Fast Landmark Reconfiguration for Highway Cover Indexes

David Coudert*
Université Côte d'Azur, CNRS, Inria, I3S
Sophia Antipolis, France
david.coudert@inria.fr

Andrea D'Ascenzo*†
Gran Sasso Science Institute
L'Aquila, Italy
andrea.dascenzo@gssi.it

Mattia D'Emidio*
DISIM and Centre of Excellence EX-EMERGE,
University of L'Aquila
L'Aquila, Italy
mattia.demidio@univaq.it

Giuseppe F. Italiano*
Luiss University
Rome, Italy
gitaliano@luiss.it

Abstract

The *highway cover labeling* (HCL) is an indexing method for weighted digraphs that enables fast queries on important graph properties such as distances and constrained shortest paths. Originally introduced by [*Farhan et al., EDBT 2019*], the HCL has gained popularity in the field of large-scale graph mining due to its efficiency: in fact, an HCL index can be computed with reasonable preprocessing computational effort, answers queries in near real time, and has low space overhead – even for graphs with tens of millions of arcs. Such a remarkable performance is obtained by carefully selecting, during the preprocessing phase, a subset of the vertices of the input graph, called *landmarks*, and by computing a suitable collection of paths and distances from/to landmarks to be used to reconstruct graph properties of interest upon query.

Recent work has shown how to adapt the HCL method to dynamic graphs, allowing updates to the graph topology without full recomputation, by identifying and updating, efficiently, the portion of the index that is altered by a modification. However, no prior dynamic methods, to efficiently handle changes to the landmark set itself, are known. This paper addresses this limitation by introducing two new dynamic algorithms specifically designed for landmark updates. Our experiments show that these methods can update an HCL index in seconds even in massive graphs – achieving speedups of several orders of magnitude over full reprocessing and enabling the use of HCL indices in fully dynamic settings.

Keywords

Graph Algorithms, Shortest Paths, Graph Databases

1 Introduction

Answering shortest-path queries is a fundamental operation on graph data, as countless optimization and information-retrieval tasks rely on fast and effective responses. Examples include routing in multihop networks, trip planning, web analytics, and graph database management [7, 19, 42]. Consequently, extensive research has focused on designing efficient algorithms to compute shortest paths and related properties (e.g., distances, centralities, constrained paths) [7, 22, 28, 43]. A key challenge, studied over the past two decades in this area of research, has been the scalability of classical polynomial-time methods, which often perform poorly either on large graphs or under high query loads [2, 19, 30]. Many standard well-established algorithms, in fact, yield average query times impractical for applications involving millions of vertices or thousands of queries per minute [2, 21, 30]. To overcome this, considerable effort has been put into developing heuristic methods that accelerate baseline algorithms, achieving query times are suitable for large-scale, real-time use [2, 21, 22, 30]. A particularly successful family of approaches is that of *labeling schemes*, which rely on a two-phase process: (i) a preprocessing step builds a compact data structure called a *labeling* (or *index*), assigning *labels* to vertices that encode selected paths and/or distances; (ii) a specialized query routine exploits this structure to answer shortest-path queries orders of magnitude faster than traditional methods [2, 7, 19, 21, 22, 30].

A notable recent advancement in this family is the *Highway Cover Labeling* (*HCL*) method by Farhan et al. [30], which gained significant attention in path mining because of its ability to balance query efficiency, index compactness, and scalability. Its core idea is a preprocessing strategy that selects a subset R of vertices of the graph, called *landmarks*, and constructs an *HCL index*. This index consists of a small matrix (storing distances between landmarks) and a collection of vertex labels (storing paths to landmarks and their distances). This index can then be exploited to return extremely fast a shortest path for a given vertex pair. This is achieved by first extracting from the index, via an appropriate query routine, a landmark-constrained shortest path – that is, a shortest path passing through at least one landmark – and then by using such information as a powerful heuristic to guide a lightweight local search that identifies the exact shortest path. Thanks to an effective pruning mechanism during construction, HCL avoids redundant storage of paths traversing multiple landmarks, keeping the index compact while fully covering relevant paths. This design translates into excellent practical performance: empirical studies have demonstrated that HCL produces compact indexes requiring only a few gigabytes of memory, built within hours, even for graphs with billions of edges [30]. At query time, it enables response times several orders of magnitude faster than traditional algorithms, and it outperforms other indexing methods for shortest paths in terms of query speed, preprocessing effort, and space overhead [2, 13, 21].

Beyond standard shortest-path queries, the flexibility of HCL has recently been showcased in more complex tasks. In particular, Coudert et al. [13] extended the HCL framework to support *shortest beer path* queries – an important generalization of shortest paths that requires computing optimal paths connecting vertex

*All authors contributed equally to this research.
†Corresponding author. Part of the author's work was carried out during his research tenure at Luiss University in Rome, Italy.

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.18

pairs and passing through designated vertices called *beer vertices*. As well documented in the literature [5, 6], answering quickly to such queries at scale is crucial for many applications in logistics, transportation, and telecommunications. For example, shortest beer paths are used for adaptive route planning (e.g. when, on a given trip, users must stop at a gas station), for vehicle routing (e.g., transporting packages that require an inspection stop), or for data transmission (e.g. to route packets through specific network nodes). The extension of [13] showed that shortest beer paths and distances can be computed efficiently by precomputing and querying, for landmark-constrained shortest paths, a suitably precomputed HCL index. Thus, the extended framework preserves the performance of the original HCL while enabling more realistic decision-making in practical domains [5, 13]. Besides speed and applicability, HCL has also proven to be adaptable to dynamic graph settings. Unlike many preprocessing-based methods that assume static inputs, recent work has introduced algorithms to update the HCL index under edge or vertex insertions and deletions [28, 29, 31]. These dynamic algorithms avoid the time-consuming full recomputation, making HCL usable in real-world scenarios where graph data evolve continuously. However, efficiently updating an HCL index when the landmark set changes remains an open problem. Recent works [13, 29, 31] emphasize that supporting landmark modifications is critical, as they strongly affect both the effectiveness and applicability of the index. In fact, in standard shortest-path queries, landmark changes impact index size and performance [28, 29]. In more advanced scenarios, such as shortest beer path queries, landmarks correspond to beer vertices – entities that are inherently dynamic (e.g., stores with fluctuating availability or infrastructure nodes like routers/servers that may go temporarily offline) [13]. Therefore, updating the index under landmark changes is essential to preserve query correctness and real-time responsiveness [13, 16]. Figure 1 illustrates a graph and the effect of changing landmarks on an HCL index. We refer the reader to [30] and the next section for details on the shown data structures.

Our Contribution. In this paper, we tackle the open problem identified in [13, 29, 31] by introducing the first dynamic algorithms for updating HCL indexes when the landmark set changes. Specifically, we propose algorithms UPGRADE-LMK and DOWNGRADE-LMK that efficiently maintain the index when landmarks are added or removed. We evaluate our methods through an extensive experimental evaluation on large-scale, real-world and synthetic graphs. Our results show that they outperform existing solutions by achieving update times up to several orders of magnitude faster than full recomputation, while preserving the low memory overhead and query performance of HCL. Our experiments also show that the HCL, combined with our dynamic algorithms, provides superior query performance also compared to other strategies – not based on the HCL construction (e.g., [43]). – for retrieving landmark-constrained shortest paths under dynamically changing landmark sets.

1.1 Related Work

The design of scalable frameworks for computing graph properties has been an active research area over the past two decades, driven by the growing need to analyze massive real-world networks. Many practical approaches combine classical speed-up techniques – such as preprocessing, sampling, and parallelization – to accelerate baseline algorithms [2, 8, 21, 22, 46]. Among these, preprocessing-based methods have been proven to be especially effective, enabling sub-millisecond query times on large graphs while maintaining reasonable preprocessing cost and memory overhead for a variety of graph properties, including shortest paths, distances, and centrality measures [2, 17, 47]. Within this area, labeling-based techniques have gained unprecedented popularity due to their performance and simplicity of implementation [1, 2, 12, 13, 21, 22, 40].

A prominent example is the hub-labeling approach, originally introduced for reachability queries by Cohen et al. [12], and later extended to support various graph mining queries at scale with strong practical performance. These include shortest paths [21, 30], shortest path counts [47], top-k distances [1], K-reach [42], shortest cycles [32], and minimum weight constrained paths [13, 40]. For distance and shortest-path queries, the HCL method [30] offers an excellent trade-off between preprocessing time, index size, and query speed. Avoiding to store shortest paths passing through multiple landmarks makes it more space-efficient than alternatives (e.g. Transit Node Routing) while at the same time faster at query answering than approaches with lower preprocessing effort (e.g. Contraction Hierarchies) [2, 4, 30, 37, 45].

