BatchHL: Answering Distance Queries on Batch-Dynamic Networks at Scale

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

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

Henning Koehler
Massey University
Palmerston North, New Zealand
H.Koehler@massey.ac.nz

ABSTRACT

Many real-world applications operate on dynamic graphs that undergo rapid changes in their topological structure over time. However, it is challenging to design dynamic algorithms that are capable of supporting such graph changes efficiently. To circumvent the challenge, we propose a batch-dynamic framework for answering distance queries, which combines offline labelling and online searching to leverage the advantages from both sides - accelerating query processing through a partial distance labelling that is of limited size but provides a good approximation to bound online searches. We devise batch-dynamic algorithms to dynamize a distance labelling efficiently in order to reflect batch updates on the underlying graph. In addition to providing theoretical analysis for the correctness, labelling minimality, and computational complexity, we have conducted experiments on 14 real-world networks to empirically verify the efficiency and scalability of the proposed algorithms.

CCS CONCEPTS

• **Theory of computation → Data structures and algorithms for data management**;

KEYWORDS

Shortest-path distance; batch-dynamic graphs; 2-hop cover; highway cover; distance labelling maintenance; graph algorithms

ACM Reference Format:
Muhammad Farhan, Qing Wang, and Henning Koehler. 2022. BatchHL: Answering Distance Queries on Batch-Dynamic Networks at Scale. In *Proceedings of the 2022 International Conference onManagement of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517883

1 INTRODUCTION

Graphs in real-world applications are typically dynamic, undergoing discrete changes in their topological structure by either adding or deleting edges and vertices. However, due to the rapid nature of data acquisition, it is often unrealistic to process single changes sequentially on graphs. Rather, updates may be aggregated in batches, and graphs are updated by large batches of updates [14].

SIGMOD '22, June 12–17,2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517883

Applications. There are various real-world applications operating on graphs that undergo rapid changes [7, 8, 35, 36], such as communication networks, context-aware search in web graphs [32, 35], social network analysis in social networks [7, 36], route-planning in road networks [1, 13], management of resources in computer networks [8], and so on. We discuss a few examples below.

- In communication networks, links between network devices (e.g. routers) may become slow or broken due to congestion of information flow over a network or a deadly fault in a network device. Efficient maintenance of shortest paths to reflect the underlying changes helps vendors to activate new links and preserve the quality of their service [8].
- In social networks, Twitter is highly dynamic [28] – about 9% of all connections change in a month. Users having 100 followers on average were found to obtain 10% more new followers but lose about 3% of existing followers in a given month. Distance information is often used to recommend the relevant content or new connections [36, 39].

Although the batch-dynamic setting is increasingly important and desired for real-world applications, it poses significant challenges on algorithm design due to the combinatorial explosion of different interactions possibly occurring among updates. Very recently, several batch-dynamic algorithms have been reported, mostly focusing on traditional graph problems such as graph connectivity [2], dynamic trees [3] and k-clique counting [15]. As of yet, batch-dynamic algorithms for shortest-path distance have been left unexplored, despite the fact that computing the distance between an arbitrary pair of vertices (i.e., *distance queries*) is a fundamental problem in many real-world applications. Up to now, only several dynamic labelling algorithms for distance queries have been studied in the single-update setting, which handles one single update (e.g., edge insertion or edge deletion) at a time [5, 12, 21, 33]. Unlike previous studies, in this work, we are interested in exploring the following research questions:

– Is it possible to design batch-dynamic algorithms for distance queries, which can efficiently reflect batch updates on graphs?
– Can such batch-dynamic algorithms offer significant performance gains in comparison with state-of-the-art algorithms in the single-update setting?
– Can we parallelize such batch-dynamic algorithms to further boost performance in a parallel setting, whenever parallel computing resources are available?

Present work. The goal of this work is to answer the aforementioned research questions on complex networks (e.g., social networks and web graphs). It is known that complex networks exhibit different properties (e.g., small diameter) from road networks [4].

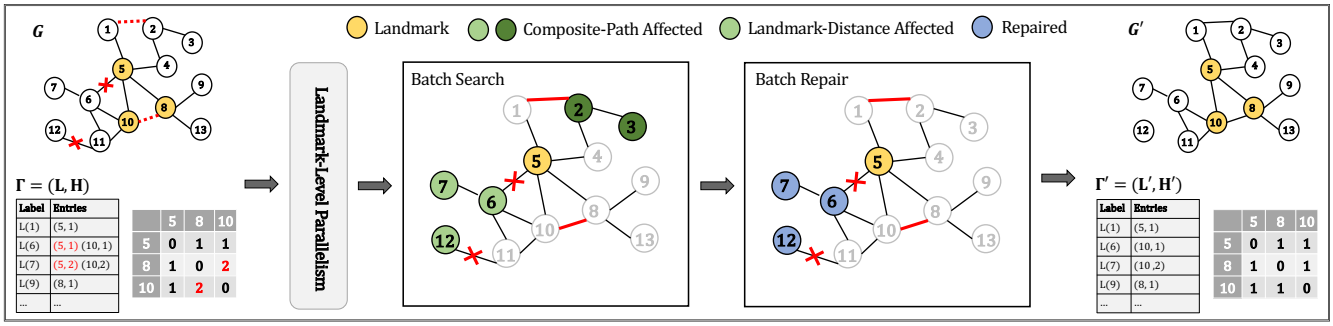


Figure 1: A high-level overview of our batch-dynamic method (BatchHL) which performs a batch update in two phases: 1) Batch Search: find vertices that are affected, and 2) Batch Repair: repair vertices returned by Batch Search.

Specifically, we thus aim to develop an efficient and scalable solution for answering distance queries on complex networks that undergo batch updates. This solution should have the following properties: (1) *Time efficiency*: it can quickly answer distance queries in a way that reflects batch updates on graphs, e.g., microseconds for large-scale graphs, since distances are used as a building block in many graph analysis tasks; (2) *Space efficiency*: it can use an efficient data structure for storing distance labellings, which ideally grows linearly or sublinearly with the number of vertices in a graph; (3) *Scalability*: it can scale to large graphs with millions of vertices and edges without compromising query and update performance.

To derive these properties, firstly, we choose to combine offline labelling and online searching so as to leverage the advantages from both sides - accelerating query processing through a *partial distance labelling* that is of limited size but provides a good approximation to bound online searches. Traditional labelling methods such as pruned landmark labelling (PLL) [4] can efficiently answer distance queries using a *full distance labelling*; however, their labelling size grows quadratically with the size of a graph and the computational cost of updating such labellings to reflect rapid changes is often unbearably high. Hence, we consider to use a partial distance labelling for providing an upper bound for online search (with theoretical guarantees for exact answers, which will be discussed in Section 4). This brings two significant computational benefits: (i) labelling construction can scale to very large graphs; (ii) labelling maintenance can be efficiently handled on dynamic graphs.

We propose a batch-dynamic method, BatchHL, to dynamize distance labellings efficiently in order to reflect large batches of updates on a graph. BatchHL consists of two phases: (1) *Batch search* finds vertices whose labels are affected by batch updates; (2) *Batch repair* changes the labels of affected vertices to ensure correctness and minimality of labelling. The main challenges are the following:

- *Unifying edge insertion and deletion:* We explore the core properties shared by edge insertion and edge deletion. Based on this, we unearth an elegant pattern that unifies these two fundamental kinds of graph updates.
- *Avoiding unnecessary and repeated computation:* We analyse how updates interact with each other, and based on that, design pruning rules to reduce search and repair spaces so as to leverage the computational efficiency of batch updates.
- *Exploiting the potential of parallelism:* We consider to parallelize batch search and batch repair in a simple but easy-to-implement way to speedup the performance.

To the best of our knowledge, this is the first study to develop a batch-dynamic solution for answering distance queries on large-scale graphs. Figure 1 presents the high-level overview of BatchHL which performs batch search and then batch repair. Figure 2 shows the gaps in the number of vertices affected by batch updates when different variants of our method are used in the batch-update setting, in comparison with the single-update setting. Notice that the number of affected vertices in the single-update setting (i.e., UHL) is much higher than the ones in the batch-update setting (i.e., BHL^s, BHL and BHL$^+$). This is because one vertex may be affected by multiple updates in a batch, which would unavoidably lead to repeated and unnecessary computations in the single-update setting.

Contributions. The main contributions of this work are as follows:

- We propose a batch-dynamic method which can handle batch updates efficiently and uniformly so as to reflect batch updates on graphs into a highway cover labelling. Previous studies [5, 12] reported that handling edge deletions on a graph has been recognized as being computational expensive and difficult, even in the single-update setting. Our method alleviates this challenge and can handle both edge insertions and edge deletions in batches efficiently.
- We develop efficient pruning strategies in our method, i.e., in both batch search and batch repair, to eliminate repeated and unnecessary computations on graphs. As a result, when dealing with batch updates, we traverse much smaller numbers of vertices than in the single-update setting where each update is handled independently. We also design an inference mechanism to compute new distances based on boundary vertices and incorporate this into batch repair in our method.
- We prove that the proposed method can preserve the minimality of labelling on batch-dynamic graphs. Notice that, as discussed in [12], minimality is a difficult but highly desirable property to have for designing a distance labelling over dynamic graphs. Otherwise, a distance labelling may have increasingly unnecessary entries left in its labels and query performance would deteriorate over time.
- Our proposed method can scale to very large dynamic graphs. This is due to several reasons: the design choices on combining offline labelling and online searching, the properties of highway cover labelling, the pruning strategies in batch search and batch repair, and landmark-based parallelism. We will discuss these in detail in Section 7.

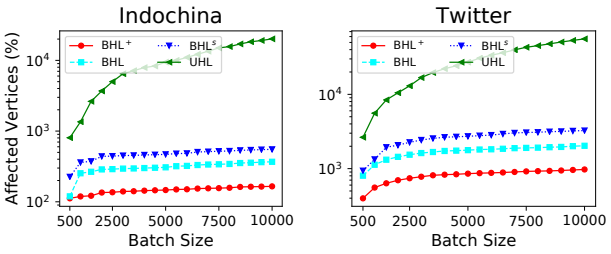


Figure 2: Number of vertices affected by batch updates of varying sizes. BHL and BHL$^+$ are our batch dynamic algorithms, BHL^s is a variant of BHL which splits edge insertions and deletions into sub-batches and performs them sequentially, and UHL handles updates in the single-update setting.

