Dual-Hierarchy Labelling: Scaling Up Distance Queries on Dynamic Road Networks

MUHAMMAD FARHAN, Australian National University, Australia
HENNING KOEHLER, Massey University, New Zealand
QING WANG, Australian National University, Australia

Computing the shortest-path distance between any two given vertices in road networks is an important problem. A tremendous amount of research has been conducted to address this problem, most of which are limited to static road networks. Since road networks undergo various real-time traffic conditions, there is a pressing need to address this problem for dynamic road networks. Existing state-of-the-art methods incrementally maintain an indexing structure to reflect dynamic changes on road networks. However, these methods suffer from either slow query response time or poor maintenance performance, particularly when road networks are large. In this work, we propose an efficient solution *Dual-Hierarchy Labelling (DHL)* for distance querying on dynamic road networks from a novel perspective, which incorporates two hierarchies with different but complementary data structures to support efficient query and update processing. Specifically, our proposed solution is comprised of three main components: *query hierarchy*, *update hierarchy*, and *hierarchical labelling*, where *query hierarchy* enables efficient query answering by exploring only a small subset of vertices in the labels of two query vertices and *update hierarchy* supports efficient maintenance of distance labelling under edge weight increase or decrease. We further develop dynamic algorithms to reflect dynamic changes by efficiently maintaining the update hierarchy and hierarchical labelling. We also propose a parallel variant of our dynamic algorithms by exploiting labelling structure which aligns well with parallel processing. We evaluate our methods on 10 large road networks and it shows that our methods significantly outperform the state-of-the-art methods, i.e., achieving considerably faster construction and update time, while being consistently 2-4 times faster in terms of query processing and consuming only 10%-20% labelling space.

CCS Concepts: • **Mathematics of computing** → **Graph algorithms**; • **Theory of computation** → **Dynamic graph algorithms**; **Shortest paths**; • **Information systems** → *Data structures*.

Additional Key Words and Phrases: Shortest-path distance; dynamic road networks; query hierarchy; update hierarchy; hierarchical 2-hop labelling

ACM Reference Format:
Muhammad Farhan, Henning Koehler, and Qing Wang. 2025. Dual-Hierarchy Labelling: Scaling Up Distance Queries on Dynamic Road Networks. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 35 (February 2025), 25 pages. https://doi.org/10.1145/3709685

1 Introduction

Typically, a road network is modeled as a dynamic weighted graph $G = (V, E, \omega)$, where vertices V represent intersections, edges E represent roads between intersections, and ω assigns weights to edges, such as travel time. In a dynamic road network, it is commonly assumed that vertices and edges remain intact, while the weights of edges may continuously change due to real-time traffic conditions, such as traffic congestion, accidents, or road closures. Given any arbitrary pair (s, t) of

Authors' Contact Information: Muhammad Farhan, Australian National University, Canberra, Australia, muhammad.farhan@anu.edu.au; Henning Koehler, Massey University, Palmerston North, New Zealand, h.koehler@massey.ac.nz; Qing Wang, Australian National University, Canberra, Australia, qing.wang@anu.edu.au.

ACM 2836-6573/2025/2-ART35
https://doi.org/10.1145/3709685

Dataset	Method	Update Time [ms]		Query Time [μs]
		Increase	Decrease	
USA	DCH	0.84	0.27	**2,915.91**
	IncH2H	**356.27**	**239.84**	3.43
	DHL (ours)	73.59	49.29	0.83
EUR	DCH	0.73	0.26	**5,440.48**
	IncH2H	**96.63**	**66.97**	3.89
	DHL (ours)	28.83	17.03	1.19

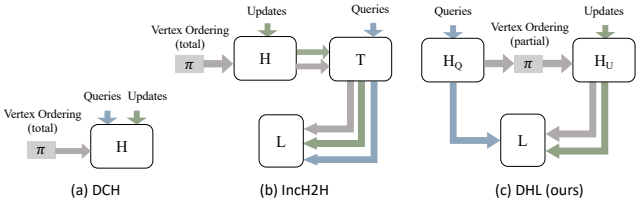


Fig. 1. A high-level comparison of our proposed method DHL with state-of-the-art methods: (a) DCH [18] and (b) IncH2H [26]. DCH processes both queries and updates based on CH (i.e., $H := CH$), IncH2H extends DCH to construct another tree hierarchy T to speed up queries, and DHL uses a tree hierarchy as H_Q and CH as H_U to process queries and updates, separately; L refers to a distance labelling. The grey, green and blue arrows indicate the processes of construction, updating, and querying, respectively.

vertices, the *distance querying* problem is to answer the up-to-date shortest-path distance between s and t under these dynamic changes.

As a fundamental building block, distance querying plays a crucial role in road applications, such as GPS navigation [12], route planning [8], traffic monitoring [16], and POI recommendation [23]. These applications often demand low-latency responses due to the need of computing thousands to millions of distance queries per second, as part of more complex tasks, which itself needs to be solved frequently. Examples include matching taxi drivers with passengers, optimizing delivery routes with multiple pick up and drop off points that can change dynamically, or providing recommendation on k-nearest POIs to customers. For instance, real-time navigation systems like Google Maps and Waze use preprocessing techniques such as contraction hierarchies [10] to accelerate the computation of shortest-path distance queries, enabling optimal route suggestions based on current traffic conditions. These conditions are obtained through various sources, such as crowdsourcing (where data like speed and location is collected from users who have the app open on their devices), road sensors, local transportation authorities, and real-time reports from users. This multifaceted approach ensures that traffic data is updated as frequently as possible, often multiple times per minute [18, 27]. In a similar vein, companies offering ride-hailing services like Uber and Lyft rely on millions of real-time distance queries to efficiently match drivers with passengers and minimize wait times [14, 24]. To meet user demands, these systems commonly use a hybrid client-server model. Complex real-time computations and traffic updates are handled by the server, while simpler tasks are managed on the client side. This allows for efficient and responsive navigation, even when the internet connection is weak or unavailable.

In this paper, we study the distance querying problem in dynamic road networks, aiming to maintain accurate distance information efficiently while still achieving superior query performance. *This task is challenging, as query time and update time are typically considered a trade-off, where improving one often comes at the expense of the other.*

Related work. Previously, several methods [5, 11, 18, 21, 25, 26] have been developed to incrementally maintain precomputed information, ensuring fast and correct responses to distance queries. These methods generally fall into two main categories:

(1) *Shortcut-based methods* [11, 18, 21], which focus on maintaining precomputed auxiliary edges, referred to as *shortcuts*, between vertices by applying vertex contraction in terms of a given vertex ordering, e.g., contraction hierarchy (CH) [10]. Such shortcuts help reduce query search space.

(2) *Hub-based labelling methods* [5, 25, 26], which maintain distance labelling in addition to shortcuts so that distance queries can be answered by searching distance labels directly. However, these methods require constructing an additional tree decomposition based on CH.

In a nutshell, shortcut-based methods require less space and shorter update time, but query time can be orders of magnitude longer than hub-based labelling methods. In contrast, hub-based labelling methods answer distance queries significantly faster at the cost of requiring additional space for storing distance labels and longer update time due to the combined maintenance of distance labels and shortcuts. The table in Figure 1 presents the empirical results on two large real-world road networks: one from USA [7] and the other from Europe [1].

Observations. To tackle the challenge of efficiently maintaining accurate distance information without compromising query performance, we make two crucial observations:

- **Separation of concerns**: Queries and updates are distinct types of operations. Therefore, different data structures can be employed to efficiently support distance queries and updates separately.
- **Vertex ordering**: Finding a vertex ordering based on the centrality of vertices in a road network plays a vital role in accelerating the performance of both queries and updates.

Previous works either use the same data structure for both queries and updates (e.g., DCH [18]) or use two data structures where the one for queries depends on the other to reflect updates (e.g., IncH2H [26]). Further, these methods, including DCH and IncH2H, all utilize a total vertex ordering based on general heuristics, such as minimum degree [4], without considering the specific properties of a road network. Figure 1(a)-(b) presents a high-level description of the methods DCH and IncH2H.

Present work. These observations guide us to develop a novel framework, *Dual-Hierarchy Labelling* (DHL), which aims to balance the trade-off between query time and update time while maintaining efficient and accurate distance querying in dynamic road networks. As depicted in Figure 1(c), the DHL framework has three cornerstones: $(\langle H_Q, H_U \rangle, L)$, where $\langle H_Q, H_U \rangle$ are two hierarchies integrated through a vertex partial ordering π, and L is a hierarchical labelling satisfying the 2-hop cover property [6]. More specifically, H_Q is a static *query hierarchy* designed to accelerate the efficiency of distance queries, while H_U is a dynamic *update hierarchy* aimed at enhancing the efficiency of distance maintenance in dynamic road networks. Both H_Q and H_U respect the partial order π on vertices. Unlike previous work, the vertex ordering π in our method is obtained by ordering vertices in terms of their occurrences in the minimum cuts of recursive partitions of a road network, reflecting their centrality in the entire network. In doing so, H_Q supports *efficient querying* by reducing the search space of labels when using L, while H_U supports *efficient maintenance* under dynamic changes, ensuring that L dynamically maintains correct distance information. Hence, distance queries are answered via H_Q and L, while dynamic changes are maintained via H_U and L.

Based on the DHL framework, we further investigate the following key aspects:

- *Design Choices:* We instantiate the DHL framework by employing a tree hierarchy for H_Q and a variant of the contraction hierarchy for H_U. Our design addresses the inherent limitations of existing shortcut-based and hub-based methods, such as DCH and IncH2H, by extracting a vertex partial ordering from H_Q to construct H_U. L is designed to consist of a *distance schema* Γ determined by H_Q and a *distance map* γ determined by H_U. The distance schema Γ captures the tree-like structural information of labelling in terms of H_Q, while the distance map γ maintains distance information efficiently by propagating dynamic changes through H_U. Furthermore, our hierarchical labelling L preserves distances within subgraphs rather

than the entire graph, while still exhibiting the 2-hop cover property for query efficiency. Restricting distances to subgraphs has the distinct advantage that a weight update only affects distance entries in the subgraph where the updated edge lies, leading to faster maintenance.

- *Dynamic Algorithms:* We propose dynamic algorithms that can efficiently maintain the update hierarchy H_U and the hierarchical labelling L to reflect edge weight decrease and increase on road networks. Specifically, under dynamic changes, we first maintain H_U to find affected shortcuts and then use these shortcuts as a starting point to update distance entries via the distance map γ. By exploring the structure of hierarchical labelling L, we also propose parallel variants of our dynamic algorithms which further improve maintenance performance by means of parallelizing searches w.r.t. multiple ancestors.
- *Theoretical Analysis:* We conduct theoretical analysis to prove the correctness of our algorithms through establishing a connection between shortcuts in H_U and distance entries in L based on H_Q. We also theoretically prove that restricting distances stored in the hierarchical labelling L to subgraphs, rather than distances in the entire graph, does not affect the 2-hop cover property of L. Furthermore, we analyse the complexity bounds of our dynamic algorithms for both edge increase and edge decrease.