The problem of computing *shortest beer paths* was recently formalized by Bacic et al. [5]. Given a source and target vertices, and a vertex category (e.g., supermarkets, routers), the goal is to find a shortest path passing through at least one such vertex. This is a special case of the *generalized shortest-path* problem which, for a given vertex pair, asks to determine a connecting path of minimum total cost that visits at least one location from each of a set of specified location categories in a specified order [38, 43]. Several works followed that of Basic et al. [5], mostly focusing on methods to efficiently compute shortest beer paths (and beer distances) in special graph classes with various performance guarantees (see [6, 16, 34, 35] and references therein). For general graphs, instead, the HCL-based approach of Coudert et al. [13] currently offers the most scalable and efficient solution, achieving sub-millisecond query times. In addition to the above, the notion of beer distance has recently been generalized in [9].

A related problem in this context is that of answering point-of-interest (POI) queries: together with an input weighted digraph, a set of POI locations is given that lie along arcs of the graph and the objective is to find the cheapest paths that traverse at least a POI arc [23]. Although easily adaptable to shortest beer path queries, with some polynomial-time computational overhead, methods for answering POI queries in the literature are significantly slower than the HCL-based solution of [13], especially on large graphs.

Nearly all studies on methods that achieve fast responses to graph mining queries through preprocessing have been followed by research into the design of corresponding dynamic algorithms, to maintain precomputed data structures efficiently under graph updates, enabling fast query answering even when the input changes over time. Such algorithms typically avoid costly full recomputation by amortizing update costs. Examples include dynamic methods for 2-hop covers [3, 15], matching and vertex cover problems [8], point-to-point shortest paths [20], k-2-hop cover labelings [17, 18], public transit labelings [25], and betweenness centrality [36]. For HCL, dynamic algorithms that support topological updates (vertex and edge insertions/deletions) have been introduced in [28, 29, 31], but these approaches do not support changes to the landmark set, a crucial feature for adapting HCL to entirely dynamic scenarios such as shortest beer path queries or evolving query patterns.

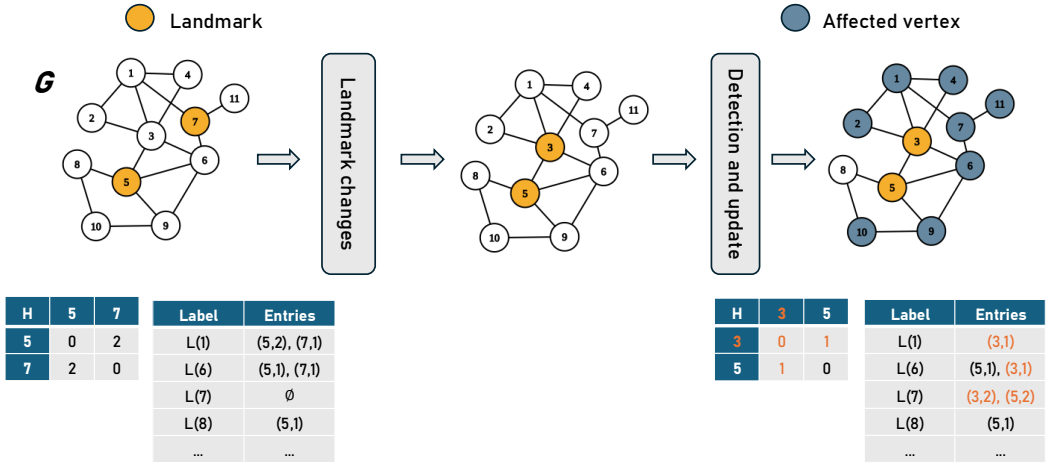

Figure 1: Effects of modifying the landmark set from $R = \{5, 7\}$ to $R' = \{3, 5\}$ for an HCL index $I = (H, L)$ covering a graph G: (left) the index with landmarks R (H is the highway, storing the distance decoding function δ_H); (right) the updated index with landmarks R' (changes are highlighted in orange). The labeling L is shown for a subset of the vertices for readability.

2 Background

We are given a weighted graph $G = (V, E, \omega)$ where V is a set of n vertices, $E \subseteq V \times V$ is a set of m edges and ω is a *weight function* $\omega : E \mapsto \mathbb{R}^+$ that assigns a positive, real value to each edge of G. A *path* $P = (s = v_1, v_2, \ldots, t = v_\eta)$ in G, connecting a pair of vertices $s, t \in V$, is a sequence of η vertices such that $(v_i, v_{i+1}) \in E$ for all $i \in [1, \eta - 1]$. The *weight* $\omega(P)$ of a path P is the sum of the weights of its edges. A *shortest path*, for a pair $s, t \in V$, is a path having minimum weight among all paths in G that connect s and t. The *distance* $d(s, t)$ from s to t is the weight of a shortest path connecting s and t. Given a vertex $v \in V$, notation $N(v) = \{u \in V \mid (v, u) \in E\}$ identifies the set of neighbors of v. For simplicity and w.l.o.g., we assume graphs are undirected. Nonetheless, all methods described in this paper can be adapted to directed graphs, by considering edge orientations and partitions of the neighbors of a vertex into outgoing and incoming neighbors, as in other works on labelings [2, 12, 15].

Highway Cover Labeling (HCL). The HCL framework is a customization of the 2-hop-cover labeling [12], a well-known method for shortest paths and distances [2, 21], introduced in [30] with the goal of reducing the preprocessing time and space requirements of the original method while preserving or slightly increasing the offered average query time. In detail, given a graph $G = (V, E, \omega)$, and a subset of its vertices $R \subseteq V$ (called *landmarks*), a *highway* $H = (R, \delta_H)$ of G over landmarks $R \subseteq V$ is a pair (R, δ_H), where δ_H is a *distance decoding function*, i.e. a function $\delta_H : R \times R \mapsto \mathbb{R}^+$ such that, for every pair $r_i, r_j \in R$, we have $\delta_H(r_i, r_j) = d(r_i, r_j)$. Given a vertex $r \in R \subseteq V$ and two vertices $s, t \in V \setminus R$, an *r-constrained shortest path* from s to t in G is a path that passes through r and has minimum weight. The weight of an r-constrained shortest path is called *r-constrained distance*. Let $H = (R, \delta_H)$ be a highway for a graph G with landmarks $R \subseteq V$. A *highway cover labeling* index (or simply *highway labeling*) of G is a pair $I = (H = (R, \delta_H), L)$ where H is a highway and L is a *(landmark) labeling* (or simply *labeling*) i.e. a collection $L = \{L(v)\}_{v \in V}$ such that a *label* $L(v)$ stores *(label) entries* in the form $(r_i, d(r_i, v))$ where $r_i \in R$. Algorithm BUILDHCL, proposed in [30] and extended to weighted graphs in [13], is the reference method to build HCL indexes. Observe that an HCL index, as many other shortest-path based data structures [15], can be adapted to retrieve paths by adding a third field to each entry, storing a predecessor vertex. For simplicity, we focus on distance retrieval only, as in most studies on indexed data structures [2, 13, 30].

Applications of the HCL. The original motivation behind the design of the HCL method was to obtain very fast and scalable responses to shortest-path queries [30]. In fact, if properly computed, an HCL index of a graph $G = (V, E, \omega)$, with landmark set R, allows to determine an upper bound $\widetilde{d}(s, t)$ on the distance, for each pair $s, t \in V$, by computing $\widetilde{d}(s, t) = \textsc{query}(s, t, H, L)$, where $\textsc{query}(s, t, H, L)$ is a query routine that returns the minimum r-constrained distances for all $r \in R$, defined as:

$$\textsc{query}(s, t, H, L) = \min_{\substack{\forall (r_i, d(r_i,s)) \in L(s) \\ \forall (r_j, d(r_j,t)) \in L(t)}} \{d(r_i, s) + \delta_H(r_i, r_j) + d(r_j, t)\}.$$

Such value, often referred to as *landmark-constrained distance*, corresponds to the weight of the shortest of the paths from s to t that pass through any pair of landmarks r_i and r_j. By combining the upper bound with a distance-bounded bidirectional search on the subgraph of G induced by vertices in $V \setminus R$, one can retrieve the distance between s and t within microseconds, even at large scale. Specifically, an HCL index is said to satisfy the *highway cover property* when, for any two vertices $s, t \in V \setminus R$, and for any $r \in R$, the r-constrained distance for s and t can be determined by accessing the function δ_H and labels $L(s)$ and $L(t)$ only. In this case, the highway labeling is said to *cover* the graph (symmetrically, the graph is said to be *covered* by the labeling). In more details, this highway cover property is guaranteed when: (i) for every pair $r_i, r_j \in R$, we have $\delta_H(r_i, r_j) = d(r_i, r_j)$; (ii) for any two vertices $s, t \in V \setminus R$ and for any $r \in R$, there exist $(r_i, d(r_i, s)) \in L(s)$ and $(r_j, d(r_j, t)) \in L(t)$ such that $r_i \in P_{rs}$, for some shortest path P_{rs} from r to s, and $r_j \in P_{rt}$, for some shortest path P_{rt} from r to t, where r_i and r_j may be equal to r. Therefore, given any two vertices s and t, an HCL index built as above allows to find any r-constrained distance by computing $d(r_i, s) + \delta_H(r_i, r_j) + d(r_j, t)$ where $r_i = r$ or $r_j = r$ and for $(r_i, d(r_i, s)) \in L(s)$ and $(r_j, d(r_j, t)) \in L(t)$. Whenever, for a given landmark r, we have $(r, d(r, t)) \in L(v)$ for a vertex $v \in V$, we say v is covered by r or r covers v. We refer the interested reader to [30] for full details about the HCL framework.