We have evaluated our method on 14 real-world networks to verify their efficiency and scalability. The results show that our method significantly improves both time and space efficiency compared to the state-of-the-art methods. It can maintain a very small labelling size, while still answering queries in the order of milliseconds, even on large-scale graphs with several billions of edges that undergo large batch updates. For example, the average distance query time for the UK dataset with 3.7 billions of edges is around 1 millisecond, and the average update time for each update is 14.45 milliseconds; similarly, the average distance query time for Twitter dataset is 0.86 millisecond and the average update time is 13.29 milliseconds, more than 300 times faster than the state-of-the-art method.

2 RELATED WORK

Answering distance queries has been an active research topic for many years. Traditionally, distance queries can be answered using Dijkstra's search on positively weighted graphs or a breadth-first search (BFS) on unweighted graphs [34] or a bidirectional scheme combining two such searches: one from the source vertex and the other from the destination vertex [31]. However, these algorithms may traverse an entire network when two query vertices are far apart from each other and become too slow for large networks. To accelerate response time in answering distance queries, labelling-based methods have emerged as an attractive way, which precompute a data structure, called *distance labelling* [1, 4, 6, 10, 11, 13, 17, 20, 22, 26, 37, 38]. For example, Akiba et al. [4] proposed the pruned landmark labelling (PLL) to pre-compute a 2-hop distance labelling [11] by performing a pruned breadth-first search from every vertex, and Li et al. [25] developed a parallel algorithm for constructing PLL which achieved the state-of-the-art results for answering distance queries on static graphs.

So far, several attempts have been made to study distance queries over dynamic graphs [5, 12, 16, 18, 21, 29, 33, 40] which only considered the unit-update setting i.e., to perform updates one at a time. Akiba et al. [5] studied the problem of updating PLL for incremental updates (i.e. edge additions). This work however does not remove outdated entries because the authors considered it too costly. Qin et al. [33] and D'angelo et al. [12] studied the problem of updating PLL for decremental updates (i.e. edge deletions). Note that, in the decremental case, outdated distance entries have to be removed; otherwise distance queries cannot be correctly answered. Their methods suffer from high time complexities and cannot scale to large graphs, e.g., the average update time of an edge deletion on a network with 19M edges is 135 seconds in [33] and on a network with 16M edges is 19 seconds in [12]. D'angelo et al. [12] combined the algorithm for incremental updates proposed in [5] with their method for decremental updates to form a fully dynamic algorithm, which can only be applied to networks with around 20M of edges. Hayashi et al. [21] proposed a fully dynamic method which combines a distance labelling with online search to answer distance queries. Their method pre-computes bit-parallel shortest-path trees (SPTs) rooted at each $r \in R$ for a small subset of vertices R and dynamically maintain the correctness of these bit-parallel SPTs for every edge insertion and deletion. Then, an online search is performed under an upper distance bound computed via the bit-parallel SPTs on a sparsified graph. Unlike these approaches, our present work considers the batch-update setting. Designing dynamic algorithms is usually quite complex and difficult, as reported by these approaches in the single-update setting, and arguably even more so in the batch-dynamic setting or parallel setting.

Another line of research studied streaming graph algorithms. In the streaming setting, a rapidly changing graph is often modeled using certain compressed data structures due to space constraints. Updates are received as a stream, but may be accumulated into batches through a sliding window and applied to the underlying graph. In this setting, a number of methods [19, 27, 30] have been proposed to address distance queries. However, these methods operate under certain constraints, e.g., limited amount of memory and accuracy of graph structure. Different from these streaming graph methods, our work considers applications which operate on batch-dynamic graphs that are explicitly stored and can be processed in the main memory of a single machine. Nevertheless, the ideas of our algorithm can be easily extended to deal with batch updates in the streaming setting.

3 PRELIMINARIES

Let $G = (V, E)$ be a graph where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. The *distance* between two vertices s and t in G, denoted as $d_G(s, t)$, is the length of a shortest path between s and t. If there does not exist any path between s and t, then $d_G(s, t) = \infty$. We use $P_G(s, t)$ to denote the set of all shortest paths between s and t in G, and $N(v)$ the set of neighbors of a vertex $v \in V$, i.e. $N(v) = \{v' \in V | (v, v') \in E\}$. Without loss of generality, we focus our discussion on unweighted, undirected graphs in this paper and discuss the extension to directed and non-negative weighted graphs in Section 6.

There are two fundamental types of updates on graphs: *edge insertion*, i.e., add an edge (a, b) into E, and *edge deletion*, i.e., delete an edge (a, b) from E. Note that node insertion or deletion can be treated as a batch update containing only edge insertions or only edge deletions, respectively. A *batch update* is a sequence of edge insertions and deletions. In the case that the same edge is being inserted and deleted within one batch update, we simply eliminate both of them. An update is *valid* if it makes a change on a graph, i.e., inserting an edge (a, b) into G when $(a, b) \notin E$, and deleting an edge (a, b) from G when $(a, b) \in E$. We ignore invalid updates.

Let $R \subseteq V$ be a subset of special vertices in G, called *landmarks*. A *label* $L(v)$ for each vertex $v \in V$ is a set of *distance entries* $\{(r_i, \delta_L(r_i, v))\}_{i=1}^{n}$ where $r_i \in R$, $\delta_L(r_i, v) = d_G(r_i, v)$ and $n \leq |R|$. We call $(r_i, \delta_L(r_i, v))$ the r_i*-label* of vertex v. The set of

labels for all vertices in V, i.e., $\{L(v)\}_{v\in V}$, form a *distance labelling* over G. The *size* of a distance labelling is defined as $\sum_{v\in V}|L(v)|$. In the literature, a distance labelling is often constructed following the 2-hop cover property [11] which requires at least one vertex $w \in L(u) \cap L(v)$ to be on a shortest path between u and v.

Definition 3.1 (2-hop cover labelling). *A distance labelling L over $G = (V, E)$ is a* 2-hop cover labeling *if for any $s, t \in V$,*

$$d_G(s,t) = \min\{\delta_L(r_i,s) + \delta_L(r_i,t) | (r_i,\delta_L(r_i,s)) \in L(s), (r_i,\delta_L(r_i,t)) \in L(t)\}. \quad (1)$$

In our work, we consider a labelling property based on the notion of highway, i.e., highway cover labelling [17].

Definition 3.2 (Highway). *A* highway $H = (R, \delta_H)$ *consists of a set R of landmarks and a distance decoding function $\delta_H : R\times R \to \mathbb{N}^+$ s.t. $\delta_H(r_1, r_2) = d_G(r_1, r_2)$ for any two landmarks $r_1, r_2 \in R$.*

Definition 3.3 (Highway cover labelling). *A* highway cover labelling $\Gamma = (H, L)$ *consists of a highway H and a distance labelling L satisfying that, for any $v \in V\backslash R$ and $r \in R$,*

$$d_G(r,v) = \min\{\delta_L(r_i,v) + \delta_H(r,r_i) | (r_i,\delta_L(r_i,v)) \in L(v)\} \quad (2)$$

Intuitively, a highway cover labelling requires that the label $L(v)$ of every vertex $v \in V$ must contain a distance entry to each landmark $r \in R$ unless there is another landmark on a shortest path between r and v. Unlike a 2-hop cover labelling that can answer distance queries for any two vertices in a graph, i.e., *a full distance labelling*, a highway cover labelling can only answer distance queries between any landmark and any vertex in a graph, i.e., *a partial distance labelling*.

Definition 3.4 (Minimality). *A highway cover labelling $\Gamma = (H, L)$ over G is* minimal *if, for any highway cover labelling $\Gamma' = (H, L')$ over G, $size(L') \geq size(L)$ holds.*

It has been shown in [17] that for any fixed set of landmarks, there exists a unique minimal highway cover labelling, which is contained in every highway cover labelling.

4 APPROACH OVERVIEW

In this section, we present how to answer distance queries for any two vertices in a batch-dynamic graph by combining highway cover labelling with online searching. The key idea is to dynamically maintain a highway cover labelling on a batch-dynamic graph, and then use such a highway cover labelling to bound online searches on a sparsified search space in order to accelerate query processing.

Given a highway cover labeling $\Gamma = (H, L)$, an upper bound on the distance between any pair of vertices $s, t \in V$ in a graph G is computed as follows:

$$d_{st}^{\top} = \min\{\delta_L(r_i,s) + \delta_H(r_i,r_j) + \delta_L(r_j,t) \mid (r_i,\delta_L(r_i,s)) \in L(s), (r_j,\delta_L(r_j,t)) \in L(t)\} \quad (3)$$

Here, $d_{st}^{\top}$ is the minimal length amongst all paths between s and t that pass through the highway. Since there may exist a shorter path not passing through the highway, we conduct 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 $d_{st}^{\top}$ to answer the distance query $Q(s, t)$ such that

$$Q(s,t) = \min(d_{G[V\backslash R]}(s,t),\ d_{st}^{\top})$$

In the implementation, $d_{G[V\backslash R]}(s, t)$ can be computed by conducting a bidirectional BFS search from both s and t [17] which terminates either after $d_{st}^{\top} - 1$ steps or when the searches from both directions meet.

The major challenge is: how to design an algorithm that can efficiently maintain a highway cover labelling for answering distance queries on graphs that undergo batch updates, particularly when graphs are very large?

Algorithm 1: BatchHL (BHL)

```
Function BatchHL(G', B, R, Γ)
    Γ' ← Γ
    foreach r ∈ R do
        V_AFF ← BatchSearch(G', B, r, Γ)
        BatchRepair(G', V_AFF, r, Γ, Γ')
    return Γ'
```

5 PROPOSED METHOD

In this section, we present our batch-dynamic method, BatchHL, which can efficiently maintain a minimal highway cover labelling for dynamic graphs. As described in Algorithm 1, BatchHL involves two phases: *Batch Search* and *Batch Repair*.

5.1 Batch Search

In the following let $G = (V, E)$ be a graph, $R \subseteq V$ a set of landmarks and B a batch update resulting in the updated graph $G' = (V', E')$. We denote the unique minimal highway labellings on G and G' by Γ and Γ', respectively. Our first aim is to identify vertices for which the set of shortest paths to a given landmark changes.

Definition 5.1 (affected). *A vertex $v \in V$ is* affected *by a batch update B w.r.t. a landmark $r \in R$ iff $P_G(r, v) \neq P_{G'}(r, v)$.*

We use $V_{\text{AFF}}(r, B) = \{v \in V | P_G(v, r) \neq P_{G'}(v, r)\}$ to denote the set of all affected vertices by a batch update B w.r.t. a landmark r. The following lemma states how affected vertices relate to a single update (either edge insertion or edge deletion).

