes of IncFD and IncHL$^+$, and thus their query times remain stable. However, the query times for IncPLL may increase over time because of the presence of outdated and redundant entries, which result in labelling of increasing size.
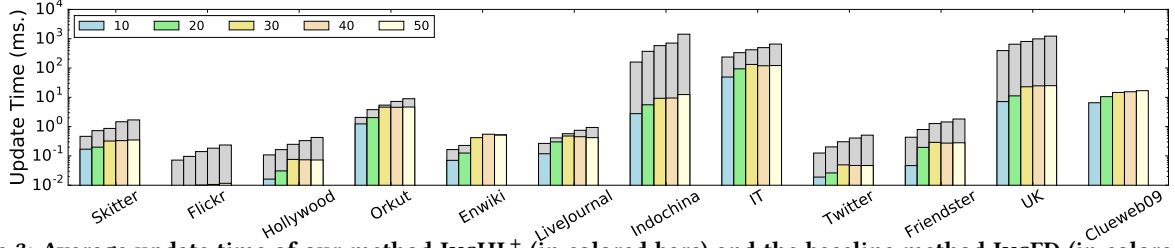
Figure 3: Average update time of our method IncHL⁺ (in colored bars) and the baseline method IncFD (in colored plus grey bars) under 10-50 landmarks. There are no results of IncFD for Clueweb09 due to the scalability issue.
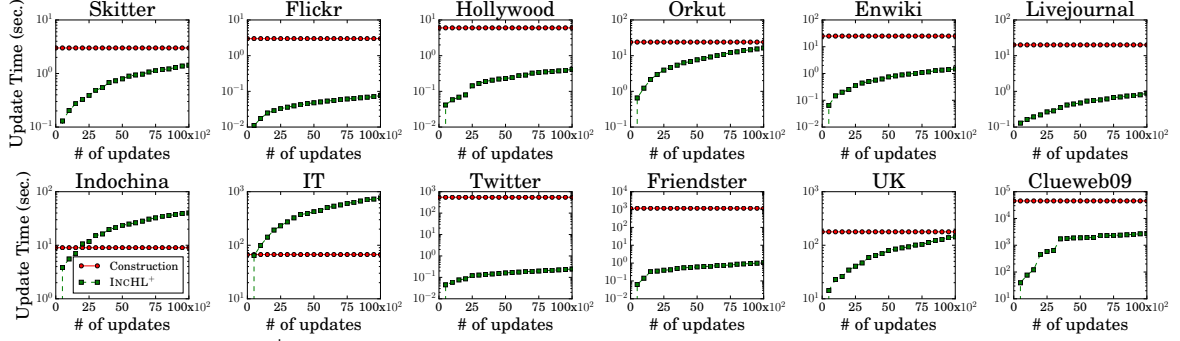

Figure 4: Update time of IncHL⁺ for performing up to 10,000 updates against construction time of labelling from scratch.

6.2 Performance with Varying Landmarks

Figure 3 shows the average update time of our method IncHL⁺ against the baseline method IncFD under varying landmarks, i.e., $|R| \in [10, 20, 30, 40, 50]$. As we can see, IncHL⁺ outperforms IncFD on all the datasets against almost every selection of landmarks. We can also see the performance gap remains stable for most of the datasets when increasing the number of landmarks. This empirically verifies the efficiency of our repair strategy.

6.3 Scalability Test

We conducted a scalability test on the update time of our method IncHL⁺, by starting with 500 updates and then iteratively adding 500 updates each time until 10,000 updates. Figure 4 shows the results. We observe that the update time of IncHL⁺ on almost all the datasets is considerably below the construction time of labelling. On Indochina and IT, IncHL⁺ performs relatively worse because these networks have large average distances as depicted in Table 2, which lead to high percentages of affected vertices as shown in Figure 1. In contrast, IncHL⁺ performs well on graphs with small average distances such as Twitter. Overall, IncHL⁺ can scale to perform a large number of updates efficiently.

7 CONCLUSION

This paper has studied the problem of answering distance queries on large dynamic networks. Our proposed algorithm exploits properties of a recent labelling technique called highway cover labelling [10] to efficiently process incremental graph updates, and can preserve the minimality property of labelling after each update operation. We have empirically evaluated the efficiency and scalability of the proposed algorithm. The results show that our proposed algorithm outperforms the state-of-the-art methods. In future, we plan to further investigate the effects of decremental updates on graphs since they are also commonly used in practice.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*. 230–241.
[2] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2012. Hierarchical hub labelings for shortest paths. In *ESA*. 24–35.
[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *ACM SIGMOD*. 349–360.
[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2014. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*. 237–248.
[5] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *ACM SIGKDD*. 44–54.
[6] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. 2006. Complex networks: Structure and dynamics. *Physics reports* 424, 4-5 (2006), 175–308.
[7] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW*. 595–601.
[8] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.
[9] Gianlorenzo D'angelo, Mattia D'emidio, and Daniele Frigioni. 2019. Fully Dynamic 2-Hop Cover Labeling. *JEA* 24, 1 (2019), 1–6.
[10] Muhammad Farhan, Qing Wang, Yu Lin, and Brendan McKay. 2019. A Highly Scalable Labelling Approach for Exact Distance Queries in Complex Networks. In *EDBT*. 13–24.
[11] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. 2013. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *VLDB* 6, 6 (2013), 457–468.
[12] Takanori Hayashi, Takuya Akiba, and Ken-ichi Kawarabayashi. 2016. Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks. In *CIKM*. 1533–1542.
[13] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. 2012. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *ACM SIGMOD*. 445–456.
[14] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *WWW*. 1343–1350.
[15] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM TKDD* 1, 1 (2007), 2–es.
[16] Jure Leskovec and Andrej Krevl. 2015. SNAP Datasets:Stanford Large Network Dataset Collection. (2015).
[17] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *AAAI*.
[18] Robert Endre Tarjan. 1983. *Data structures and network algorithms*. Vol. 44. Siam.
[19] Antti Ukkonen, Carlos Castillo, Debora Donato, and Aristides Gionis. 2008. Searching the wikipedia with contextual information. In *CIKM*. 1351–1352.
[20] Monique V Vieira, Bruno M Fonseca, Rodrigo Damazio, Paulo B Golgher, Davi de Castro Reis, and Berthier Ribeiro-Neto. 2007. Efficient search ranking in social networks. In *CIKM*. 563–572.
[21] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P Gummadi. 2009. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*. 37–42.
[22] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *ACM SIGMOD*. 99–110.