We have conducted extensive experiments to evaluate the performance of our algorithms on 10 real-world large road networks, including the whole road network of USA and western Europe road network. The results show that our algorithms consistently outperforms the state-of-the-art method IncH2H on all of these road networks in both query time and update time. Figure 1 presents some results on two largest datasets (refer to Section 7.1 for details). In general, our method is about 3-4 times faster than IncH2H in terms of update time, while being 2-4 times faster in terms of query processing and consuming only 10%-20% labelling space.

Outline. The rest of the paper is organized as follows: Section 2 summarizes related work on distance queries in dynamic road networks. Section 3 presents two state-of-the-art methods for dynamic road networks, Dynamic Contraction Hierarchy (DCH) [18] and Incremental Hierarchical 2-Hop (IncH2H) [26] in detail. In Section 4, we introduce the key components of our proposed solution. Section 5 discusses two dynamic algorithms, DHL$^-$ and DHL$^+$, designed to efficiently maintain hierarchical labeling while handling edge weight decreases and increases, respectively. Section 6 examines the theoretical aspects, including correctness, 2-hop cover property, and complexity. Experimental results are presented in Section 7. Section 8 explores extensions to directed graphs, edge/node insertions and deletions, and boundedness. The paper concludes in Section 9.

2 Related Work

The classical approach for computing the shortest-path distance between a pair of vertices is to use Dijkstra's algorithm [20], which has a time complexity of $O(|E| + |V|log|V|)$ and may traverse an entire network to answer queries for vertices that are far apart from each other. Specifically, given a source vertex s and a target vertex t, Dijkstra's algorithm [20] traverses vertices in ascending order of their distances from s until t is reached. Bidirectional Dijkstra's algorithm [22], a variation of Dijkstra's algorithm, performs two Dijkstra's searches simultaneously from s and t, traversing vertices in ascending order of their distances to s and t, respectively. A* algorithm [13] estimates for each visited vertex v the heuristic distance value from v to the target vertex t and use it as the searching guidance. A* algorithm reduces Dijkstra's search space to a smaller conceptual ellipse [2]. Since search-based methods do not require any precomputed information and have no construction or maintenance cost, they are naturally adopted to dynamic road networks. However, these methods

are very inefficient in query processing and vulnerable to large search space, particularly on large road networks (e.g., millions of vertices).

Below, we focus on discussing two lines of work that leverage and maintain precomputed information for answering distance queries on dynamic road networks.

Shortcut-Based Methods. To improve query time, which is crucial for real-world applications, several works [10, 11, 18, 21, 28] precompute shortcuts between vertices. *Contraction hierarchy* (CH) [10] is the most widely used method in this direction. Given a pre-defined vertex order τ, CH performs vertex contraction on each vertex v one by one following τ to add shortcuts among its neighbors which preserve their distance information through v. Later, CH was extended to the dynamic setting [11, 18, 21], which can be divided into two categories: (1) *vertex-centric methods*; (2) *shortcut-centric methods*. The vertex-centric methods [11] first identify affected vertices and then re-contract them using the original vertex order to update shortcuts. The shortcut-centric methods [18, 21] first identify affected shortcuts and then update the weights of these shortcuts. The former requires the minimality of shortcuts during recontraction under the *shortest distance constraint* [10], which is highly inefficient because new or old shortcuts may need to be added or removed accordingly. The latter allows redundant shortcuts to avoid insertion or deletion of shortcuts during maintenance and updates only the weights of affected shortcuts due to the *minimum weight property* [18]. However, the shortcut-centric methods may add much more shortcuts than the vertex-centric methods, making it difficult to scale to graphs with large treewidth due to potentially extremely dense structures.

Hub-Based Labelling Methods. Recent works [5, 15, 25, 26] are mostly hub-based labelling methods, which precompute *distance labels* that capture distance information between all pairs of vertices in G. Then, only distance labels are searched at query time to compute distances. These works are typically built upon *hierarchical 2-hop index* (H2H-Index) [17] which can reduce search to a subset of labels $L(s)$ and $L(t)$ for two querying vertices s and t by exploiting a vertex hierarchy using tree decomposition. Specifically, they extend H2H-Index to dynamic road networks by incrementally maintaining H2H-Index to reflect edge weight updates on G. DynH2H [5] is the first dynamic algorithm which maintains H2H-Index in order to efficiently answer distance queries under dynamic changes on road networks. Zhang et al. [25] propose *dynamic tree decomposition based hub labelling* (DTDHL), an optimized version of DynH2H. Nonetheless, DTDHL may take seconds even for a single change, failing to scale to large road networks. Recently, Zhang et al. [26] studied the theoretical boundedness of dynamic CH index [18, 21] and dynamic H2H index [5, 25] for edge weight increase and decrease separately. IncH2H has shown to achieve the state-of-the-art performance on dynamic road networks. However, it suffers from maintaining a huge index which is constructed upon CH using a total ordering of vertices based on *minimum degree heuristic* [4].

3 State-of-the-Art Solutions

Let $G = (V, E, \omega)$ be a road network where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. Each edge $(u, v) \in E$ is associated with a non-negative weight $\omega(u, v) \in \mathbb{R}_{\geq 0}$. A simple path p is a sequence of distinct vertices $(v_1, v_2, \ldots, v_k)$ where $(v_i, v_{i+1}) \in E$ for each $i \in [1, k)$. The weight of a path p is defined as $\omega(p) = \sum_{i=1}^{k-1} \omega(v_i, v_{i+1})$. A shortest path p between two arbitrary vertices s and t is a path starting at s and ending at t such that $\omega(p)$ is minimised. The *distance* between s and t, denoted as $d_G(s, t)$, is the weight of any shortest path between s and t. We use $N(v)$ to denote the set of neighbors of a vertex $v \in V$, i.e. $N(v) = \{(u, \omega(u, v)) \mid u \in V, (u, v) \in E\}$, and $V(G)$ and $E(G)$ the set of vertices and edges in G, respectively. We assume that G is undirected (unless explicitly indicated otherwise) and treat dynamic changes on G as edge weight updates [17, 26].

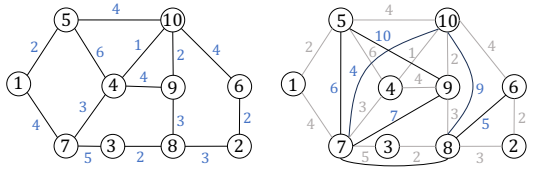


Fig. 2. (a) A road network G; (b) A shortcut graph S_G of G.

Below, we discuss two state-of-the-art methods for dynamic road networks: (1) Dynamic Contraction Hierarchy (DCH) [18], and (2) Incremental Hierarchical 2-Hop (IncH2H) [26].

3.1 Dynamic Contraction Hierarchy

Dynamic Contraction Hierarchy (DCH) [18] is built upon a variant [11] of the original Contraction Hierarchy (CH) [10]. This variant ensures that the presence of shortcuts is weight-independent, albeit at the cost of adding more shortcuts compared to the original CH. This trade-off significantly improves update performance.

Indexing. Given a graph G and a total ordering π over $V(G)$, DCH constructs a shortcut graph S_G of G by adding shortcuts as follows. A shortcut (v_1, v_k) is added between two vertices v_1 and v_k if and only if G contains a valley path [21]. A *valley path* is defined to be a simple path $p = (v_1, v_2, \ldots, v_k)$ satisfying $\pi(v_i) < \min\{\pi(v_1), \pi(v_k)\}$ for $\forall 1 < i < k$. In other words, the rank of any intermediate vertex of p is lower than the rank of its endpoints. The weight of the shortcut (v_1, v_k) is defined as the weight of the valley path p, i.e., $\omega(v_1, v_k) = \omega(p)$.

For each vertex v, let $N^+(v)$ be its upward neighbors in S_G which are ranked higher than v, i.e., $N^+(v) = \{u \mid (u, v) \in E(S_G) \wedge \pi(u) > \pi(v)\}$, and $N^-(v)$ be its downward neighbors in S_G which are ranked lower than v, i.e., $N^-(v) = \{u \mid (u, v) \in E(S_G) \wedge \pi(u) < \pi(v)\}$. Shortcuts in DCH satisfy the *minimum weight property* [18].

Property 3.1. *For any $(u, v) \in E(S_G)$, the following holds:*

$$\omega(u, v) = \min\left\{\omega_G(u, v), \omega(x, u) + \omega(x, v) \mid x \in N^-(u) \cap N^-(v)\right\}. \tag{1}$$

Here $\omega_G(u, v)$ denotes the weight of edge (u, v) in G if it exists, or ∞ otherwise. Note that every edge in G is a valley path, and thus a shortcut in S_G, and that its weight in G and S_G may differ.

Updates and Queries. DCH maintains shortcuts affected by edge weight decrease and increase separately. Let $(u, v) \in E(G)$ be an updated edge with $\pi(u) < \pi(v)$. For each $x \in N^+(u)$, if the weight of (u, v) is decreased and $\omega(v, x) > \omega(u, v) + \omega(u, x)$, then the weight of shortcut $\omega(v, x)$ is updated to $\omega(u, v) + \omega(u, x)$; if the weight of (u, v) is increased and $\omega(v, x) = \omega(u, v) + \omega(v, x)$, then the weight of shortcut $\omega(v, x)$ is updated based on Equation 1. When answering a distance query between two vertices $s, t \in V(G)$, a bidirectional Dijkstra's search from s to t is performed over S_G with the restriction that edges are only searched in an upward direction w.r.t. the vertex order π.

Example 3.1. Figure 2(b) illustrates a shortcut graph S_G obtained from an example road network G shown in Figure 2(a) following a total vertex ordering 1<2<3<4<5<6<7<8<9<10. Consider a query pair $(6, 9)$ on the road network G in Figure 2. A bidirectional search from 6 and 9 is conducted such that only the edges $(6, 10)$, $(6, 8)$, $(8, 9)$ are considered in the search from vertex 6, and only the edge $(9, 10)$ is considered in the search from vertex 9 because they lead to vertices with higher ranks. Thus we get $d_G(6, 9) = \min\{\omega(6, 10) + \omega(9, 10), \omega(6, 8) + \omega(8, 9)\} = 6$.

3.2 Incremental Hierarchical 2-Hop

Incremental Hierarchical 2-Hop (IncH2H) [26] is built upon H2H-Index [17] to incrementally maintain distance labels and shortcuts for distance queries on dynamic road networks.