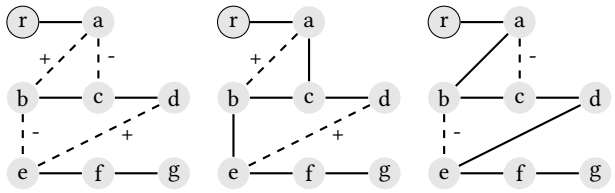
Lemma 5.2. *A vertex v is affected w.r.t. a landmark r iff there exists a shortest path between v and r in $G \cup G'$ that passes through an inserted edge (a, b) in G' or a deleted edge (a, b) in G.*

An edge insertion or deletion (a, b) can create or eliminate shortest paths starting from r and passing through (a, b). By this lemma, we know that any update on an edge (a, b) with $d_G(r, a) = d_G(r, b)$ is *trivial* w.r.t. a landmark r, since such an update does not affect any vertices w.r.t. the landmark r.

A naive way of finding affected vertices would be to apply Definition 5.1 directly, by computing the set of all shortest paths from a landmark to each vertex on G and G', respectively, and comparing them. However, the computational cost of this would be prohibitive, even for small graphs. Until now, the standard way of

Algorithm 2: Batch Search

```
1  Function BatchSearch(G', B, r, Γ)
2      foreach (a, b) ∈ B do
3          if d_G(r, a) < d_G(r, b) then
4              add (d_G(r, a) + 1, b) to Q
5          else if d_G(r, a) > d_G(r, b) then
6              add (d_G(r, b) + 1, a) to Q
7      while Q is not empty do
8          remove minimal (d, v) from Q
9          if v ∉ V_AFF+ then
10             add v to V_AFF+
11             foreach w ∈ N_G'(v) do
12                 if d + 1 ≤ d_G(r, w) then
13                     add (d + 1, w) to Q
14     return V_AFF+
```



	$v =$	a	b	c	d	e	f	g
	$d_G(r,v) =$	1	3	2	3	4	5	6
Anchor b	$d_{G'}(b,v) =$	1	0	1	2	3	4	5
	Eq. 4 =	*False*	*True*	*False*	*False*	*False*	*False*	*False*
Anchor c	$d_{G'}(c,v) =$	2	1	0	1	2	3	4
	Eq. 4 =	*False*	*True*	*True*	*True*	*True*	*True*	*True*
Anchor e	$d_{G'}(e,v) =$	4	3	2	1	0	1	2
	Eq. 4 =	*False*	*False*	*False*	*False*	*True*	*True*	*True*

Figure 3: Example graphs, where edges marked by + are inserted and edges marked by − are deleted.

handling graph changes is to treat edge insertion and edge deletion separately, since they have opposite effects on a graph. A natural extension on batch updates would then be to devise an incremental algorithm for batch edge insertions and a decremental algorithm for batch edge deletions. However, for a batch update that contains both edge insertions and edge deletions, we would then need to split it into two sub-batches - one for edge insertions and the other for edge deletions, and apply incremental and decremental algorithms, respectively. Thus, repeated computations across edge insertions and deletions cannot be eliminated because no interaction between edge insertion and deletion can be captured.

Example 5.3. *Consider Figure 3.a with four updates. If handling edge insertions and deletions separately in two sub-batches as shown in Figure 3.b-3.c, insertions of (a,b) and (d,e) lead to affected vertices $\{b,e,f,g\}$, while deletions of (a,c) and (b,e) lead to affected vertices $\{c,d,e,f,g\}$. The traversal on edges (e,f) and (f,g) is repeated.*

To overcome the aforementioned shortcomings, we propose an efficient algorithm that unifies edge insertions and deletions. The key idea is based on our observation of a "*shared pattern*" that characterises affected vertices w.r.t. a landmark in a unified way for both edge insertions and edge deletions.

Let $r \in R$ and $(a,b) \in B$. Here, (a,b) is any update, i.e., either inserted or deleted edge. The *anchor* of (a,b) is either a or b, whichever is further away from r, and the *pre-anchor* of (a,b) is a vertex in $\{a,b\}$ that is not the anchor. The *anchor distance* of (a,b) is defined as $d_G(r,u')+1$ where u' is the pre-anchor of (a,b). Note that when $d_G(r,a) = d_G(r,b)$, there is no anchor nor pre-anchor corresponding to the update (a,b). For each B, there exists a set of anchors corresponding to updates in B. An affected vertex v in G w.r.t. r by a batch update B can be found if the following condition is satisfied by at least one anchor u from B:

$$d_G(r,v) \geq (d_G(r,u')+1) + d_{G'}(u,v). \tag{4}$$

Example 5.4. *Consider Figure 3.a again, which has three anchors b, c and e corresponding to the four updates. By applying Eq. 4, we can identify affected vertices $\{b,c,d,e,f,g\}$ as shown in the table.*

This striking pattern enables us to design a simple yet efficient algorithm for finding affected vertices which only needs to traverse local neighbors v of each anchor u on G' recursively, i.e., computing $d_{G'}(u,v)$, regardless whether updates are edge insertions or deletions. The anchor distance $d_G(r,u')+1$ and the distance $d_G(r,v)$ on G can be efficiently computed from the highway cover labelling Γ. The searches by different updates can be combined into a single search in the order of the anchor distances plus their distances to the anchors to avoid unnecessary computation.

We note that due to this unified handling of insertions and deletions, optimization that apply to only one of these operations cannot simply be applied to the combined algorithm. However, we show how one such optimization can still be leveraged in Section 5.2.

Armed with these ideas, Algorithm 2 eliminates unnecessary searches on unaffected vertices v with $d_G(r,v) < d_G(r,u')+1$ and also avoids traversing vertices affected by multiple updates more than once. However, Algorithm 2 does not precisely compute the set of all affected vertices, but a superset of it. The following example illustrates why this happens, and why it is difficult to avoid.

Example 5.5. *Consider the graph in Figure 4.a. The dotted edge between r and u indicates a long path between them, and the dotted edge between r and v indicates an even longer path. When both edge deletion (r,u) and edge insertion (u,v) occur, the distance between r and u in G is used to compute the anchor distance of v for the update (u,v), ignoring that the distance between r and u has changed. It is difficult to identify whether v is affected – it hinges on whether the long path between r and v is longer than the long path between r and u plus 1, which cannot be ascertained by Γ.*

We now characterize the set of vertices returned by Algorithm 2.

Definition 5.6 (composite path). *A path from r to v in $G \cup G'$ is a* composite path *iff it consists of two parts: a part that lies in G followed by a part in G'.*

A composite path is *significant* iff it passes through at least one deleted and at least one inserted edge. In Figure 3.a, $r-a-b-c$ and $r-a-c-d$ are insignificant composite paths, $r-a-c-d-e$

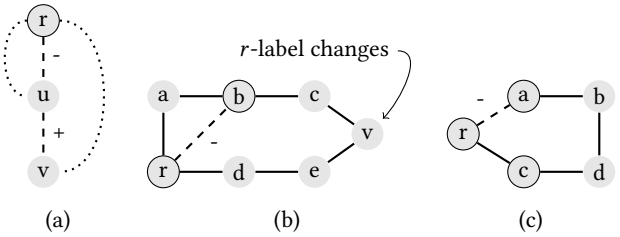


Figure 4: Example graphs for illustrating batch search.

is a significant composite path, and $r - a - b - e$ is not a composite path as a deleted edge comes after an inserted edge.

Definition 5.7 (composite-path affected). *A vertex $v \in V$ is* composite-path-affected (CP-affected) *by a batch update B w.r.t. a landmark $r \in R$ iff*

(i) v is affected w.r.t. r, or
(ii) there exists a significant composite path from r to v of length $d_G(r, v)$ or less.

We will show that Algorithm 2 returns the set of all composite-path-affected vertices. Clearly this includes all affected vertices. Additional vertices due to condition (ii) are undesirable but hard to avoid, as illustrated in Example 5.5. From an algorithmic perspective, it happens because our starting distance is calculated w.r.t. G, so we are effectively considering paths for which the first part (from r to an anchor) lies in G, and the rest in G'.

Lemma 5.8. *Algorithm 2 returns the set of all CP-affected vertices.*

Proof. We show that a vertex is CP-affected iff it lies in $V_{\text{AFF+}}$ returned by Algorithm 2. We prove the "if" and "only if" below.

(if) Let $v \in V_{\text{AFF+}}$. Then there must exist a composite path p from r to v of length at most $d_G(r, v)$ that passes through at least one edge in B. If p lies in G then it lies in $P_G(r, v)$ but not in $P_{G'}(r, v)$, so v is affected. If p lies in G', then either it lies in $P_{G'}(r, v)$ or there exists an strictly shorter path p' in $P_{G'}(r, v)$. Neither p nor p' lies in $P_G(r, v)$, so v is affected. If p lies neither in G nor in G' then it must be significant. Thus v is CP-affected. in all cases.

(only if) Reversely, let v be CP-affected. If $P_G(r, v) \nsubseteq P_{G'}(r, v)$ then there exists a path p in G of length $d_G(r, v)$ that passes through a deleted edge. If $P_G(r, v) \subsetneq P_{G'}(r, v)$ then there exists a path p in G' of length at most $d_G(r, v)$ that passes through an inserted edge. Otherwise the exists a significant composite path of length at most $d_G(r, v)$. Thus, in all cases, there exists a composite path p of length at most $d_G(r, v)$ that passes through an edge in B.

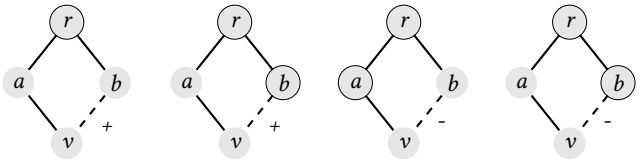
Let (a, b) be either the last deleted edge that p passes through, or the first inserted edge, with $d_G(r, a) < d_G(r, b)$. Then p can be split into p_{ra} from r to a, (a, b) and p_{bv} from b to v such that p_{ra} lies in G and p_{bv} in G'. The search in Algorithm 2 starting at b will use $|p_{rb}| = d_G(r, a) + 1$ as the anchor distance for b, and proceed along p_{bv}. Thus for every vertex $w \in p_{bv}$, including v, it will obtain $|p_{rw}| \leq d_G(r, w)$ and add w to $V_{\text{AFF+}}$. □

5.2 Improved Batch Search

So far we aimed at computing affected vertices. However, changes to shortest paths between r and v do not always cause a change in distance. Thus we shall differentiate between new and eliminated paths, and strengthen the pruning condition $d + 1 \leq d_G(r, w)$ in Line 12 of Algorithm 2 to $d + 1 < d_G(r, w)$ for new paths.

