The VLDB Journal (2024) 33:101–129
https://doi.org/10.1007/s00778-023-00799-9

REGULAR PAPER

BatchHL$^+$: batch dynamic labelling for distance queries on large-scale networks

Muhammad Farhan[1] · Henning Koehler[2] · Qing Wang[1]

Received: 18 June 2022 / Accepted: 15 May 2023 / Published online: 7 June 2023

Abstract
Many real-world applications operate on dynamic graphs to perform important tasks. In this article, we study batch-dynamic algorithms that are capable of updating distance labelling efficiently in order to reflect the effects of rapid changes on such graphs. To explore the full pruning potentials, we first characterize the minimal set of vertices being affected by batch updates. Then, we reveal patterns of interactions among different updates (edge insertions and edge deletions) and leverage them to design pruning rules for reducing update search space. These interesting findings lead us to developing a new batch-dynamic method, called BatchHL$^+$, which can dynamize labelling for distance queries much more efficiently than existing work. We provide formal proofs for the correctness and minimality of BatchHL$^+$ which are non-trivial and require a delicate analysis of patterns of interactions. Empirically, we have evaluated the performance of BatchHL$^+$ on 15 real-world networks. The results show that BatchHL$^+$ significantly outperforms the state-of-the-art methods with up to 3 orders of magnitude faster in reflecting updates of rapidly changing graphs for distance queries.

Keywords Shortest-path distance · Batch-dynamic graphs · 2-Hop cover · High-way cover · Distance labelling maintenance · Graph algorithms

1 Introduction

Batch-dynamic algorithms on graphs have attracted considerable interest in recent years, for both fundamental research and practical applications [2, 3, 14, 15, 18, 29, 41]. Different from traditional dynamic algorithms which handle single changes sequentially—one at a time, batch-dynamic algorithms focus on dynamically maintaining graph properties, albeit graphs may undergo rapid changes through large batches of updates. Nowadays, many real-world applications operate on rapidly changing graphs, such as communication networks [34], context-aware search in web graphs [37], social network analysis [7, 38], route-planning in road networks [1, 13], and management of resources in computer networks [8].

Given a graph, a *distance query* computes the shortest path distance between two vertices in the graph, which is a fundamental problem both theoretically and in practice. A classical approach to answer distance queries is to run the Dijkstra's algorithm for non-negative weighted graphs or breadth-first search algorithm for unweighted graphs [36]. However, these algorithms are inefficient for large graphs. A well-established technique for speeding up query response time is to pre-compute and store distance labelling such as 2-hop labelling [11] and in particular pruned landmark labelling [4]. Although these labelling techniques have been shown to be effective in improving query response time on static graphs, the use of distance labelling poses an additional challenge for dynamic graphs—"How to efficiently update distance labelling in order to reflect changes on graphs, particularly when changes occur rapidly?"

Several studies have attempted to address this challenge by developing dynamic 2-hop labellings (in particular pruned landmark labellings) in the sequential setting which process one single update (edge insertion or edge deletion) at a time

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

Henning Koehler
h.koehler@massey.ac.nz

Qing Wang
qing.wang@anu.edu.au

1 School of Computing, Australian National University, Canberra, Australia

2 School of Mathematical and Computational Sciences, Massey University, Palmerston North, New Zealand

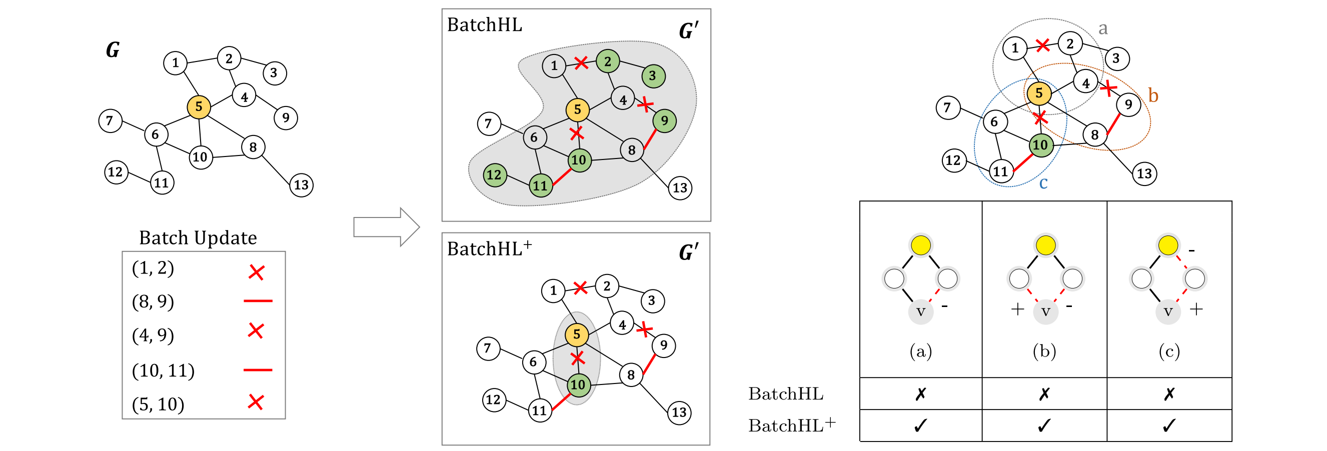


Fig. 1 A high-level overview of our batch-dynamic method BatchHL$^+$: (*left*) A graph G with a landmark 5 on which a batch update (a set of edge insertions and deletions) is performed; (*middle*) A comparison of affected vertices being identified by the state-of-the-art method BatchHL and our proposed method BatchHL$^+$; (*right*) Three different interaction patterns in which v can be pruned by BatchHL$^+$ but cannot be pruned by BatchHL

[5, 12, 23, 35]. However, the size of such distance labellings can grow quadratically with the size of a graph. This not only limits the applicability of these distance labellings to graphs with up to millions of nodes and edges, but also increases the difficulty and complexity of updating labellings. That is to say, even if such distance labellings can be constructed on graphs, the computational cost of updating them to reflect rapid changes is still unbearably high.

Recently, a batch-dynamic method to answer distance queries, namely BatchHL, has been proposed [18]. The key idea of BatchHL is to combine pre-computed labelling with online searches so as to exploit the merits of both sides—accelerating query response time through a *partial distance labelling* that is of limited size but can bound online search space. Further, BatchHL can efficiently dynamize a partial distance labelling to reflect large batches of updates on a graph. It has been shown [18] that BatchHL offers significant performance gains in comparison with other state-of-the-art methods for answering distance queries on dynamic graphs and can scale to billion-scale graphs without compromising query and update performance.

In this work, we further explore possible ways to advance the design of batch-dynamic algorithms for distance queries. We first analyse how different types of updates (edge insertion and edge deletion) interact with each other. Then, based on that, we unearth new patterns of update interactions that can be leveraged to design pruning rules for reducing update search space. These interesting findings lead us to developing a new batch-dynamic algorithm, called BatchHL$^+$, which can dynamize labelling for distance queries much more efficiently than BatchHL on graphs that undergo rapid changes. Figure 1 presents a high-level overview of comparing BatchHL and BatchHL$^+$ in terms of several typical update interaction patterns. We can see that the search space of BatchHL, the state-of-the-art method, is more exhaustive than BatchHL$^+$ when updating labelling to reflect batch updates on a graph. This is because BatchHL$^+$ further prunes search space by exploiting interactions between different types of updates. Concretely, in the lower right-hand corner of Fig. 1, three update interaction patterns are presented, and vertex v cannot be pruned by BatchHL but can be pruned by BatchHL$^+$ successfully. This enables BatchHL$^+$ to perform much more efficiently than BatchHL.

Novelty. This article is an extended version of the previous work [18]. The following contributions are novel.

– We explore update interaction patterns and characterize the minimal set of vertices affected by batch updates, for which a "perfect" batch-dynamic algorithm may wish to identify. We show that algorithms for only edge insertions or only edge deletions can precisely identify such vertices in each separate case. However, combining these two algorithms fails to compute the exact set of vertices affected by batch updates in the general case (Sect. 5).
– We propose an improved batch-dynamic method, namely BatchHL$^+$, which is equipped with optimized batch search and batch repair algorithms. In the original work of BatchHL, a vertex is flagged as affected if *any* shortest path is eliminated. In contrast, BatchHL$^+$ proposed in this work only flags a vertex as affected if *every* shortest path is eliminated. As a result, edge deletions can be processed almost as quickly as edge insertions (Sect. 6).
– We present a comprehensive discussion on distinctive characteristics of different batch-dynamic algorithms and their relationships in terms of their capacity of identi-

fying vertices that are "affected" by batch updates on dynamic graphs. This leads to several different notions of "affected" vertices and the connections between these notions manifest the source of efficiency differences among different batch-dynamic algorithms. For example, BatchHL$^+$ handles a much smaller subset of "affected" vertices than BatchHL (Sect. 7).

– We provide detailed proofs for the correctness and minimality of BatchHL$^+$. We note that the proof of correctness for the batch search algorithm (i.e. Algorithm 6) of BatchHL$^+$ is non-trivial and requires a delicate analysis of interaction patterns of batch updates. We also conduct the time and space complexity analysis of BatchHL$^+$ (Sect. 8).

Empirically, we have evaluated our method on 15 real-world networks to verify efficiency and scalability. The results show that our method significantly improves update time compared to the state-of-the-art methods. It can maintain labelling of a very small size, while still answering distance queries in the order of milliseconds, even on large-scale graphs with several billions of edges that undergo large batches of updates. The average update times on all these real-world networks are less than one millisecond, and in general, our method is up to 3 orders of magnitude faster than the recent state-of-the-art method BatchHL.

2 Related work

Traditionally, distance queries can be answered using Dijkstra's search, breadth-first search (BFS), or a bidirectional scheme combining two such searches: one from the source vertex and the other from the destination vertex [33, 36]. 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, 19, 22, 24, 28, 39, 40]. 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. [27] developed a parallel algorithm for constructing PLL which achieved the state-of-the-art results for answering distance queries on static graphs.

Previously, several attempts have been made to study distance queries over dynamic graphs [5, 12, 16, 20, 23, 31, 35, 42] 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. [35] 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 19 M edges is 135 s in [35] and on a network with 16 M edges is 19 s 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 however does not scale beyond networks with around 20 M of edges. Hayashi et al. [23] 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. Very recently, several methods have attempted to perform updates in the batch-update setting (i.e. to perform multiple updates in a batch) [17, 18]. For instance, BatchHL [18] has shown to be much faster in performing batch updates as compared to the other state-of-the-art methods. In our present work, we further advance the design of BatchHL by studying patterns that were left unexplored in the design of BatchHL.

Another line of research studied streaming graph algorithms. In the streaming setting, a rapidly changing graph is often modelled 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 [21, 30, 32] 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

Without loss of generality, we focus our discussion on unweighted, undirected graphs in this paper and discuss the extension to directed graphs in Sect. 9. Table 1 summarises the frequently used notations.

Table 1 Summary of notations

Notation	Description
$G = (V, E)$	A graph G with the set of vertices V and the set of edges E
R	A subset of vertices, called landmarks
$L(v)$	Label of a vertex v
$\Gamma = (H, L)$	A highway cover labelling, consisting of a highway H and a distance labelling L
$N(v)$	Set of vertices adjacent to v in a graph G
$d_G(u, v)$	Shortest path distance between u and v in a graph G
$P_G(u, v)$	Set of all shortest paths between u and v in a graph G
$\delta_H(u, v)$	Highway distance between u and v
$\delta_L(u, v)$	Labelling distance between u and v
$G[V\backslash R]$	A sparsified graph after removing R from G
$d_{uv}^{\top}$	An upper distance bound between u and v
$Q(u, v)$	An exact query between u and v
$d_G^L(r, v)$	Landmark distance between r and v in G
$V_{\text{AFF}}/V_{\text{AFF+}}$	Set of affected vertices, for some notion of "affected"
B	A batch update
$G' = (V', E')$	Graph after batch update B
$\Gamma' = (H', L')$	An updated highway cover labelling
$\mathbb{N}/\mathbb{B}$	Set of all natural numbers / Boolean values
$\oplus$	Append operator to update the landmark length of a path
$d_G^L(r, v)$	Landmark distance between r and v in G
$d_C(r, v)$	Composite distance between r and v
$\|S\|$	Number of elements in a set S
$d_{\text{BOU}}(v, S)$	Distance bound of a vertex v w.r.t. a set S
$d_{\text{BOU}}^L(v, S)$	Landmark distance bound of a vertex v w.r.t. a set S

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, $d_G(s, t) = \infty$. 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 neighbours of a vertex v, i.e. $N(v) = \{v' \in V | (v, v') \in E\}$.

There are two types of edge updates on graphs: *edge insertion* and *edge deletion*. A *batch update* is a set of edge insertions and deletions. Node insertion or deletion can be treated as a batch update containing only edge insertions or only edge deletions, respectively. 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 $(a, b) \in E$. Without loss of generality, we ignore invalid updates.

Let $R \subseteq V$ be a subset of special vertices in G, called *landmarks*. A *label* $L(v)$ for a vertex 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. A *distance labelling* over G is the set of labels for all vertices in V. 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 common vertex in $L(u)$ and $L(v)$ to be on a shortest path between any two vertices u and v.

Definition 1 (*2-hop cover labelling*) A distance labelling L over $G = (V, E)$ is a *2-hop cover labelling* if for any $s, t \in V$,

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

3.1 Highway cover labelling

Our work considers the highway cover labelling [19].

Definition 2 (*Highway*) A *highway* $H = (R, \delta_H)$ consists of a set R of landmarks and a distance decoding function $\delta_H : R \times R \rightarrow \mathbb{N}^+$ 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 (*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 $r \in R$,

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

A highway cover labelling requires that every label $L(v)$ must contain a distance entry to each landmark $r \in R$ unless there is another landmark on a shortest path between r and v. We shall refer to this distance entry (or its absence) as the *r-label* of 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*.

As discussed in the original paper [19], highway cover labelling enjoys several nice theoretical properties, such as minimality and order independence. 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. A highway cover labelling $\Gamma = (H, L)$ over G is *order-independent* if Γ remains the same, regardless of the order of applying landmarks in R. Given any fixed set of landmarks, there exists a unique minimal highway cover labelling, which is contained in every highway cover labelling [19].

3.2 Query answering

A distance query can be answered via bounding online searches on a sparsified search space based on the highway cover labelling. Specifically, given a highway cover labelling $\Gamma = (H, L)$, an upper bound on the distance between any pair of vertices $s, t \in V$ in a graph G is computed as

$$d^\top_{st} = \min\{\delta_L(r_i, s) + \delta_H(r_i, r_j) + \delta_L(r_j, t) \mid (r_i, \delta_L(r_i, s)) \in L(s), (r_j, \delta_L(r_j, t)) \in L(t)\}.$$

Here, $d^\top_{st}$ 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 $d^\top_{st}$ to answer the distance query $Q(s, t)$ such that

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

In the implementation, $d_{G[V\backslash R]}(s, t)$ can be computed by conducting a bidirectional BFS search from both s and t [19] which terminates either after $d^\top_{st} - 1$ steps or when the searches from both directions meet.

4 BatchHL: basic algorithm

In this section, we start by presenting the basic algorithm of BatchHL, a batch-dynamic method that can efficiently maintain a highway cover labelling for dynamic graphs [18]. Generally, BatchHL involves two phases: *Batch Search* and *Batch Repair*, as described in Algorithm 1. Batch Search identifies vertices for which labels may need to be updated, while Batch Repair updates these vertices. These two phases are done independently for each landmark.