Indexing. H2H-Index is a 2-hop labelling constructed using tree decomposition based on DCH. Let S_G be a shortcut graph over G [18]. H2H-Index first constructs a tree decomposition T based on S_G and π as follows. Each vertex $u \in V(S_G)$ corresponds to a tree node in T. For a vertex $u \in V(S_G)$ with a non-empty $N^+(u)$, the vertex $v \in N^+(u)$ with lowest rank $\pi(v)$ is assigned as the parent of the tree node of u. This construction ensures that for each $u \in V(S_G)$, all vertices in $N^+(u)$ correspond to tree nodes that are ancestors of u in T. Another property of T is that every shortest path between any two vertices s and t in G must pass through at least one vertex in $\{u\} \cup N^+(u)$, where the tree node of u is the lowest common ancestor of s and t in T. Then, a 2-hop labelling is constructed using T. The label $L(v)$ of each vertex v consists of three arrays: (i) *ancestor array* $[w_1, \ldots, w_k]$ representing the path from the root to v in T, (ii) *distance array* $[\delta_{vw_1}, \ldots, \delta_{vw_k}]$ where $\delta_{vw_i} = d_G(v, w_i)$ and $\{w_1, \ldots, w_k\}$ is the set of vertices that are ancestors of v in T, and (iii) *position array* $[i_1, \ldots, i_k]$ storing positions of $\{w_1, \ldots, w_k\}$ in T which are their depths in T.

Updates and Queries. H2H-Index is dynamically maintained in two phases: 1) *shortcut maintenance* identifies affected shortcuts in S_G and updates their weights; 2) *labelling maintenance* updates distance labels based on affected shortcuts in S_G. For each affected shortcut $(u, w) \in E(S_G)$ with $\pi(u) < \pi(w)$, it finds all ancestors a of w for which the distance between u and a has changed, and updated the corresponding values in the distance array. Finally, descendants x of u are processed to identify pairs (x, a) whose distance has changed, and updates distances in $L(x)$.

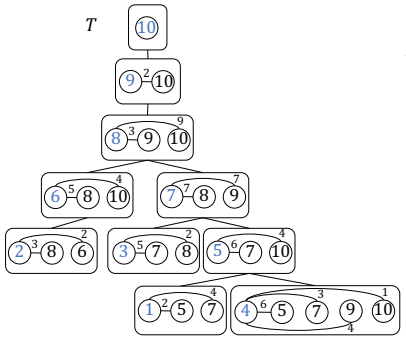
Given two vertices $s, t \in V(G)$, their lowest common ancestor $lca(s, t)$ in T is first searched and then $d_G(s, t)$ is computed as

$$d_G(s,t) = \min\{L(s).dist(i) + L(t).dist(i) \mid i \in L(x).pos, x = lca(s,t)\}. \quad (2)$$

Example 3.2. Figure 3(a) and 3(b) illustrate a tree decomposition T and the H2H-Index L for a road network shown in Figure 2(a). $L(1)$ stores an ancestor array $[10, 9, 8, 7, 5, 1]$ containing all ancestors of 1 in T, a distance array $[6, 8, 11, 4, 2, 0]$ storing the distances from vertex 1 to its ancestors, and a position array $[4, 5, 6]$ storing the positions in the ancestor array of 1 for $\{7, 5, 1\}$, the vertices inside the tree node of 1 in T. Suppose the weight of an edge $(6, 10)$ has changed, IncH2H first updates the weight of the affected shortcut $(8, 10)$ in S_G shown in Figure 2(b). Then, starting from $\{(6, 10), (8, 10)\}$, it first updates the distance between vertices $\{6, 8\}$ to their ancestor 10 and iteratively identifies affected downward neighbors of 6 and 8 whose distance to 10 has also changed. $L(2)$ will be updated as its distance to 10 will be affected as well. For a distance query between vertices 1 and 2, $lca(1, 2) = 8$ is first obtained, and then using the distances in $L(1)$ and $L(2)$ at the positions $[1, 2, 3]$ in $L(8)$, $d_G(1, 2) = 12$ is obtained according to Equation 2.

3.3 Discussion

Both state-of-the-art methods, DCH and IncH2H, exploit a vertex ordering to construct a shortcut graph S_G. This graph is then used in querying and maintenance by DCH, and to construct a tree decomposition by IncH2H. It is known that such a vertex ordering is crucial for constructing S_G with a minimum number of shortcuts. However, finding an optimal vertex ordering that minimizes the number of shortcuts is challenging [3]. As a result, DCH suffers from scalability issues, leading to slow querying in large road networks. Similarly, IncH2H produces tree decompositions with very large height and width, resulting in huge H2H-Index sizes that hinder efficient maintenance.



(a)

L

Label	Ancestors	Distance	Position
L(1)	[10, 9, 8, 7, 5, 1]	[6, 8, 11, 4, 2, 0]	[4, 5, 6]
L(2)	[10, 9, 8, 6, 2]	[6, 6, 3, 2, 0]	[3, 4, 5]
L(3)	[10. 9, 8, 7, 3]	[7, 5, 2, 5, 0]	[3. 4, 5]
L(4)	[10, 9, 8, 7, 5, 4]	[1, 4, 7, 3, 5, 0]	[1, 2, 4, 5, 6]
L(5)	[10, 9, 8, 7, 5]	[4, 6, 9, 6, 0]	[1, 4, 5]
L(6)	[10, 9, 8, 6]	[4, 6, 5, 0]	[1, 3, 4]
L(7)	[10, 9, 8, 7]	[4, 7, 7, 0]	[1, 2, 3, 4]
L(8)	[10, 9, 8]	[5, 3, 0]	[1, 2, 3]
L(9)	[10, 9]	[2, 0]	[1, 2]
L(10)	[10]	[0]	[1]

(b)

Fig. 3. (a) A tree decomposition T; (b) H2H-Index L.

Additionally, IncH2H requires a complex mechanism for computing the least common ancestor of two vertices, which further degrades query performance.

4 Our Proposed Solution

Our proposed solution consists of three components: $(\langle H_Q, H_U \rangle, L)$, where H_Q is a static *query hierarchy*, H_U is a dynamic *update hierarchy*, and L is a *hierarchical labelling*. Further, H_Q and H_U are *order invariant* with respect to a vertex partial order $\preceq_H$ over $V(G)$, and L satisfies the 2-hop cover property. In the following, we will discuss the details of our solution.

4.1 Hierarchies: H_Q and H_U

Query Hierarchy. The purpose of H_Q is to improve query efficiency. A natural candidate for H_Q is a tree hierarchy, as tree-like structures are known to be very efficient for search [9, 17]. This is also evidenced by the state-of-the-art method IncH2H [26] where H2H-Index employs tree decomposition to accelerate query performance by restricting search to part of the labels of two given query vertices. However, as H2H-Index constructs a tree decomposition based on CH, it is vulnerable to produce very large heights h and widths w due to the ordering employed in CH. Consequently, query performance becomes slower as it needs to explore labels proportional to w for answering a distance query.

Definition 4.1 (Query Hierarchy). A *query hierarchy* H_Q over a graph G is a balanced binary tree, $H_Q = (V_Q, E_Q, \omega_Q)$, where V_Q is a set of tree nodes, E_Q is a set of tree edges, and $\ell : V(G) \to V_Q$ is a total surjective function, satisfying the following properties:

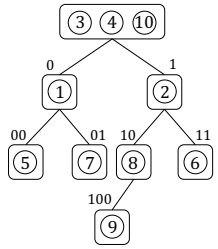
(1) Each $N \in \mathcal{N}$ satisfies

$$|T_{\downarrow}(N_L)|, |T_{\downarrow}(N_R)| \leq (1 - \beta) \cdot |T_{\downarrow}(N)|$$

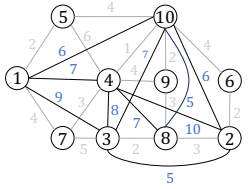
where $0 < \beta \leq 0.5$, $T_{\downarrow}(N)$ denotes a subtree rooted at N, and N_L and N_R are the left and right children of N, respectively.

(2) Every path connecting $\{s, t\} \subseteq V(G)$ contains a vertex a such that $\ell(a)$ is a common ancestor of $\ell(s)$ and $\ell(t)$.

Example 4.2. Figure 4(a) shows a query hierarchy H_Q of the road network G in Figure 2(a). Consider two vertices 6 and 9 in Figure 2(a) and their corresponding tree nodes in Figure 4(a). The common ancestors of 6 and 9 are $\{\{3, 4, 10\}, \{2\}\}$ and there is a shortest-path between 6 and 9 i.e., $\langle 6, 10, 9 \rangle$ containing vertex 10 from a common ancestor $\{3, 4, 10\}$.



(a)



(b)

Label	Distance
L(1)	[9, 7, 6], [0]
L(2)	[5, 7, 6], [0]
L(3)	[0]
L(4)	[8, 0]
L(5)	[11, 5, 4], [2], [0]
L(6)	[7, 5, 4], [2], [0]
L(7)	[5, 3, 10], [4], [0]
L(8)	[2, 6, 5], [3], [0]
L(9)	[5, 3, 2], [6], [3], [0]
L(10)	[7, 1, 0]

(c)

Fig. 4. (a) Query hierarchy H_Q, (b) Update hierarchy H_U, and (c) Hierarchical labelling L.

Following [9], we construct a tree hierarchy for H_Q using the recursive bi-partitioning algorithm which finds balanced and minimal cuts to partition a graph. For each cut, we compute a partition bitstring and depth – the number of vertices mapped to its ancestors in the tree hierarchy. Partition bitstrings allow us to find the lowest common ancestor of two nodes s, t in constant time, and the depth of this ancestor is then used to identify which parts of $L(s)$ and $L(t)$ to use in computing their distance. Compared to [9], a query hierarchy requires that paths only need to intersect *some* common ancestor rather than the lowest one, which thus allows omission of shortcuts during construction. This in turn leads to smaller cuts, reducing both the number of common ancestor vertices and overall labelling size. However, we need to inspect all common ancestors to answer distance queries, not just the lowest one.

Vertex Partial Order. The tree structure of H_Q defines an ancestor-descendant relationship among tree nodes, which can be extended to a vertex partial order as defined below.

Definition 4.3 (Vertex Partial-Order $\preceq_H$). Let H_Q be a tree hierarchy and $\preceq$ an arbitrary total order between vertices. The *vertex partial-order* $\preceq_H$ induced by H_Q and $\preceq$ is defined as:

$$v \preceq_H u \quad \Leftrightarrow \quad \ell(v) \text{ is a strict ancestor of } \ell(u) \text{ in } H_Q, \text{ or}$$
$$\ell(v) = \ell(u) \text{ and } v \preceq u$$

We use $anc(v) = \{w \in V(G) \mid w \preceq_H v\}$ and $desc(v) = \{w \in V(G) \mid v \preceq_H w\}$ to denote the *ancestors* and *descendants* of v, respectively, and $lca(v, w)$ the common ancestor of vertices v and w which is largest w.r.t. $\preceq_H$, i.e., their *lowest common ancestor*.

Example 4.4. Consider H_Q in Figure 4(a). By ordering the vertices within each tree node using their node identifiers, we obtain the vertex partial order

$$\preceq_H := \{3 < 4 < 10 < \ldots, 1 < 5, 1 < 7, 2 < 6, 2 < 8 < 9\}.$$

We also have $anc(7) = \{3, 4, 10, 1, 7\}$, $anc(4) = \{3, 4\}$, $lca(6, 7) = 10$, and $lca(6, 9) = 2$.