Things get a little trickier though, as we may need to eliminate redundant labels, or restore previously eliminated labels when they become non-redundant. Thus even if the distance between r and v does not change, the highway labeling may need to be updated.

Example 5.9. *Consider the following graphs and updates, where the landmarks are circled. In all cases, vertex v is affected, but the distance between r and v does not change. For case (a) adding the edge (b, v) does not cause a label change for v. It does however for case (b) where b is a landmark, causing the r-label of v to be deleted. Deletion of (b, v) does not cause a change on the label of v in case (c), but causes a change in case (d) where an r-label needs to be inserted.*



(a) no change *(b) change* *(c) no change* *(d) change*

A core difficulty in identifying whether affected vertices have changes on their labels is that label changes can happen far away from updates, and computing the changed labels of such vertices may require the consideration of vertices whose labels do not change, as illustrated by the example below.

Example 5.10. *Consider the graph in Figure 4.b, where r and b are landmarks and the edge (r, b) is deleted. The distance between r and c changes, but the label of c does not change. That is because the shortest path between r and c goes through landmark b without change. At the same time the label of v does change, as the edge (r, b) eliminates a shortest path between r and v that passes through landmark b, similar to case (d) in Example 5.9. Although the label of c does not change, the changed distance between r and c is needed for computing the changed label of v. Therefore, c needs to be captured as well.*

We thus need to reexamine exactly which vertices need to be returned. Firstly, this includes any vertex v for which the highway labeling must be updated. For non-landmarks the only possible change is to their r-label. For landmarks their distance to r is stored as part of the highway, and needs to be updated when it changes. Secondly, we must return any vertex for which the distance to r changes. That is because the batch repair algorithm computes the updated distance of a vertex to r from that of its neighbors, so using outdated values for a neighbor could lead to errors.

Example 5.11. *Consider the graph in Figure 4.c, where r, a and c are landmarks and the edge (r, a) is getting deleted. The only node for which the highway labeling needs to be updated is a. For b the distance to r changes, but its r-label is still redundant. Using the old distance between r and b would cause our batch repair algorithm to compute $d_{G'}(r, a)$ as $d_G(r, b) + 1 = 3$.*

By considering vertices for which either label or distance changes, we can address both of the issues illustrated in Examples 5.10 and 5.11. This motivates the following definition.

Definition 5.12 (landmark-distance affected). *A vertex v is* landmark-distance-affected (LD-affected) *by a batch update B w.r.t. a landmark $r \in R$ iff it is*

(i) label-affected: *the r-label of v changes, or*
(ii) distance-affected: *the distance between r and v changes.*

As seen in Example 5.9, changes to r-label without changes to distance happen whenever a new shortest path passing through another landmark is created where none existed previously, or when the last such path is deleted. To identify such cases, we track whether a shortest path to r passes through another landmark.

Definition 5.13 (landmark length). *The* landmark length *of a path p starting from $r \in R$ is a tuple $(d, l) \in \mathbb{N} \times \mathbb{B}$ where*

- *d is the length of p (number of edges), and*
- *l is the* landmark flag, *with l = True iff p passes through a landmark other than r.*

We denoted this landmark length as $|p|_L$. The landmark distance *between r and v in G is the minimal landmark length of paths between them, denoted as*

$$d_G^L(r, v) := \min \{\ |p|_L \mid \ p \text{ is a path between } r \text{ and } v \text{ in } G\ \}$$

The ordering used to compare landmark length tuples is the lexicographical one, with True < False. *The latter ensures that the landmark flag of $d_G^L(r, v)$ is set iff* any *of the shortest paths between r and v passes through another landmark.*

Lemma 5.14. *Let $d_{G'}^L(r, v) = (d, l)$. If $d = \infty$ or l = True then v has no r-label in Γ'. Otherwise v has the r-label (r, d).*

Proof. If v has any r-label in Γ' it must be (r, d). As Γ' is minimal, this r-label exists iff it is not redundant. For $d = \infty$ redundancy of (∞, r) is obvious. Otherwise (d, r) is redundant iff the correct distance could also be computed using the highway. This happens iff a shortest path between r and v passes through another landmark, which is indicated by the landmark flag. □

Lemma 5.15. *A vertex v is LD-affected iff $d_G^L(r, v) \neq d_{G'}^L(r, v)$.*

Proof. Let l_G and $l_{G'}$ denote the landmark flags of $d_G^L(r, v)$ and $d_{G'}^L(r, v)$, respectively. Condition (ii) of Definition 5.12 states $d_G(r, v) \neq d_{G'}(r, v)$. It suffices to show that for $d_G(r, v) = d_{G'}(r, v)$ condition (i) holds iff $l_G \neq l_{G'}$. This is trivial for $d_G(r, v) = d_{G'}(r, v) = \infty$. For finite distances it follow from Lemma 5.14. □

Like Algorithm 2, our improved batch search algorithm computes a superset of the set of all LD-affected vertices, albeit a smaller one. By Lemma 5.15 we need to return a vertex whenever its landmark distance changes. Thus we improve upon Algorithm 2 by tweaking the pruning conditions:

- Insertion: To affect the landmark distance, the landmark length of a new path p_{new} from r to v must be strictly smaller than the current landmark distance between r and v. Thus we check $|p_{\text{new}}|_L < d_G^L(r, v)$.
- Deletion: A deleted path p_{del} can only affect landmark distance if its landmark length was minimal, i.e., equal to the old landmark distance. This suggests checking $|p_{\text{del}}|_L = d_G^L(r, v)$. However, deleted paths may be obscured by shorter composite paths, so we check $|p_{\text{del}}|_L \leq d_G^L(r, v)$ instead.

The effects of these optimizations can be observed in Example 5.9, where v will not be returned for case (a) and case (c).

To apply the new pruning conditions, we must know the landmark length of a path we are following, and whether or not it passes through a deleted edge. Thus we track not only the length of each path, but also a landmark flag and a deletion flag.

Definition 5.16 (extended landmark length). *The* extended landmark length *of a path p starting from $r \in R$ is a tuple $(d, l, e) \in \mathbb{N} \times \mathbb{B} \times \mathbb{B}$ where*

- *(d, l) is the landmark length of p, and*
- *e is the* deletion flag, *with e = True iff p passes through a deleted edge.*

We use lexicographical order for comparison, with True < False.

For ease of extending landmark length values we will flatten tuples implicitly, i.e., we treat $((d, l), e)$ as (d, l, e). The choice of the ordering True < False for the deletion flag is not arbitrary. When multiple search paths merge, we only track the length of the shorter one w.r.t. extended landmark length. To ensure that deleted paths will not be pruned using the stricter condition for insertion, we need to keep the deletion flag if *any* path has it, which is achieved by ordering True < False.

We apply our pruning conditions by comparing the extended landmark lengths computed for paths ending in v to the landmark distance of v in G. For this we identify the minimal extended landmark length that indicates LD-affectedness.

Lemma 5.17. *Let v be LD-affected w.r.t. r, and β defined as*

$$\beta(r, v) := \left(d_G^L(r, v),\ \text{True}\right)$$

Any composite path of minimal extended landmark length equals to $\beta(r, v)$ or less and pass through an updated edge.

Proof. In the following we shall always refer to composite paths from r to v. By Lemma 5.15 we have $d_G^L(r, v) \neq d_{G'}^L(r, v)$.

(1) If $d_G^L(r, v) < d_{G'}^L(r, v)$ then all paths of minimal landmark length must pass through a deleted edge. That makes their extended landmark length $\beta(r, v)$ or less.

(2) If $d_G^L(r, v) > d_{G'}^L(r, v)$ then all paths of minimal landmark length must pass through an inserted edge. Their landmark length is at most $d_{G'}^L(r, v)$, so their extended landmark length is strictly less than $\beta(r, v)$. □

Batch search with improved pruning is described in Algorithm 3. As we frequently need to update the landmark length of a path when appending another vertex, we define an operator for this:

$$(d, l) \oplus w := \begin{cases} (d+1, \text{True}) & \text{if } w \text{ is a landmark} \\ (d+1, l) & \text{otherwise} \end{cases}$$

We finally show the correctness of Algorithm 3, i.e., that all LD-affected vertices are included in its result set. Note that some additional vertices may be returned as well.

Lemma 5.18. *Algorithm 3 returns all LD-affected vertices.*

Proof sketch. Let v be LD-affected, and P_{min} be the set of all composite paths from r to v of minimal landmark length. By

Algorithm 3: Improved Batch Search

```
1  Function BatchSearch(G', B, r, Γ)
2      foreach (a, b) ∈ B do
3          e ← (a, b) is deleted
4          if d_G(r, a) < d_G(r, b) then
5              add (d^L_G(r, a) ⊕ b, e, b) to Q
6          else if d_G(r, a) > d_G(r, b) then
7              add (d^L_G(r, b) ⊕ a, e, a) to Q
8      while Q is not empty do
9          remove minimal (d, l, e, v) from Q
10         if v ∉ V_AFF+ then
11             add v to V_AFF+
12             foreach w ∈ N_G'(v) do
13                 d_w ← ((d, l) ⊕ w, e)
14                 if d_w ≤ β(r, w) then
15                     add (d_w, w) to Q
16     return V_AFF+
```

Algorithm 4: Batch Repair

```
1  Function BatchRepair(G', V_AFF, r_i, Γ, Γ')
2      foreach v ∈ V_AFF do
3          D_BOU[v] ← d^L_BOU(v, V_AFF) // use Γ to compute
4      while V_AFF is not empty do
5          V_min ← {v ∈ V_AFF | D_BOU[v].d is minimal}
6          remove V_min from V_AFF
7          foreach v ∈ V_min do
8              if D_BOU[v].d = ∞ ∨ D_BOU[v].l then
9                  remove r-label from Γ'(v)
10             else
11                 set r-label of Γ'(v) to (r_i, D_BOU[v].d)
12             if v is a landmark then
13                 δ'_H(r_i, v) ← D_BOU[v].d
14             foreach w ∈ N_G'(v) ∩ V_AFF do
15                 D_BOU[w] ← min(D_BOU[w], D_BOU[v] ⊕ w)
```

Lemma 5.17 these (and all their prefixes) meet the pruning condition in line 14 and pass through an updated edge. Thus the search in Algorithm 3 will follow them, starting from the last deleted or first inserted edge. While some paths may be pruned in line 10, the search will still follow at least one path $p \in P_{\min}$ with minimal landmark length. While its extended landmark length may not be minimal, this only causes p to be pruned if its landmark length equals $d^L_G(r, v)$ and p does not pass through a deleted edge. But in this case, v is not LD-affected. □

5.3 Batch Repair

In the following, we develop an efficient algorithm to repair labels. At its core is an inference mechanism for the distances of affected vertices, which allows us to update their labels. Here we start with *boundary vertices* that lie on the boundary of affected and unaffected vertices, and for which the distance to r can be computed from neighboring vertices whose distance did not change. Importantly, even though a vertex may be affected by multiple edge updates in a batch, its r-label only needs to be updated once.