Algorithm 1: BatchHL (Basic Algorithm)

```
1 Function BatchHL($G'$, $B$, $R$, $\Gamma$)
2     $\Gamma' \leftarrow \Gamma$
3     foreach $r \in R$ do
4         $V_{AFF} \leftarrow$ BatchSearch($G'$, $B$, $r$, $\Gamma$)
5         BatchRepair($G'$, $V_{AFF}$, $r$, $\Gamma$, $\Gamma'$)
6     return $\Gamma'$
```

4.1 Batch search

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.

Definition 4 (*P-affected*) A vertex $v \in V$ is *path-affected (P-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 $V_{AFF}(r, B) = \{v \in V | P_G(v, r) \neq P_{G'}(v, r)\}$ to denote the set of all P-affected vertices by a batch update B w.r.t. a landmark r. An edge insertion or deletion (a, b) can create or eliminate shortest paths starting from r and passing through (a, b). 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.

For a non-trivial update (a, b), we call the vertex $u \in \{a, b\}$ further away from r in G the *anchor* of (a, b), and the other one its *pre-anchor*. We denote the *anchor distance* of (a, b) by $d_G(r, u) = d_G(r, u') + 1$, where u' is the pre-anchor of (a, b).

As every newly created or eliminated path must pass through an updated edge, we can use anchors to identify P-affected vertices.

Lemma 1 *For every P-affected vertex v we have*

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

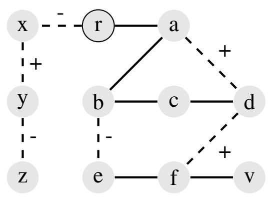
Algorithm 2: Batch Search

```
Function BatchSearch($G'$, $B$, $r$, $\Gamma$)
    foreach $(a, b) \in B$ do
        if $d_G(r, a) < d_G(r, b)$ then
            add $(d_G(r, a) + 1, b)$ to $\mathcal{Q}$
        else if $d_G(r, a) > d_G(r, b)$ then
            add $(d_G(r, b) + 1, a)$ to $\mathcal{Q}$
    while $\mathcal{Q}$ is not empty do
        remove minimal $(d, v)$ from $\mathcal{Q}$
        if $v \notin V_{\text{AFF+}}$ then
            add $v$ to $V_{\text{AFF+}}$
            foreach $w \in N_{G'}(v)$ do
                if $d + 1 \leq d_G(r, w)$ then
                    add $(d + 1, w)$ to $\mathcal{Q}$
    return $V_{\text{AFF+}}$
```

for some non-trivial update $(a, b) \in B$ with anchor u and pre-anchor u'.

We make use of this characterization in Algorithm 2. Specifically, we perform a search with anchors as starting points, compute the minimal value of the right hand side of Eq. 1, and collect all vertices for which Eq. 1 holds.

Example 1 Consider the graph below, where edges marked by + are inserted and edges marked by − are deleted in one batch update.

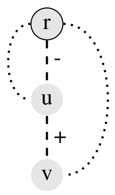


Insertion of (a, d) triggers a search starting from the anchor d with the anchor distance $d_G(r, a) + 1$ computed from Γ. The search then follows edges in G', including the newly inserted edge (d, f), and finds that d, c, f and v satisfy (1). Insertion of (d, f) is trivial since d and f are equidistant from r in G, and does not lead to any search. Deletion of (b, e) triggers a search starting from the anchor e with the anchor distance $d_G(r, b) + 1$ computed from Γ.

Combining the searches for (a, d) and (b, e) into one search means that the edge (f, v) is traversed only once.

Algorithm 2 eliminates unnecessary search paths by not following vertices violating Eq. 1 and also avoids traversing vertices affected by multiple updates more than once. However, Algorithm 2 does not precisely compute the set of all P-affected vertices, but the set of all vertices satisfying Eq. 1, which is a superset. The following example illustrates why the characterization in Lemma 1 is not precise, and why precise discovery of P-affected vertices is difficult.

Example 2 Consider the graph below, where the dotted edge between r and u indicates a long path between them, and the dotted edge between r and v indicates another long 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 P-affected—it hinges on whether the long path between r and v is longer (or equal) than the long path between r and u plus 1, which cannot be ascertained by Γ.

The set of vertices returned by Algorithm 2 can be precisely characterized as follows.

Definition 5 (*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. Consider Example 1 again. $r - x$ is an insignificant composite path, $r - x - y$ is a significant composite path, and $r - x - y - z$ is not a composite path. To see the latter, consider that any inserted edge must belong to the G' part, and thus later edges would need to lie in G' as well, i.e. cannot be deleted.

Definition 6 (*CP-affected*) A vertex v is *composite-path-affected* (*CP-affected*) w.r.t. $r \in R$ iff

(i) v is P-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.

Algorithm 2 returns the set of all CP-affected vertices which includes all P-affected vertices. Additional vertices due to condition (ii) are undesirable but hard to avoid, as illustrated in Example 2. This happens because the starting distance is calculated w.r.t. G, and we are thus effectively considering paths for which the first part (from r to an anchor) lies in G, and the rest in G'. For example, the vertex y in Example 1 is not P-affected but CP-affected, and Algorithm 2 returns it.

Lemma 2 *A vertex v is CP-affected w.r.t. r iff there exists a composite path from r to v of length at most $d_G(r, v)$ that passes through an updated edge.*

Proof We show each direction separately.

- (if) Let p be a composite path 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 P-affected. If p lies in G' then either it lies in $P_{G'}(r, v)$, or there exists a strictly shorter path p' in $P_{G'}(r, v)$. Neither p not p' lies in $P_G(r, v)$, so v is P-affected. If p lies neither in G nor in G' then it must be significant.
- (only if) 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, there exists a significant composite path of length at most $d_G(r, v)$, which passes through edges in B by definition.

□

Lemma 3 *Algorithm 2 returns the set of all CP-affected vertices.*

Proof We show that a vertex v is added to $V_{\text{AFF+}}$ iff there exists a composite path p of length at most $d_G(r, v)$ that passes through an updated edge.

- (if) Let $v \in V_{\text{AFF+}}$. Vertices are only added to $\mathcal{Q}$ and thus to $V_{\text{AFF+}}$ if they pass one of three distance checks. In each case, this compares the length of a composite path passing through an updated edge to the distance from r in G.
- (only if) 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 distance bound for b, and proceed along p_{bv}. Thus, for every vertex $w \in p_{bv}$, including v, it will obtain $|p_{rw}| \le d_G(r, w)$ as distance bound for w, and add w to $V_{\text{AFF+}}$.

□

In particular, it follows that (1) characterizes CP-affected vertices, rather than P-affected ones.

4.2 Batch repair

Algorithm 3 describes the approach for repairing the labels of affected vertices returned by Algorithm 2. At its core, the algorithm starts with *boundary vertices* that lie on the boundary of affected and unaffected vertices, and whose distances to r are computed from neighbouring 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.

Definition 7 (*Boundary vertex*) A vertex v is a *boundary vertex* w.r.t. a landmark r if v is an affected vertex (for some version of "affected", e.g. CP-affected) but has an unaffected neighbour w.

Denote by $V_{\text{AFF+}}$ the set of affected vertices, and 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 – while we employ different variants of "affected" throughout the paper, "unaffected" vertices will always retain their distance to r. By taking the minimum of such known upper bounds, we get a readily available distance bound for v.

Definition 8 (*Distance bound*) Let $S \subset V \setminus \{r\}$ be a set of vertices. The *distance bound* of v w.r.t. S is:

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

The following lemma allows us to compute the distance of vertices in $V_{\text{AFF+}}$ from r in G' using their distance bounds.

Lemma 4 *Let $S \subset V\backslash\{r\}$ and $v \in S$ with minimal distance bound. Then $d_{G'}(r, v) = d_{\text{BOU}}(v, S)$.*

Proof For $d_{G'}(r, v) = \infty$ this is trivial. Otherwise, let p be a shortest-path from r to v in G', v' be the first vertex in p that lies in S, and w its predecessor in p. Since $w \notin S$ we have $d_{\text{BOU}}(v', S) \le d_{G'}(r, w) + 1 = d_{G'}(r, v')$. If $v' \ne v$ 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 $d_{\text{BOU}}(v, S) = d_{G'}(r, v)$. □

Note that $d_{G'}(r, v) = d_{\text{BOU}}(r, S)$ does not generally hold for every boundary vertex v. Based on Lemma 4, we repair labels by starting with boundary vertices that have the smallest distance bound. After each iteration, we treat affected vertices with repaired labels as being unaffected and find boundary vertices that have the smallest distance bound again. This process terminates only when the labels of all affected vertices are repaired.

Algorithm 3 shows the pseudo-code of the batch repair algorithm used by BatchHL. Given a graph G' and a set of all affected vertices $V_{\text{AFF+}}$, we first compute the distance bounds of vertices in $V_{\text{AFF+}}$ using their unaffected neighbours. We

Algorithm 3: Batch Repair

```
1  Function BatchRepair($G'$, $V_{\text{AFF+}}$, $r_i$, $\Gamma$, $\Gamma'$)
2      foreach $v \in V_{\text{AFF+}}$ do
3          $D_{\text{BOU}}[v] \leftarrow d_{\text{BOU}}(v, V_{\text{AFF+}})$ // use $\Gamma$ to compute
4      while $V_{\text{AFF+}}$ is not empty do
5          $V_{\min} \leftarrow \{v \in V_{\text{AFF+}} \mid D_{\text{BOU}}[v]$ is minimal$\}$
6          remove $V_{\min}$ from $V_{\text{AFF+}}$
7          foreach $v \in V_{\min}$ do
8              set $r$-label of $\Gamma'(v)$ to $(r_i, D_{\text{BOU}}[v])$
9              foreach $w \in N_{G'}(v) \cap V_{\text{AFF+}}$ do
10                 $D_{\text{BOU}}[w] \leftarrow \min(D_{\text{BOU}}[w], D_{\text{BOU}}[v] + 1)$
```

then find vertices in $V_{\text{AFF+}}$ with the minimal distance bounds and remove them from $V_{\text{AFF+}}$. By Lemma 4, their distances to r in G' equal their distance bounds. For each $v \in V_{\min}$, we set the r-label of v to $(r_i, D_{\text{BOU}}[v])$ (Line 8) and assign new distances to the affected children of v in $V_{\text{AFF+}}$ (Lines 9-10). We continue this process until V_{AFF} is empty.

Note that Algorithm 3 does not eliminate redundant r-labels. We will address this issue in Sect. 6.2.

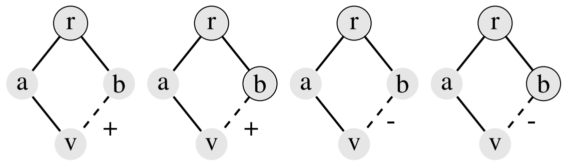
Remark 1 We note that the work in [18] has proposed several optimization techniques to reduce the set of CP-affected vertices returned by Algorithm 2 and can thus efficiently repair the affected vertex set. These optimization techniques have been shown to significantly improve the performance of updating labelling. Thus, in our experiments (Sect. 10), we will compare the performance of our proposed method BatchHL$^+$ against the method of BatchHL that has incorporated these optimization techniques.

5 Separate algorithms—insertion and deletion

We observe that changes to shortest paths between r and v do not always cause a change in distance. Based on this observation, we shall differentiate between new and eliminated paths, and strengthen the pruning condition $d + 1 \le d_G(r, w)$ in Line 12 of Algorithm 2 to $d + 1 < d_G(r, w)$ for new paths. For eliminated paths, we examine all neighbouring vertices to ensure that *all* shortest paths have been eliminated.

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

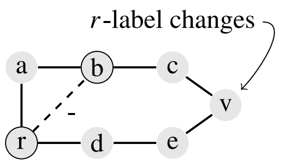
Example 3 Consider the following graphs and updates, where the landmarks are circled. In all cases, vertex v is affected, but the distance between r and v does not change. For case (a) adding the edge (b, v) does not cause a label change for v. It does however for case (b) where b is a landmark, causing 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 due to being redundant, as illustrated by the example below.

Example 4 Consider the graph below, 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 the landmark b without changes. 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 the landmark b, similar to case (d) in Example 3. 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 want to return (at least) all those vertices for which either label or distance changes.

Definition 9 (*LD-affected*) A vertex v is *landmark-distance-affected (LD-affected)* by a batch update B w.r.t. a landmark $r \in R$ iff there is a

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

As seen in Example 3, 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 10 (*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 $|p|_{\mathrm{L}}$. The *landmark distance* between r and v in G is the minimal landmark length of paths between them, denoted as

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

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 *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 6 *A vertex v is LD-affected iff it satisfies:*

$$d_G^L(r, v) \neq d_{G'}^L(r, v).$$

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 9 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 follows from Lemma 5. □

5.1 Insertion-only batch search

To identify LD-affected vertices in the case of edge insertions, for an inserted edge (a, b), we need to compute the minimal landmark length of paths from r to b passing through a as an upper bound for the landmark distance of b. Since this may lead to frequently updating the landmark length of a path when appending another vertex, we define an operator for this:

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

Batch search for insertions is described in Algorithm 4. Its correctness is straightforward.

Lemma 7 *Algorithm 4 returns the set of LD-affected vertices.*

Algorithm 4: Insertion-Only Batch Search

```
Function BatchSearchInsert($G'$, $B$, $r$, $\Gamma$)
    foreach $(a, b) \in B$ do
        if $d_G^L(r, a) \oplus b < d_G^L(r, b)$ then
            add $\left(d_G^L(r, a) \oplus b,\ b\right)$ to $\mathcal{Q}$
        else if $d_G^L(r, b) \oplus a < d_G^L(r, a)$ then
            add $\left(d_G^L(r, b) \oplus a,\ a\right)$ to $\mathcal{Q}$
    while $\mathcal{Q}$ is not empty do
        remove minimal $(d, l, v)$ from $\mathcal{Q}$
        if $v \notin V_{\text{LD-AFF}}$ then
            add $v$ to $V_{\text{LD-AFF}}$
            foreach $w \in N_{G'}(v)$ do
                if $(d, l) \oplus w < d_G^L(r, w)$ then
                    add $((d, l) \oplus w, w)$ to $\mathcal{Q}$
    return $V_{\text{LD-AFF}}$
```

Proof Any vertex added to $V_{\text{LD-AFF}}$ must be LD-affected, as its landmark distance in G' is lower than in G.

Conversely, if v is LD-affected, there must exist a strictly shorter path p in G'. Let p be the shortest such path, b be the first LD-affected vertex in p and a its predecessor. Then the edge (a, b) must be inserted, and all vertices from b onwards must be LD-affected. As vertices are processed in order of distance, the computed distance bound will be minimal, causing b and all later vertices in p to be added to $V_{\text{LD-AFF}}$. □

5.2 Deletion-only batch search

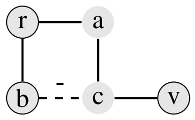
The key idea for identifying an LD-affected vertex v in the case of edge deletions is to examine its neighbourhood. While this will not always give us its landmark distance in G' as neighbours may not have been updated yet, it does provide a lower bound that is sufficient for determining LD-affectedness. In case v turns out to be LD-affected, we will need to examine its neighbours anyways to identify other LD-affected vertices, and some of the required computations can be reused.