Update Hierarchy. Unlike H_Q, the purpose of H_U is to support efficient maintenance of the labelling under dynamic changes. Thus, instead of tree-like structures, a shortcut graph (i.e., the original graph G plus a set of shortcuts) from CH [10] would be a good candidate for H_U, which can not only preserve distances between vertices in G but also enable fast search for updating shortcuts and distance labels being affected. A central concept of CH is valley path, where intermediate vertices have higher ranks than their endpoints. We extend this notion to partial orders in our work.

Definition 4.5 (Valley Path). A path p between v and w with $w \preceq_H v$ is a *valley path* iff $v \preceq_H u$ for all $u \in V(p) \setminus \{v, w\}$.

These valley paths form the basis of our update hierarchy.

Definition 4.6 (Update Hierarchy). An update hierarchy H_U over a graph G and a vertex partial order $\preceq_H$ is a shortcut graph which contains a shortcut (v, w) for every valley path between v and w. The weight associated with a shortcut (v, w) is the length of the shortest valley path between v and w.

Example 4.7. Figure 4(b) illustrates update hierarchy H_U over an example road network G shown in Figure 2(a) using the vertex partial order from Example 4.4. There is a shortcut edge $(1, 4)$ in H_U because valley paths $\langle 1, 7, 4\rangle$ and $\langle 1, 5, 4\rangle$ between 1 and 4 exist in G. Note that $\langle 1, 5, 10, 4\rangle$ is not a valley path since $10 \preceq_H 1$. We have $\omega(1, 4) = 7$, the weight of the shortest valley path $\langle 1, 7, 4\rangle$.

Let $G = (V, E)$ be a road network, $\Delta(E)$ be a set of edge weight updates on G, and $G \oslash \Delta(E)$ the resulting graph after applying $\Delta(E)$ to G. We also use $H_U = (V_U, E_U, \omega_U)$ and $H_U^\Delta = (V_U^\Delta, E_U^\Delta, \omega_U^\Delta)$ to refer to the update hierarchies corresponding to G and $G \oslash \Delta(E)$, respectively. The following properties are satisfied by H_U and H_U^Δ :

(U1). **Structural stability**: H_U and H_U^Δ differ only in edge weights, i.e., $(V_U, E_U) = (V_U^\Delta, E_U^\Delta)$.
(U2). **Bounded searching**: A weight update of $(v, w) \in \Delta(E)$ only affects shortcuts (v', w') with $v', w' \preceq_H v, w$.

We construct H_U using the approach [18]. However, unlike [18, 21] which use a pre-defined total vertex order, we perform vertex contractions following $\preceq_H$ in decreasing order. The following lemma shows that the update hierarchy based on $\preceq_H$ is identical to the classic CH based on any total order extending $\preceq_H$.

Lemma 4.8. *Let $\preceq$ be a total ordering on $V(G)$ extending $\preceq_H$, $w \preceq v$, and p a valley path between v and w w.r.t. $\preceq$. Then $w \preceq_H v$ and p is also a valley path w.r.t. $\preceq_H$.*

Proof. Definition 4.1 implies that p must contain a common ancestor r with $r \preceq_H v, w$. As p is a valley path w.r.t. $\preceq$, we must have $w \preceq r$. Since $\preceq$ extends $\preceq_H$, it follows that $r = w$. This shows $w \preceq_H v$. Consider now any intermediate vertex $u \in V(p) \setminus \{v, w\}$ and the subpath p' of p connecting v and u. Again, Definition 4.1 implies the existence of $r \in V(p')$ with $r \preceq_H v$ and $r \preceq_H u$. Since p is a valley path w.r.t. $\preceq$, we must have $v \preceq r$. It follows that $v = r \preceq_H u$, which makes p a valley path w.r.t. $\preceq_H$. □

While our update hierarchy is essentially a contraction hierarchy, there are some subtle differences. To maintain the roles of upward and downward neighbors in our context, we define

$$N^+(v) = \{u \mid (v, u) \in E(H_U) \wedge u \preceq_H v\};$$
$$N^-(v) = \{u \mid (v, u) \in E(H_U) \wedge v \preceq_H u\}.$$

4.2 Hierarchical Labelling: L

We present the design of the hierarchical labelling L, which is based on *the principle of separation* between H_Q and H_U. This ensures that distance queries can be answered by searching L via H_Q, while dynamic changes in a road network can be reflected in L via H_U.

Conceptually, the hierarchical labelling L in our work is associated with a *distance scheme* Γ_L and a *distance map* γ_L defined over Γ_L. Below, we provide detailed definitions of Γ and γ, omitting the subscript L to simplify the notation.

Distance Scheme. The distance scheme Γ describes how distances are to be stored in L, which is determined by query hierarchy H_Q. Since each vertex $v \in V(G)$ is associated with a set of ancestor tree nodes $\mathcal{N}(v) = \{\ell(w) \mid w \in anc(v)\}$ in H_Q, we start with defining the notion of *tree node scheme* to describe the positions of vertices that are associated with each tree node.

Algorithm 1: Hierarchical Labelling Construction

```
1 Function DHL(τ, H_U)
2     initialize L_*[*] = ∞
      // copy shortcuts
3     foreach (v, w) ∈ E(H_U) with τ(v) > τ(w) do
4         L_v[w] ← ω(v, w)
      // compute label distances top-down
5     foreach v ∈ V in increasing order of τ(v) do
6         foreach w ∈ N^+(v) do
7             foreach i ∈ [0, τ(w)] do
8                 L_v[i] ← min(L_v[i], ω(v, w) + L_w[i])
```

Definition 4.9 (Tree Node Scheme). Let $N^t \in \mathcal{N}(v)$, $depth(N^t)$ be the depth of N^t in H_Q, and $t = depth(N^t)$. The *tree node scheme* of v at N^t is $\Gamma^t(v) = [w_1, \ldots, w_k]$ where $w_1 \preceq_H \ldots \preceq_H w_k$, and $\{w_1, \ldots, w_k\} = \{w_i | w_i \in \ell^{-1}(N^t), w_i \preceq_H v\}$.

Based on tree node schemes for ancestor tree nodes, we define the distance scheme Γ for each vertex $v \in V(G)$.

Definition 4.10 (Distance Scheme). Let $N = \ell(v)$. Then, the *distance scheme* $\Gamma(v)$ of v is defined as

$$\Gamma(v) = \Gamma^0(v) \mid \Gamma^1(v) \mid \ \ldots \ \mid \Gamma^m(v), \tag{3}$$

where $\Gamma^t(v)$ is the tree node scheme of v at a tree node $N^t \in \mathcal{N}(v)$ and $m = depth(N)$.

Our distance scheme is purely conceptual (no data is actually stored) and does not change as edge weights in G are modified.

Distance Map. The distance map γ stores distance entries into tree node schemes of the distance scheme Γ. Note that the distance entries we store are not necessarily distances in G as for IncH2H or HC2L, but distances within a subgraph of H_U.

Definition 4.11 (Subgraph Distance). We denote by $d_{H_U}^{[w,v]}$ the distance between w and v in the subgraph of H_U induced by

$$desc(w) \cap anc(v) = \{u \mid w \preceq_H u \preceq_H v\}$$

We use $L_v[w]$ to denote the distance between v and w in this subgraph, i.e., $L_v[w] = d_{H_U}^{[w,v]}(v, w)$. As we shall see later, these distance entries can also be expressed as distances in certain subgraphs of G. This close relationship to H_U enables us to update L efficiently based on changes to H_U.

Definition 4.12 (Distance Map). A *distance map* is a function $\gamma : V(G) \times V(G) \to \mathbb{R}_{\geq 0}$ such that, for any $\Gamma^t(v) = [w_1, \ldots, w_k]$,

$$L^t(v) = \gamma(\Gamma^t(v)) = [\gamma(v, w_1), \ldots, \gamma(v, w_k)], \tag{4}$$

where $\gamma(v, w_i) = L_v[w_i]$ for $i = 1, \ldots, k$. Let $N = \ell(v)$. Accordingly, γ over $\Gamma(v)$ defines the label $L(v)$ as

$$L(v) = L^0(v) \mid L^1(v) \mid \ \ldots \ \mid L^m(v). \tag{5}$$

Example 4.13. Consider Figure 4. The distance scheme of vertex 7 is $\Gamma(7) = \Gamma^0(7) \mid \Gamma^1(7) \mid \Gamma^2(7)$ with $\Gamma^0(7) = \{3, 4, 10\}, \Gamma^1(7) = \{1\}$ and $\Gamma^2(7) = \{7\}$ at depth 0, 1 and 2 in Figure 4(a). Accordingly, the distance map of vertex 7 is shown as $L(7)$ in Figure 4(c) storing distances to ancestors in $\Gamma(7)$. $L_7[\tau(10) = 2]$ is 10 rather than 4, the distance in the subgraph of H_U induced by $\{10, 1, 7\}$.

We shall denote by $\tau(v) = |\{w \in V(G) | w \preceq_H v\}|$ the number of ancestors of vertex v w.r.t. $\preceq_H$. In our implementation we use $\tau(u) \leq \tau(v)$ to check whether $u \preceq_H v$ holds. By Lemma 4.8 shortcut edge endpoints are always comparable w.r.t. $\preceq_H$, so this characterization is sound. Note that τ plays the same role as the vertex ordering π used in contraction hierarchies, except that the ordering is reversed and vertices incomparable w.r.t. $\preceq_H$ may share τ values.

Our bottom-up approach to construct the hierarchical labelling L is described as Algorithm 1. We compute the label $L(v)$ for each vertex v in increasing order of $\tau(v)$. Specifically, we inspect ancestors of upward neighbors $w \in N^+(v)$ to compute $L_v[i]$ (Lines 7-9).

4.3 Queries and Updates

In the following, we discuss how distance queries and weight updates are processed in our work.

Distance Queries. Given two vertices $s, t \in V(G)$, a distance query between s and t is processed in two phases via H_Q and L:

(1) The common ancestors $anc(s) \cap anc(t)$ of s and t are quickly found through $lca(s, t)$ in H_Q;
(2) By restricting the search of $L(s)$ and $L(t)$ to only vertices occurring in $anc(s) \cap anc(t)$, $d_G(s, t)$ is defined as

$$d_G(s, t) = \min\{L_s[r] + L_t[r] \mid L_s[r] \in L(s), L_t[r] \in L(t), r \in anc(s) \cap anc(t)\}. \tag{6}$$

As we identify tree nodes in H_Q via bitstrings, we can compute the level l of $lca(s, t)$ as the common prefix of the bitstrings of s and t. Then the common ancestors of s and t are the vertices at the depths between 0 and l which can also be efficiently found.

Example 4.14. Consider a query pair $(6, 9)$. The bitstrings of vertices 6 and 9 are 11 and 100, respectively, shown in Figure 4(a). The depth of $lca(6, 9) = \{2\}$ is 1 and their common ancestors are at the depths $0 \leq i \leq 1$, i.e., $anc(6) \cap anc(9) = \{3, 4, 10, 2\}$.