Let $v \in V_{\text{AFF+}}$. For every neighbour w of v in G', $d_{G'}(r, v)$ must be upper-bounded by $d_{G'}(r, w) + 1$. If such a neighbour lies outside of $V_{\text{AFF+}}$, the value of $d_{G'}(r, w) = d_G(r, w)$ can easily be obtained. By taking the minimum of such known upper bounds, we get a readily available distance bound for v. As we wish to eliminate redundant r-labels, we track landmark distance.

Definition 5.19 (Landmark distance bound). *Let $S \subset V \setminus \{r\}$ be a set of vertices. The* landmark distance bound *of v w.r.t. S is:*

$$d^L_{BOU}(v, S) := \min\{d^L_{G'}(r, w) \oplus v \mid w \in N_{G'}(v) \setminus S\};$$

and the distance bound *of v w.r.t. S is:*

$$d_{BOU}(v, S) := \min\{d_{G'}(r, w) + 1 \mid w \in N_{G'}(v) \setminus S\}.$$

Note that the distance bound of a vertex is simply the distance component of its landmark distance bound. The following lemma allows us to compute the (landmark) distance of vertices in $V_{\text{AFF+}}$ from r in G' using their (landmark) distance bounds.

Lemma 5.20. *Let $S \subset V \setminus \{r\}$ and $v \in S$ with minimal distance bound. Then $d^L_{G'}(r, v) = d^L_{BOU}(v, S)$.*

Proof. For $d_{G'}(r, v) = \infty$ this is trivial. Otherwise let p be a shortest path from r to v in G' w.r.t. landmark length, v' the first vertex in p that lies in S, and w its predecessor in p. Since $w \notin S$ we have $d^L_{\text{BOU}}(v', S) \le d^L_{G'}(r, w) \oplus v = d^L_{G'}(r, v')$. If $v' \neq v$ then $d_{G'}(r, v') < d_{G'}(r, v) \le d_{\text{BOU}}(v, S)$, and therefore $d_{\text{BOU}}(v', S) < d_{\text{BOU}}(v, S)$. This contradicts the minimality of $d_{\text{BOU}}(v, S)$, so $v' = v$. It follows that $d^L_{\text{BOU}}(v, S) = d^L_{G'}(r, v)$. □

Note that $d^L_{G'}(r, v) = d^L_{\text{BOU}}(r, S)$ does not generally hold for every boundary vertex v. This can e.g. be seen in the graph of Example 5.22: when computing distances to r_1, e must be repaired before f, as the new shortest path between r_1 and f passes through e. The situation is reversed when computing distance to r_2. Algorithm 4 shows the pseudo-code of our batch repair algorithm. Given a graph G' and a set of all affected vertices V_{AFF}, we first compute the landmark distance bounds of vertices in V_{AFF} using their unaffected neighbors. We then find vertices in V_{AFF} with minimal distance bounds and remove them from V_{AFF}. By Lemma 5.20 their landmark distance to r in G' equals their landmark distance bounds. We use these landmark distances to update their r-labels, as well as their highway distances in the case of landmarks. Finally we update the landmark distance bounds of neighboring vertices in V_{AFF}. We continue this process until V_{AFF} is empty.

5.4 Analysis of BatchHL

In the following we will show correctness of Algorithm 1 and analyse its time complexity.

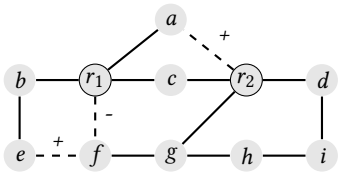
Theorem 5.21. *The highway labeling Γ' returned by Algorithm 1 is the minimal highway labeling for G'.*

Proof. By Lemmas 5.8 and 5.18, the vertex set V_{AFF} returned by BatchSearch contains all LD-affected vertices, regardless of which Algorithm (2 or 3) is used. By Lemma 5.15 this means that for vertices outside of V_{AFF} the landmark distance to r_i does not change, so that in line 3 of Algorithm 1 the value of $d^L_{\text{BOU}}(v, V_{\text{AFF}})$ can be computed from Γ. From Lemma 5.20 it follows that $D_{\text{BOU}}[v] = d^L_{G'}(r_i, v)$ whenever vertex v lies in $V_{\min}$.

For each landmark r and each vertex LD-affected w.r.t. r we update the r-label of v in Γ' based on its landmark distance to r in G'. By Lemma 5.14 these updates are correct. As the r-labels of vertices outside of V_{AFF} do not change, and we initialized Γ' using Γ, this leave all vertices with correct r-labels, for all $r \in R$, so the distance labeling of Γ' is correct and minimal. Highway is updated for vertices in V_{AFF} as well, for all $r \in R$, and do not change for others by Definition 5.12. □

The following example illustrates the individual steps that our BatchHL algorithm runs through.

Example 5.22. *Consider the following graph and updates:*



The initial highway labeling $\Gamma = (H, L)$ *will look like this:*

$H = \{\delta_H(r_1, r_2) = 2\}$,

	a	b	c	d	e	f	g	h	i
$L =$	$(r_1, 1)$	$(r_1, 1)$	$(r_1, 1)$		$(r_1, 2)$	$(r_1, 1)$	$(r_1, 2)$	$(r_1, 3)$	
			$(r_2, 1)$	$(r_2, 1)$		$(r_2, 2)$	$(r_2, 1)$	$(r_2, 2)$	$(r_2, 2)$

BatchHL will initialize Γ' *as* Γ*, and then run BatchSearch and BatchRepair for both* r_1 *and* r_2*.*

For r_1 *the basic BatchSearch described as Algorithm 2 returns*

$$V_{AFF+} = \{r_2, d, e, f, g, h, i\}$$

Here vertex e *is not actually affected, but still returned due to the composite path* $r_1 - f - e$*. Algorithm 3 returns only*

$$V_{AFF+} = \{e, f, g, h\}$$

For r_2, d *and* i*, the new paths through* a *have the same landmark length as existing ones and are thus pruned. The eliminated path* $r_1 - f - g - h - i$ *has strictly greater landmark length than the existing path through* r_2*, and thus is pruned. Note that* e *is still returned due to the composite path* $r_1 - f - e$*, despite not being LD-affected.*

One of these sets is then used as input for BatchRepair, say $V_{AFF} = \{e, f, g, h\}$*. The initial landmark bounds for this set are*

$$d^L_{BOU}(r_1, \ldots) = \begin{array}{c|c|c|c} e & f & g & h \\ \hline (2, \text{False}) & (\infty, \text{False}) & (3, \text{True}) & (5, \text{True}) \end{array}$$

Here e *has the minimal distance bound, so we update* $L(e)$ *by setting its* r_1*-label to* 2 *(which does not actually change* $L'(e)$*). Afterwards* e *is removed from* V_{AFF} *and the landmark bound of* f *is updated to* $(3, \text{False})$*. In the next iteration* f *and* g *are minimal, so the* r_1*-label in* $L(f)$ *is updated to* $(r_1, 3)$ *and the* r_1*-label in* $L'(g)$ *is removed. Finally* $d^L_{BOU}(r_1, h)$ *is updated to* $(4, \text{True})$ *and the* r_1*-label in* $L'(h)$ *is removed. This leaves* L' *as*

	a	b	c	d	e	f	g	h	i
$L' =$	$(r_1, 1)$	$(r_1, 1)$	$(r_1, 1)$		$(r_1, 2)$	$(r_1, 3)$			
			$(r_2, 1)$	$(r_2, 1)$		$(r_2, 2)$	$(r_2, 1)$	$(r_2, 2)$	$(r_2, 2)$

Running BatchSearch for r_2 *gives us one of*

$$V_{AFF} = \{r_1, a, b, e\} \text{ or } V_{AFF} = \{a, e\}$$

depending on which algorithm (Algorithms 2 or 3) is used. Running BatchRepair on either of those inserts $(r_2, 1)$ *into* $L'(a)$ *and* $(r_2, 2)$ *into* $L'(e)$ *for the final updated highway labeling*

	a	b	c	d	e	f	g	h	i
$L' =$	$(r_1, 1)$	$(r_1, 1)$	$(r_1, 1)$		$(r_1, 2)$	$(r_1, 3)$			
	$(r_2, 1)$		$(r_2, 1)$	$(r_2, 1)$	$(r_2, 3)$	$(r_2, 2)$	$(r_2, 1)$	$(r_2, 2)$	$(r_2, 2)$

Complexity analysis. Table 1 compares the time and space complexity of the state-of-the-art methods and our proposed method BatchHL for constructing and updating a distance labelling, and querying a distance. Let a be the total number of affected vertices, l be the maximum label size and d be the maximum degree. In our method, a refers to CP-affected vertices in the batch-update setting which is different from FulFD [21] and FulPLL [12] in the single-update setting. We perform $|R|$ BFSs to construct highway labelling in $O(|R| \cdot |V|)$ time and space. Then, we update highway labelling in Algorithm 1 where Algorithm 2 visits $O(a)$ vertices and for each affected vertex performs d queries to check its affected neighbors in $O(d \cdot l)$ time. Thus, the time complexity of Algorithms 2 and 3 is $O(a \cdot d \cdot l)$. Note that Algorithm 3 further reduces the total number of CP-affected vertices and is naturally faster than Algorithm 2. In practice, l and d are closer to the average values, and a is usually orders of magnitudes smaller than the total number of vertices in a graph. Next, Algorithm 4 repairs CP-affected vertices returned by Algorithm 2 which in the worse case could repair the labels of all CP-affected vertices. To decide whether the label of an affected vertex needs to be repaired, we check its neighbors in $O(d)$. Thus, the time complexity of Algorithm 4 is $(a \cdot d)$, and the overall time complexity of Algorithm 1 is $O(|R| \cdot a \cdot d \cdot l)$ using $O(V)$ space. We omit l from the time complexity of Algorithm 4 because we store distances for all unaffected neighbors of affected vertices in Algorithms 2 and 3.