To identify a vertex v as LD-affected, we use two criteria:

1. There must exist a path between r and v in G of length $d_G^L(r, v)$ that passes through a deleted edge.
2. There must not exist a path between r and v in G' of length $d_G^L(r, v)$.

The first criterion provides us with candidates to examine further, while the second criterion is used to confirm or reject candidates. However, simply storing whether a vertex is LD-affected is not enough for this purpose.

Example 5 Consider the graph below, where r, b and v are landmarks and edge (b, c) is getting deleted.

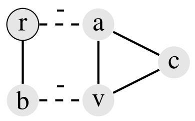


Here, c is LD-affected as its landmark distance reduces from $(2, L)$ to $(2, -)$. However, v is not LD-affected as its landmark distance remains as $(3, L)$. Deciding the latter is impossible if all we know is that its only neighbour is LD-affected.

While the issue could be solved by storing the new landmark distance for LD-affected vertices, we do not have this information available during batch search (e.g. for vertex a in Example 6). However, we *can* decide whether a path of equal length—but not passing through another landmark—still exists in G', and storing this information is sufficient for deciding LD-affectedness of other landmarks.

Another key observation concerns the neighbours of v through which a path violating the second criterion (path in G' of equal length) might pass. By processing vertices in order of their distance from r in G, we ensure that these will already have been processed.

Example 6 Consider the graph below, where edges (r, a) and (b, v) are getting deleted.



To confirm that v is LD-affected, we examine its neighbours a and c in G'. Here a has already been processed and marked as LD-affected, as it is closer to r than v. Vertex c may not have been processed yet, and thus may wrongly suggest a path between r and v of length 3 in G'. However, this is irrelevant for deciding whether a path of length $d_G(r, v)$ or $d^L_G(r, v)$ exists in G'.

The resulting algorithm for batch search in case of deletion is given as Algorithm 5. Here $V_{\text{D- AFF}}$ contains distance-affected vertices. We show its correctness next.

Algorithm 5: Deletion-Only Batch Search

```
1  Function BatchSearchDelete($G'$, $B$, $r$, $\Gamma$)
2      foreach $(a, b) \in B$ do
3          if $d^L_G(r, a) \oplus b = d^L_G(r, b)$ then
4              add $(d^L_G(r, a) \oplus b,\ b)$ to $\mathcal{Q}$
5          else if $d^L_G(r, b) \oplus a = d^L_G(r, a)$ then
6              add $(d^L_G(r, b) \oplus a,\ a)$ to $\mathcal{Q}$
7      while $\mathcal{Q}$ is not empty do
8          remove minimal $(d_v, l_v, v)$ from $\mathcal{Q}$
9          if $v \notin V_{\text{LD- AFF}}$ then
10             $\mathcal{N} \leftarrow \emptyset$, v_daff $\leftarrow$ True
11             foreach $w \in N_{G'}(v) \setminus V_{\text{D- AFF}}$ do
                   // check for path of equal (landmark) length
12                 if $d_v = d_G(r, w) + 1$ then
13                     v_daff $\leftarrow$ False
14                     if $(d_v, l_v) = d^L_G(r, w) \oplus v$ then
15                         if $w \notin V_{\text{LD- AFF}} \lor v \in R$ then
16                             continue from line 7
                   // check if w might be LD-affected
17                 else if $(d_v, l_v) \oplus w = d^L_G(r, w)$ then
18                     add $((d_v, l_v) \oplus w, w)$ to $\mathcal{N}$
19             add $v$ to $V_{\text{LD- AFF}}$
20             if v_daff then
21                 add $v$ to $V_{\text{D- AFF}}$
22             add tuples in $\mathcal{N}$ to $\mathcal{Q}$
23     return $V_{\text{LD- AFF}}$
```

Lemma 8 *Algorithm 5 returns the set of LD-affected vertices.*

Proof Let v be added to $V_{\text{LD- AFF}}$. This only happens if (d_v, l_v, v) was previously added to $\mathcal{Q}$, at which point it was checked that (d_v, l_v) equals the landmark distance $d^L_G(r, v)$. Additionally, line 16 must not have been reached, meaning the checks for a path in G' of landmark length that equal to (d_v, l_v) must have failed. For the correctness of this check, consider that any such path would need to pass through some neighbour of v in G', and this neighbour would need to be closer to r and not distance-affected. It follows that G' contains no path from r to v of landmark length $d^L_G(r, v)$, meaning that v is LD-affected.

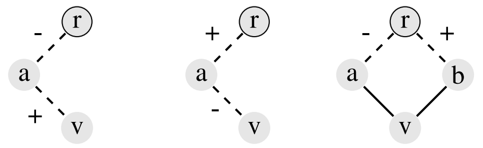
Conversely, let v be LD-affected and p be some shortest path from r to v in G w.r.t. landmark length. As p does not lie in G', it must pass through one or more deleted edges. Let (a, b) be the deleted edge in p closest to v, with b closer to v than a. Then, all vertices in p between b and v must be LD-affected. By induction we may assume that all of them except v will be added to $V_{\text{LD- AFF}}$. As all neighbours of vertices in $V_{\text{LD- AFF}}$ are checked for being LD-affected, v will be checked as well and added to $V_{\text{LD- AFF}}$. □

Remark 2 Both insertion-only and deletion-only batch search algorithms can compute LD-affected vertices when they handle only edge insertions or only edge deletions. However, when applying these two algorithms together on batch updates with mixed inserted and deleted edges, e.g. applying insertion-only algorithm on edge insertions first and then deletion-only algorithm on edge deletions, or the reversed order, they fail to compute the exact set of LD-affected vertices. The next section will discuss the reasons in detail, which motivates our development of an improved algorithm BatchHL$^+$.

6 BatchHL$^+$: improved algorithm

In principle, one could handle any batch update consisting of both insertions and deletions by processing insertions first and deletions after, or vice versa. However, the issue with such an approach is that vertices may be affected by both insertions and deletion, causing them to be processed twice. It is even possible that insertions and deletions cancel each other out so that vertices are processed twice even though they would not need to be processed at all.

Example 7 Consider the following graphs and updates. When considering insertions and deletions together, v is not LD-affected in either of the three cases. However, for cases (a) and (b), processing insertions first and deletions after will cause v to be LD-affected at both times. Similarly, for case (c), processing deletions first and insertions after will cause v to be LD-affected at both times.

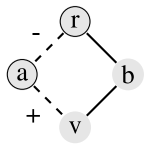


While it is challenging to identify all cases where insertions and deletions cancel each other out, we will be able to avoid some of them. For example, we can avoid returning v for cases (b) and (c) in Example 7, but not for case (a). For case (a), the difficulty lies in not knowing the distance between r and a in the changed graph G'. As illustrated in Example 6, computing this during batch search is tricky.

Even though we will present an improved algorithm for processing mixed batches in the next section, it may be worth pointing out a small trick when splitting a batch B into insertions B^+ and deletions B^-. Instead of running Algorithm 4 on the graph obtained by applying B^+ to G, we can run it on the graph obtained by apply the whole batch B. This prevents us from following paths that pass through a deleted edge after first passing through an inserted edge, as depicted in case (b) of Example 7.

Clearly, there are still cases where this trick cannot stop an unaffected vertex from getting flagged as affected. Case (a) of Example 6 was one such case. There are other cases as shown in the example below.

Example 8 Consider the following graph and updates.



Here, v is not LD-affected but still returned by Algorithm 4, even with all of B used in constructing G'.

The improved algorithm presented next will avoid returning v, as the insertion of (a, v) will only be processed after the deletion of (r, a) has already been considered. Here the existence of the alternate path $r - b - v$, which has the same length as $r - a - v$, is crucial in avoiding the difficulties illustrated in Example 6. By the time v is being processed, we can be sure that b is unaffected and can thus conclude that v is unaffected.

6.1 Improved batch search

To combine Algorithms 4 and 5, consider their search process. Both algorithms track the lengths of paths $r \dots a - b \dots v$ passing through an updated edge (a, b). As the distance between r and a is computed using the highway labelling Γ, and edges used to extend the paths lie in G', we are effectively considering paths that lie partially in G and partially in G'. Therefore, we define the notion of composite distance.

Definition 11 (*Composite distance*) The *composite distance* between r and v over (G, G') is the length of the shortest composite paths between them. We denote it as

$$d_C(r, v) := \min \{ |p| \mid p \text{ is composite path from } r \text{ to } v \}$$

Example 9 Consider again the cases from Example 7. Here $r - a - v$ is a composite path for case (a) but not for case (b). The composite distance between r and v is 2 for case (a) and ∞ for case (b).

Our improved algorithm is presented as Algorithm 6. A detailed discussion is deferred until Sect. 8, where we prove that the vertex set identified by it can be characterized as follows.

Definition 12 (*LCD-affected*) A vertex v is *landmark-composite-distance-affected (LCD-affected)* w.r.t. a landmark r iff one of the following conditions is satisfied:

- Landmark distance changes: $d^L_{G'}(r, v) \neq d^L_G(r, v)$;
- Composite distance changes: $d_C(r, v) \neq d_G(r, v)$.

Clearly, this includes all LD-affected vertices, but may also include others, e.g. case (a) in Example 7, which is undesirable. The reason for those additional vertices is that we cannot easily compute $d^L_{G'}(r, v)$ during batch search. However, we *can* compute $d_C(r, v)$ and decide whether a path of length $d_C(r, v)$ exists in G', essentially the same way as this was done in Algorithm 5. We call vertices where such a path exists *stable*, and use this to resolve situations similar to Example 8.

We note that whenever B contains only inserted or only deleted edges, vertices are LCD-affected iff they are LD-affected. That is because edge deletions do not change the composite distance, while the absence of edge deletions means that a change in composite distance implies a change in landmark distance.

6.2 Improved batch repair

Now, we introduce improved batch repair algorithm to repair LCD-affected vertices $V_{\text{AFF+}}$ returned by Algorithm 6. To eliminate redundant r-labels, we need to track landmark distance, which requires the following notion of landmark distance bound.

Definition 13 (*Landmark distance bound*) Let $S \subset V \setminus \{r\}$ be a set of vertices. The *landmark distance bound* of vertex v w.r.t. S is:

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

Algorithm 6: Improved Batch Search

```
1  Function BatchSearch($G'$, $B$, $r$, $\Gamma$)
2      $\mathcal{Q}, \mathcal{Q}^+ \leftarrow$ InitQueues($B, r, \Gamma$)
3      while $\mathcal{Q} \cup \mathcal{Q}^+$ is not empty do
4          if minimal tuple in $\mathcal{Q} \cup \mathcal{Q}^+$ lies in $\mathcal{Q}$ then
5              remove minimal $(d_v, l_v, \text{stable}, v)$ from $\mathcal{Q}$
6          else
7              remove minimal $(d_v, l_v, \text{stable}, v, a)$ from $\mathcal{Q}^+$
8              if $a \in V_{\text{AFF+}}$ then
9                  continue
10         if $v \notin V_{\text{AFF+}}$ then
11             $\mathcal{N} \leftarrow \emptyset$
12             if $\neg l_v \wedge d_v = d_G(r, v)$ then
                   // check for path of equal (landmark) length
13                 foreach $w \in N_{G'}(v) \setminus V_{\text{UNSTABLE}}$ do
14                     $(d_w, l_w) \leftarrow$ GetLD($w$)
15                     if $d_v = d_w + 1$ then
16                         stable $\leftarrow$ True
17                         if $d^L_G(r, v) = (d_w, l_w) \oplus v$ then
18                             continue from line 3
19             foreach $w \in N_{G'}(v) \setminus V_{\text{AFF+}}$ do
                   // check if w might be LCD-affected
20                 if $d_v + 1 \leq d_G(r, w)$ then
21                     add $w$ to $\mathcal{N}$
22             add $v$ to $V_{\text{AFF+}}$
23             if stable then
24                 LD[$v$] $\leftarrow (d_v, l_v)$
25                 foreach $w \in \mathcal{N}$ do
26                     if $d^L_G(r, v) \oplus w = d^L_G(r, w) < (d_v, l_v) \oplus w$ or
                         $(d_v, l_v) \oplus w < d^L_G(r, w)$ then
27                         add $\big((d_v, l_v) \oplus w,\ \text{True},\ w\big)$ to $\mathcal{Q}$
28             else
29                 add $v$ to $V_{\text{UNSTABLE}}$
30                 foreach $w \in \mathcal{N}$ do
31                     if $d_v + 1 < d_G(r, w)$ or
                         $d^L_G(r, v) \oplus w = d^L_G(r, w)$ then
32                         add $(d_v + 1,\ \text{False},\ \text{False},\ w)$ to $\mathcal{Q}$
33     return $V_{\text{AFF+}}$
```

Algorithm 6: (Continued)

```
1  Function InitQueues($B$, $r$, $\Gamma$)
2      foreach inserted $(a, b) \in B$ do
3          if $d^L_G(r, a) \oplus b < d^L_G(r, b)$ then
4              add $\big(d^L_G(r, a) \oplus b,\ \text{True},\ b,\ a\big)$ to $\mathcal{Q}^+$
5          else if $d^L_G(r, b) \oplus a < d^L_G(r, a)$ then
6              add $\big(d^L_G(r, b) \oplus a,\ \text{True},\ a,\ b\big)$ to $\mathcal{Q}^+$
7      foreach deleted $(a, b) \in B$ do
8          if $d^L_G(r, a) \oplus b = d^L_G(r, b)$ then
9              add $\big(d_G(r, b),\ \text{False},\ \text{False},\ b\big)$ to $\mathcal{Q}$
10         else if $d^L_G(r, b) \oplus a = d^L_G(r, a)$ then
11             add $\big(d_G(r, a),\ \text{False},\ \text{False},\ a\big)$ to $\mathcal{Q}$
12     return $(\mathcal{Q}, \mathcal{Q}^+)$
13 Function GetLD($w$)
14     if $w \in V_{\text{AFF+}}$ then
15         return LD[$w$]
16     else
17         return $d^L_G(r, w)$
```

The following lemma allows us to compute the landmark distances of vertices in $V_{\text{AFF+}}$ from a landmark r in the changed graph G' using their landmark distance bounds.

Lemma 9 *Let $S \subset V\setminus\{r\}$ and $v \in S$ with minimal landmark distance bound. Then $d^L_{G'}(r, v) = d^L_{\text{BOU}}(v, S)$.*

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 $d^L_{\text{BOU}}(v', S) \leq d^L_{G'}(r, w) \oplus v = d^L_{G'}(r, v')$. If $v' \neq v$ then $d_{G'}(r, v') < d_{G'}(r, v) \leq d_{\text{BOU}}(v, S)$, and thus $d_{\text{BOU}}(v', S) < d_{\text{BOU}}(v, S)$. 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 again $d^L_{G'}(r, v) = d^L_{\text{BOU}}(r, S)$ does not generally hold for every boundary vertex v. Algorithm 7 shows the pseudo-code of our improved batch repair algorithm. For a graph G' and a set of all affected vertices $V_{\text{AFF+}}$, we first compute the landmark distance bounds of vertices in $V_{\text{AFF+}}$ using their unaffected neighbours. We then find vertices in $V_{\text{AFF+}}$ with minimal distance bounds and remove them from $V_{\text{AFF+}}$. By Lemma 9 their landmark distances to r in G' equal 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 neighbouring vertices in $V_{\text{AFF+}}$. We continue this process until $V_{\text{AFF+}}$ is empty.