Weight Updates. Given any weight updates $\Delta(E)$ on G, H_U and L are processed to reflect $\Delta(E)$, which also involves two phases:

(1) H_U is processed to find all shortcuts $\Delta(S)$ on H_U affected by $\Delta(E)$ and the weights of these shortcuts are updated.
(2) The distance entries of L affected by $\Delta(S)$ are updated.

Observe that using our definitions of upward and downward neighbors, Property 3.1 holds for H_U as well, and can be used to maintain shortcuts. Let ω_U and ω_U^Δ refer to the shortcut weights w.r.t. G and $G \oslash \Delta(E)$, respectively.

Definition 4.15 (Affected Shortcut). A shortcut (v, w) in H_U is *affected* by $\Delta(E)$ iff $\omega_U(v, w) \neq \omega_U^\Delta(v, w)$.

Lemma 4.16. *If $(v, w) \in E(H_U)$ is affected by $\Delta(E)$, then either*

(1) $(v, w) \in \Delta(E)$, or
(2) (v, x) or (x, w) is affected for some $x \in N^-(v) \cap N^-(w)$.

Algorithm 2: H_U Maintenance (Decrease)

Function $\mathtt{DH}^-_U(\tau, H_U, \Delta(E))$
  $Q \leftarrow \emptyset$ // *a priority queue*
  foreach $(v, w, \omega_{new}) \in \Delta(E)$ *with* $\tau(v) > \tau(w)$ **do**
    if $\omega(v, w) > \omega_{new}$ **then**
      $\omega(v, w) \leftarrow \omega_{new}$
      add (v, w) into Q
  foreach $(v, w) \in Q$ *in decreasing order of* $\tau(v)$ **do**
    foreach $w' \in N^+(v) \setminus \{w\}$ **do**
      if $\omega(w, w') > \omega(v, w) + \omega(v, w')$ **then**
        $\omega(w, w') \leftarrow \omega(v, w) + \omega(v, w')$
        add $(\max_\tau(w, w'), \min_\tau(w, w'))$ into Q
  return affected shortcuts

Algorithm 3: H_U Maintenance (Increase)

Function $\mathtt{DH}^+_U(\tau, H_U, \Delta(E))$
  $Q \leftarrow \emptyset$ // *a priority queue*
  foreach $(v, w, \omega_{old}) \in \Delta(E)$ *with* $\tau(v) > \tau(w)$ **do**
    if $\omega(v, w) = \omega_{old}$ **then**
      add (v, w) into Q
  foreach $(v, w) \in Q$ *in decreasing order of* $\tau(v)$ **do**
    // *recompute shortcut distance*
    $\omega_{new} \leftarrow \begin{cases} \omega & \text{if } (v, w, \omega) \in E(G) \oslash \Delta(E) \\ \infty & \text{otherwise} \end{cases}$
    foreach $x \in N^-(v) \cap N^-(w)$ **do**
      $\omega_{new} \leftarrow \min(\omega_{new},\ \omega(x, v) + \omega(x, w))$
    if $\omega(v, w) \neq \omega_{new}$ **then**
      foreach $w' \in N^+(v) \setminus \{w\}$ **do**
        if $\omega(w, w') = \omega(v, w) + \omega(v, w')$ **then**
          add $(\max_\tau(w, w'), \min_\tau(w, w'))$ into Q
      $\omega(v, w) \leftarrow \omega_{new}$
  return affected shortcuts

This enables maintenance of H_U by focusing on the intermediate vertex x of the shortcut triangle v–x–w. A bottom-up search (w.r.t. $\preceq_H$) ensures that the weights of (v, x) and (x, w) have already been updated before using (v, w). We describe this, and how to update L based on H_U, in the next section.

Algorithm 4: DHL Maintenance (Decrease)

```
1  Function DHL⁻(τ, H_U, L, Δ(E))
2      Δ(S) ← DH⁻_U(τ, H_U, Δ(E)) // shortcut updates
3      Q ← ∅ // a priority queue
       // update distances involving ancestors
4      foreach (v, w, ω_new) ∈ Δ(S) with ω_new < L_v[w] do
5          foreach i ∈ [0, τ(w)] do
6              if ω_new + L_w[i] < L_v[i] then
7                  L_v[i] ← ω_new + L_w[i]
8                  add (v, i) into Q
       // identify and update distances involving descendants
9      foreach (v, i) ∈ Q in increasing order of τ(v) do
10         foreach u ∈ N⁻(v) do
11             if L_u[v] + L_v[i] < L_u[i] then
12                 L_u[i] ← L_u[v] + L_v[i]
13                 add (u, i) into Q
```

5 Dynamic Algorithms

In this section we propose two dynamic algorithms, DHL$^-$ and DHL$^+$, to efficiently maintain the hierarchical labelling L, handling edge weight decrease and increase, respectively. Since L is constructed upon the update hierarchy H_U, we first maintain H_U and then maintain L using H_U. Note that each $L_v[w]$ stores the distance between two vertices v and w in a (fairly small) induced subgraph of H_U, and not the distance in G, which makes some update operations simpler in our work, compared to IncH2H.

5.1 Edge Weight Decrease

Algorithm 4 shows *DHL*$^-$. It first maintains the update hierarchy H_U using DH^-_U described as Algorithm 2. Then, in Algorithm 2, for each update $(v, w, \omega_{new}) \in \Delta(E)$ with $\tau(v) > \tau(w)$, DH^-_U tests whether the old weight in H_U is greater than ω_{new}. If so, it updates the weight of the edge (v, w) in H_U and pushes (v, w) to a priority queue Q (Lines 3-8). Each affected shortcut $(v, w) \in Q$ is processed in the decreasing order of $\tau(v)$, i.e., for each $w' \in N^+(v)$ it checks whether the weight of the shortcut (w, w') in H_U is greater than the weight of the new path through v (Lines 7-9), updates it accordingly and pushes (w, w') to Q (Lines 10-11). The maintenance of H_U continues until all affected shortcuts are updated and finally affected shortcuts $\Delta(S)$ are returned to Algorithm 4 (Line 2).

At Lines 4-8 of Algorithm 4, for each shortcut $(v, w, \omega_{new}) \in \Delta(S)$, it examines distance entries in the label of v w.r.t. ancestors of w. Specifically, for each ancestor $i \in [0, \tau(w)]$, if the old distance $L_v[i]$ is greater than the sum of the new distance ω_{new} between v and w and $L_w[i]$, it updates $L_v[i]$ and adds (v, i) to a priority queue Q, as the distance update between v and ancestor i may have affected the labels of descendants of v. Next, at Lines 9-13, it processes pairs $(v, i) \in Q$ in increasing order of $\tau(v)$ to ensure the availability of correct labels of ancestors when updating the labels of descendants. That is, for each pair $(v, i) \in Q$, it examines each $u \in N^-(v)$ to check whether distance between u to i is changed via ancestor v ($L_v[i]$ which has been updated) and accordingly update and enqueue pair (u, i) into Q. This process continues iteratively until no further labels are

Algorithm 5: DHL Maintenance (Increase)

```
1  Function DHL+(τ, H_U, L, Δ(E))
2      Δ(S) ← DH+_U(τ, H_U, Δ(E)) // shortcut updates
3      Q ← ∅ // a priority queue
       // identify distances to ancestors to update
4      foreach (v, w, ω_old) ∈ Δ(S) with ω_old = L_v[w] do
5          foreach i ∈ [0, τ(w)] do
6              if ω_old + L_w[i] = L_v[i] then
7                  add (v, i) into Q
       // identify and update distances involving descendants
8      foreach (v, i) ∈ Q in increasing order of τ(v) do
9          ω_new ← ∞ // new distance from v to i
10         foreach w ∈ N+(v) with τ(w) ≥ i do
11             ω_new ← min(ω_new, ω(v, w) + L_w[i])
           // distance may not have changed after all
12         if ω_new > L_v[i] then
13             foreach u ∈ N−(v) do
14                 if L_u[v] + L_v[i] = L_u[i] then
15                     add (u, i) into Q
16             L_v[i] ← ω_new
```

updated. We note that this approach does not consider all paths of Definition 4.11, but only paths in which vertices are ordered by τ. This will be justified later by Lemma 6.3.

5.2 Edge Weight Increase

Algorithm 5 describes DHL$^+$ for edge weight increase. Similar to DHL$^-$, it first maintains H_U using DH^+_U described as Algorithm 3. In Algorithm 3, for each update $(v, w, \omega_{new}) \in \Delta(E)$ with $\tau(v) > \tau(w)$, DH^+_U tracks path(s) through the updated edge and pushes (v, w) to a priority queue Q (Lines 3-5). Then, each affected shortcut $(v, w) \in Q$ is processed in the decreasing order of $\tau(v)$. Different from the weight decrease case, the updated weight ω_{new} of shortcut (v, w) is obtained using Equation 1 (Lines 7-9) and whether the weight obtained is different from the old weight is checked (Line 10). If so, it examines upward neighbors $w' \in N^+(v)$ to identify potentially affected shortcuts (w, w') and pushes them to Q, before updating (v, w) (Lines 11-14). The maintenance of H_U continues until all affected shortcuts are updated, and finally affected shortcuts $\Delta(S)$ are returned to Algorithm 5 (Line 2).

Then, at Lines 4-7, for each affected shortcut $(v, w, \omega_{old}) \in \Delta(S)$, it examines distance entries in the label of v w.r.t. ancestors of w. Those with the same distance as a path ending in the shortcut (v, w) are potentially affected, and added to a priority queue Q. Next, at Lines 8-16, it processes pairs $(v, i) \in Q$ in increasing order of $\tau(v)$. For each pair $(v, i) \in Q$, it computes the new distance ω_{new} for $L_v[i]$ (Lines 10-11). If ω_{new} is greater than the old distance $L_v[i]$, it examines each $u \in N^-(v)$ to check whether a shortest path between u and i passes through vertex v (Line 14) and might thus be affected and if so enqueues (u, i). Finally, we update the label entry $L_v[i]$ (Line 16). This process continues iteratively until no further labels are updated. Note that unlike IncH2H, DH^+_U and DHL$^-$

do not track the support for shortcuts or labels as in practice most of them have a support of one. This helps to save space at the cost of recomputing distances of some unaffected shortcuts and label entries. Experimentally, we found the fraction of unnecessary recomputations to be small enough to justify this trade-off.

Example 5.1. Consider that the weight of edge $(7, 4)$ is decreased to 1 from 3 in the road network depicted in Figure 2(a). As the weight is decreased, $\omega(7, 4)$ of the shortcut $(7, 4)$ is updated in H_U and inserted into Q (Lines 4-6). Lines 7-11 then process $(7, 4)$ and inspect weights of the shortcuts $\{(1, 4), (4, 3)\}$. As $\omega(1, 4) > \omega(7, 1) + \omega(7, 4)$ and $\omega(4, 3) > \omega(7, 3) + \omega(7, 4)$, there weights are updated to 5 and 6, respectively. Then in Algorithm 4, labels of affected vertices are updated w.r.t. $\Delta(S) = \{(7, 4, 1), (1, 4, 5), (4, 3, 6)\}$. The label entries $L_7[\tau[4] = 1], L_1[\tau[4] = 1]$ and $L_4[\tau[3] = 0]$ are updated to 1, 5 and 6, respectively, and inserted into Q at Lines 4-8. Due to the updates of $(1, 4)$ and $(4, 3)$, the downward neighbors $\{5, 7\} \in N^-(1)$ and $\{1, 2, 5, 7, 8, 9, 10\} \in N^-(4)$ are inspected in Lines 9-13 to see if their distances to 4 and 3 change, but no more labels are updated.