A very recent application of the HCL is answering queries on *shortest beer paths* [13], a problem concerned with the identification of optimal detours of significant practical relevance, formalized and rigorously investigated for the first time in [5] through the concept of beer graphs. A *beer graph* is a weighted graph $G = (V, E, \omega)$ with a weight function $\omega : E \mapsto \mathbb{R}^+$ and a set of special vertices $B \subseteq V$, called *beer vertices*. A *beer path*, between two vertices s and t of a beer graph, is any path of G from s to t that visits at least one vertex in B while a *shortest beer path* for two vertices s and t is a beer path having minimum total weight (called *beer distance*). Although a beer path may be a non-simple path (i.e. it might self-intersect), it can be easily shown that any shortest beer path, for a pair of vertices s, t, always consists of two shortest paths: one from s to a beer vertex, say w, and one from w to t. In other terms, a beer distance is always equal to the minimum, overall beer vertices $w_i \in B$, of the sums of the shortest path distances from s to w_i and from w_i to t. This characterization provides a baseline algorithm for computing shortest beer paths and beer distances for a pair s, t, that is: grow two shortest-path trees rooted, respectively, at s and t and select the beer vertex that minimizes the sum of the distances from s and to t [5, 6]. Currently, the fastest known approach to answer queries on shortest beer paths and distances at scale on general digraphs is that in [13]. Their method builds an HCL index (H, L) where the beer vertex set B is used as the landmark set R. Using this index, beer distances can be computed without any graph traversal at query time: given a vertex pair (s, t), the beer distance is retrieved as the landmark-constrained distance $\textsc{query}(s, t, H, L)$. Figure 1 (left) visually illustrates how beer distances can be efficiently derived from an HCL index when beer vertices are used as landmarks.

3 Dynamic Algorithms

In this section, we design two dynamic algorithms to update a given HCL index $I = (H, L)$, covering a graph G with landmark set $R \subseteq V$, so that the resulting index covers G through a different set of landmarks R'. Specifically, we present algorithm UPGRADE-LMK (DOWNGRADE-LMK, respectively) that updates an HCL index when the new set R' is larger (smaller, respectively) than R by one landmark.

3.1 Algorithm UPGRADE-LMK

The strategy of algorithm UPGRADE-LMK is to enrich a given HCL index (both the highway and the labeling) with enough information to ensure that the highway cover property holds with the enlarged set of landmarks. For efficiency, this enrichment is performed by leveraging the data already stored in the index. The first step is to determine the distance from the newly added landmark, say r, to other landmarks, to be added to H: this is either retrieved directly from $L(r)$ (for landmarks that cover r) or derived from the highway H (otherwise). Next, a traversal of G is initiated from r with a twofold objective: (i) to determine any landmark $r' \in R \setminus \{r\}$ that covered r before the update (i.e., for which there exists a shortest path from r to r' not intersecting other landmarks); (ii) to find vertices that are covered by the new landmark r (any vertex u such that a shortest path from r to u, not traversing other landmarks, exists). For efficiency, this traversal discovers only r-constrained shortest paths, by suitably executing the QUERY routine on the HCL index and pruning whenever a non-shortest path is found. Finally, for each vertex v covered by r, the algorithm checks whether the entries associated with other landmarks must be removed from $L(v)$, i.e., whether a landmark $r' \in R \setminus \{r\}$ still covers v after the addition of r to R. The latter occurs when there exists, in the graph, a shortest path from r' to v that does not traverse r. To test this property, vertices covered by r' are analyzed in order of distance from r' (for efficiency, they are tracked during the traversal of the graph in the previous phase).

Algorithm 1: Algorithm UPGRADE-LMK.

Input: Graph $G = (V, E, \omega)$, HCL index $I = (H = (R, \delta_H), L)$ covering G with landmarks $R \subseteq V$, a vertex $r \in V \setminus R$.
Output: HCL index $I = (H = (R', \delta_H), L)$ covering G with landmarks $R' = R \cup \{r\}$

1 **foreach** $(r', \delta) \in L(r)$ **do**
2 $\quad \delta_H(r, r') \leftarrow \delta$; ▷Update H to include each r' that covers r
3 $\widetilde{R} \leftarrow \{r' \in R \mid (r', \delta) \in L(r)\}$; ▷Landmarks that cover r
4 **foreach** $r' \in R \setminus \widetilde{R}$ **do**
5 $\quad \delta_H(r, r') \leftarrow \min_{\hat{r} \in \widetilde{R}} \{\delta_H(r, \hat{r}) + \delta_H(\hat{r}, r')\}$;
6 $L(r) \leftarrow \emptyset$; ▷Reset $L(r)$
7 **foreach** $r' \in R$ **do** REACHED-VER$[r'] \leftarrow \emptyset$;
8 $R' \leftarrow R \cup \{r\}$;
9 REACHED-LAN $\leftarrow \emptyset$;
10 $Q \leftarrow \emptyset$; ▷Empty priority queue, e.g. min-heap
11 $Q.enqueue(r, 0)$; ▷Add r to Q with priority $d(r, r) = 0$
12 **foreach** $v \in V \setminus R'$ **do**
13 $\quad P[v] \leftarrow \infty$; ▷Init distance from r to other vertices
14 $P[r] \leftarrow 0$;
15 **while** $Q \neq \emptyset$ **do**
16 $\quad (u, \delta) \leftarrow Q.dequeueMin()$; ▷Here $P[u]$ equals δ
17 $\quad$ **if** $u \in R' \setminus \{r\}$ **then**
18 $\quad\quad$ Add u to REACHED-LAN;
19 $\quad\quad$ **continue**;
20 $\quad$ **if** $\textsc{query}(r, u, H, L) < \delta$ **then continue**;
21 $\quad$ **foreach** $(r', \delta') \in L(u)$ **do** Add u to REACHED-VER$[r']$;
22 $\quad$ Add (r, δ) to $L(u)$;
23 $\quad$ **foreach** $w \in N(u) : P[w] > P[u] + \omega(u, w)$ **do**
24 $\quad\quad$ **if** $P[w] = \infty$ **then** $Q.enqueue(w, P[u] + \omega(u, w))$;
25 $\quad\quad$ **else** $Q.decreaseKey(w, P[u] + \omega(u, w))$;
26 $\quad\quad$ $P[w] \leftarrow P[u] + \omega(u, w)$;
27 **foreach** $r' \in$ *REACHED-LAN* **do**
28 $\quad Q \leftarrow \emptyset$; ▷Empty priority queue, e.g. min-heap
29 $\quad$ **foreach** $x \in$ *REACHED-VER*$[r']$ **do** ▷$x \in$ reached-ver$[r']$
30 $\quad\quad$ Find (r', ρ) in $L(x)$; ▷$\implies$ x covered by r'
31 $\quad\quad$ $Q.enqueue(x, \rho)$;
32 $\quad$ **while** $Q \neq \emptyset$ **do**
33 $\quad\quad$ $(u, \delta) \leftarrow Q.dequeueMin()$; ▷Here δ equals $d(r', u)$
34 $\quad\quad$ **if** $\nexists\, w \in N(u) : (r', \rho) \in L(w) \wedge \delta = \rho + \omega(u, w)$ **then** $L(u) \leftarrow \{(l, \sigma) \in L(u) \mid l \neq r'\}$;