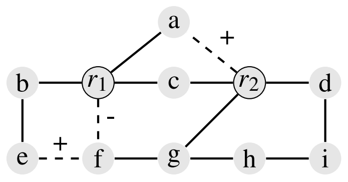
Algorithm 7: Improved Batch Repair

```
1  Function BatchRepair($G'$, $V_{AFF}$, $r_i$, $\Gamma$, $\Gamma'$)
2      foreach $v \in V_{AFF}$ do
3          $D_{BOU}[v] \leftarrow d^L_{BOU}(v, V_{AFF})$ // use $\Gamma$ to compute
4      while $V_{AFF}$ is not empty do
5          $V_{min} \leftarrow \{v \in V_{AFF} \mid D_{BOU}[v].d$ is minimal$\}$
6          remove $V_{min}$ from $V_{AFF}$
7          foreach $v \in V_{min}$ do
8              if $D_{BOU}[v].d = \infty \vee D_{BOU}[v].l$ then
9                  remove $r$-label from $\Gamma'(v)$
10             else
11                 set $r$-label of $\Gamma'(v)$ to $(r_i, D_{BOU}[v].d)$
12             if $v$ is a landmark then
13                 $\delta'_H(r_i, v) \leftarrow D_{BOU}[v].d$
14             foreach $w \in N_{G'}(v) \cap V_{AFF}$ do
15                 $D_{BOU}[w] \leftarrow \min(D_{BOU}[w], D_{BOU}[v] \oplus w)$
```

6.3 An illustrative example

The following example illustrates the individual steps which our improved algorithm BatchHL$^+$ runs through.

Example 10 Consider the following graph and updates:



The initial highway labelling $\Gamma = (H, L)$ looks 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$^+$ initializes Γ' as Γ and then runs BatchSearch (Algorithm 6) and BatchRepair (Algorithm 7) for both landmarks r_1 and r_2.

For landmark r_1, Algorithm 6 returns the following set of affected vertices:

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

Due to the inserted edge (a, r_2), the new paths from a to r_2, d and i 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 is also pruned. For vertex e, the stable path $r_1 - b - e$ ensures us that e is not LCD-affected.

For comparison, the BatchSearch of BatchHL described by Algorithm 2 would return the following larger set of vertices:

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

Note that, here, vertex e is not P-affected, but still returned due to the composite path $r_1 - f - e$.

Now, we run Algorithm 7 on $V_{AFF+} = \{f, g, h\}$. The initial landmark distance bounds for this set are

	f	g	h
$d^L_{BOU}(r_1, \ldots) =$	(3, False)	(3, True)	(5, True)

In the first iteration f and g have minimal distance bounds. Thus, the r_1-label in $L'(f)$ is updated to $(r_1, 3)$ and the r_1-label in $L'(g)$ is removed. After that, $d^L_{BOU}(r_1, h)$ is updated to (4, True) and the r_1-label in $L'(h)$ is removed. This leaves L' as

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

For landmark r_2, Algorithm 6 and Algorithm 2 return $V_{AFF+} = \{a, e\}$ and $V_{AFF+} = \{r_1, a, b, e\}$, respectively.

Then, running Algorithm 7 on $V_{AFF+} = \{a, e\}$ inserts $(r_2, 1)$ into $L'(a)$ and $(r_2, 2)$ into $L'(e)$ for the updated highway labelling as

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

7 Comparison of batch search algorithms

In this section, we provide a brief theoretical comparison between different batch search algorithms, including the algorithms used in BatchHL (Sect. 4), the algorithms of handling insertions and deletions separately (Sect. 5), and the improved algorithm used in BatchHL$^+$ (Sect. 6). Recall that, ideally, we would only want to return vertices that are LD-affected w.r.t. an entire batch update, but all these algorithms fail at that and return additional vertices.

Figure 2 summarizes the relationships between different concepts defined and vertex sets returned by these algorithms. Here, BatchHL was originally introduced in [18], BHL^{I+D} and BHL^{D+I} denote the sequential applications of insertions before deletions or deletions before insertions as described by Algorithms 4 and 5, and BHL$^{I+D}_{trick}$ sequential application with the trick described in Sect. 5, and BatchHL$^+$ refers to

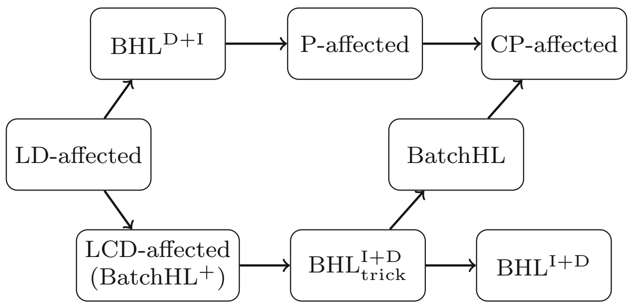


Fig. 2 Relationships between “affected” vertex sets defined and/or returned by different batch search algorithms. Each arrow indicates a subset containment

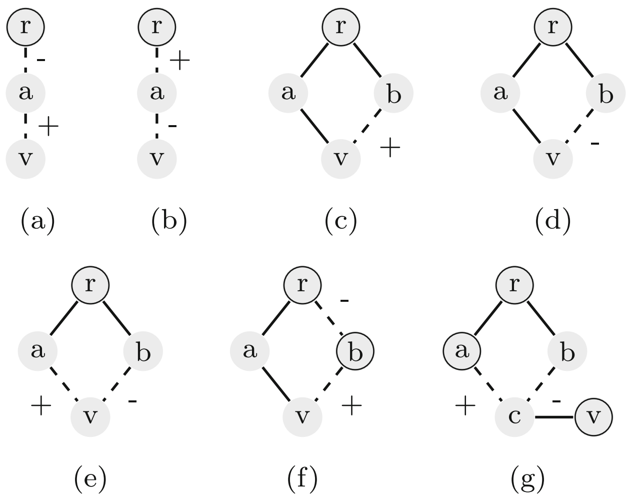


Fig. 3 Graphs illustrating differences between “affected” vertex sets defined and/or returned by batch search algorithms

Table 2 Containment of vertex v in Fig. 3 in the vertex sets of different batch search algorithms

	(a)	(b)	(c)	(d)	(e)	(f)	(g)
P-affected	No	No	Yes	Yes	Yes	No	Yes
LD-affected	No	No	No	No	No	No	No
CP-affected	Yes	No	Yes	Yes	Yes	Yes	Yes
LCD-affected (BatchHL$^+$)	Yes	No	No	No	No	No	No
BatchHL	Yes	No	No	Yes	Yes	Yes	No
BHL$^{\text{D+I}}$	No	No	No	No	Yes	No	Yes
BHL$^{\text{I+D}}$	Yes	Yes	No	No	No	Yes	No
BHL$^{\text{I+D}}_{\text{trick}}$	Yes	No	No	No	No	Yes	No

our improved algorithm as described by Algorithms 6 and 7 in Sect. 6.

Figure 3 illustrates cases that differentiate the vertex set definitions and algorithm outputs. For each of these cases, Table 2 lists whether vertex v can be identified by different “affected” notions and algorithms.

We note that case (g) in Fig. 3 is designed to trigger the corner case of Algorithm BatchHL, described in the proof of Lemma 5.18 in [18], where the search follows a path whose extended landmark length is not minimal. This corner case prevents BHL$^{\text{D+I}}$ from always returning a subset of BatchHL.

In the following, we denote by V_{I+D} the set of vertices returned by either Algorithm 4 or 5 if insertions are processed first, and by V_{D+I} if deletions are processed first. We note that V_{D+I} does not need to contain all LCD-affected vertices, as evidenced by case (a) of Example 7, where v is LCD-affected but not returned. However, it may also contain vertices that are not LCD-affected, as seen in case (c) of Example 7, where v is returned but it is not LCD-affected. Thus we can only compare it against Algorithm 6 experimentally. However, we can easily establish a relationship between V_{D+I} and the set of P-affected vertices as in Definition 4.

Lemma 10 *V_{D+I} contains only P-affected vertices.*

Proof If v is returned by Algorithm 5, then by Lemma 8 v is LD-affected w.r.t. deletions. Thus, a shortest path in G is eliminated, and v is P-affected.

If v is not returned by Algorithm 5 but returned by Algorithm 4, then by Lemma 7 v is LD-affected w.r.t. insertions into the intermediate graph G^-, the result of applying edge deletions to G. Thus the landmark distance between r and v is strictly smaller in G' than it is in G^-. Since v was not returned by Algorithm 5, the landmark distance between r and v in G is the same as in G^-. Thus, v is LD-affected and P-affected w.r.t. the combined batch update. □

For V_{I+D}, the situation is nonetheless different. As seen in case (b) of Example 7, V_{I+D} may contain vertices that are not LCD-affected. However, as we will show below, it will never return fewer.

Lemma 11 *V_{I+D} contains all LCD-affected vertices.*

Proof Let v be LCD-affected. Definition 12 requires

$$d_C(r,v) < d_G(r,v) \quad \text{or} \quad d^L_{G'}(r,v) \neq d^L_G(r,v)$$

If $d_C(r,v) < d_G(r,v)$ then v is LD-affected w.r.t. insertions only, and thus returned by Algorithm 4. If $d^L_{G'}(r,v) \neq d^L_G(r,v)$ then v is LD-affected w.r.t. the combined batch update, and thus returned. □

We note that Lemma 11 still holds if the final graph G' is passed to Algorithm 4 (the “trick” mentioned in Sect. 5). This can be seen by considering that for $d_C(r,v) < d_G(r,v)$ the algorithm will use the last inserted edge in a shortest composite path as a starting point, and by Definition 5 all subsequent edges must lie in G'. Example 8 shows that non-LCD-affected vertices may still be returned.

8 Theoretical discussion

8.1 Proof of correctness

Now we prove that Algorithm 6 returns precisely the set of LCD-affected vertices. As illustrated in Example 8, we are particularly interested in vertices for which the distance to r in G' is precisely the composite distance. In these cases we can compute the landmark distance to r in G', which enables us to determine with certainty whether or not its landmark distance changes.

Definition 14 (*Stable path and stable vertex*) We call a path from r to v *stable* iff it lies in G' and has length $d_C(r, v)$. We call v *stable* w.r.t. r iff such a stable path exists, i.e. iff $d_{G'}(r, v) = d_C(r, v)$.

As mentioned before, we can determine the updated landmark distance of stable vertices during batch search. For others we can only determine the composite distance. This motivates the following definition.

Definition 15 (*Composite landmark distance*) Given a composite path p, the *composite landmark length* of p is its landmark length if it lies in G', and $(|p|, \text{False})$ otherwise. Accordingly, the *composite landmark distance* of a vertex is defined as

$$d_C^L(r, v) := \begin{cases} d_{G'}^L(r, v) & \text{if } v \text{ is stable;} \\ (d_C(r, v), \text{False}) & \text{otherwise.} \end{cases}$$

Definition 16 (*Stable landmark distance*) The *stable landmark distance* additionally stores whether v is stable:

$$d_C^S(r, v) := \begin{cases} (d_{G'}^L(r, v), \text{True}) & \text{if } v \text{ is stable;} \\ (d_C(r, v), \text{False}, \text{False}) & \text{otherwise.} \end{cases}$$

The distance component of $d_C^L(r, v)$ is $d_C(r, v)$ in both cases, while the landmark flag indicates the existence of a stable path that passes through another landmark. When comparing stable landmark length values, we use lexicographical ordering with True $<$ False, as for landmark length. This is needed to ensure that stable paths are processed first, so that the stable variable is set correctly if the check in line 12 fails in Algorithm 6.

We first prove the following lemma.

Lemma 12 *A vertex v is LCD-affected iff*

(i) v is unstable, or
(ii) $d_C^L(r, v) \neq d_G^L(r, v)$

Proof We prove the "if" and "only if" below.

– (*if*) Let v be unstable, meaning $d_C(r, v) \neq d_{G'}(r, v)$. If $d_C(r, v) \neq d_G(r, v)$ then its composite distance changes. If $d_C(r, v) = d_G(r, v)$ then $d_{G'}(r, v) \neq d_G(r, v)$ and its landmark distance changes. Otherwise v must be stable and $d_C^L(r, v) \neq d_G^L(r, v)$. This means $d_{G'}^L(r, v) \neq d_G^L(r, v)$ and its landmark distance changes.
– (*only if*) Let v be LCD-affected but stable so that $d_C^L(r, v) = d_{G'}^L(r, v)$. If $d_{G'}^L(r, v) \neq d_G^L(r, v)$ then $d_C^L(r, v) \neq d_G^L(r, v)$ follows. Otherwise $d_C(r, v) \neq d_G(r, v)$ and $d_C^L(r, v) \neq d_G^L(r, v)$ follows.

□

Note that together with the landmark distance, the stable landmark distance provides precisely the information needed to check the conditions of Lemma 12.

The following terminology will be useful for identifying paths through which vertices are visited by Algorithm 6. In particular, this is restricted by the checks in lines 26 and 31 of Algorithm 6.

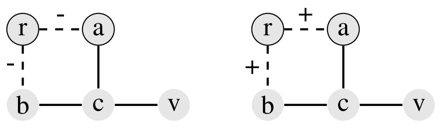
Definition 17 (*LCD-parent*) We call v an *LCD-parent* of w, and w an *LCD-child* of v, iff

(a) v and w are LCD-affected, and
(b) w is a neighbour of v in G', and
(c) $d_C^L(r, w) = \begin{cases} d_C^L(r, v) \oplus w & \text{if } v \text{ is stable,} \\ (d_C(r, v) + 1, \text{False}) & \text{otherwise, and} \end{cases}$
(d) $d_C(r, v) + 1 < d_G(r, w)$, or
$d_{G'}^L(r, v) \oplus w < d_G^L(r, w)$, or
$d_G^L(r, v) \oplus w = d_G^L(r, w) < d_{G'}^L(r, w)$.

We use the terms LCD-ancestor and LCD-descendant to denote the reflexive and transitive closures of the LCD-parent and LCD-child relationships.

Note that conditions (2) and (3) mean that v is the predecessor of w on a path from r to w of minimal composite landmark length. The conditions of (4) correspond precisely to those of Definition 12: either a path through v reduces the composite distance to w, or it reduces the landmark distance to w, or the landmark distance of w increases due to elimination of a shortest path passing through v.

Example 11 Consider the following graphs and updates.



For the graph on the left, the LCD-ancestors of v are a, c and v itself. Here, b is not an LCD-parent of c as the conditions of (4) are violated: $d_G^L(r, c) < d_{G'}^L(r, c)$ but $d_G^L(r, b) \oplus c \neq d_G^L(r, c)$. For the graph on the right, the LCD-ancestors of v

are again a, c and v itself. b is not an LCD-parent of c as the conditions of (3) are violated.

The following lemma shows that all LCD-affected vertices can be traced back to a vertex initially enqueued in Algorithm 6.

Lemma 13 *Let b be LCD-affected with no LCD-parent. Then, there exists a vertex a such that either*

(a) *edge (a, b) is inserted, $d^L_G(r, a) \oplus b = d^L_C(r, b) < d^L_G(r, b)$, and a is not LCD-affected, or*
(b) *edge (a, b) is deleted, $d^L_G(r, a) \oplus b = d^L_G(r, b)$, and $(d_G(r, b), \text{False}) = d^L_C(r, b)$.*

Proof Since b is LCD-affected, Definition 12 requires

- $d_C(r, b) < d_G(r, b)$ or
- $d^L_{G'}(r, b) < d^L_G(r, b)$ or
- $d^L_{G'}(r, b) > d^L_G(r, b)$.