Now consider the weight of the edge $(4, 7)$ increased to 5 from 3 in the road network depicted in Figure 2(a). Again, the shortcuts $\{(1, 4), (4, 3)\}$ are inspected w.r.t. $(7, 4)$ at Lines 11-13 and found to be affected as $\omega(1, 4) = \omega(7, 1) + \omega(7, 4)$ and $\omega(4, 3) = \omega(7, 4) + \omega(7, 3)$. The weights of $\Delta(S) = \{(7, 4), (1, 4), (4, 3)\}$ are updated in H_U at Line 14 (Algorithm 3) and returned to Algorithm 5 (Line 2). Then, in Algorithm 5, after updating $L_7[\tau[4] = 1], L_1[\tau[4] = 1]$ and $L_4[\tau[3] = 0]$ there are no further updates of downward neighbors.

5.3 Parallel Maintenance

We now introduce parallel variants for our dynamic algorithms DHL$^-$ and DHL$^+$, namely DHL^-_p and DHL^+_p described as Algorithm 6-7, respectively. We observe that the most time consuming part of Algorithms 4 and 5 is to compute updated distances to descendants at Lines (9-13) and (8-16) respectively. We can parallelize these parts to significantly improve performance. The main idea is to distribute this work amongst multiple threads. We group entries $(v, i) \in Q$ w.r.t. ancestors i in their respective queues Q_i (Line 3). This way all entries with the same i are processed within the same thread. To make this work without costly synchronization operations, we must ensure that distance labels accessed by different threads are distinct. Observe that all entries generated when processing an entry $(v, i) \in Q_i$ are in the form (u, i), and thus exclusive to the processing thread. Furthermore, the only labels used for comparison are $L_u[v]$ and $L_v[i]$. The latter also "belongs" to the processing thread and will not be updated externally, but $L_u[v]$ could be updated by the thread processing $\tau(v)$, which is problematic. However, we can simply replace $L_u[v]$ with $\omega(u, v)$ which is not updated by any thread, and still obtain correct updates (by Lemma 6.3).

6 Theoretical Results

We discuss several theoretical aspects of our solution, including correctness, 2-hop cover property, and complexity.

Correctness Analysis. We first establish a connection between shortcuts in H_U and distance entries in L.

Definition 6.1 (Shortcut Chain). A *shortcut chain* is a descending sequence of vertices

$$v_1 \preceq_H \ldots \preceq_H v_n$$

such that v_i and v_{i+1} (for $i \in [1, n)$) are connected by a shortcut in H_U, i.e., the shortcut chain *connects* v_1 and v_n. The *length* of a shortcut chain is the sum of weights associated with the shortcuts.

Algorithm 6: DHL Parallel Maintenance (Decrease)

```
1  Function DHL⁻_p(τ, H_U, L, Δ(E))
2      ...
       // identify and update distances involving descendants
3      partition Q into Q_0, Q_1, ... with (v, i) ∈ Q_i
4      foreach Q_i in parallel do
5          foreach (v, i) ∈ Q_i in increasing order of τ(v) do
6              foreach u ∈ N⁻(v) do
7                  if ω(u, v) + L_v[i] < L_u[i] then
8                      L_u[i] ← ω(u, v) + L_v[i]
9                      add (u, i) into Q_i
```

Algorithm 7: DHL Parallel Maintenance (Increase)

```
1  Function DHL⁺_p(τ, H_U, L, Δ(E))
2      ...
       // identify and update distances involving descendants
3      partition Q into Q_0, Q_1, ... with (v, i) ∈ Q_i
4      foreach Q_i in parallel do
5          foreach (v, i) ∈ Q_i in increasing order of τ(v) do
6              ω_new ← ∞ // new distance from v to i
7              foreach w ∈ N⁺(v) with τ(w) ≥ i do
8                  ω_new ← min(ω_new, ω(v, w) + L_w[i])
               // distance may not have changed after all
9              if ω_new > L_v[i] then
10                 foreach u ∈ N⁻(v) do
11                     if ω(u, v) + L_v[i] = L_u[i] then
12                         add (u, i) into Q_i
13                 L_v[i] ← ω_new
```

Example 6.2. Consider the update hierarchy H_U from Figure 4(b). Here 3-2-6 forms a shortcut chain of length 7, connecting 3 and 6.

Let $d_G^w(v,u)$ denote the distance between two vertices v and u in the subgraph of G induced by $desc(w)$. We have the following.

Lemma 6.3. *For any w, v with $w \preceq_H v$, there exists a shortcut chain from w to v of length $d_G^w(w,v)$.*

Proof. Let p be a shortest path from w to v passing only through the descendants of w. We show that p can be decomposed into subpaths p_i between vertices $w = v_1 \preceq_H \ldots \preceq_H v_n = v$ such that each p_i is a valley path. Once shown the claim follows.

Let $\preceq$ be a total ordering on $V(G)$ extending $\preceq_H$. By Lemma 4.8 it suffices to show that the subpaths p_i are valley paths w.r.t. $\preceq_H$. Let $u \in V(p) \setminus \{v, w\}$ be the smallest intermediate vertex w.r.t. $\preceq$. If $v \prec u$, then p is already a valley path and the decomposition is trivial. Otherwise, we must

have $w \prec u \prec v$ and can decompose p into p_{wu} and p_{uv} connecting w to u and u to v, respectively. As u is the minimal intermediate vertex, p_{wu} is a valley path, and p_{uv} passes only through the descendants of u. By induction on the length of p, p_{uv} can be decomposed into a chain of valley paths, which combined with p_{wu} forms a decomposition of p. □

Theorem 6.4. *Algorithms 4 and 5 correctly update distance entries in L to contain distances w.r.t. the updated graph.*

Proof (Sketch). By Definition 6.1, distance entries in L are the lengths of minimal shortcut chains. The first for loop in Algorithms 4 and 5 collects all (potentially) affected distance entries associated with shortcut chains that had their last shortcut updated. The second for loop iteratively extends those shortcut chains. □

Recall that the subgraph distance $d_{H_U}^{[w,v]}$ in Definition 4.11 is based on the subgraph of H_U containing only vertices between w and v. Since shortcut chains between w and v are paths within these subgraphs, we immediately obtain the following corollary.

Corollary 6.5. *Let $w \preceq_H v$. Then $d_G^w(w,v) = d_{H_U}^{[w,v]}(w,v)$.*

2-Hop Cover Proof. We show that the hierarchical labelling L is indeed a 2-hop labelling [6]. Recall that $L_v[w] = d_{H_U}^{[w,v]}(w,v)$.

Lemma 6.6. *For any two vertices $s,t \in V(G)$, there exists at least one vertex r with $r \preceq_H s$ and $r \preceq_H t$ s.t. $L_s[r] + L_t[r] = d_G(s,t)$.*

Proof. Let p be a shortest path between s and t in G, and r the vertex in p with the minimal $\tau(r)$. Then $r \preceq_H v$ for all $v \in V(p)$. Thus, p lies in the subgraph of G induced by the descendants of r. By Corollary 6.5 $L_s[r]$ and $L_t[r]$ store distances within this subgraph. It follows that $L_s[r] + L_t[r] \leq |p| = d_G(s,t)$. The direction $L_s[r] + L_t[r] \geq d_G(s,t)$ is obvious. □

Complexity Analysis. Let E_Δ denote the number of affected edges in G, S_Δ the number of affected shortcuts in H_U, and L_Δ the number of affected distance entries in L. Further, denote by $d_{\max}$ the maximum degree of vertices in H_U, and by h the maximum number of ancestors of a vetex w.r.t. $\preceq_H$. We assume constant time access to shortcut weights in H_U.

Theorem 6.7. *Algorithms 2 and 3 operate in $O(E_\Delta + S_\Delta \cdot d_{\max})$ and $O(E_\Delta \cdot d_{\max} + S_\Delta \cdot d_{\max}^2)$, respectively.*

Proof. Consider Algorithm 2. The shortcuts that are *potentially* affected are affected edges, plus shortcuts incident to affected shortcuts. The number of the latter is bounded by $O(S_\Delta \cdot d_{\max})$. For Algorithm 3, the *potentially* affected shortcuts are similarly bounded, but processing each shortcut requires $O(d_{\max})$ time to compute the new distance value. □

Theorem 6.8. *Algorithms 4 and 5 operate in $O(S_\Delta \cdot h + L_\Delta \cdot d_{\max})$ and $O(S_\Delta \cdot h \cdot d_{\max} + L_\Delta \cdot d_{\max}^2)$, respectively.*

Proof. Consider Algorithm 4. The distance entries that are *potentially* affected in the first for loop correspond to the affected shortcuts, plus distance entries pointing to ancestors of one endpoint of an affected shortcut. The number of these is bounded by $O(S_\Delta \cdot h)$. The second for loop investigates potentially affected distance entries that form a triangle with an affected distance entry and a shortcut; their number is thus bounded by $O(L_\Delta \cdot d_{\max})$. For Algorithm 5, the distance entries being processed are similarly bounded, but processing each of them requires $O(d_{\max})$ time to compute the new distance value. □

Under the assumptions that $E_\Delta \leq S_\Delta$ and $S_\Delta \cdot h \leq L_\Delta \cdot d_{\max}$, which hold in practice, the above complexity bounds can be simplified to

– Algorithms 2 and 3 : $O(S_\Delta \cdot d_{\max})$ and $O(S_\Delta \cdot d_{\max}^2)$;
– Algorithms 4 and 5 : $O(L_\Delta \cdot d_{\max})$ and $O(L_\Delta \cdot d_{\max}^2)$.

Note further that the extra $d_{\max}$ factors for the weight increase algorithms are due to weight calculations for shortcuts or distance entries where a shortest path is eliminated, but other paths of the same length remain. In practice, such cases are rare and observed performance is much closer to the weight decrease case than the analysis suggests. Tracking supports as for IncH2H could eliminate these factors from the theoretical bounds.

Table 1. Summary of datasets - 10 real-world road networks.

Network	Region	$\|V\|$	$\|E\|$	Memory
NY	New York City	264,346	733,846	17 MB
BAY	San Francisco	321,270	800,172	18 MB
COL	Colorado	435,666	1,057,066	24 MB
FLA	Florida	1,070,376	2,712,798	62 MB
CAL	California	1,890,815	4,657,742	107 MB
E	Eastern USA	3,598,623	8,778,114	201 MB
W	Western USA	6,262,104	15,248,146	349 MB
CTR	Central USA	14,081,816	34,292,496	785 MB
USA	United States	23,947,347	58,333,344	1.30 GB
EUR	Western Europe	18,010,173	42,560,279	974 MB

7 Experiments

We conducted experiments to verify the effectiveness of our proposed solution. All the experiments were performed on a Linux server Intel Xeon W-2175 with 2.50GHz CPU, 28 cores, and 512GB of main memory. All the algorithms were implemented in C++20 and compiled using g++ 9.4.0 with the -O3 option. Distance results are exact for all methods considered, and correctness has been verified using Dijkstra.