In more detail, Algorithm UPGRADE-LMK, whose pseudocode is given in Algorithm 1, takes as input a weighted graph $G = (V, E, \omega)$, an HCL index $I = (H = (R, \delta_H), L)$ with landmark set $R \subseteq V$, and a vertex $r \in V \setminus R$ to be *promoted* as a new landmark (hence added to R). It computes a highway labeling $I = (H = (R', \delta_H), L)$ such that H is a highway for the set of landmarks $R' = R \cup \{r\}$, L is a landmark labeling, and I covers G with landmark set R'. Specifically, the algorithm works as follows: first, in Lines (1)–(5) we update the highway H by storing in δ_H the distance from r to every other landmark in R. On the one hand, for each landmark r' that covers r, we obtain the distance to r from entry $(r', \delta) \in L(r)$. On the other hand, for

each landmark r' that does not cover r, we compute $d(r, r')$ as $\min\{\delta_H(r, \hat{r}) + \delta_H(\hat{r}, r')\}$ over all $\hat{r} \in \widetilde{R} = \{r' \in R \mid (r', \delta) \in L(r)\}$. Note that no graph search is required to update H. Next, all entries in $L(r)$ are deleted and, to check whether a vertex $v \in V \setminus R'$ is covered by the new landmark r, we perform a Dijkstra-like search rooted at r, using a min-priority queue Q (with initial priority and distance of r set to 0). When a vertex u is found at distance δ from r (i.e., when u is dequeued from Q with priority δ), the visit is pruned and no entry is added to $L(u)$ in two cases: (i) u is a landmark different from r; or (ii) δ is larger than the value returned by the routine QUERY for (r, u) on the HCL index. If neither condition holds, then it follows that there exists a shortest path from r to u: (i) that does not traverse any landmark $r' \neq r$; and (ii) whose weight is smaller than all values $d(r, s) + d(s, u)$ with $(s, d(r, s)) \in L(r)$ and $(s, d(s, u)) \in L(u)$, i.e., smaller than any path obtained by concatenating shortest paths through s. Therefore, entry (r, δ) is added to $L(u)$, and r covers non-landmark vertex u. Simultaneously, we add u to each set REACHED-VER$[r']$ (initially empty) for every $(r', \delta') \in L(u)$ with $r' \neq r$. This tracks the landmarks that covered u before adding (r, δ) to $L(u)$. The visit then continues by enqueueing neighbors of u or decreasing their priority in Q. Finally, after the Dijkstra-like visit, we post-process each vertex $x \in$ REACHED-VER$[r']$, for each $r' \in$ REACHED-LAN, to preserve the *minimality* of the HCL index and the *order-invariance* of UPGRADE-LMK. These two properties, defined by Farhan et al. [30], positively impact query performance and space usage. Minimality ensures that no entry in the index can be removed without breaking the highway cover property, while order-invariance guarantees that the index size does not depend on the order of landmarks in the labeling process. To enforce minimality and order-invariance, we delete *superfluous* label entries from each $L(x)$, i.e., entries removable without breaking the highway cover property. In other words, these are entries that would inflate the index size compared to recomputing it from scratch or considering a different labeling order (cf. Line 27). Formally, an entry $(r', \rho) \in L(x)$ is superfluous if and only if: (i) there exists at least one shortest path from r' to x not traversing other landmarks before adding r (so r' covered x); and (ii) all shortest paths from r' to x pass through the new landmark r. To identify and remove superfluous entries, we proceed as follows: for each $r' \in$ REACHED-LAN and each vertex $x \in$ REACHED-VER$[r']$, we add x to a priority queue Q with priority equal to its distance from r'. We then process vertices of REACHED-VER$[r']$ in order of distance from r'. If x has no neighbor closer to r' that is covered by r', then any entry (r', σ) can be safely removed from $L(x)$; otherwise, the entry remains in the index. We are now able to prove the correctness of our update routine.

THEOREM 3.1. *Algorithm 1 computes an HCL index $I = (H = (R', \delta_H), L)$ that covers G with landmark set $R' = R \cup \{r\}$.*

PROOF. At the beginning of the algorithm, $I = (H = (R, \delta_H), L)$ covers G. When r is added to the set of landmarks, to preserve the highway cover property the algorithm enforces two conditions at termination: (i) δ_H is a distance decoding function for the landmark set, i.e., $\delta_H(r, r') = d(r, r')$ for all $r' \in R \cup \{r\}$; and (ii) for any two vertices $s, t \in V \setminus R$ and for any $r \in R$, labels $L(s)$ and $L(t)$ store entries sufficient to compute the r-constrained distance when combined with δ_H. We now show that if $I = (H = (R, \delta_H), L)$ covers G before execution, then (i) and (ii) hold at termination.

Property (i). Suppose, by contradiction, that there exists a vertex $r' \in R \cup \{r\}$ such that $\delta_H(r, r') \neq d(r, r')$. If r' covered r before execution, this contradicts the fact that $d(r, r')$ is stored in $L(r)$. Otherwise, if r' did not cover r, then since (H, L) covered G, it follows that all shortest paths between r and r' traverse at least one landmark different from r'. Let $\hat{r}$ be the closest such landmark to r in these paths, in terms of distance. By the cover property, $\hat{r} \in L(r)$, and hence $\delta_H(\hat{r}, r) = d(\hat{r}, r)$. By optimal substructure of shortest paths, we have $d(r, r') = d(r, \hat{r}) + d(\hat{r}, r')$. Since $d(\hat{r}, r')$ is stored in $\delta_H(\hat{r}, r')$, Line (5) would have selected $d(r, r')$ as the minimum value for (r, r'). Thus, if $\delta_H(r, r') \neq d(r, r')$, then $\delta_H(\hat{r}, r')$ must be incorrect, contradicting the assumption that (H, L) covers G.

Property (ii). By (i), H is a highway for $R' = R \cup \{r\}$ and G. We must show that for any $r' \in R'$, any r'-constrained distance between vertices $s, t \in V \setminus R'$ can be retrieved from L and H via QUERY. Observe that the only labels affected by Algorithm 1 are those of r and of vertices reached by the Dijkstra-like search. Entries in $L(r)$ are all removed, and after the first dequeue operation, $(r, 0)$ is added to $L(r)$, since r is now a landmark.

Consider a vertex u reached by the search. If $u \in R'$, then it is skipped (except for the base case $u = r$). By (i), the distance between u and r is stored in H. If $u \notin R'$ and is reached by a shortest path of weight δ not traversing other landmarks, then: – If QUERY$(r, u, H, L) < \delta$, it follows that there exists a shorter path from r to u passes through a landmark $r' \neq r$, and pruning is safe. – Otherwise, $\delta = d(r, u)$ and the shortest path does not traverse a different landmark, so entry (r, δ) is correctly added to $L(u)$. Now consider $u \in V$ not reached by the search. Either: – a shortest path from r to u traverses some landmark $r' \neq r$, and then $d(r, u)$ is already stored in (H, L); or – the path visits a vertex w where the search was pruned (Line 20). By optimal substructure, the path through w would be longer than a shortest path, a contradiction. Finally, to complete the proof, we show that in Lines (27)–(34) no entry necessary for the highway cover property is removed. Observe that, for each $r' \in$ REACHED-LAN reached during the Dijkstra-like search, each set REACHED-VER$[r']$ contains vertices v with entries in $L(v)$ for both r' and r. The procedure checks whether (r', ρ) can be deleted from $L(v)$, depending on whether r' still covers v. Vertices are processed in order of distance from r' (i.e. starting with a vertex u whose $(r', \rho) \in L(u)$ is such that ρ is minimum and proceeding in non-decreasing order). We prove by induction on the number of extractions from Q that if (r', ρ) is removed from $L(v)$, the highway cover property remains valid. *Base case.* The first extraction is u, δ with $\delta = d(r', u)$ minimal. If there exists a neighbor $w \in N(u)$ with $(r', \rho) \in L(w)$ and $\rho + 1 = \delta$, then $w \notin$ REACHED-VER$[r']$ (possibly $w = r'$). Hence, a shortest path from r' to u through w not traversing r still exists, so (r', δ) cannot be removed. Otherwise, all r'-constrained shortest paths from r' to u traverse r, so (r', δ) can be safely removed. *Inductive step.* Suppose the property holds after k extractions. For the $(k+1)$-th extraction, the same argument applies: (r', ρ) is removed from $L(u)$ only if no neighbor of u closer to r' is covered by r', i.e., if all shortest paths from r' to u pass through r. Hence, the highway cover property holds at termination. □

In what follows, we show that HCL indices computed by UPGRADE-LMK are minimal and order-invariant.