FulFD constructs a bit-parallel shortest-path tree for each $r \in R$ and 64 of its neighbors N and requires $O(R \cdot N)$ time and space for every vertex $v \in V$ which is huge for a large network because V could be very large. Similarly, the construction time and space of FulPLL (PLL) and PSL* [25] (Parallel PLL using t cores) is prohibitive and this is the reason why we do not have results on large datasets for these methods in our experiments. For state-of-the-art dynamic methods FulFD and FulPLL, a is the sum over all affected vertices by each update in a batch and is very large in practice because of unnecessary computations due to which they take longer to update a distance labelling.

6 VARIANTS

Parallel batch updates. BatchHL can be parallelized at the landmark level. Let $\Gamma = (H, L)$ be the unique minimal highway cover labelling over G. Then the unique minimal highway cover labelling

Table 1: Complexity analysis and comparison.

Method	Construction		Update		Query	
	Time	Space	Time	Space	Time	Space
BatchHL	$O(R{\cdot}V)$	$O(R{\cdot}V)$	$O(R{\cdot}a{\cdot}d{\cdot}l)$	$O(V)$	$O(E)$	$O(V)$
FulFD	$O(R{\cdot}N{\cdot}V)$	$O(R{\cdot}N{\cdot}V)$	$O(R{\cdot}N{\cdot}a{\cdot}d{\cdot}l)$	$O(V)$	$O(E)$	$O(V)$
FulPLL	$O(l^2{\cdot}E)$	$O(V{\cdot}l)$	$O(a{\cdot}(E{+}V{\cdot}l))$	$O(V)$	$O(l)$	-
PSL*	$O(l^2{\cdot}E/t)$	$O(V{\cdot}l)$	-	-	$O(d{\cdot}l)$	-
BiBFS	-	-	-	-	$O(E)$	$O(V)$

$\Gamma' = (H', L')$ over G' may differ from Γ in: (1) *highway:* H is changed to H'; and (2) *labels:* L is changed to L'.

To enable the parallelism on highway, we store H using a *highway matrix* such that $h_{ij} = h_{ji}$ for each pair of landmarks (r_i, r_j). Then, searches can be conducted in parallel to update the entries in this highway matrix. In the second case, for any vertex v, distance entries in $L(v)$ w.r.t. different landmarks are disjoint subsets. Thus updating distance entries in $L(v)$ w.r.t. different landmarks can be processed in parallel. Putting it all together, for any batch update, we run batch search and batch repair w.r.t. each landmark in parallel to speed up the performance.

Directed and weighted graphs. Our methods can be extended to directed and non-negative weighted graphs. For directed graphs, we use $d_G(s, t)$ to refer to the distance from vertex s to vertex t and store two sets of labels for each vertex v, forward $L_f(v)$ and backward $L_b(v)$ labels, containing pairs $(r_i, \delta_{r_i v})$ after performing forward and backward pruned BFSs w.r.t. every $r_i \in R$. Accordingly, we store forward $H_f = (R, \delta_{H_f})$ and 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 $\delta_{H_b}(r_i, r_j) = d_G(r_i, r_j)$. To repair the affected labels and highways affected by a batch update, we perform our batch search and batch repair methods twice: once in the forward direction and once in the backward direction. Then the upper bound for a distance query (s, t) can be computed using $L_f(s)$, $L_b(t)$, δ_{H_f} and δ_{H_b} in the same way as described in Equation 3. For weighted graphs, we can use pruned Dijkstra's algorithm in place of pruned BFSs. We consider updates in the form of edge weight increase or decrease instead of edge insertion or deletion. Our methods can then handle weight increases in a similar way to edge deletions, and weight decreases in a similar way to edge insertions.

7 EXPERIMENTS

We have implemented our algorithm to experimentally verify its efficiency and scalability on real-world large networks.

7.1 Experimental Setup

In our experiments, all algorithms are implemented in C++11 and compiled with g++ 5.5.0 using the -O3 option. All the experiments are performed on a Linux server Intel Xeon W-2175 (2.50GHz CPU) with 28 cores and 512GB of main memory.

Baseline methods. We consider the following variants of our batch dynamic algorithm, (1) BHL: which uses the batch search described in Algorithm 2 and the batch repair described in Algorithm 4, (2) BHL$^+$: which uses the improved batch search described in Algorithm 3 and the batch repair described in Algorithm 4, and (3) BHL^p: which is a parallel variant of BHL$^+$. We compare these variants with the state-of-the-art methods as follows:

- FulFD [21]: A fully dynamic method that incorporates two algorithms IncFD and DecFD to update distance labelling for edge insertions and deletions, and then combines it with a graph traversal algorithm to answer distance queries.
- FulPLL [12]: A fully dynamic 2-hop cover labelling method which is composed of two separate dynamic algorithms. The first algorithm was proposed in [5] to answer distance queries on graphs undergoing edge insertions and the second algorithm was proposed in [12] to answer distance queries on graphs undergoing edge deletions. This method is based on the pruned landmark labelling (PLL) [4].
- PSL* [25]: A parallel algorithm which constructs pruned landmark labelling for static graphs to answer distance queries.
- BiBFS [21]: An online bidirectional BFS algorithm which answers distance queries using an optimized strategy to expand searches from the direction with fewer vertices.

Table 2: Summary of datasets.

Dataset	Type	$\|V\|$	$\|E\|$	avg. deg	max. deg
Youtube	social	1.1M	3M	5.265	28754
Skitter	comp	1.7M	11M	13.08	35455
Flickr	social	1.7M	16M	18.13	27224
Wikitalk	comm	2.4M	5M	3.890	100029
Hollywood	social	1.1M	114M	98.91	11467
Orkut	social	3.1M	117M	76.28	33313
Enwiki	social	4.2M	101M	43.75	432260
Livejournal	social	4.8M	69M	17.68	20333
Indochina	web	7.4M	194M	40.73	256425
Twitter	social	42M	1.5B	57.74	2997487
Friendster	social	66M	1.8B	55.06	5214
UK	web	106M	3.7B	62.77	979738
Italianwiki	social	1.2M	35M	33.25	81090
Frenchwiki	social	2.2M	59M	26.36	137021

Note that FulFD and FulPLL can handle only a single edge insertion/deletion at a time. Thus, for a fair comparison, we also consider a unit-update variant of our algorithm: treating our method BHL$^+$ in the unit update setting by performing one update at a time. We call this unit-update variant as UHL$^+$. The code for FulFD, FulPLL and PSL* was provided by their authors and implemented in C++. We use the same parameter settings as suggested by their authors unless otherwise stated. For a fair comparison, we also select high degree landmarks and set them to 20 in the same way as FulFD for our methods. We set the number of threads to 20 for PSL* as well as for the parallel variant of our method BHL^p.

Datasets. We use 14 large real-world networks from a variety of domains to verify the efficiency, scalability and robustness of our algorithm. Among them, Italianwiki and Frenchwiki are two real dynamic networks whose topology evolves over time. We treat these networks as undirected and unweighted graphs, and their statistics are summarized in Table 2. They are accessible at Stanford Network Analysis Project [24], Laboratory for Web Algorithmics [9], and the Koblenz Network Collection [23].

Test data generation. For our batch dynamic variants, we generate 10 batches for the first 12 datasets, where each batch contains 1,000 edges randomly selected. We use three batch update settings for testing: (1) *decremental* - delete these batches and measure the average deletion time, (2) *incremental* - add these batches followed by decremental updates and measure the average insertion time,

Table 3: Comparing update time of our methods BHL$^+$, BHL and BHL^p with the state-of-the-art dynamic methods, where the batch size is 1,000 and thus the update time reported for every method is for 1,000 updates.

Dataset	Fully Dynamic Batch Update Time (sec.)						Incremental Batch Update Time (sec.)					Decremental Batch Update Time (sec.)				
	BHL^p	BHL$^+$	BHL	UHL$^+$	FulFD	FulPLL	BHL^p	BHL$^+$	UHL$^+$	IncFD	IncPLL	BHL^p	BHL$^+$	UHL$^+$	DecFD	DecPLL
Youtube	0.046	0.070	0.208	0.091	1.249	9.110	0.003	0.008	0.048	0.154	0.194	0.070	0.169	0.239	3.181	9.850
Skitter	0.147	0.601	0.902	1.587	5.986	8.770	0.002	0.006	0.069	0.117	1.312	0.163	0.751	2.382	14.15	31.50
Flickr	0.024	0.026	0.130	0.099	2.152	6.300	0.003	0.008	0.072	0.053	1.259	0.030	0.041	0.107	3.364	13.40
Wikitalk	0.029	0.025	0.101	0.134	2.926	4.550	0.002	0.005	0.097	0.029	0.081	0.046	0.044	0.147	5.674	9.820
Hollywood	0.008	0.014	0.115	0.056	4.423	-	0.001	0.002	0.046	0.090	27.53	0.017	0.031	0.071	8.401	-
Orkut	0.537	1.775	5.855	4.539	13.30	-	0.005	0.014	0.127	0.367	-	0.677	0.035	5.921	23.94	-
Enwiki	0.508	1.681	10.50	3.952	121.7	-	0.008	0.012	0.168	0.316	4.916	0.770	3.079	8.194	251.2	-
Livejournal	0.221	0.306	0.873	0.379	4.736	-	0.006	0.010	0.202	0.244	-	0.299	0.570	0.731	4.736	-
Indochina	0.543	1.181	1.547	9.575	20.63	-	0.015	0.011	0.308	0.141	4.680	0.553	1.346	19.20	44.92	-
Twitter	13.29	49.62	115.7	125.6	5103	-	0.125	0.024	13.09	0.263	-	19.17	68.85	231.8	9460	-
Friendster	0.409	0.410	0.811	21.93	23.27	-	0.163	0.035	20.96	0.254	-	0.420	0.738	21.87	30.38	-
UK	14.45	41.46	40.79	56.50	110.1	-	0.218	0.055	4.349	0.258	-	14.99	42.29	75.20	257.3	-
Italianwiki	0.001	0.001	0.025	0.051	6.623	-	-	-	-	-	-	-	-	-	-	-
Frenchwiki	0.003	0.004	0.067	0.098	5.289	-	-	-	-	-	-	-	-	-	-	-

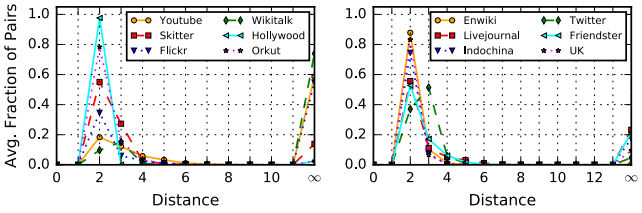


Figure 5: Distance distribution of batch updates.

and (1) *fully dynamic* - randomly select 50% updates in each of these 10 batches to delete and then measure the average update time after applying these batches. For the last two datasets, we select 10 batches in the order of their timestamps, each containing 1,000 real-world inserted/deleted edges and measure the average update time after applying them in a streaming fashion.