As b has no LCD-parents, every vertex a must violate one of the conditions (1)-(4) of Definition 17.

(1) Assume $d_C(r, b) < d_G(r, b)$. Let p be a composite path from r to b of minimal composite landmark length, and a the predecessor of b in p. Then, $d_C(r, a) + 1 = d_C(r, b)$, so condition (4) holds. If a is stable, then $d^L_C(r, a) \oplus b = d^L_C(r, b)$; otherwise, the landmark flag of $d^L_C(r, b)$ is False by Definition 15. Thus, condition (3) holds in both cases. Condition (2) must hold as well—otherwise the last edge (a, b) in the composite path p is a deleted edge, which implies that p lies in G, contradicting $d_C(r, b) < d_G(r, b)$. This only leaves condition (1) to be violated, so a cannot be LCD-affected. Thus, $d^L_G(r, a) \oplus b = d^L_C(r, a) \oplus b = d^L_C(r, b)$, and therefore $d_G(r, a) + 1 < d_G(r, b)$. It follows that (a, b) must be inserted and $d^L_G(r, a) \oplus b = d^L_C(r, b) < d^L_G(r, b)$ holds.

(2) Assume $d_C(r, b) = d_G(r, b)$ and $d^L_{G'}(r, b) < d^L_G(r, b)$. Then, $d_C(r, b) \le d_{G'}(r, b) \le d_G(r, b) = d_C(r, b)$ so b is stable. Let p be a stable path from r to b of minimal landmark length, and a the predecessor of b in p. Then, conditions (2) and (3) clearly hold. Since $d^L_{G'}(r, a) \oplus b = d^L_{G'}(r, b) < d^L_G(r, b)$ condition (4) holds as well, so condition (1) must be violated and a cannot be LCD-affected. Thus, $d^L_G(r, a) \oplus b = d^L_C(r, a) \oplus b = d^L_C(r, b) < d^L_G(r, b)$, implying that (a, b) is inserted.

(3) Assume $d_C(r, b) = d_G(r, b)$ and $d^L_{G'}(r, b) > d^L_G(r, b)$. If b is unstable we have $d^L_C(r, b) = (d_C(r, b), \text{False}) = (d_G(r, b), \text{False})$ by Definition 15. Otherwise $d_G(r, b) = d_C(r, b) = d_{G'}(r, b)$, so $d^L_{G'}(r, b) > d^L_G(r, b)$ is only possible if $d^L_{G'}(r, b) = (d_G(r, b), \text{False})$. Thus $d^L_C(r, b) = d^L_{G'}(r, b) = (d_G(r, b), \text{False})$, so the last condition in case (b) of Lemma 13 holds in both cases.

Let p be a path from r to b in G of minimal landmark length, and a the predecessor of b in p. This implies $d^L_G(r, a) \oplus b = d^L_G(r, b)$, so it only remains to show that (a, b) is deleted. Assume to the contrary that (a, b) lies in G', so condition (2) holds immediately. Since p has length $d_C(r, b)$ we have $d_C(r, b) = d_C(r, a) + 1$. Together with earlier results this gives us $d^L_C(r, b) = (d_C(r, b), \text{False}) = (d_C(r, a)+1, \text{False})$. Thus condition (3) holds if a is unstable. If a is stable, then $d^L_C(r, b) \le d^L_C(r, a) \oplus b$. This is only possible if the landmark flag of $d^L_C(r, a) \oplus b$ is False, so condition (3) holds again. Finally $d^L_G(r, a) \oplus b = d^L_G(r, b) < d^L_{G'}(r, b)$ so condition (4) holds as well. Thus, condition (1) must be violated, so a cannot be LCD-affected. It follows that $d^L_{G'}(r, a) = d^L_G(r, a)$, and thus

$$d^L_{G'}(r, b) \le d^L_{G'}(r, a) \oplus b = d^L_G(r, a) \oplus b = d^L_G(r, b)$$

violating $d^L_{G'}(r, b) > d^L_G(r, b)$, a contradiction. □

The next lemma shows that every LCD-child meets the pruning conditions in lines 20, 26 and 31 of Algorithm 6.

Lemma 14 *Let w be an LCD-child of v. Then,*

(i) $d_C(r, v) + 1 \le d_G(r, w)$
(ii) *if v is stable, then $d^L_C(r, v) \oplus w < d^L_G(r, w)$ or $d^L_G(r, v) \oplus w = d^L_G(r, w) < d^L_C(r, v) \oplus w$*
(iii) *if v is unstable, then $d_C(r, v) + 1 < d_G(r, w)$ or $d^L_G(r, v) \oplus w = d^L_G(r, w)$*

Proof Consider the cases in condition (4) of Definition 17. If $d_C(r, v) + 1 < d_G(r, w)$ then conditions (1), (2) and (3) follow immediately.

If $d^L_{G'}(r, v) \oplus w < d^L_G(r, w)$ then $d_{G'}(r, v) + 1 \le d_G(r, w)$, and condition (1) follows from $d_C(r, v) \le d_{G'}(r, v)$. If v is stable, then $d^L_C(r, v) = d^L_{G'}(r, v)$ and condition (2) follows. If v is unstable, then $d_C(r, v) < d_{G'}(r, v)$ and thus $d_C(r, v) + 1 < d_{G'}(r, v) + 1 \le d_G(r, w)$, showing condition (3).

If $d^L_G(r, v) \oplus w = d^L_G(r, w) < d^L_{G'}(r, w)$ then condition (3) follows immediately, and condition (1) from $d_C(r, v) + 1 \le d_G(r, v) + 1 = d_G(r, w)$. For condition (2) stability of v gives us

$$\begin{aligned} d^L_G(r, v) \oplus w = d^L_G(r, w) &< d^L_{G'}(r, w) \\ &\le d^L_{G'}(r, v) \oplus w = d^L_C(r, v) \oplus w \end{aligned}$$

showing condition (2). □

We are now ready to prove our main result.

Theorem 1 *Algorithm 6 returns the set of all LCD-affected vertices.*

Proof We prove it by showing the following:

(a) For each $(d, l, s, v) \in \mathcal{Q}$, we have

$$d_C^L(r, v) \leq (d, l) \quad \text{and} \quad d \leq d_G(r, v).$$

If $l =$ True, then $s =$ True and $(d, l) < d_G^L(r, v)$.

(b) From line 10 onwards, we have

$$d_C^L(r, v) \leq (d_v, l_v) \quad \text{and} \quad d_v \leq d_G(r, v).$$

If $l_v =$ True, stable $=$ True and $(d_v, l_v) < d_G^L(r, v)$.

(c) From line 11, we have $d_v = d_C(r, v)$.

(d) In line 14 either $d_w \geq d_C(r, w) \geq d_v$, or $(d_w, l_w) = d_C^L(r, w)$ and w is stable.

(e) If line 18 is reached, then v is not LCD-affected.

(f) If line 19 is reached, then v is LCD-affected.

(g) From line 19, we have $(d_v, l_v, stable) = d_C^S(r, v)$.

(h) The **while** loop has the following invariant:
If v is LCD-affected with $v \notin V_{\text{AFF+}}$, then $\mathcal{Q} \cup \mathcal{Q}^+$ contains a tuple $(d, l, \ldots)$ with $(d, l) \leq d_C^L(r, v)$.

Once shown, (h) guarantees that every LCD-affected vertex is included in $V_{\text{AFF+}}$, while (f) ensures that these are the only ones. By considering only the first time any condition would be violated, we may assume they all hold for earlier steps.

Case (a): For insertions prior to the while loop, the conditions clearly hold. Let (d, l, s, w) be the first tuple inserted during the while loop that violates the condition, and consider the iteration during which this insertion occurred. By (g) we have $(d_v, l_v) = d_C^L(r, v)$, and the check in line 23 succeeds iff v is stable.

If v is stable, then insertion occurs in line 27. Thus,

$$\begin{aligned}(d, l) &= (d_v, l_v) \oplus w = d_C^L(r, v) \oplus w \\ &= d_{G'}(r, v) \oplus w \geq d_{G'}(r, w) \geq d_C^L(r, w).\end{aligned}$$

Otherwise, insertion occurs in line 32, and we have

$$\begin{aligned}(d, l) &= (d_v + 1, \text{False}) = (d_C(r, v) + 1, \text{False}) \\ &\geq (d_C(r, w), \text{False}) \geq d_C^L(r, w).\end{aligned}$$

This shows the first inequality. The check in line 20 ensures the second inequality $d \leq d_G(r, w)$.

Finally, let $l =$ True. Then, (d, l, s, w) must have been inserted in line 27. Thus, we have $(d, l) = (d_v, l_v) \oplus w$, $s =$ True and either $(d_v, l_v) \oplus w < d_G^L(r, w)$ or $d_G^L(r, w) < (d_v, l_v) \oplus w$. In the first case $(d, l) < d_G^L(r, w)$ follows immediately. Otherwise, we have $d_G^L(r, w) < (d, l)$ and $d_G(r, w) \leq d$, which together with the second inequality implies $d = d_G(r, w)$. Thus, $d_G^L(r, w) < (d, l)$ is only possible for $l =$ False, a contradiction.

Case (b): If $(d_v, l_v, \ldots)$ comes from $\mathcal{Q}$, the claim follows from (a). Note that variable stable will never be updated to False. Otherwise, $(d_v, l_v, \text{stable}, v, a)$ comes from $\mathcal{Q}^+$. Then, stable is True, (a, v) lies in G', and $(d_v, l_v) = d_G^L(r, a) \oplus v < d_G^L(r, v)$. Thus, $d_v \leq d_G(r, v)$ holds. Since the check in line 8 succeeded $a \notin V_{\text{AFF+}}$. By (h) and minimality of $d_v > d_G(r, a) \geq d_C(r, a)$, it follows that a is not LCD-affected. Thus $d_C^L(r, v) \leq d_C^L(r, a) \oplus v = (d_v, lv)$.

Case (c): The check in line 10 must have succeeded, so $v \notin V_{\text{AFF+}}$. If v is LCD-affected, then $(d_v, l_v) \leq d_C^L(r, v)$ by (h) and the claim follows from (b). If v is not LCD-affected, then $d_C^L(r, v) = d_G^L(r, v)$ by Lemma 12 and the claim follows from (b).

Case (d): If $w \in V_{\text{AFF+}}$, then $(d_w, l_w) = LD[w] = d_C^L(r, w)$ by (g), and w is stable by (g); thus, (d) holds. Otherwise, $(d_w, l_w) = d_G^L(r, w)$; thus, $d_w = d_G(r, w) \geq d_C(r, w)$. If $d_C(r, w) \geq d_v$, then (d) holds. Let $d_C(r, w) < d_v$. Since d_v was minimal in $\mathcal{Q} \cup \mathcal{Q}^+$ and $w \notin V_{\text{AFF+}}$, w cannot be LCD-affected by (h). By Lemma 12 w is stable and $d_C(r, w) = d_G^L(r, w) = (d_w, l_w)$, so (d) holds.

Case (e): Assume to the contrary that v is LCD-affected. Then, (h) implies $(d_v, l_v) \leq d_C^L(r, v)$ as (d_v, l_v) is minimal in $\mathcal{Q} \cup \mathcal{Q}^+$, and thus $(d_v, l_v) = d_C^L(r, v)$ by (b). Denote by w the neighbour of v for which line 18 is reached. The checks in lines 12 and 15 ensure that $d_G(r, v) = d_w + 1$. Thus, w is stable and $(d_w, l_w) = d_C^L(r, w)$ by (d). This implies $d_C(r, v) = d_C(r, w) + 1$, so v is stable. The check in line 17 ensures that

$$d_G^L(r, v) = d_C^L(r, w) \oplus v \geq d_C^L(r, v).$$

Hence, either $d_C^L(r, v) < d_G^L(r, v)$ or v is not LCD-affected by Lemma 12. Assume the former. Since $d_G(r, v) = d_v = d_C(r, v)$, this requires $d_C(r, v) = (d_v, \text{True})$. But that means $l_v =$ True and the check in line 12 would have failed.

Case (f): Assume to the contrary that v is not LCD-affected. We will show that execution would have reached line 18 first.

If $l_v =$ True, we would have $d_C^L(r, v) \leq (d_v, l_v) < d_G^L(r, v)$ by (b) which contradicts Lemma 12, and thus $l_v =$ False. By (c) and Lemma 12, we have $d_v = d_C(r, v) = d_G(r, v)$, and the check in line 12 succeeds.

By Lemma 12, v is stable. Hence, there must exist a stable path p from r to v of landmark length $d_C^L(r, v)$. Let w be the predecessor of v in p so that w is stable and $d_C^L(r, w) \oplus v = d_C^L(r, v)$. From (g) it follows that V_{UNSTABLE} contains precisely the unstable vertices in $V_{\text{AFF+}}$. Hence, w lies in $N_{G'}(v) \setminus V_{\text{UNSTABLE}}$.

By (d) and $d_C(r, w) < d_C(r, v) = d_v$, we have $(d_w, l_w) = d_C^L(r, w)$. It follows that $d_v = d_C(r, v) = d_C(r, w) + 1 = d_w + 1$ and $d_G^L(r, v) = d_C^L(r, v) = d_C^L(r, w) \oplus v = (d_w, l_w) \oplus v$. Thus, w passes all checks required to reach line 18, and line 19 is not reached.

Case (g): By (f), v is LCD-affected. Hence, (h) implies $(d_v, l_v) \leq d^L_C(r, v)$, and $(d_v, l_v) = d^L_C(r, v)$ by (b). It remains to show that variable stable is True iff v is stable.

If t_v came from $\mathcal{Q}^+$ with anchor a, then stable is True and $d_C(r, a) \leq d_G(r, a) < d_G(r, a) + 1 = d_v$. Since the check in line 8 must have succeeded, we have $a \notin V_{\text{AFF+}}$. By (h) it follows that a is not LCD-affected. Since $d_C(r, v) = d_v = d_G(r, a) + 1 = d_C(r, a) + 1$ we may conclude that v is stable.

Let t_v have come from $\mathcal{Q}$. If stable was initially True, then the tuple must have been inserted in line 27 while examining a parent vertex u. As (g) holds for all earlier iterations, u must have been stable with $d_C(r, u) + 1 = d_v = d_C(r, v)$, and v is stable. So let stable have been initially False. l_v = False by (a). We distinguish two cases. (i) If $d_v = d_G(r, v)$ then the check in line 12 succeeds. By Definition 15 v is stable iff v has a stable neighbour w in G' with $d_C(r, w) + 1 = d_C(r, v)$. By (d) line 16 is reached iff such a neighbour exists, so the claim holds. (ii) If $d_v \neq d_G(r, v)$, then the check in line 12 fails. Thus, stable will still be false by the time when line 19 is reached. Assume to the contrary that v is stable. Then, it must have a stable neighbour a in G' with $d_C(r, a) + 1 = d_C(r, v) = d_v < d_G(r, v)$.

If a is not LCD-affected, then (a, v) must have been inserted. Hence, a tuple $t_a = (d_C(r, v), l, \text{True}, v, a)$ must have been inserted into $\mathcal{Q}^+$. Minimality of t_v in $\mathcal{Q} \cup \mathcal{Q}^+$ means that t_a must have been removed from $\mathcal{Q}^+$ earlier. At this point the checks in lines 8 and 12 would have failed, causing v to be added to $V_{\text{AFF+}}$, a contradiction.