LEMMA 3.2. *The HCL index $I = (H = (R', \delta_H), L)$ computed by Algorithm 1 satisfies minimality.*

Proof. Suppose there exists a labeling L' covering G with landmarks R', a vertex v, and a landmark $\bar{r} \in R'$ such that $(\bar{r}, \delta) \in L(v)$ but $(\bar{r}, \delta) \notin L'(v)$. Two cases may occur: either $(\bar{r}, \delta)$ was in $L(v)$ before the update, or it was added to L by UPGRADE-LMK. In the first case, $(\bar{r}, \delta) \notin L'(v)$ implies that all shortest paths from $\bar{r}$ to v traverse another landmark other than $\bar{r}$. Let r' be the landmark closest to $\bar{r}$, in terms of distance, on such paths. Since the initial HCL index covered G, r' must be the new landmark r. Then, UPGRADE-LMK adds $\bar{r}$ to REACHED-LAN and v to REACHED-VER $[\bar{r}]$. As all shortest paths from $\bar{r}$ to v traverse a different landmark, Line (34) evaluates to true, removing $(\bar{r}, \delta)$ from $L(v)$, which is a contradiction. In the second case, if $\bar{r} = r$, the absence of $(\bar{r}, \delta)$ in $L'(v)$ would imply that another landmark lies on all shortest paths from r to v. However, UPGRADE-LMK adds entries for r only when shortest paths exist that avoid other landmarks. Combined with Theorem 3.1, this contradicts the assumption that $(\bar{r}, \delta) \in L(v)$. Hence, I is minimal. □

Lemma 3.3. *Algorithm 1 preserves order-invariance.*

Proof. Assume I, before executing Algorithm 1, was built by some algorithm A that constructs HCL indices in an order-invariant manner (i.e., the structure of the index does not depend on landmark order during the labeling process). We show that applying UPGRADE-LMK to I preserves this property. Observe that UPGRADE-LMK first explores a shortest-path tree rooted at the new landmark r. This is the same traversal A would perform since it follows shortest paths and prunes when encountering other landmarks. Thus, the entries added to labels, as well as values stored in δ_H for r, are identical to those produced by A, since I already covers G. Finally, in Lines (27)–(34), for each $r' \in$ REACHED-LAN, the algorithm removes (r', δ) from $L(v)$ only if v lacks a neighbor w with $(r', \rho) \in L(w)$ and $\rho = \delta + \omega(v, w)$, processing vertices in order of distance from r'. This means entries are removed only from the labels of vertices in subgraphs of shortest-path trees rooted at r' that would not be explored by A. Therefore, the resulting labeling matches the one A would construct, proving order-invariance. □

Theorem 3.4. *Algorithm 1 runs in $O(|R|(m + n(\log n + |R|)))$ time and uses $O(|R|n)$ additional space.*

Proof. Assume that the queue Q is implemented as a min-heap (e.g., a Fibonacci heap), where insert and decrease-key take constant time and delete-min takes $O(\log n)$ time (both amortized). Each label $L(v)$ of a vertex v is stored as an array, and w.l.o.g. we assume $L(v)$ contains at most one entry per landmark $r \in R$, i.e., $|L(v)| \leq |R|$. Hence, adding or removing entries from a label costs $O(|R|)$. The QUERY routine therefore runs in $O(|R|)$ time for a pair (s, t) if either $s \in R$ or $t \in R$, and in $O(|R|^2)$ otherwise. Lines (1)–(5) require $O(|R|^2)$ time to compute δ_H for landmarks not covering r. During the modified Dijkstra search (Lines 15–26), each vertex is extracted at most once from Q, and running QUERY for $u \neq r$ takes $O(|R|)$ time. This dominates the cost of adding u to sets REACHED-VER. Thus, this phase costs $O(m + n \log n + n|R|)$ overall.

Finally, in Lines (27)–(34), for each landmark in REACHED-LAN (with $|\text{REACHED-LAN}| = O(|R|)$), we insert $O(n)$ vertices into Q, which takes $O(n \log n)$ time. Extracting $O(n)$ vertices costs $O(\log n)$ per operation, plus scanning neighbors and labels. This yields $O(\log n + |R| + |N(v)|)$ time per vertex, or $O(m + n(\log n + |R|))$ total time for all vertices, per landmark. Therefore, this phase runs in $O(|R|(m + n(\log n + |R|)))$ time. Summing up, the time complexity of Algorithm 1 is:

$$O\big(|R|^2 + m + n \log n + n|R| + |R|(m + n(\log n + |R|))\big) = O(|R|(m + n(\log n + |R|))).$$

Concerning the space complexity, the algorithm uses three auxiliary structures: P, REACHED-LAN, and REACHED-VER. Their sizes are $O(n)$ (one distance per vertex), $O(|R|)$ (one entry per landmark), and $O(|R|n)$ (one entry per landmark–vertex pair), respectively. Since each vertex can appear in Q at most once, the overall extra space is $O(|R|n)$, as claimed. □

3.2 Algorithm DOWNGRADE-LMK

The strategy of algorithm DOWNGRADE-LMK, in contrast to algorithm UPGRADE-LMK, is to remove from the HCL index (both from the highway and the labeling) all information related to the downgraded landmark r and to restore the cover property with the remaining landmarks, if it is broken by this removal. This is done by starting a graph search from r, serving two purposes: (i) delete all entries associated with r from the labels of visited vertices; (ii) identify landmarks $r' \in R \setminus \{r\}$ such that a shortest path from r to r' does not traverse other landmarks (i.e., landmarks that might cover r and vertices reachable from r by shortest paths). At this point, the distance from r to each such r' is then retrieved from $L(r)$ and used to start a second graph search, rooted at r', from r. This search locates vertices previously covered by r that can now be covered by a different landmark r'. In more detail, Algorithm DOWNGRADE-LMK, whose pseudo-code is given in Algorithm 2, takes as input a weighted graph $G = (V, E, \omega)$, an HCL index $I = (H = (R, \delta_H), L)$ with landmark set R, and a landmark $r \in R$ to be *downgraded* to be a non-landmark vertex, hence removed from R. It outputs a highway labeling $I = (H = (R', \delta_H), L)$ such that $R' = R \setminus \{r\}$, H is a highway for R', L is a landmark labeling, and I covers G with landmark set R'. The algorithm begins by removing r from R and deleting the corresponding entry in $L(r)$. Next, a Dijkstra-like search starts from r. When a vertex u is discovered at distance δ from r (i.e., dequeued from the min-priority queue Q with priority δ), the algorithm proceeds as follows: if u is a landmark, we check whether δ is smaller than or equal to the distance $\delta_H(r, u)$, stored in H. If so, then landmark u covers vertex r, so we add entry (u, δ) to $L(r)$ and prune the search at u (i.e., do not continue the traversal to its neighbors). If $u \notin R'$, instead, we check whether r covers u. If true, the entry for r is removed from $L(u)$, and the search continues by enqueueing neighbors of u or decreasing their priority in Q. At the end of the search, all entries in H related to r – that is, values $\delta_H(r, x)$ for each $x \in R'$ –are deleted.

Then, for each landmark l reached in the previous phase (identified by label entries stored in REACHED-ENT), a modified Dijkstra search is performed. This search is rooted at l and starts from r, aiming to find vertices previously covered by r that can now be covered by l. To begin, a min-priority queue Q is initialized with vertex r at priority ρ, where ρ is the distance from l to r. The search proceeds by dequeuing pairs (δ, u), meaning a shortest path of weight δ from l to u has been found. The search is pruned at u if $u \in R'$ or if the query routine QUERY on the HCL index for r and u returns a value smaller than δ. If neither condition holds, then there exists a path from l to u that: (i) does not traverse any landmark other than l, and (ii) has weight smaller than any alternative path encoded in the HCL index (i.e., obtained by concatenating two shortest paths from l and u to a common landmark). In this case, entry (l, δ) is added to $L(u)$, and the

Algorithm 2: Algorithm DOWNGRADE-LMK.

Input: Graph $G = (V, E, \omega)$, HCL index $I = (H = (R, \delta_H), L)$ covering G with landmarks $R \subseteq V$, a landmark $r \in R$.
Output: HCL index $I = (H = (R', \delta_H), L)$ covering G with landmarks $R' = R \setminus \{r\}$.

1 $R' \leftarrow R \setminus \{r\}$;
2 $L(r) \leftarrow \emptyset$; ▷Initialize new label, add it to L
3 REACHED-ENT $\leftarrow \emptyset$;
4 $Q \leftarrow \emptyset$; ▷Empty priority queue, e.g. min-heap
5 $Q.enqueue(r, 0)$; ▷Add r to Q with priority $d(r, r) = 0$
6 **foreach** $v \in V$ **do**
7 $\quad P[v] \leftarrow \infty$; ▷Init distance from r to other vertices to ∞
8 $P[r] \leftarrow 0$;
9 **while** $Q \neq \emptyset$ **do**
10 $\quad (u, \delta) \leftarrow Q.dequeueMin()$; ▷Here $P[u]$ equals δ
11 $\quad$ **if** $u \in R'$ **then**
12 $\quad\quad$ **if** $\delta_H(r, u) < \delta$ **then**
13 $\quad\quad\quad$ **continue**;
14 $\quad\quad$ Add (u, δ) to REACHED-ENT;
15 $\quad\quad$ Add (u, δ) to $L(r)$;
16 $\quad\quad$ **continue**;
17 $\quad L(u) \leftarrow \{(l, \rho) \in L(u) \mid l \neq r\}$;
18 $\quad$ **foreach** $w \in N(u) : P[w] > P[u] + \omega(u, w)$ **do**
19 $\quad\quad$ **if** $P[w] = \infty$ **then** $Q.enqueue(w, P[u] + \omega(u, w))$;
20 $\quad\quad$ **else** $Q.decreaseKey(w, P[u] + \omega(u, w))$;
21 $\quad\quad P[w] \leftarrow P[u] + \omega(u, w)$;
22 Delete all entries related to r in δ_H;
23 **foreach** $(l, \rho) \in$ *REACHED-ENT* **do**
24 $\quad$ **foreach** $u \in V$ **do**
25 $\quad\quad P[u] \leftarrow \infty$; ▷Reset distance from l to other vertices
26 $\quad P[l] \leftarrow 0$;
27 $\quad P[r] \leftarrow \rho$;
28 $\quad Q \leftarrow \emptyset$; ▷Empty priority queue, e.g. min-heap
29 $\quad Q.enqueue(r, \rho)$; ▷Add r to Q with priority $d(l, r) = \rho$
30 $\quad$ **while** $Q \neq \emptyset$ **do**
31 $\quad\quad (u, \delta) \leftarrow Q.dequeueMin()$; ▷Here $P[u]$ equals δ
32 $\quad\quad$ **if** $u \in R'$ **then**
33 $\quad\quad\quad$ **continue**;
34 $\quad\quad$ **if** $u \neq r$ *and* $QUERY(l, u, H, L) < \delta$ **then continue**;
35 $\quad\quad$ Add (l, δ) to $L(u)$;
36 $\quad\quad$ **foreach** $w \in N(u) : P[w] > P[u] + \omega(u, w)$ **do**
37 $\quad\quad\quad$ **if** $P[w] = \infty$ **then**
$\quad\quad\quad\quad Q.enqueue(w, P[u] + \omega(u, w))$;
38 $\quad\quad\quad$ **else** $Q.decreaseKey(w, P[u] + \omega(u, w))$;
39 $\quad\quad\quad P[w] \leftarrow P[u] + \omega(u, w)$;