Datasets. We use 10 undirected real-world road networks. Nine of these road networks are from the US and publicly available at the webpage of the 9th DIMACS Implementation Challenge [7], while the other one is from Western Europe managed by PTV AG [1]. Table 1 summarises these datasets where the largest dataset is the whole road network in the USA.

Baselines. We compare our algorithms with the state-of-the-art method IncH2H [26] for distance queries on dynamic road networks: We use IncH2H$^+$, IncH2H$^-$ and IncH2H$_p^+$, IncH2H$_p^-$ to denote the sequential and parallel version of IncH2H for edge weight increase and decrease, respectively. We do not consider DCH [18] in our comparison because their query performance is generally orders of magnitude slower than the hub-based labelling methods. Further, we omit the comparison with DTDHL [25] since it has been significantly outperformed by IncH2H. The code for IncH2H was kindly provided by their authors and implemented in C++. We select the balance partition threshold $\beta = 0.2$ and set the number of threads to 28, the number of available cores.

Table 2. Comparing update times of our algorithms and the state-of-the-art algorithms for batch updates.

Network	Increase [ms]				Decrease [ms]			
	DHL_p^+	$IncH2H_p^+$	DHL^+	$IncH2H^+$	DHL_p^-	$IncH2H_p^-$	DHL^-	$IncH2H^-$
NY	0.209	0.234	0.790	2.900	0.116	0.187	0.522	2.006
BAY	0.153	0.178	0.543	2.498	0.103	0.134	0.394	1.769
COL	0.257	0.318	0.933	4.613	0.179	0.241	0.696	3.306
FLA	0.311	0.390	1.906	4.981	0.216	0.320	1.368	3.585
CAL	0.786	1.185	5.079	20.20	0.539	0.855	3.614	13.89
E	1.913	2.481	12.20	43.57	1.314	1.820	8.197	29.33
W	2.420	3.841	18.11	68.99	1.757	2.772	12.69	47.76
CTR	8.721	15.13	58.72	309.7	5.570	10.75	38.48	213.1
USA	9.321	18.20	73.59	356.3	6.004	13.06	49.29	239.8
EUR	5.634	8.283	26.83	96.63	3.273	6.969	17.03	66.97

7.1 Performance Comparison

Update Time. We randomly sampled 10 batches for each network, where each batch contains 1,000 updates. Then, for each update (a, b, ω) in a batch, we increase its weight to $2.0 \times \omega$ and maintain the labelling using algorithms for weight increase and decrease (restore) its weight to original (i.e., to ω) and maintain the labelling using algorithms for weight decrease. We process updates in both batch update setting and single update setting (one by one) and report the average update time over 10 batches in Tables 2 and 3.

Table 2 and 3 show that our algorithms DHL^+ and DHL^- significantly outperform $IncH2H^+$ and $IncH2H^-$. In particular, DHL^+ and DHL^- are about 3-4 times faster in update time compared to $IncH2H^+$ and $IncH2H^-$. The reason behind our outstanding performance is two-fold. Firstly, H_U is constructed based on an order induced by H_Q which exploits structure of road networks via minimal balanced cuts. This ensures that DHL contains fewer labels than IncH2H to begin with. Secondly, unlike IncH2H whose labels contain distances in G, DHL stores distances to ancestors within induced subgraphs. This further reduces the search space when updating distance entries. Table 3 shows difference in the number of affected labels updated by DHL and IncH2H, respectively. E.g. for NY 8 out of 31 million DHL label entries (26%) had their distance value changed. We observe that the fraction of affected labels ($L_\Delta/|L|$) tends to be smaller for DHL than for IncH2H, which can be attributed to the reduced search space. We do not report results w.r.t. E_Δ and S_Δ as the cost to maintain G and H_U is negligible compared to L. The parallel variants of our algorithms DHL_p^+ and DHL_p^- also significantly outperform $IncH2H_p^+$ and $IncH2H_p^-$ on all datasets.

Query Time. We randomly sample 1,000,000 pairs of vertices from each network. Table 4 reports the average query time over 1 million query pairs. DHL clearly outperforms IncH2H, i.e., about 3 times faster over all road networks. The reason is that DHL produces significantly smaller labelling size compared to IncH2H. Smaller labels allow DHL to use caching more effectively. Further, DHL processes a significantly smaller number of distance entries related to common ancestors in the labels of a query pair.

We also evaluate the query performance using query pairs with varying distances, similar to [17, 19]. We generate 10 sets of query pairs $Q_1, Q_2, \ldots, Q_{10}$ for each network. Let $x = (\frac{l_{max}}{l_{min}})^{1/10}$, where $l_{min} = 1{,}000$ and l_{max} is the maximum distance of any query pair. For $\forall 1 \leq i \leq 10$, we generate 10,000 queries in each set Q_i, where their distances are in the range $(l_{min} \cdot x^{i-1},\ l_{min} \cdot x^i]$. We report the average query time for all road networks in Figure 6. We can see that DHL significantly

Table 3. Comparing update times of our algorithms and the state-of-the-art algorithms for single updates.

Network	Increase [ms]		Decrease [ms]		Affected Labels L_Δ (Million)	
	DHL$^+$	IncH2H$^+$	DHL$^-$	IncH2H$^-$	DHL	IncH2H
NY	1.190	4.069	0.847	3.021	8/31 (26%)	40/99 (40%)
BAY	1.069	3.910	0.827	2.992	6/24 (25%)	43/94 (46%)
COL	2.172	9.693	1.708	7.279	13/41 (32%)	96/166 (58%)
FLA	2.546	5.703	2.010	4.531	15/97 (15%)	58/283 (20%)
CAL	6.951	27.05	5.447	19.99	42/252 (17%)	297/1,023 (29%)
E	13.22	58.97	9.967	42.28	92/736 (13%)	864/2,627 (33%)
W	19.74	83.27	15.01	60.27	119/1,210 (10%)	3,430/4,595 (75%)
CTR	59.56	334.3	42.65	233.2	331/5,092 (7%)	3,604/23,250 (16%)
USA	71.19	349.7	51.41	240.1	458/9,206 (5%)	944/40,220 (2%)
EUR	23.24	87.02	15.99	65.11	157/9,511 (2%)	567/42,300 (1%)

Table 4. Comparing query times, labelling sizes and construction times of our method DHL with the state-of-the-art method IncH2H.

Network	Query Time [μs]		Labelling Size		Shortcuts Size		Const. Time [s]	
	DHL	IncH2H	DHL	IncH2H	DHL	IncH2H	DHL	IncH2H
NY	0.287	0.913	130 MB	826 MB	15 MB	42 MB	2	4
BAY	0.299	0.841	105 MB	797 MB	12 MB	40 MB	2	3
COL	0.349	1.018	176 MB	1.35 GB	15 MB	51 MB	4	5
FLA	0.396	1.019	425 MB	2.38 GB	40 MB	129 MB	10	11
CAL	0.490	1.333	1.03 GB	8.12 GB	73 MB	233 MB	25	30
E	0.630	1.683	2.92 GB	20.5 GB	136 MB	444 MB	64	74
W	0.664	1.702	4.83 GB	36.0 GB	231 MB	758 MB	107	126
CTR	0.812	2.483	19.7 GB	177 GB	558 MB	1.77 GB	455	858
USA	0.834	3.428	35.6 GB	307 GB	931 MB	2.97 GB	710	1,081
EUR	1.185	3.888	36.4 GB	320 GB	733 MB	2.38 GB	907	1,254

outperforms IncH2H for long distance pairs. Long distance pairs have a significantly small number of common ancestors because their lowest common neighbor generally lies at higher levels of a hierarchy. For short distance pairs, DHL often examines more hops compared to IncH2H because the lowest common ancestor (LCA) lies at lower levels of the query hierarchy, resulting in more ancestors to search in the labels. Despite this, DHL performs comparably on almost all road networks for short distance queries. This is due to two factors: (1) the hops are stored in a continuous memory block, allowing for efficient access, and (2) our data structure enables faster computation of LCA.

Labelling Size. We also compare the memory consumed by DHL and the state-of-the-art method IncH2H. Table 4 shows that our method DHL requires significantly less memory than IncH2H. In particular, the labelling size of DHL is about 9 times smaller than IncH2H on the largest three datasets. This is because DHL is constructed based on a vertex partial order from H_Q. Since H_Q is generated using a partitioning technique which produces small and minimal cuts to partition a graph, our distance labels store a smaller number of label entries than IncH2H. Additionally, IncH2H employs auxiliary data structures to speed up maintenance and ensure strong theoretical bounds, such as support, which inflate memory requirements further. We also compared the memory sizes

of both methods for storing shortcuts, and find that IncH2H uses about 3 times more memory than DHL, for similar reasons.

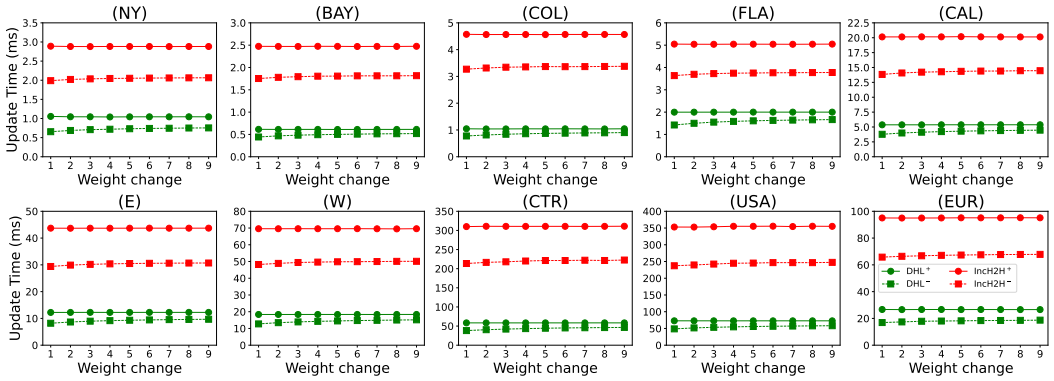


Fig. 5. Maintenance performance under varying edge weights for both decrease and increase case on all datasets.