If a is LCD-affected, then $a \in V_{\text{AFF+}}$ by (h) and minimality of d_v in $\mathcal{Q} \cup \mathcal{Q}^+$. Since a is stable, a tuple $t_a = (d_C(r, v), l, \text{True}, v)$ must have been inserted in $\mathcal{Q}$ in line 27 when processing a. Minimality of $(d_v, \text{False}, \text{False})$ in $\mathcal{Q} \cup \mathcal{Q}^+$ means that t_a must have been removed from $\mathcal{Q}$ earlier. At this point the check in line 12 would have failed, causing v to be added to $V_{\text{AFF+}}$, a contradiction.

Case (h): We will show a stronger invariant, namely that if v is LCD-affected with $v \notin V_{\text{AFF+}}$ then either

- ($\mathcal{Q}$) $\mathcal{Q}$ contains a tuple (d, l, s, b) with $(d, l) \leq d^L_C(r, v)$ where $b \notin V_{\text{AFF+}}$ is an LCD-ancestor of v, or
- ($\mathcal{Q}^+$) $\mathcal{Q}^+$ contains a tuple (d, l, s, b, a) with $(d, l) \leq d^L_C(r, v)$ where $b \notin V_{\text{AFF+}}$ is an LCD-ancestor of v and a is not LCD-affected.

For every LCD-affected vertex v, there exists an LCD-ancestor of v that has no LCD-parent, possibly v itself. Thus InitQueues will add a tuple satisfying either ($\mathcal{Q}$) or ($\mathcal{Q}^+$) by Lemma 13, so the invariant holds initially.

Consider now an iteration of the while loop where v is LCD-affected with $v \notin V_{\text{AFF+}}$. We need to show that the invariant is maintained for its LCD-descendants. Denote by t_v the tuple removed from $\mathcal{Q} \cup \mathcal{Q}^+$. If $t_v \in \mathcal{Q}^+$, then by (f) $a \notin V_{\text{AFF+}}$, and the check in line 8 fails. Thus, execution reaches line 10 in both cases, and the check here passes by assumption. By (e) line 18 is never reached, and execution reaches line 22 where v is added to $V_{\text{AFF+}}$. Hence, the invariant is maintained for v.

Finally, let w be an LCD-child of v. By Lemma 14 and (g), the checks in lines 20 and 26 (if v is stable) or 31 (otherwise) succeed. Thus, a tuple (d_w, l_w, s, w) is added to $\mathcal{Q}$ in line 27 or 32. By condition (3) of Definition 17, we have $(d_w, l_w) = d^L_C(r, w)$, and the invariant is maintained for w and its LCD-descendants. □

8.2 Proof of minimality

Theorem 2 *The highway labelling Γ' that BatchHL$^+$ returns is the minimal highway labelling for G'.*

Proof By Lemma 6 and Theorem 1, the vertex set $V_{\text{AFF+}}$ returned by Algorithm 6 contains all LD-affected vertices. By Lemma 6, for vertices outside of $V_{\text{AFF+}}$ the landmark distance to r_i does not change. In line 3 of Algorithm 7 the value of $d^L_{\text{BOU}}(v, V_{\text{AFF+}})$ can be computed from Γ. From Lemma 9, 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 LD-affected vertex v 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, these updates are correct. As the r-labels of vertices outside of $V_{\text{AFF+}}$ 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 labelling of Γ' is correct and minimal. Highway is updated for vertices in $V_{\text{AFF+}}$ for all $r \in R$, and do not change for others by Definition 9. □

8.3 Complexity analysis

We analyse the time and space complexity of BatchHL$^+$. Let a be the total number of affected vertices, l be the maximum label size, and d be the maximum degree. We perform $|R|$ BFSs to construct highway labelling in $O(|R| \cdot |V|)$ time and space. The time complexity of Algorithm 6 is $O(a \cdot d \cdot l)$, where it visits $O(a)$ vertices and for each affected vertex performs d queries to check its affected neighbours in $O(d \cdot l)$ time. Note that a in Algorithm 6 refers to the total number of LCD-affected vertices which is much smaller than the total number of CP-affected vertices being referred to as a by Algorithm 2 of BatchHL. 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. Further, in contrast to Algorithm 3 of BatchHL which repairs CP-affected vertices returned by Algorithm 2, Algorithm 7 of BatchHL$^+$ repairs LCD-affected vertices returned by Algorithm 6. In the worse case, both Algorithm 3 and Algorithm 7 could repair the labels of all affected vertices. When repairing the label of an affected vertex, we check its neighbours in $O(d)$. Thus, the time complexity of Algorithm 7 is $(a \cdot d)$, and the overall time complexity of BatchHL$^+$ is $O(|R| \cdot a \cdot d \cdot l)$ using $O(V)$ space. We omit l from the time complexity of Algorithms 3 and 7 because we store distances for all unaffected neighbours of affected vertices in Algorithms 2 and 6.

9 Implementation optimizations

When implementing Algorithm 6, landmark length and stable landmark length values can be encoded as single integers to save storage space and speed up computation. Further, storing computed landmark distances in G for later reuse can also speed up processing.

For undirected graphs, the **for** loop in line 19 can be integrated into the **for** loop in line 13 if the latter is reached. For directed graphs, it can be eliminated altogether, in which case the **for** loops in lines 25 and 30 iterate over the outgoing neighbours.

Case (g) in Theorem 1 ensures that $\text{LD}[v] = d^L_{G'}(r, v)$ for all stable LCD-affected vertices. Hence the highway labels for these can be updated directly, and batch repair can be restricted to unstable vertices. Eliminating duplicate values in $\mathcal{Q}$ or checking whether a value has already been processed in line 10 stops vertices from getting processed multiple times (whereas checking containment in $V_{\text{AFF+}}$ only prevents this for LCD-affected vertices). Case (e) shows that this will not cause any LCD-affected vertices to be missed. However, the overhead of tracking visited vertices may not be worth it in situations where repeated visits are rare.

10 Experiments

We have implemented our proposed method BatchHL$^+$ to answer the following questions:

(1) How efficiently can BatchHL$^+$ perform in comparison with the state-of-the-art dynamic algorithms for answering distance queries?
(2) How does the number of landmarks affect the performance of BatchHL$^+$?
(3) How does the number of affected vertices effect the performance of BatchHL$^+$ in comparison with the state-of-the-art dynamic algorithms?
(4) What is the effect of the size of batch updates on the performance of BatchHL$^+$?

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

Baseline methods. We compare our proposed method BatchHL$^+$ with the following state-of-the-art methods.

– BatchHL [18]: A batch-dynamic method which performs graph changes in the batch-update setting to efficiently maintain a highway cover distance labelling. Then, it combines the updated labelling with graph traversal algorithm to answer distance queries. We compare our proposed method with the optimised version of BatchHL in our experiments.
– FulHL [20]: A fully dynamic method which presents two algorithms IncHL and DecHL to maintain a highway cover distance labelling as a result of graph changes in the single-update setting. Then, it combines the updated labelling with graph traversal algorithm to answer distance queries.
– FulFD [23]: A fully dynamic method that incorporates two algorithms IncFD and DecFD to maintain a distance labelling in the single-update setting as a result of graph changes. Then, it combines the updated distance labelling with a graph traversal algorithm to answer distance queries.
– PSL* [27]: A parallel algorithm which constructs pruned landmark labelling for static graphs to answer distance queries.
– BiBFS [23]: An online bidirectional BFS algorithm which answers distance queries using an optimized strategy to expand searches from the direction with fewer vertices.

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*.
Datasets. We use 15 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 3. These datasets are accessible at Stanford Network Analysis Project [26], Laboratory for Web Algorithmics [9], and the Koblenz Network Collection [25].
Test data generation. For batch-dynamic algorithms, we generate 10 batches of updates for all the 13 datasets, where each batch update contains 1000 edges that are randomly selected from E. We consider three batch update settings for testing:

(1) *Decremental*—delete these batches of updates and measure the average deletion time;

Table 3 Datasets, where $size(G)$ denotes the size of a graph G with each edge appearing in the forward and reverse adjacency lists and being represented by 8 bytes

Dataset	Network	n	m	m/n	Avg. deg	Max. deg	Avg. dist	$size(G)$
Youtube	social (u)	1.1M	3 M	2.63	5.265	28754	5.3	23 MB
Skitter	comp (u)	1.7M	11 M	6.54	13.08	35455	5.0	85 MB
Flickr	social (u)	1.7M	16 M	9.07	18.13	27224	5.3	119 MB
Wikitalk	comm (d)	2.4M	5 M	1.95	3.890	100029	3.9	36 MB
Hollywood	social (u)	1.1M	114 M	49.5	98.91	11467	3.9	430 MB
Orkut	social (u)	3.1M	117 M	38.1	76.28	33313	4.2	894 MB
Enwiki	social (d)	4.2M	101 M	21.9	43.75	432260	3.4	701 MB
Livejournal	social (d)	4.8M	69 M	8.84	17.68	20333	5.6	327 MB
Indochina	web (d)	7.4M	194 M	20.4	40.73	256425	7.7	1.1 GB
IT	web (d)	41 M	1.2B	24.9	49.77	1326744	7.0	7.7 GB
Twitter	social (d)	42 M	1.5B	28.9	57.74	2997487	3.6	9.0 GB
Friendster	social (u)	66 M	1.8B	27.4	55.06	5214	5.0	13 GB
UK	web (d)	106 M	3.7B	31.4	62.77	979738	6.9	25 GB
Italianwiki	web (d)	1.2M	35 M	16.6	33.25	81090	3.5	153 MB
Frenchwiki	web (d)	2.2M	59 M	13.2	26.36	137021	3.9	223 MB

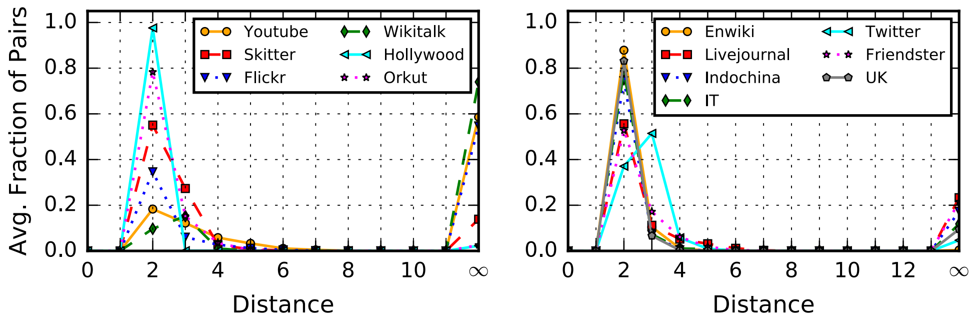


Fig. 4 Distance distribution of batch updates

(2) *Incremental*—add these batches of updates after decremental updates and measure the average insertion time;
(3) *Fully dynamic*—randomly select 50% updates from each of 10 batches of decremental updates as insertions and the remaining 50% as deletion. We report the average update time.

For the two datasets that are real-world dynamic networks, we select 10 batches in the order of their timestamps, each containing 1000 real-world inserted/deleted edges and measure the average update time after applying them in a streaming fashion.

For the methods FULHL and FULFD, 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 effect.

Figure 4 shows the distance distribution of edges in these batches after deleting. 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. Only a small number of updates are disconnected 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 labelling size produced by our batch-dynamic method BHL^{++} after performing fully dynamic batch updates.

11 Experimental results

11.1 Performance comparison

We compare our methods against the baseline methods in terms of update time, labelling size and query time.

11.1.1 Update time

Table 4 presents the average update time in the fully dynamic, incremental, and decremental settings of our proposed method and the baseline methods.

Table 4 Comparing update time of our methods BatchHL$^+$ and BHL^D with the state-of-the-art dynamic methods

Dataset	Fully Dynamic Batch Update Time [s]				Incremental Batch Update Time [s]				Decremental Batch Update Time [s]				
	BatchHL$^+$	BatchHL	FulHL	FulFD	BatchHL$^+$	BatchHL	IncHL	IncFD	BatchHL$^+$	BHL^D	BatchHL	DecHL	DecFD
Youtube	0.010	0.070	0.078	1.250	0.009	0.008	0.113	0.155	0.010	0.009	0.169	0.032	3.181
Skitter	0.009	0.601	0.914	5.968	0.007	0.006	1.347	0.117	0.012	0.010	0.751	0.551	14.15
Flickr	0.011	0.026	0.059	2.153	0.010	0.008	0.052	0.054	0.012	0.010	0.041	0.066	3.365
Wikitalk	0.006	0.025	0.049	2.927	0.006	0.005	0.052	0.030	0.005	0.005	0.044	0.055	5.674
Hollywood	0.005	0.014	0.058	4.424	0.003	0.002	0.055	0.091	0.007	0.006	0.031	0.062	8.401
Orkut	0.024	1.775	2.900	13.30	0.012	0.014	4.087	0.368	0.034	0.031	0.035	1.274	23.95
Enwiki	0.021	1.681	8.401	121.8	0.010	0.012	10.28	0.317	0.026	0.025	3.079	6.275	251.3
Livejournal	0.015	0.306	0.383	4.736	0.011	0.010	0.505	0.244	0.020	0.019	0.570	0.331	6.816
Indochina	0.016	1.181	2.085	20.63	0.011	0.011	3.628	0.142	0.018	0.016	1.346	0.810	44.93
IT	0.026	4.502	6.471	129.6	0.020	0.033	10.23	0.147	0.035	0.027	4.699	3.238	285.6
Twitter	0.014	49.62	208.1	5103	0.005	0.024	208.4	0.264	0.021	0.020	68.85	207.4	9460
Friendster	0.014	0.410	0.461	23.28	0.007	0.035	0.422	0.255	0.024	0.022	0.738	0.482	30.38
UK	0.027	41.46	13.88	110.1	0.017	0.055	26.76	0.258	0.033	0.030	42.29	1.686	257.4
Italianwiki	0.018	0.163	0.623	6.623	–	–	–	–	–	–	–	–	–
Frenchwiki	0.006	0.189	0.965	5.289	–	–	–	–	–	–	–	–	–

The batch size is 1000, and thus, the update time reported for every method is for 1000 updates. [s] is an abbreviation of second

Fully dynamic setting. Table 4 shows that BatchHL$^+$ significantly outperforms BatchHL. We observe that BatchHL$^+$ is up to 3 orders of magnitude faster than BatchHL on large datasets. BatchHL$^+$ also significantly outperforms FULHL and FULFD on all datasets. In particular, BatchHL$^+$ is over 15 times faster than FULHL on most of the datasets and several orders of magnitude faster than FULFD. Further, the performance difference of BatchHL$^+$ and BatchHL is due to the fact that our improved batch search in BatchHL$^+$ further prunes affected vertices that do not need a repair in their labels, and in practice they are significant in amount as can be seen in Table 6. Our method BatchHL$^+$ is also much faster than BatchHL, FULHL and FULFD on the real-world dynamic networks: Italianwiki and Frenchwiki. We can observe that the average update time of BatchHL$^+$ is always by far smaller than recomputing labelling from scratch, i.e. construction time of BatchHL$^+$ in Table 5. Further, the construction time of BatchHL$^+$ is significantly smaller than the construction of the baseline methods on all datasets. We can also see that the parallel variant of PLL (PSL*) still failed to construct labelling for the largest three datasets.