visit continues by enqueueing neighbors of u or decreasing their priority in Q.

We now prove the correctness of Algorithm 2.

THEOREM 3.5. *Algorithm 2 computes an HCL index $I = (H = (R', \delta_H), L)$ that covers G with landmark set $R' = R \setminus \{r\}$.*

PROOF. When r is removed from R, the algorithm must ensure two properties to preserve the highway cover property: (i) δ_H remains a distance decoding function for R', i.e., $\delta_H(r', r'') = d(r', r'')$ for all $r', r'' \in R' = R \setminus \{r\}$; (ii) for each pair $s, t \in V \setminus R$, and for any $r' \in R'$, labels $L(s)$ and $L(t)$ combined with δ_H suffice to compute the r'-constrained distance. We prove (i) and (ii) separately. For (i), note that the highway cover property of I implies that for any $r', r'' \neq r$, $\delta_H(r', r'') = d(r', r'')$. Since Algorithm 2 only deletes entries involving r, all distances for pairs in $R' \times R'$ remain unchanged. Thus, at termination, δ_H correctly stores all distances between landmarks in R'.

For (ii), since (i) ensures that H is a highway for R', we must show that for any $r' \in R'$ and non-landmark vertices $s, t \in V \setminus R'$, the r'-constrained distance can still be retrieved from H and L. Specifically, it must hold that

$$d(r_i, s) + \delta_H(r_i, r_j) + d(r_j, t),$$

where $r_i = r'$ or $r_j = r'$, and entries $(r_i, d(r_i, s)) \in L(s)$ and $(r_j, d(r_j, t)) \in L(t)$ exist. This guarantees that constrained shortest paths involving r in the original index are still represented after its removal. Notice that the only labels modified by algorithm DOWNGRADE-LMK are: (i) $L(r)$; (ii) $L(u)$ for each $u \in V \setminus R'$ with $r \in L(u)$; (iii) $L(v)$ for each $v \in V \setminus R'$ reached by the Dijkstra-like search from r' for entries $(l, \rho) \in$ REACHED-ENT.

First, we show that, by the end of the algorithm, all entries related to r are removed from the labeling. Suppose, for contradiction, that there exists a vertex $u \in V \setminus R'$ such that $(r, \delta) \in L(u)$. This would imply that the search initiated from r (in Lines (9)-(21)) did not reach vertex u. Consequently, the (shortest) path from r to u traversed during the search must have encountered another landmark $r' \neq r$, which caused the search to be pruned before reaching u (per Line 16). But the cover property of the initial index guarantees the existence of a shortest path from r to u avoiding other landmarks, which leads to a contradiction.

Now assume that the final index $I = (H, L)$ fails at covering G with the updated landmark set R'. Then there exists $u \in V \setminus R'$ and $r' \in R'$ such that $(r', d(r', u)) \notin L(u)$. Three cases arise: (a) vertex u is the removed landmark r. However, all updates to $L(r)$ occur solely in Lines (9)–(21), where any missing entry implies all shortest paths from r to $\bar{r} \in R'$ traverse another landmark, contradicting the assumption; (b) No pair $(r', \rho) \in$ REACHED-ENT. This mirrors case (a), since any landmark added to $L(r)$ during the first phase is also recorded in REACHED-ENT through entries added to $L(r)$; (c) entry $(r', \rho) \in$ REACHED-ENT, but the resumed search from r' (Lines (23)–(39)) fails to reach u. Then either: – every shortest path from r' to u traverses another landmark $\bar{r} \neq r'$, contradicting the assumption that $(r', d(r', u))$ should exist; or – pruning at Line (34) occurs at some vertex $u' \in V \setminus R'$ on a shortest path $P_{r'u}$ whose weight is less than δ, again implying $(\bar{r}, d(\bar{r}, u)) \in L(u)$ for some $\bar{r} \neq r'$, contradicting the absence of $(r', d(r', u))$. In all cases we reach a contradiction. Hence, the final HCL index covers G with landmark set R'. □

We now show that HCL indices computed by DOWNGRADE-LMK are minimal and order-invariant.

LEMMA 3.6. *The HCL index $I = (H = (R', \delta_H), L)$, computed by Algorithm 2 satisfies minimality.*

PROOF. Suppose, for contradiction, that there exists a labeling L' covering G such that for some vertex v and landmark $\bar{r}$, we have $(\bar{r}, \delta) \in L(v)$ but $(\bar{r}, \delta) \notin L'(v)$. Two cases can arise: either $(\bar{r}, \delta)$ was already in $L(v)$ before the update, or it was added by DOWNGRADE-LMK.

Case 1: $(\bar{r}, \delta)$ was in $L(v)$ before the update. If $\bar{r} = r$, then $(\bar{r}, \delta) \in L(v)$ implies the existence of a shortest path from r to v that does not include other landmarks. Hence, v is reached during the traversal rooted at r, and $(\bar{r}, \delta)$ is removed, a contradiction. If $\bar{r} \neq r$, the absence of $(\bar{r}, \delta)$ from $L'(v)$ implies that all shortest paths from $\bar{r}$ to v pass through another landmark. This contradicts the assumption that the index covered G before execution, since $\bar{r}$ should cover v.

Case 2: $(\bar{r}, \delta)$ is added by DOWNGRADE-LMK. Entries are added only if a shortest path exists from the landmark to u without traversing other landmarks. The absence of $(\bar{r}, \delta)$ from L' contradicts the assumption that L' covers G. In all cases, the assumption leads to a contradiction, proving minimality. □

LEMMA 3.7. *Algorithm 2 preserves order-invariance.*

PROOF. Assume that I, before executing Algorithm 2, was built by an order-invariant algorithm A (i.e., an algorithm that outputs an index whose structure does not depend on landmark order during the labeling process). We show that applying DOWNGRADE-LMK to I preserves this property. Specifically, note that the algorithm modifies the index in two phases. In the first phase, a Dijkstra search of the graph starts from r, the removed landmark. This traversal explores the same shortest-path tree as A, since it is pruned only when landmarks are encountered and since only shortest paths are traversed. Thus, entries added to $L(r)$ in Line (15) correspond to shortest paths avoiding other landmarks, the same as A would add. A similar argument holds for the corresponding removal of entries, associated with r, in both the labels of visited vertices and in δ_H (these would not be added by A). Hence, order-invariance is guaranteed by this step. In the second phase, a sequence of Dijkstra searches are resumed from r, for each landmark l such that, by the highway cover property holding on I before execution, there exists a shortest path from l to r not traversing other landmarks. Hence, the algorithm extends a pruned shortest-path tree, from r, and adds corresponding entries, as A would do, and therefore the claim follows. Finally, the last phase of DOWNGRADE-LMK performs a sequence of Dijkstra searches that are resumed from r, for each landmark l such that, by the highway cover property holding on I before execution, there exists a shortest path from l to r not traversing other landmarks. □

THEOREM 3.8. *Algorithm 2 runs in $O(|R|(m + n\log(n) + n|R|))$ and uses $O(n)$ additional space.*