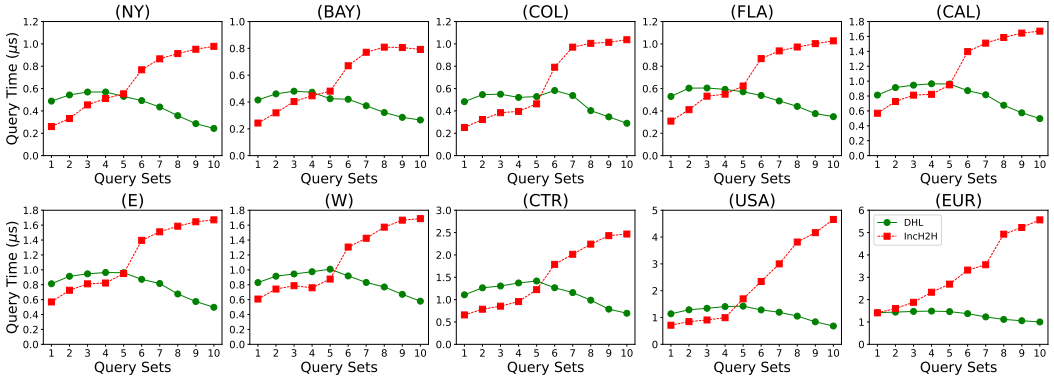


Fig. 6. Query performance for 10 sets of query pairs with varying distances on all datasets.

7.2 Update Time with Varying Weights

Following [18, 26], we randomly sampled 9 batches, each containing 1,000 edges. We first increase weights of updates (a, b, ω) in batch t to $(t+1) \times \omega$ and then restore their weights to original i.e., ω to test the performance of weight increase and decrease case, respectively. Figure 5 shows the average update time of our algorithms DHL$^+$, DHL$^-$ and the state-of-the-art algorithms IncH2H$^+$, IncH2H$^-$. We can see that DHL$^+$ and DHL$^-$ consume significantly less time to update the labelling compared to IncH2H$^+$ and IncH2H$^-$. This is because DHL has significantly reduced labelling sizes compared to IncH2H. As a result, the percentages of affected labels maintained by our algorithms are much smaller than IncH2H, as shown in Table 3.

7.3 Scalability Test

We test the scalability of our methods DHL$^+$ and DHL$^-$ by randomly sampling 5,000 updates for each network and processing them in batches of sizes $\{5, 10\ 15, 20, 25, 30, 35, 40, 45, 50\} \times 10^2$. Figure

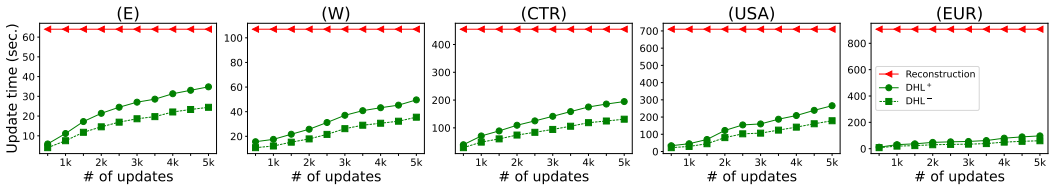


Fig. 7. Maintenance performance against reconstruction time, for batches of updates of varying sizes.

7 presents the results for the large 6 datasets E, W, CTR, USA and EUR, while the results for other datasets are omitted as their trends are similar. We process each group with weight increase first and then with weight decrease, which is compared with the time taken by DHL to construct the labelling from scratch. We see that, even for the largest group with 5,000 updates, the update times of DHL$^+$ and DHL$^-$ are significantly less than reconstruction.

8 Extensions

We discuss how our solution can be extended to directed graphs and to edge/node insertions/deletions, and the boundedness.

Directed Road Networks. Our method can be easily extended to the directed version of dynamic road networks. We can create *forward* and *reverse* labels for each vertex $v \in V(G)$ to store distances from both directions when constructing the labelling L. This may be carried out by running Algorithm 1 for both forward and reverse directions. Then, we can use DHL$^-$ and DHL$^+$ twice to maintain L, once on the forward labels and again on the backward search. For directed versions of dynamic road networks, existing labelling-based methods require increased memory to store precomputed labels. However, road networks are often nearly undirected, with a few notable exceptions (e.g., Stockholm). In such cases, two distances stored within each label are often identical. This raises two intriguing possibilities for future research: (1) how to exploit this symmetry to reduce the size of labels in directed road networks, and (2) how to avoid performing separate distance calculations for each direction during both the construction and updating of labels.

Edge/Vertex Insertion/Deletion. In practice, new roads are rarely created and old roads are rarely deconstructed. Thus, the structure of a road network is considered to be intact. As a result, structural changes such as edge/vertex insertion and deletion are extremely infrequent on road networks. Previous studies address scenarios related to such changes [17, 25, 26]. Similarly, our algorithms consider such changes in the context of DHL as follows. Edge deletions can be handled by increasing the weights of the deleted edges to ∞ and similarly a vertex deletion can be handled by increasing the weights of its adjacent edges to ∞. For edge insertions, we first update the query hierarchy H_Q by identifying the largest affected induced subgraph and recursively repartitioning it to fix the affected nodes in H_Q. Then, based on the ordering induced by H_Q, we fix the update hierarchy H_U by invoking the algorithm of [18]. Accordingly, we compute new labels of vertices using Algorithm 1.

Boundedness. Compared to IncH2H$^-$ and IncH2H$^+$ which are *relative bounded* and *relative sub-bounded* [26], respectively, DHL$^-$ is *relative bounded* and DHL$^+$ is not *relative subbounded* because Algorithm 5 may recompute label entries if the length of a shortest path increases, even if other shortest paths of the same length remain. IncH2H addresses the challenge by tracking the *support* of labels — a measure conceptually similar to the number of shortest paths — and only updating label entries when the support drops to zero. While this approach reduces unnecessary recomputations, it also increases the index size and adds extra maintenance time. This trade-off would typically be

beneficial in cases where there are a large number of unnecessary label recomputations. However, in road networks, shortest paths are often unique, resulting in a low fraction of unnecessary label recomputations. To avoid overhead, we chose not to maintain additional support information. Our experiments confirmed this decision: most distance recomputations were necessary for the networks analyzed.

9 Conclusion

We critically examined the limitations of existing state-of-the-art solutions for efficiently answering distance queries on dynamic road networks. To address these limitations, we propose a novel solution called Dual-Hierarchy Labelling (DHL). Our solution integrates two hierarchies of distinct yet complementary data structures designed to enhance query efficiency while minimizing the maintenance time required for label updates. DHL leverages these hierarchies to balance the trade-off between fast query response times and efficient update processes, which is a persistent challenge in dynamic settings. The experimental results validate the effectiveness of DHL: our approach consistently produces smaller index sizes and achieves faster construction times compared to current state-of-the-art methods. These results highlight the practical value of DHL in real-world applications, demonstrating its potential for scalable and efficient deployment in large-scale, dynamic road networks. This work sets the stage for further exploration into hybrid data structures that can optimize both query speed and maintenance efficiency in increasingly complex network environments.

Acknowledgments

This research was supported partially by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (project DP210102273).

References

[1] PTV AG. [n.d.]. Western europe dataset. http://www.ptv.de

[2] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. 2016. Route planning in transportation networks. *Algorithm engineering: Selected results and surveys* (2016), 19–80.

[3] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. 2010. Preprocessing speed-up techniques is hard. In *International Conference on Algorithms and Complexity*. Springer, 359–370.

[4] Anne Berry, Pinar Heggernes, and Genevieve Simonet. 2003. The minimum degree heuristic and the minimal triangulation process. In *Graph-Theoretic Concepts in Computer Science: 29th International Workshop, WG 2003. Elspeet, The Netherlands, June 19-21, 2003. Revised Papers 29*. Springer, 58–70.

[5] Zitong Chen, Ada Wai-Chee Fu, Minhao Jiang, Eric Lo, and Pengfei Zhang. 2021. P2h: Efficient distance querying on road networks by projected vertex separators. In *Proceedings of the 2021 International Conference on Management of Data*. 313–325.

[6] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[7] Camil Demetrescu, Andrew V Goldberg, and David S Johnson. 2009. *The shortest path problem: Ninth DIMACS implementation challenge*. Vol. 74. American Mathematical Soc.

[8] DongKai Fan and Ping Shi. 2010. Improvement of Dijkstra's algorithm and its application in route planning. In *2010 seventh international conference on fuzzy systems and knowledge discovery*, Vol. 4. IEEE, 1901–1904.

[9] Muhammad Farhan, Henning Koehler, Robert Ohms, and Qing Wang. 2023. Hierarchical Cut Labelling–Scaling Up Distance Queries on Road Networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[10] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International workshop on experimental and efficient algorithms*. Springer, 319–333.

[11] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact routing in large road networks using contraction hierarchies. *Transportation Science* 46, 3 (2012), 388–404.
[12] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory. In *SODA*, Vol. 5. 156–165.
[13] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.
[14] Shuai Huang, Yong Wang, Tianyu Zhao, and Guoliang Li. 2021. A learning-based method for computing shortest path distances on road networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 360–371.
[15] Henning Koehler, Muhammad Farhan, and Qing Wang. 2025. Stable Tree Labelling for Accelerating Distance Queries on Dynamic Road Networks. In *Then 28th International Conference on Extending Database Technology (EDBT)*.
[16] Hans-Peter Kriegel, Peer Kröger, Peter Kunath, Matthias Renz, and Tim Schmidt. 2007. Proximity queries in large traffic networks. In *Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*. 1–8.
[17] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data*. 709–724.
[18] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. *Proceedings of the VLDB Endowment* 13, 5 (2020), 602–615.
[19] Ira Sheldon Pohl. 1969. *Bi-Directional and Heuristic Search in Path Problems*. Ph.D. Dissertation. Stanford, CA, USA. AAI7001588.
[20] Robert Endre Tarjan. 1983. *Data structures and network algorithms*. SIAM.
[21] Victor Junqiu Wei, Raymond Chi-Wing Wong, and Cheng Long. 2020. Architecture-intact oracle for fastest path and time queries on dynamic spatial networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1841–1856.
[22] Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, Andy Diwen Zhu, and Shuigeng Zhou. 2012. Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation. *Proceedings of the VLDB Endowment* 5, 5 (2012).
[23] Pranali Yawalkar and Sayan Ranu. 2019. Route recommendations on road networks for arbitrary user preference functions. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 602–613.
[24] Mengxuan Zhang. 2021. *Efficient shortest path query processing in dynamic road networks*. Ph.D. Dissertation.
[25] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. 2021. Dynamic hub labeling for road networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 336–347.
[26] Yikai Zhang and Jeffrey Xu Yu. 2022. Relative Subboundedness of Contraction Hierarchy and Hierarchical 2-Hop Index in Dynamic Road Networks. In *Proceedings of the 2022 International Conference on Management of Data*. 1992–2005.
[27] Yu Zheng, Yukun Chen, Quannan Li, Xing Xie, and Wei-Ying Ma. 2010. Understanding transportation modes based on GPS data for web applications. *ACM Transactions on the Web (TWEB)* 4, 1 (2010), 1–36.
[28] Andy Diwen Zhu, Hui Ma, Xiaokui Xiao, Siqiang Luo, Youze Tang, and Shuigeng Zhou. 2013. Shortest path and distance queries on road networks: towards bridging theory and practice. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 857–868.

Received July 2024; revised September 2024; accepted November 2024