For the methods FulFD, FulPLL and UHL$^+$, we randomly sample 1000 edges and follow the same update settings as above to measure the update time of performing updates one by one. These settings enable us to explore the impacts of edge insertions and edge deletions respectively, in addition to their combined impact. In Figure 5, we report the distance distribution of edges in these batches after deleting. As we can see, the distances in all datasets are small ranging from 1 to 6. This shows that the updates are mostly from densely connected components of these networks which may cause fewer vertices to be affected in the *incremental* setting. Further, only a small number of updates are disconnected (i.e., have distance ∞) in most of these datasets.

For queries, we randomly sample 100,000 pairs of vertices in each dataset to evaluate the average querying time on graphs being changed as a result of fully dynamic batch updates. We also report the average size of labelling in the fully dynamic setting.

7.2 Performance Comparison

7.2.1 Update Time. Tables 3 and 4 show the average update time of our proposed and the baseline methods.

Fully dynamic setting. From Table 3, we see that our proposed methods BHL^p, BHL$^+$, and BHL significantly outperform FulFD and FulPLL on all datasets w.r.t. update time. In particular, our methods BHL^p and BHL$^+$ are over 15 times faster than FulFD on most of the datasets and several orders of magnitude faster than FulPLL. FulPLL only works on four graphs and fails to scale to large graphs with more than 100 millions. Further, the performance difference of BHL$^+$ and BHL is due to the fact that our improved batch search in BHL$^+$ can further prune away affected vertices that do not need to be repaired, and in practice they are significant in amount as can be seen in Table 5. Our methods also significantly outperform FuLFD on the real-world dynamic networks: Italianwiki and Frenchwiki. We can also observe that the average update time of BHL$^+$, BHL and BHL^p is always by far smaller than recomputing labelling from scratch, i.e., construction time of BHL$^+$ in Table 4. Notice that, we consider the same construction time for BHL^p and BHL as BHL$^+$, which is smaller than the construction time of baseline methods FulFD [21] and PSL* [25] on all datasets. We can see that the parallel variant of PLL (PSL*) still failed to construct labelling for the largest three datasets.

Incremental setting. Table 3 also shows that our methods BHL$^+$, BHL^p are considerably faster than the baseline methods IncFD and IncPLL. Even though IncFD and IncPLL do not preserve the minimality of distance labellings and thus do not spend time to delete outdated and redundant label entries, they are still slower than our methods. We can also see BHL$^+$ and BHL^p in the batch update setting are significantly faster than UHL$^+$ in the unit update setting. This is because UHL$^+$ requires extra usage of resource for each single update and involves in repeated and unnecessary computations. Here it is also to note that BHL^p does not perform well on the last four datasets as compared to BHL. This is because there only exist a very small number of average affected vertices against the total number of affected vertices as shown in Table 5. This confirms that the parallel variant of our method works very well when a large number of vertices are affected by batch updates; otherwise it may introduce unneeded thread overhead.

Decremental setting. It is evident from Table 3 that our methods BHL$^+$ and BHL^p are much faster than DecFD and DecPLL on all the datasets in this setting. Especially, BHL and BHL^p can achieve outstanding performance on networks which have a high average

Table 4: Comparing performance of our method BHL$^+$ with the baseline methods in terms of construction time, query time and labelling size. Note that when a method did not finish the labelling construction in 24 hours, we denote it as "-".

Dataset	Construction Time (CT) [s]				Query Time (QT) [ms]				Labelling Size (LS)			
	BHL$^+$	FulFD	FulPLL	PSL*	BHL$^+$	FulFD	FulPLL	PSL*	BHL$^+$	FulFD	FulPLL	PSL*
Youtube	2	4	84	4	0.005	0.010	0.045	0.002	20 MB	83 MB	3.14 GB	318 MB
Skitter	3	8	511	21	0.029	0.020	0.082	0.007	42 MB	153 MB	11.9 GB	1.01 GB
Flickr	3	10	546	23	0.007	0.013	0.102	0.005	34 MB	152 MB	13.1 GB	0.98 GB
Wikitalk	2	5	92	4	0.006	0.008	0.031	0.001	41 MB	74 MB	5.22 GB	160 MB
Hollywood	6	24	9,782	377	0.026	0.036	-	0.143	27 MB	263 MB	-	4.15 GB
Orkut	24	88	-	26,310	0.102	0.156	-	0.203	70 MB	711 MB	-	121 GB
Enwiki	25	88	7,382	389	0.053	0.051	-	0.021	82 MB	608 MB	-	7.04 GB
Livejournal	20	46	-	4,441	0.043	0.051	-	0.047	122 MB	663 MB	-	50.5 GB
Indochina	9	30	3,205	86	0.788	0.767	-	0.007	85 MB	838 MB	-	3.39 GB
Twitter	549	1,928	-	-	0.868	0.174	-	-	1.14 GB	3.83 GB	-	-
Friendster	1,181	3,365	-	-	0.815	0.902	-	-	2.43 GB	9.14 GB	-	-
UK	178	621	-	-	1.174	5.233	-	-	1.78 GB	11.8 GB	-	-
Italianwiki	6	15	-	215	0.008	0.014	-	0.006	23 MB	159 MB	-	0.81 GB
Frenchwiki	11	25	-	433	0.009	0.016	-	0.006	46 MB	272 MB	-	1.54 GB

Table 5: Average number of vertices affected by BHL$^+$ and BHL after performing batch updates on all the datasets.

Method	Type	Youtube	Skitter	Flickr	Wikitalk	Hollywood	Orkut	Enwiki	Livejournal	Indochina	Twitter	Friendster	UK	Italianwiki	Frenchwiki
BHL$^+$	Delete	366 K	971 K	55 K	127 K	14 K	503 K	1,220 K	276 K	2,079 K	10,622 K	66 K	54,515 K	-	-
	Add	23 K	11 K	22 K	16 K	2 K	3 K	4 K	12 K	15 K	2 K	6 K	12 K	-	-
	Mix	166 K	834 K	42 K	81 K	7 K	293 K	712 K	156 K	200 K	8,341 K	36 K	54,026 K	337	3 K
BHL	Mix	476 K	1,266 K	157 K	474 K	41 K	982 K	3,587 K	454 K	3,085 K	20,705 K	80 K	54,864 K	9 K	45 K

degree such as Twitter, Flickr and Hollywood. Due to inherent complexity of edge deletion on graphs (i.e., increasing distances), DecFD and DecPLL take very long in identifying and updating labels of affected vertices. As we can see, DecPLL does not have results on 8 out of 12 datasets. This is because while applying decremental updates their software either crashed or did not finish when the datasets are large that is why we don't have query time and labelling size after updates for these datasets in Table 4. Furthermore, our methods BHL$^+$ and BHL^p outperform UHL$^+$ because both leverage the benefit of handling updates in a batch and significantly reduce repeated computations during identifying and repairing the labels of affected vertices.

7.2.2 Labelling Size. Table 4 shows that BHL$^+$ has significantly smaller labelling size than FulFD, FulPLL and PSL* on all the datasets. When an update occurs, the labelling size of FulFD remains unchanged because they store complete shortest-path trees at all times. In contrast, BHL$^+$ stores pruned shortest-path trees preserving the property of minimality. Nonetheless, the labelling size of BHL$^+$ remains stable in practice because the average label size is bounded by a constant, i.e., the number of landmarks. The labelling size of FulPLL may increase significantly because IncPLL does not remove outdated and redundant distance entries and there is also no bound on labelling size. The parallel variant of PLL (PSL*) which exploit PLL properties to reduce labelling size still produces labelling of very large size as compared to BHL$^+$.

7.2.3 Query Time. Table 4 shows that the average query time of BHL$^+$ is comparable with FulFD and faster than FulPLL. It has been previously shown [12] that the average query time is largely dependent on labelling size. Since the dynamic operations do not considerably affect the labelling size for BHL$^+$ and FulFD, their query times remain stable. On Twitter, the query time of BHL$^+$ underperforms FulFD because FulFD also maintains the shortest-path information for the neighborhood of landmarks and we can see that the maximum degree of Twitter is very high which might cause many pairs to be covered by landmarks. However, the query time for FulPLL may considerably increase over time because they do not remove outdated entries, leading to labelling of increasing sizes. Although the query time of PSL* in Table 4 is better than BHL$^+$ on some datasets, it only handles static graphs. For dynamic graphs, it has the following limitations: (1) the cost of re-constructing labelling from scratch after each batch update is too high to afford, particularly when batch updates are frequent or when underlying dynamic graph is large which is evident from Table 4, (2) the labelling size is much larger than BHL$^+$. As we can see in Table 4, PSL* produces the labelling of size almost 99% larger than the labelling of BHL$^+$ for Orkut thus possess a high query cost as well. Considering the overall performance w.r.t. three main factors i.e., query time, labelling size and construction time, BHL$^+$ stands out in claiming the best trade-off between these factors.