PROOF. The most time-consuming components of Algorithm 2 are the two modified Dijkstra searches. The first, in Lines (9)-(21), prunes entries related to the removed landmark r. This subroutine runs in $O(m + n\log n + n|R|)$ time, since it scans all entries in $L(u)$ for each vertex u dequeued from Q. The second search, in Lines (23)-(39), is executed once per landmark identified in the previous phase. With $O(|R|)$ such landmarks and each search costing $O(m + n\log n + n|R|)$ time, the total time for this phase is $O(|R|(m + n\log n + n|R|))$. This complexity arises because each execution of QUERY takes $O(|R|)$ time (as $l \in R'$), and we execute one QUERY per dequeued vertex (see proof of Thm. 3.4 for supporting analysis on the cost of queue operations and queries). Overall, this second phase's time complexity dominates the runtime of Algorithm 2. Regarding the space complexity, the algorithm stores: (i) the distances from the root of each Dijkstra-like search in array P; (ii) a priority queue with at most n vertices, and (iii) set REACHED-ENT with at most one entry per landmark. Therefore, the space complexity is $O(n + n + |R|) = O(n)$. □

In Figure 1 we give a visual representation of the changes an HCL index undergoes if updated by our dynamic algorithms. In particular, on the left, we show an input graph G (unweighted, for readability) and an initial set of landmarks $R = \{5, 7\}$. Both landmarks are at distance 2 one from the other, and this is stored in H. Label $L(1)$, initially, contains entries $(5, 2)$ and $(7, 1)$, since $(1, 3, 5)$ and $(1, 7)$ are two shortest paths to 5 and 7, respectively. Similarly, vertex 6 is connected to both landmarks by an edge, hence the corresponding entries with distance equal to 1 are stored $L(6)$. Finally, $L(8)$ contains only an entry associated with landmark 5, since the 7-constrained shortest path from landmark 7 to 8 traverses landmark 5. The update begins when vertex 3 is promoted to be a landmark. Note that, for readability, we show only a portion of the HCL index, next to the effects of executing our algorithms when a landmark is added or removed on such portion. In particular, Algorithm UPGRADE-LMK begins by scanning label $L(3) = \{(5, 1), (7, 2)\}$ to populate the highway for the new landmark, i.e. to set $\delta_H(3, 5) = 1$ and $\delta_H(3, 7) = 2$. Successively, the algorithm initiates a graph traversal from the new landmark 3, visits landmark 5 and vertices $\{1, 2, 4, 6\}$ at distance 1, adds landmark 5 to set REACHED-LAN, and inserts entry $(3, 1)$ into the $L(v)\ \forall v \in \{1, 2, 4, 6\}$. Then, at distance 2, landmark 7 is reached and added to REACHED-LAN, while on vertex 9 the search adds the label entry $(3, 2)$ to $L(9)$. A similar situation occurs for vertex 10 at distance 3, while the visit is pruned on 8, at distance 4, since $\text{QUERY}(3, 8, H, L)$ returns 2, due to the path from 3 to 8 traversing vertex 5. As a result of the above, set REACHED-VER$[5] = \{1, 2, 4, 6, 9, 10\}$ as these vertices were covered by 5, before the addition of 3, and now are covered by 3. Therefore, in the last phase, UPGRADE-LMK proceeds by testing, following an order of distance from 5 (dictated by the priority in Q), whether entries associated with 5 must be kept in the index or removed (when all shortest paths from 5 to a vertex in REACHED-VER$[5]$ pass through 3). For example, vertices 9 and 6 are the first two to be extracted from Q, as they are at distance 1 from 5. They have a neighbor closer to 5 (5 itself) and hence $(5, 1)$ is not deleted from both $L(9)$ and $L(6)$, since there exist two shortest paths from 5 to 6 that do not traverse any other landmark, namely $(5, 6)$ and $(5, 9)$. Instead, for each $v \in \{1, 2, 4, 10\}$ entry $(5, 2)$ is removed from $L(v)$, since no neighbor, closer to 5, satisfying the test of Line (34) exists in the graph (all shortest paths from v to 5 pass through the new landmark 3). Once UPGRADE-LMK terminates, landmark 7 is downgraded to be non-landmark. Algorithm DOWNGRADE-LMK starts a graph traversal from this vertex and: at distance 1, it removes entry $(7, 1)$ from both $L(1), L(6)$ and $L(11)$. When visiting vertices at distance 2, it deletes entry $(7, 2)$ from $L(2), L(4)$ and $L(9)$, while, respectively, adding entries $(3, 2)$ and $(5, 2)$ to $L(7)$; these entries are also inserted in set REACHED-ENT. Finally, vertex 10 is reached at distance 3, and the entry relative to 7 is removed from $L(10)$. Next, the highway is updated to delete values $\delta_H(3, 7)$ and $\delta_H(5, 7)$. Finally, for each entry (l, ρ) in REACHED-ENT, the algorithm performs a second Dijkstra-like visit starting at l with priority ρ. In the example, this yields the addition of entries $(3, 3)$ and $(5, 3)$ to $L(11)$. The only vertex whose label is unchanged is 8. The final result is shown in Figure 1 (right).

4 Experimental Evaluation

In this section, we present an experimental study to evaluate the performance of our proposed algorithms UPGRADE-LMK and DOWNGRADE-LMK. Our investigation pursues two main goals:

(G1) Assessing Dynamic Maintenance Efficiency. We first evaluate whether UPGRADE-LMK and DOWNGRADE-LMK can maintain an HCL index more efficiently than recomputing it from scratch using BUILDHCL, the only baseline method currently available to update an HCL index when the landmark set changes. This comparison is crucial to determine whether dynamic landmark support is practically feasible

with HCL. Although buildHCL has a time complexity of $O(|R|(n \log n + m))$, which is asymptotically better than the $O(n^3)$ worst-case time of upgrade-lmk and downgrade-lmk (when $|R| = n$), the practical performance picture is quite different. Our algorithms localize updates by efficiently identifying only the affected parts of the index – specifically, modified labels and distances – via few graph traversals. As our results will show, this leads to substantial real-world speedups, often by orders of magnitude.

(G2) Comparing Dynamic Query Effectiveness. Second, for completeness, we examine whether the HCL index, combined with our dynamic algorithms, outperforms other existing methods, not based on HCL, for computing landmark-constrained distances in dynamic settings. For example, the frameworks by Rice et al. [43] and Kaffes et al. [38], originally designed for generalized shortest-path queries, are adaptable to our setting: by treating landmarks as a single category, they can compute landmark-constrained distances on demand and naturally accommodate dynamic changes to landmarks. Similarly, POI query systems such as [23] can be extended to support this task with minimal computational overhead. All of these represent valid alternatives to our HCL-based dynamic framework and are therefore included in our comparative analysis.

Experimental Setup. To address goals (G1) and (G2), we design an extensive experimental evaluation, as described below. For (G1), we implemented the new dynamic algorithms upgrade-lmk and downgrade-lmk, along with the baseline buildHCL for full recomputation. For (G2), as an alternative to HCL, and motivated by its status as a state-of-the-art solution for generalized shortest paths, we implemented the method of Rice et al. [43], which we denote as ch-gsp. We do not include the method of Kaffes et al. [38], as it has been empirically shown to be consistently outperformed by the HCL-based approach in [13]. In the same work, other natural baselines – such as computing distances via multiple Dijkstra searches or precomputing and storing the full distance matrix for landmarks – were also evaluated and found to be significantly slower and less scalable than HCL. For similar reasons, we exclude POI query systems like [23], as their performance on landmark-constrained queries was reported to be orders of magnitude worse than that of HCL [13]. All algorithms were implemented in two versions to support both weighted and unweighted graphs. The latter are obtained by replacing Dijkstra-like traversals with BFS-like traversals (i.e., FIFO queues instead of priority queues), and by assigning unit weights to all edges. Our codebase is written in C++ and compiled with G++ 10.5 using `-O3` optimization. The implementation of ch-gsp builds on the RoutingKit library, which provides efficient Contraction Hierarchies for point-to-point shortest paths [26]. To evaluate the behavior of the proposed solutions under dynamic landmarks, we simulate environments in which vertices randomly become or cease to be landmarks. Queries for landmark-constrained distances are issued on-the-fly using each competing method.

Input Datasets. For our experiments, we select real-world graphs from publicly available repositories [11, 24, 27, 39, 41, 44], focusing on domains where fast retrieval of landmark-constrained distances is critical – such as road, social, web, and communication networks [2, 13, 30]. For completeness, we also include synthetic instances generated using the *Barabási–Albert* algorithm, known to model well real-world networks with power-law degree distributions, and the Erdős–Rényi method for generating uniform random graphs [10]. Details of the graphs used are provided in Table 1.