Incremental setting. Table 4 also shows that BatchHL$^+$ is comparable with BatchHL because of no technical difference in the batch search approach of both methods. On the other hand, it significantly outperforms INCHL and INCFD. This is because INCHL and INCFD involve in repeated and unnecessary computations while performing updates and require extra usage of resource for each single update.

Decremental setting. In Table 4, we can also see that BatchHL$^+$ is much faster than BatchHL in this setting. This is because it further categorises and prunes out affected vertices whose labels do not need to be changed. In contrary, BatchHL failed to recognise such vertices. The difference in the amount of affected vertices repaired by BatchHL$^+$ and BatchHL is shown in Table 6. It is clear that BatchHL$^+$ has to repair much smaller fraction of affected vertices in the decremental setting. Furthermore, BatchHL$^+$ significantly outperforms DECHL and DECFD on all the datasets. Especially, BatchHL$^+$ can achieve outstanding performance on networks which have a high average degree such as Twitter, Flickr and Hollywood. Due to inherent complexity of edge deletion on graphs (i.e. increasing distances), DECHL and DECFD take very long in identifying and updating labels of affected vertices. BatchHL$^+$ outperforms these methods because it leverages the benefit of handling updates in a batch and significantly reduce repeated computations during identifying and repairing the labels of affected vertices.

11.1.2 Labelling size

Table 5 shows that BatchHL$^+$ yields significantly smaller labelling sizes than FULFD and PSL* on all the datasets. Note that BatchHL$^+$, BatchHL and FULHL produce labelling of equal sizes after performing update operations because they are designed using highway cover property. When an update occurs, the labelling size of FULFD remains unchanged because they store complete shortest-path trees at all times. In contrast, BatchHL$^+$ similar to state-of-the-art BatchHL and FULHL stores pruned shortest-path trees based on highway cover property. Nonetheless, the labelling size of BatchHL$^+$ remains stable in practice because the average label size is bounded by a constant, i.e. the number of landmarks. The parallel variant of PLL (PSL*) which exploit PLL properties to reduce labelling size still produces labelling of very large size as compared to BatchHL$^+$.

11.1.3 Query time

Table 5 shows that the average query time of BatchHL$^+$ is comparable with FULFD. It was reported [12] that the average query time is largely dependent on labelling size. Since the dynamic operations do not considerably affect the labelling size for BatchHL$^+$ and FULFD, their query times remain stable. On Twitter, the query time of BatchHL$^+$ underperforms FULFD. This is because FULFD also maintains the shortest-path information for the neighbourhood of landmarks and the maximum degree of Twitter is very high which might cause many pairs to be covered by landmarks.

Although the query time of PSL* in Table 5 is better than BatchHL$^+$ on some datasets, it only handles static graphs. For dynamic graphs, it has several limitations including: (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 5, (2) the labelling size is much larger than BatchHL$^+$. As we can see in Table 5, PSL* produces the labelling of size almost 99% larger than the labelling of BatchHL$^+$ 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, BatchHL$^+$ stands out in claiming the best trade-off between these factors.

11.2 Performance under varying landmarks

Figure 8 and Fig. 9 show how the update time of our method BatchHL$^+$ in the fully dynamic setting behaves when increasing the number of landmarks. We can see that the update time in Fig. 8 for almost all datasets grows to 30 landmarks and then either decreases or grows linearly. This is because selecting a larger number of landmarks can better leverage the pruning power of our method. Figure 9 shows 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 decreases because of a very high average degree and selecting a larger number of high

Table 5 Comparing performance of our method BatchHL$^+$ with the baseline methods FulFD, FulPLL and PSL* in terms of construction time (CT), query time (QT) and labelling size (LS)

Dataset	Construction Time (CT) [s]				Query Time (QT) [ms]				Labelling Size (LS)			
	BatchHL$^+$	FulFD	FulPLL	PSL*	BatchHL$^+$	FulFD	FulPLL	PSL*	BatchHL$^+$	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	9782	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	7382	389	0.053	0.051	–	0.021	82 MB	608 MB	–	7.04 GB
Livejournal	20	46	–	4441	0.043	0.051	–	0.047	122 MB	663 MB	–	50.5 GB
Indochina	9	30	3205	86	0.788	0.767	–	0.007	85 MB	838 MB	–	3.39 GB
IT	9	30	3205	86	0.788	0.767	–	0.007	85 MB	838 MB	–	3.39 GB
Twitter	549	1928	–	–	0.868	0.174	–	–	1.14 GB	3.83 GB	–	–
Friendster	1181	3365	–	–	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

We omit results for BatchHL because there is no performance difference between BatchHL$^+$ and BatchHL as they both extend the same highway labelling approach to answer distance queries on dynamic networks. Note that when a method did not finish the labelling construction in 24h, we denote it as "–"

degree landmarks contributes greatly towards shortest-path coverage and makes querying process faster.

11.3 Performance under varying batch sizes

We compare the total time of querying and updating on dynamic graphs. We randomly select 10,000 updates from E and process them in batches of varying size. For each batch, we first delete 50% updates and then perform a batch in the fully dynamic setting. To make a fair comparison, the total time of our method BatchHL$^+$, and the baseline methods BatchHL, FULHL and FULFD is the total time to perform a batch update plus the query time to perform 1000 queries after performing a batch update and then averaged over 1000 queries, denoted as BatchHL$^+$+QT, BatchHL+QT, FULHL and FULFD+QT, respectively.

Figure 5 presents the results. For the baseline method BiBFS, we take only the query time averaged over 1000 queries after applying a batch update. The overall performance of BatchHL$^+$+QT is significantly better than the baseline methods on all datasets. It is worth noticing that BatchHL$^+$+QT is not only more efficient than BatchHL, but also their efficiency gap remain significant with the increased size of batch updates on majority of datasets. This shows that the novel batch search strategy used in BatchHL$^+$+QT is effective in reducing the size of affected vertex set. We 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.

11.4 Scalability test

We analyse the update time performance of our method BatchHL$^+$ under varying batches of insertions and deletions separately. We randomly select 10,000 updates from E and process them in batches of varying size. We start with a batch of 500 updates and then iteratively increase the batch by 500 updates to 10,000 updates. We process these batches in the incremental setting followed by the decremental setting. Figures 6 and 7 show the average update times after constructing the labelling from scratch, and updating the labelling after each batch increase. We observe from Fig. 6 that the update time of BatchHL$^+$ for batches of insertions is always below the construction time of labelling and significantly faster than the baseline methods. As compared with BatchHL$^+$, it does not show any performance difference because both algorithms repair roughly the same set of affected vertices (as can be seen in Table 6). Figure 7 shows that BatchHL$^+$ performs well on all the datasets. The performance gap of the proposed method BatchHL$^+$ compared to state-of-the-art methods BatchHL in the batch-update setting as well

Table 6 Average number of vertices affected by BatchHL and BatchHL$^+$ after performing batch updates on all the datasets

Method	Type	Youtube	Skitter	Flickr	Wikitalk	Hollywood	Orkut	Enwiki	Livejournal	Indochina	IT	Twitter	Friendster	UK	Italianwiki	Frenchwiki
	BatchHL	366 K	971 K	55 K	127 K	14 K	503 K	1220 K	276 K	2079 K	5878 K	10,622 K	66 K	54,515 K	–	–
Delete	BatchHL$^+$	23 K	11 K	22 K	17 K	2 K	3 K	5 K	13 K	16 K	28 K	2 K	6 K	12 K	–	–
	BatchHL	22 K	10 K	21 K	16 K	2 K	3 K	4 K	12 K	15 K	26 K	2 K	6 K	12 K	–	–
Add	BatchHL$^+$	22 K	10 K	21 K	16 K	2 K	3 K	4 K	12 K	15 K	26 K	2 K	6 K	12 K	–	–
	BatchHL	166 K	834 K	42 K	81 K	7 K	293 K	712 K	156 K	200 K	5681 K	8341 K	36 K	54,026 K	45 K	48 K
Mix	BatchHL$^+$	22 K	10 K	21 K	16 K	2 K	3 K	4 K	12 K	15 K	26 K	2 K	6 K	12 K	4 K	480

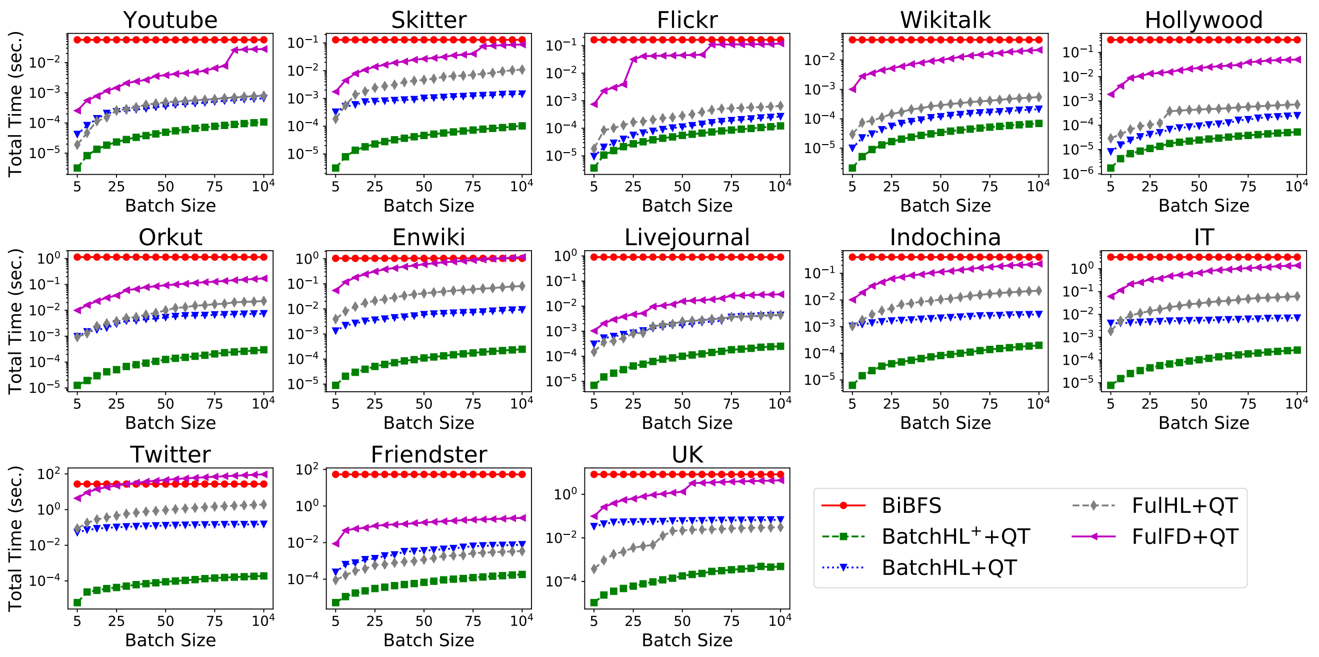


Fig. 5 Total time of querying and updating by the proposed and baseline methods under varying batches of mixed updates ranging from 500 to 10,000

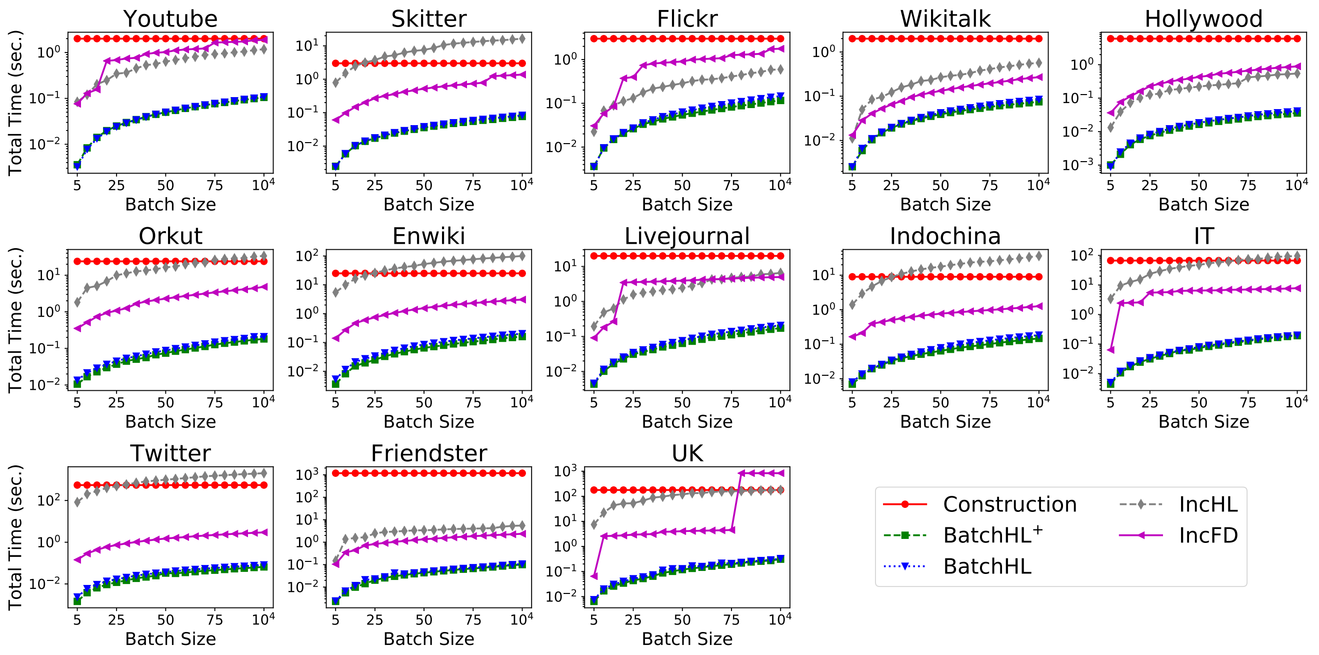


Fig. 6 Update time of the proposed and baseline methods under varying batches of insertions ranging from 500 to 10,000

as DecHL and DecFD in the single-update setting is huge which can be verified by the amount of repairs each algorithm has to perform in Table 6. Overall, the update time performance is dependent on the fraction of affected vertices to be repaired and our method can scale to perform large batches of updates efficiently.