7.3 Performance under Varying Landmarks

Figure 7 shows how the update time of our method BHL$^+$ in the fully dynamic setting behaves when increasing the number of landmarks. We can see that the update time for almost all datasets grows till 30 landmarks and then either decreases or remains stable. This is because selecting a larger number of landmarks can better leverage the pruning power of our method. On Twitter, we observe that the update time grows linearly due to its very high average degree which leads to a large fraction of vertices to be affected as can

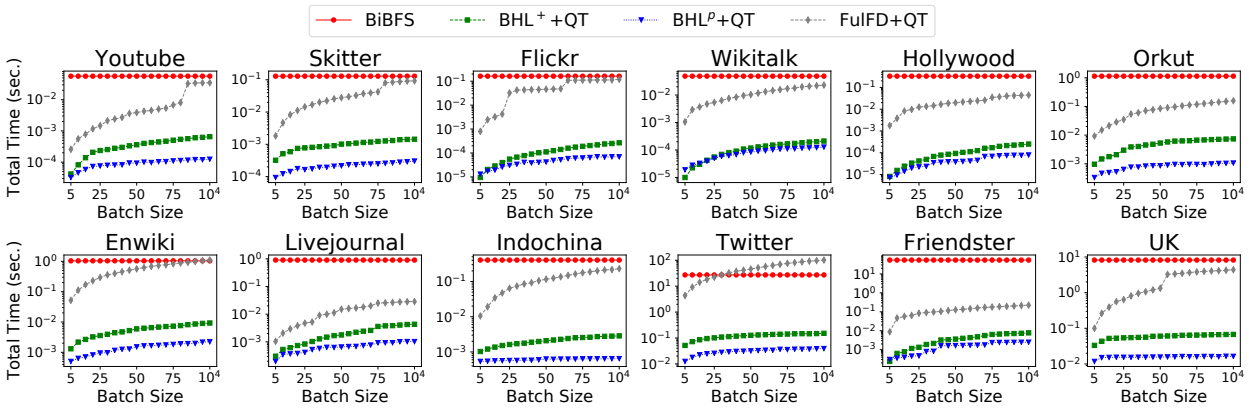


Figure 6: Total time of querying and updating by the proposed methods against online search methods.

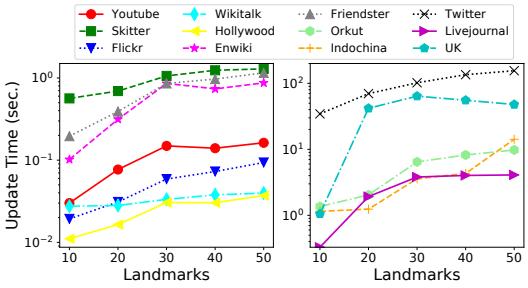


Figure 7: Update time under 10-50 landmarks.

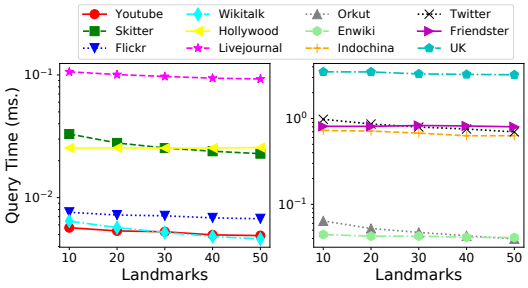


Figure 8: Query time under 10-50 landmarks.

be seen in Table 5 for 20 landmarks. We can also see in Figure 8 that the query time decreases or remains the same for almost all datasets with the increased number of landmarks. Particularly, the query time of Twitter, Orkut and Livejournal decreases because they have a very high average degree and selecting a larger number of high degree landmarks contributes greatly towards shortest-path coverage and makes querying process faster.

7.4 Performance under Varying Batch Sizes

We also compare the total time of querying and updating on dynamic graphs. To make a fair comparison, the total time of our methods BHL$^+$ and BHL^p, and the baseline method FulFD is the total time to perform a batch update plus the query time to perform 1000 queries after the batch update and then averaged over 1000 queries, denoted as BHL$^+$+QT, BHL^p+QT and FulFD+QT, respectively. We conduct the experiments for 5 randomly sampled fully dynamic batch updates of varying sizes, i.e., 500 to 10,000 in each batch. Figure 6 presents the results. For the baseline method BiBFS, we take only the query time averaged over 1000 queries after applying a batch update. We see that, the overall performance of our methods is significantly better than the baseline methods on all the datasets. It is worth noticing that BHL^p is not only more efficient than BHL$^+$, but also their efficiency gap becomes larger when the size of batch updates increases. This shows that the parallelism power of BHL^p can be better leveraged for batch updates of larger sizes. We can also observe that the update time along with the query time of our methods grows fast for batches of smaller sizes (with up to 1000 updates) and then grows very slowly when batch sizes become very large which shows that our methods are robust w.r.t the increased batch size.

7.5 Performance on Directed Graphs

We also conduct experiments on directed graphs. We can see in Table 6 that the update time of our methods is significantly smaller than the construction time of labelling from scratch. The update time of our optimized method BHL$^+$ is faster than the method BHL on all datasets except Livejournal. On Livejournal, the amount of affected vertices traversed by both BHL and BHL$^+$ is the same; however, due to additional overhead of computing extended landmark lengths, BHL$^+$ under-performs BHL. BHL^p is still the fastest among all methods. Our methods are also efficient in performing queries and have small labelling sizes.

Table 6: Comparing update time, construction time (CT), query time (QT) and labelling size (LS) on directed graphs.

Datasets	BHL^p[s]	BHL$^+$[s]	BHL[s]	CT[s]	QT[ms]	LS
Wikitalk	0.02	0.04	0.17	2.03	0.001	54 MB
Enwiki	2.98	12.5	28.0	46.8	0.023	177 MB
Livejournal	7.54	18.9	15.1	44.6	0.050	222 MB
Twitter	16.2	64.4	142	931	0.312	1.7 GB

8 CONCLUSION

We have proposed a novel method for answering distance queries on dynamic graphs undergoing batch updates. Our proposed approach exploits properties of updates in a batch to improve efficiency of maintaining distance labelling. We have analyzed the correctness and complexity of our approach and showed that they preserve the labelling minimality. We have empirically verified the efficiency and scalability of our approach on 14 real-world networks. For future work, we plan to explore the following directions: 1) the applicability and extension of the proposed method to road networks; 2) potential optimizations in designing separate batch-dynamic algorithms for edge insertion and edge deletion in dynamic graphs.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2012. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*. Springer, 24–35.

[2] Umut A Acar, Daniel Anderson, Guy E Blelloch, and Laxman Dhulipala. 2019. Parallel batch-dynamic graph connectivity. In *SPAA*. 381–392.

[3] Umut A Acar, Daniel Anderson, Guy E Blelloch, Laxman Dhulipala, and Sam Westrick. 2020. Parallel batch-dynamic trees via change propagation. In *ESA*.

[4] 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.

[5] 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.

[6] Takuya Akiba, Christian Sommer, and Ken-ichi Kawarabayashi. 2012. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *EDBT*. 144–155.

[7] 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.

[8] 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.

[9] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW*. 595–601.

[10] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. 2012. The exact distance to destination in undirected world. *The VLDB Journal* 21, 6 (2012), 869–888.

[11] 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.

[12] Gianlorenzo D'angelo, Mattia D'emidio, and Daniele Frigioni. 2019. Fully Dynamic 2-Hop Cover Labeling. *JEA* 24, 1 (2019), 1–6.

[13] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. 2014. Robust distance queries on massive networks. In *ESA*. 321–333.

[14] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. 2020. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *SODA*. 1300–1319.

[15] Laxman Dhulipala, Quanquan C Liu, and Julian Shun. 2020. Parallel Batch-Dynamic k-Clique Counting. *arXiv preprint arXiv:2003.13585* (2020).

[16] Muhammad Farhan and Qing Wang. 2021. Efficient Maintenance of Distance Labelling for Incremental Updates in Large Dynamic Graphs. *arXiv preprint arXiv:2102.08529* (2021).

[17] Muhammad Farhan, Qing Wang, Yu Lin, and Brendan Mckay. 2018. A Highly Scalable Labelling Approach for Exact Distance Queries in Complex Networks. In *EDBT*.

[18] Muhammad Farhan, Qing Wang, Yu Lin, and Brendan McKay. 2021. Fast fully dynamic labelling for distance queries. *The VLDB Journal* (2021), 1–24.

[19] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. 2005. Graph distances in the streaming model: the value of space.. In *SODA*, Vol. 5. Citeseer, 745–754.

[20] 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.

[21] Takanori Hayashi, Takuya Akiba, and Ken-ichi Kawarabayashi. 2016. Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks. In *CIKM*. 1533–1542.

[22] 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.

[23] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *WWW*. 1343–1350.

[24] Jure Leskovec and Andrej Krevl. 2015. SNAP Datasets:Stanford Large Network Dataset Collection. (2015).

[25] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling distance labeling on small-world networks. In *Proceedings of the 2019 International Conference on Management of Data*. 1060–1077.

[26] Ye Li, Man Lung Yiu, Ngai Meng Kou, et al. 2017. An experimental study on hub labeling based shortest path algorithms. *VLDB* 11, 4 (2017), 445–457.

[27] Andrew McGregor. 2014. Graph stream algorithms: a survey. *ACM SIGMOD Record* 43, 1 (2014), 9–20.

[28] Seth A Myers and Jure Leskovec. 2014. The bursty dynamics of the twitter information network. In *WWW*. 913–924.

[29] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. *Proceedings of the VLDB Endowment* 13, 5 (2020), 602–615.

[30] Anil Pacaci, Angela Bonifati, and M Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *ACM SIGMOD*. 1415–1430.

[31] Ira Pohl. 1971. Bi-derectional search. *Machine intelligence* 6 (1971), 127–140.

[32] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast shortest path distance estimation in large networks. In *CIKM*. 867–876.

[33] Yongrui Qin, Quan Z Sheng, Nickolas JG Falkner, Lina Yao, and Simon Parkinson. 2017. Efficient computation of distance labeling for decremental updates in large dynamic graphs. *WWW* 20, 5 (2017), 915–937.

[34] Robert Endre Tarjan. 1983. *Data structures and network algorithms*. Vol. 44. Siam.

[35] Antti Ukkonen, Carlos Castillo, Debora Donato, and Aristides Gionis. 2008. Searching the wikipedia with contextual information. In *CIKM*. 1351–1352.

[36] 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.

[37] Ye Wang, Qing Wang, Henning Koehler, and Yu Lin. 2021. Query-by-sketch: Scaling shortest path graph queries on very large networks. In *Proceedings of the 2021 International Conference on Management of Data*. 1946–1958.

[38] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *ACM SIGMOD*. 99–110.

[39] Sihem Amer Yahia, Michael Benedikt, Laks VS Lakshmanan, and Julia Stoyanovich. 2008. Efficient network aware search in collaborative tagging sites. *VLDB* 1, 1 (2008), 710–721.

[40] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2021. Efficient 2-hop labeling maintenance in dynamic small-world networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 133–144.