11.5 Performance on directed graphs

Table 7 shows our experimental results on directed graphs in the *fully dynamic* setting (as described under "Test data generation"). The update time of our proposed method BatchHL$^+$ significantly outperforms the state-of-the-art method

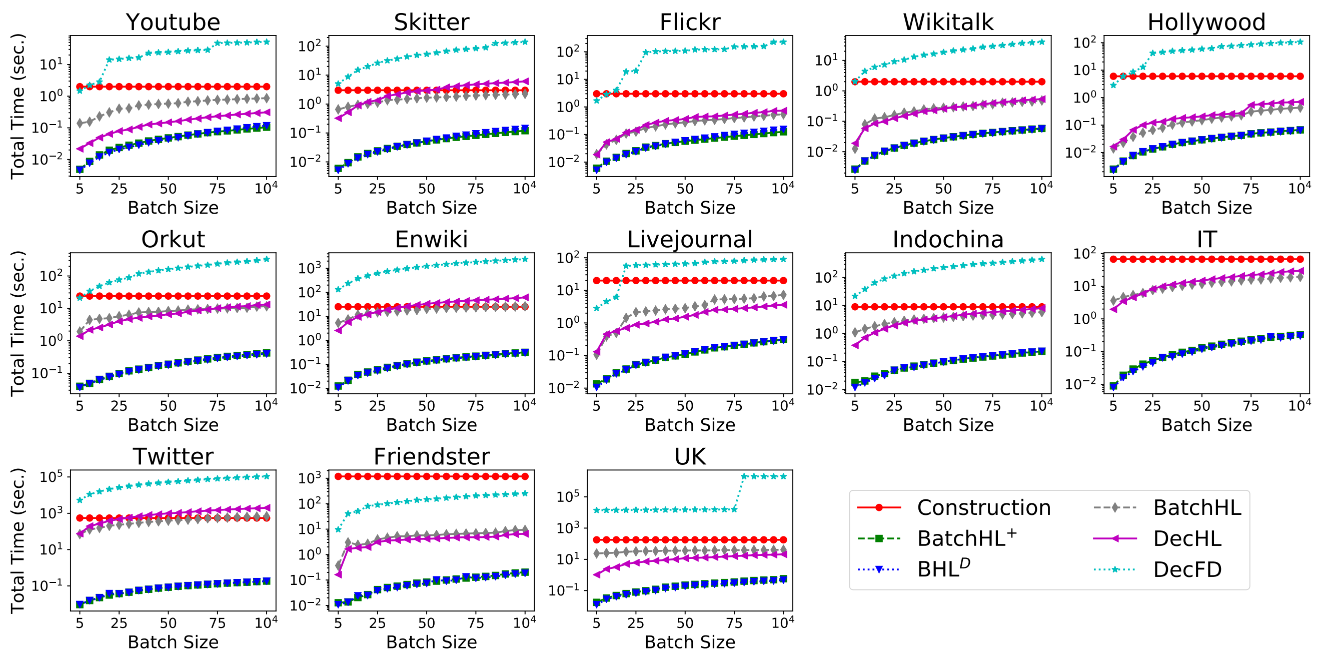


Fig. 7 Update time of the proposed and baseline methods under varying batches of deletions ranging from 500 to 10,000

Table 7 Comparing update time, construction time (CT), query time (QT) and labelling size (LS) on directed graphs

Datasets	BatchHL$^+$[s]	BatchHL[s]	CT[s]	QT[ms]	LS
Wikitalk	0.013	0.098	2.53	0.007	131 MB
Enwiki	0.034	27.91	55.8	0.037	436 MB
Livejournal	0.028	27.51	50.9	0.064	548 MB
Indochina	0.067	66.92	16.91	1.311	161 MB
Twitter	0.051	150.1	1122	0.368	4.19 GB
UK	12.12	1127	348.1	10.51	4.76 GB

Table 8 Average number of affected vertices after performing batch updates on directed graphs

Methods	Wikitalk	Enwiki	Livejournal	Indochina	Twitter	UK
BatchHL	217K	6,224K	11,224K	89,999K	17,746K	1,094,136K
BatchHL$^+$	14K	4K	11K	78K	5K	9,596K

BatchHL. The difference in the amount of affected vertices repaired by BatchHL$^+$ and BatchHL is shown in Table 8. It is clear that BatchHL$^+$ has to repair a much smaller fraction of affected vertices. Note that the performance of BatchHL$^+$ and BatchHL in terms of CT, QT and LS is the same because both of them dynamically maintain the highway labelling for fast distance querying. We also see that BatchHL$^+$ is several orders of magnitude faster in updating labels w.r.t. 1000 updates than constructing labelling from scratch for most of the datasets (Figs. 8, 9).

12 Extension to weighted graphs

For non-negative weighted graphs, we use the Dijkstra's algorithm in place of BFSs in our batch search and batch repair algorithms in order to compute and maintain a highway cover distance labelling for fast querying. We consider graph 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.

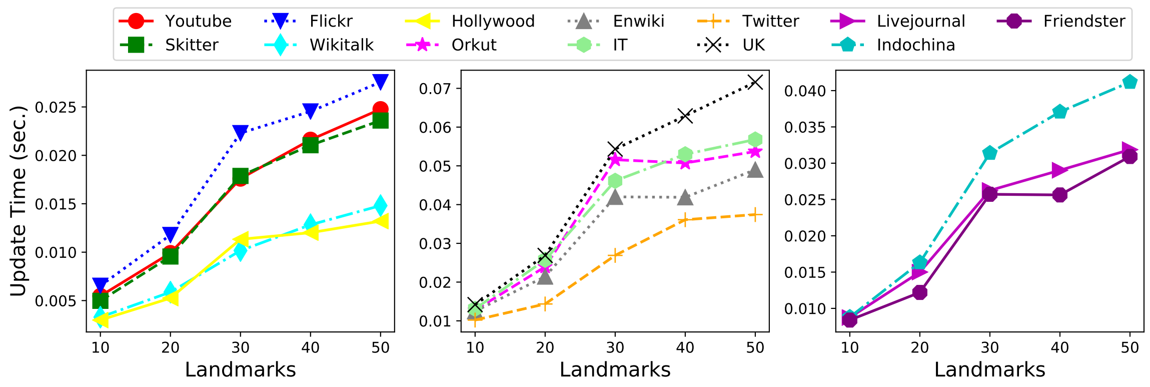


Fig. 8 Update time of the proposed method BatchHL$^+$ under varying landmarks ranging from 10 to 50

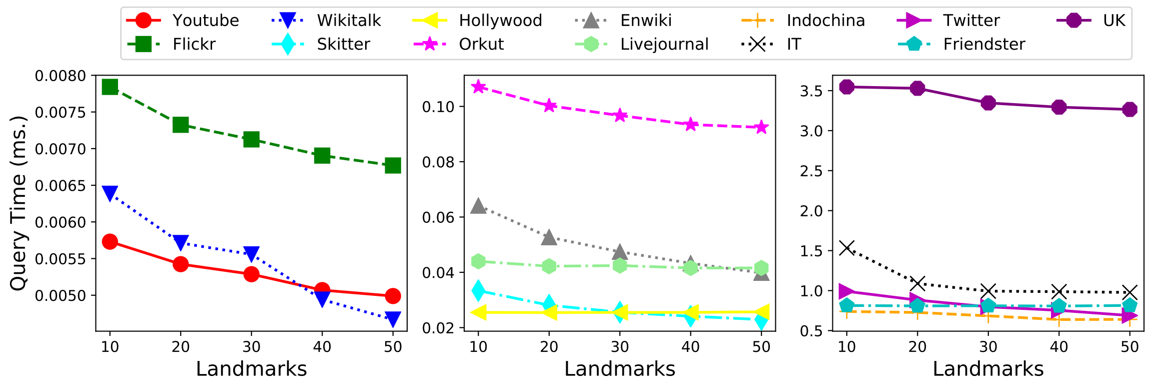


Fig. 9 Query time of the proposed method BatchHL$^+$ under varying landmarks ranging from 10 to 50

13 Conclusion

In this article, we have proposed an efficient method to answer distance queries on dynamic graphs undergoing batch updates. We have explored combined and separate effects of different types of updates in a batch to further reduce search space for maintaining distance labelling against graph changes. We have analysed the correctness and complexity of our method and showed that it preserves the labelling minimality. We have empirically verified the efficiency and scalability of our method on 15 real-world networks. For future work, we plan to explore the applicability and extension of the proposed method to road networks.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

References

1. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Hierarchical hub labelings for shortest paths. In: Proceedings of the 20th Annual European Conference on Algorithms, ESA'12, pp. 24–35. Springer-Verlag, Berlin (2012). https://doi.org/10.1007/978-3-642-33090-2_4
2. Acar, U.A., Anderson, D., Blelloch, G.E., Dhulipala, L.: Parallel batch-dynamic graph connectivity. In: The 31st ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '19, pp. 381–392. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3323165.3323196
3. Acar, U.A., Anderson, D., Blelloch, G.E., Dhulipala, L., Westrick, S.: Parallel batch-dynamic trees via change propagation. In: ESA (2020)
4. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, pp. 349–360. Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2463676.2465315
5. Akiba, T., Iwata, Y., Yoshida, Y.: Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In: Proceedings of the 23rd International Con-

ference on World Wide Web, WWW '14, pp. 237–248. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2566486.2568007
6. Akiba, T., Sommer, C., Kawarabayashi, K.i.: Shortest-path queries for complex networks: Exploiting low tree-width outside the core. In: Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12, pp. 144–155. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2247596.2247614
7. Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group formation in large social networks: Membership, growth, and evolution. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD, pp. 44–54 (2006). https://doi.org/10.1145/1150402.1150412
8. Boccaletti, S., Latora, V., Moreno, Y., Chavez, M., Hwang, D.U.: Complex networks: structure and dynamics. Phys. Rep. **424**(4–5), 175–308 (2006). https://doi.org/10.1016/j.physrep.2005.10.009
9. Boldi, P., Vigna, S.: The webgraph framework I: compression techniques. In: Proceedings of the 13th International Conference on World Wide Web, WWW, pp. 595–602 (2004). https://doi.org/10.1145/988672.988752
10. Chang, L., Yu, J.X., Qin, L., Cheng, H., Qiao, M.: The exact distance to destination in undirected world. VLDB J. **21**(6), 869–888 (2012). https://doi.org/10.1007/s00778-012-0274-x
11. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '02, pp. 937–946. Society for Industrial and Applied Mathematics, USA (2002)
12. D'angelo, G., D'emidio, M., Frigioni, D.: Fully dynamic 2-hop cover labeling. J. Exp. Algorithmics (2019). https://doi.org/10.1145/3299901
13. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Robust distance queries on massive networks. In: European Symposium on Algorithms, pp. 321–333. Berlin (2014). https://doi.org/10.1007/978-3-662-44777-2_27
14. Dhulipala, L., Durfee, D., Kulkarni, J., Peng, R., Sawlani, S., Sun, X.: Parallel batch-dynamic graphs: Algorithms and lower bounds. In: SODA, pp. 1300–1319 (2020)
15. Dhulipala, L., Liu, Q.C., Shun, J., Yu, S.: Parallel batch-dynamic k-clique counting. In: Symposium on Algorithmic Principles of Computer Systems (APOCS), pp. 129–143 (2021)
16. Farhan, M., Wang, Q.: Efficient maintenance of distance labelling for incremental updates in large dynamic graphs. In: 24th International Conference on Extending Database Technology EDBT (2021)
17. Farhan, M., Wang, Q.: Efficient maintenance of highway cover labelling for distance queries on large dynamic graphs. World Wide Web, pp. 1–26 (2023). https://doi.org/10.1007/s11280-023-01146-2
18. Farhan, M., Wang, Q., Koehler, H.: Batchhl: Answering distance queries on batch-dynamic networks at scale. SIGMOD '22, pp. 2020–2033. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3514221.3517883
19. Farhan, M., Wang, Q., Lin, Y., Mckay, B.: A highly scalable labelling approach for exact distance queries in complex networks. In: EDBT (2018)
20. Farhan, M., Wang, Q., Lin, Y., McKay, B.: Fast fully dynamic labelling for distance queries. VLDB J. (2021). https://doi.org/10.1007/s00778-021-00707-z
21. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: Graph distances in the streaming model: the value of space. In: SODA, vol. 5, pp. 745–754. Citeseer (2005)
22. Fu, A.W.C., Wu, H., Cheng, J., Wong, R.C.W.: Is-label: an independent-set based labeling scheme for point-to-point distance querying. Proc. VLDB Endow **6**(6), 457–468 (2013). https://doi.org/10.14778/2536336.2536346
23. Hayashi, T., Akiba, T., Kawarabayashi, K.I.: Fully dynamic shortest-path distance query acceleration on massive networks. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, CIKM '16, pp. 1533–1542. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2983323.2983731
24. Jin, R., Ruan, N., Xiang, Y., Lee, V.: A highway-centric labeling approach for answering distance queries on large sparse graphs. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, pp. 445–456. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2213836.2213887
25. Kunegis, J.: Konect: The Koblenz network collection. In: Proceedings of the 22nd International Conference on World Wide Web, WWW, pp. 1343–1350 (2013). https://doi.org/10.1145/2487788.2488173
26. Leskovec, J., Sosič, R.: Snap: a general-purpose network analysis and graph-mining library. ACM Trans. Intell. Syst. Technol. (2016). https://doi.org/10.1145/2898361
27. Li, W., Qiao, M., Qin, L., Zhang, Y., Chang, L., Lin, X.: Scaling distance labeling on small-world networks. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, pp. 1060–1077. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3299869.3319877
28. Li, Y., U, L.H., Yiu, M.L., Kou, N.M.: An experimental study on hub labeling based shortest path algorithms. Proc. VLDB Endow. **11**(4), 445–457 (2017). https://doi.org/10.1145/3186728.3164141
29. Liu, Q.C., Shi, J., Yu, S., Dhulipala, L., Shun, J.: Parallel batch-dynamic algorithms for k-core decomposition and related graph problems. In: Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '22, pp. 191–204. Association for Computing Machinery, New York (2022). https://doi.org/10.1145/3490148.3538569
30. McGregor, A.: Graph stream algorithms: a survey. SIGMOD Rec. **43**(1), 9–20 (2014). https://doi.org/10.1145/2627692.2627694
31. Ouyang, D., Yuan, L., Qin, L., Chang, L., Zhang, Y., Lin, X.: Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. Proc. VLDB Endow. **13**(5), 602–615 (2020)
32. Pacaci, A., Bonifati, A., Özsu, M.T.: Regular path query evaluation on streaming graphs. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, pp. 1415–1430. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3318464.3389733
33. Pohl, I.: Bi-derectional search. Mach. Intell. **6**, 127–140 (1971)
34. Potamias, M., Bonchi, F., Castillo, C., Gionis, A.: Fast shortest path distance estimation in large networks. In: Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM '09, pp. 867–876. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1645953.1646063
35. Qin, Y., Sheng, Q.Z., Falkner, N.J., Yao, L., Parkinson, S.: Efficient computation of distance labeling for decremental updates in large dynamic graphs. World Wide Web **20**(5), 915–937 (2017). https://doi.org/10.1007/s11280-016-0421-1
36. Tarjan, R.E.: Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia (1983). https://doi.org/10.1137/1.9781611970265
37. Ukkonen, A., Castillo, C., Donato, D., Gionis, A.: Searching the wikipedia with contextual information. In: Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM, pp. 1351–1352 (2008). https://doi.org/10.1145/1458082.1458274
38. Vieira, M.V., Fonseca, B.M., Damazio, R., Golgher, P.B., Reis, D.d.C., Ribeiro-Neto, B.: Efficient search ranking in social net-

works. In: Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management, CIKM, pp. 563–572 (2007). https://doi.org/10.1145/1321440.1321520

39. Wang, Y., Wang, Q., Koehler, H., Lin, Y.: Query-by-sketch: Scaling shortest path graph queries on very large networks. In: Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21, pp. 1946–1958. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3448016.3452826
40. Wei, F.: Tedi: Efficient shortest path query answering on graphs. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pp. 99–110. Association for Computing Machinery, New York (2010). https://doi.org/10.1145/1807167.1807181
41. Yu, M., Qin, L., Zhang, Y., Zhang, W., Lin, X.: Dptl+: Efficient parallel triangle listing on batch-dynamic graphs. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 1332–1343 (2021). https://doi.org/10.1109/ICDE51399.2021.00119
42. Zhang, M., Li, L., Hua, W., Zhou, X.: Efficient 2-hop labeling maintenance in dynamic small-world networks. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 133–144 (2021). https://doi.org/10.1109/ICDE51399.2021.00019