Graph	Type	|V|	|E|	$\bar{\Delta}$	W
erd [10]	Uniform	10 000	24 998 846	4 999.77	●
lux [44]	Road	30,647	37 773	2.46	●
cai [44]	Internet	32 000	40 204	2.51	●
uk-w [44]	Web	129 632	11 744 049	181.19	○
nw [24]	Road	1 207 945	1 410 387	2.33	●
ne [24]	Road	1 524 453	1 934 010	2.53	●
yah [27]	Ratings	1 625 951	256 804 235	315.88	○
ita [44]	Road	2 077 709	2 589 431	2.49	●
deu [44]	Road	4 047 577	4 907 447	2.42	●
u-bar [10]	Power-Law	50 000 000	149 985 000	6.00	○
w-bar [10]	Power-Law	50 000 000	149 985 000	6.00	●
usa [24]	Road	23 947 347	28 854 312	2.40	●
twi [11]	Social	52 579 682	1 614 106 500	61.39	○

Table 1: Summary of datasets: graph name, type, size (num. of vertices |V| and edges |E|), average degree $\bar{\Delta}$, graph is weighted (W = ●) or not (W = ○). Rows are sorted by non-decreasing $|V|$.

Experimental Setup and Methodology. To evaluate our dynamic framework (termed dyn-hcl, consisting of algorithms upgrade-lmk and downgrade-lmk operating on HCL indexes), we perform the following sequence of tests on each graph of Table 1:

(1) We first construct an initial HCL index I on the graph using a landmark set R, built via the buildHCL routine.
(2) In parallel, we preprocess the graph using ch-gsp, following the approach in [43], and measure the setup time. This step is restricted to sparse graphs, as Contraction Hierarchy-based methods suffer significant performance degradation – particularly long preprocessing times – when applied to other types of networks (e.g., social or dense graphs) [33].
(3) Next, we simulate dynamic behavior by applying $\sigma = |R|/4$ landmark updates: a randomly interleaved sequence of $\sigma/2$ insertions (moving vertices from $V \setminus R$ to R) and $\sigma/2$ deletions (moving vertices from R to $V \setminus R$), with equal probability.
(4) After each insertion or deletion, we invoke upgrade-lmk or downgrade-lmk to update the index and record the runtime.
(5) Once all σ updates have been applied, we recompute a new HCL index from scratch using buildHCL with the final landmark set, and measure its execution time. To assess query performance and pursue objective (G2), we issue $q = 10^7$ queries on randomly selected vertex pairs. We run query using both the updated and the recomputed HCL index, and execute the query routine of ch-gsp (the latter for sparse graphs only). We measure and compare average query times. The value of q is chosen to cover a sufficiently large fraction of the total $\approx |V|^2$ vertex pairs, averaging around 10^4 queries per update [30]. Whenever an index is altered – either by dyn-hcl or buildHCL– we measure its space occupancy. This is done solely for validation purposes, i.e., to verify that the memory footprint of the updated index matches that of the newly built one, as guaranteed by theory. For this reason, corresponding measurements are omitted.

We vary the initial landmark set size $|R|$ in $\{20, 40, 80\}$ for all graphs, as [30] suggests this range is suitable for most networks. Furthermore, for road graphs and communication networks, we consider larger values $|R| \in \{800, 1600, 3200\}$, identified in [13] as

Graph	$\|R\| = 20$			$\|R\| = 40$			$\|R\| = 80$		
	T_{BUILD}	T_{FDYN}	SPEED-UP	T_{BUILD}	T_{FDYN}	SPEED-UP	T_{BUILD}	T_{FDYN}	SPEED-UP
ERD	1 676.18	51.34	32.65	3 342.40	59.50	56.17	6 697.29	82.98	80.71
LUX	0.03	< 0.01	19.21	0.06	< 0.01	39.32	0.12	< 0.01	39.29
CAI	0.18	< 0.01	36.23	0.20	< 0.01	61.86	0.38	< 0.01	174.19
UK-W	0.84	< 0.01	1 123.68	1.73	< 0.01	1 266.68	3.28	< 0.01	1 152.99
NW	5.84	0.18	30.99	10.48	0.16	65.31	20.81	0.13	153.47
NE	8.16	0.59	13.66	15.74	0.46	34.09	29.64	0.40	72.36
YAH	94.80	10.27	9.22	234.61	23.39	10.02	560.38	55.90	10.02
ITA	8.58	0.72	11.88	17.42	0.62	27.90	35.00	0.51	68.35
DEU	26.89	2.07	12.98	41.59	0.53	77.04	84.79	1.68	50.43
U-BAR	1 214.05	87.23	13.92	2 574.39	162.33	15.86	6 711.91	320.63	20.93
W-BAR	1 799.19	129.85	13.86	3 758.61	210.92	17.82	307.19	6072.08	19.76
USA	231.56	15.31	15.11	427.74	6.38	67.04	1 012.62	17.05	59.36
TWI	2 876.37	173.86	16.54	6 043.94	408.10	14.80	12 485.80	519.11	24.05
Graph	$\|R\| = 800$			$\|R\| = 1600$			$\|R\| = 3200$		
	T_{BUILD}	T_{FDYN}	SPEED-UP	T_{BUILD}	T_{FDYN}	SPEED-UP	T_{BUILD}	T_{FDYN}	SPEED-UP
LUX	1.25	< 0.01	641.50	3.15	< 0.01	1 501.53	8.86	< 0.01	2 287.82
CAI	3.92	< 0.01	2 717.60	8.34	< 0.01	5 146.90	20.46	< 0.01	6 346.80
UK-W	33.41	0.02	1 242.49	64.28	0.04	1 482.26	139.00	0.12	1 108.52
NW	205.09	0.25	814.82	418.75	0.39	1 053.96	809.16	0.35	2 310.98
NE	304.48	2.63	115.71	631.39	2.77	227.24	1 132.35	1.18	958.53
ITA	357.68	1.04	341.45	677.71	0.94	718.32	1 374.71	1.49	921.28
DEU	802.17	3.33	240.49	1 577.84	2.41	652.02	3 058.22	2.21	1 383.41
USA	8 879.29	42.66	208.15	13 990.10	43.26	323.40	28 528.40	82.88	344.21

Table 2: Comparison of DYN-HCL versus full recomputation by BUILDHCL in terms of runtime.

typical for applications using the HCL index for shortest beer path queries. We adopt standard landmark selection policies based on degree and approximate betweenness [21, 30]: the former yields better results for unweighted graphs, while the latter is more effective for weighted ones. All experiments were conducted on a workstation equipped with an Intel Xeon E5-2643 @ 3.40 GHz, 96 GB RAM, running Ubuntu Linux. For completeness, we also tested purely incremental and decremental sequences of landmark modifications. The observed trends closely match those of the mixed-update case, and results are omitted for brevity.

Experimental Results and Analysis. The results of our experimental evaluation are reported in Tables 2 (G1) and 3 (G2). In Table 2, we analyze the efficiency of DYN-HCL in maintaining the HCL index under landmark updates, and compare it against BUILDHCL, which reconstructs the index from scratch. We consider the following performance indicators:

- T_{FDYN}: average runtime per update (insertion or deletion) of UPGRADE-LMK and DOWNGRADE-LMK, executed once for each of the σ modifications to the landmark set;
- T_{BUILD}: time to rebuild the index from scratch after all updates, using BUILDHCL;
- SPEED-UP: speedup factor T_{BUILD}/T_{FDYN}, indicating how faster the dynamic approach is compared to full recomputation.

The main conclusion drawn from the data in Table 2 is that, despite their worst-case time complexities (see Theorems 3.4 and 3.8), the dynamic algorithms UPGRADE-LMK and DOWNGRADE-LMK are highly efficient in practice for updating HCL indexes, consistently outperforming full recomputation – by the BUILDHCL algorithm – by one to three orders of magnitude. Even on massive datasets and with large landmark sets, our dynamic algorithms maintain high efficiency. For instance, on graph CAI with $|R| = 3200$, the speedup factor exceeds 6,000×, while on NW and DEU we observe improvements of approximately 2,300× and 1,300×, respectively. Similarly, for the largest graphs considered (e.g., YAH, USA, and TWI), our dynamic framework is consistently at least two orders of magnitude faster than BUILDHCL. Remarkably, the speed-up tends to increase with the size of the landmark set and is only weakly affected by changes in the graph size, for a same topology (see, e.g., in Table 2, graph USA compared to ITA or NE, or graph TWI compared to YAH). These results clearly demonstrate the excellent scalability of our methods in practical scenarios. This is mostly due to their ability to identify and process only the graph regions impacted by a change, rather than the full graph.

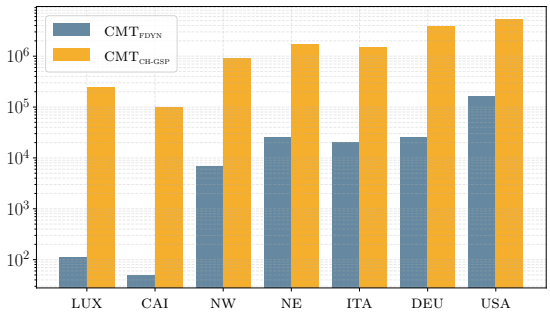

Figure 2: Cumulative runtimes of DYN-HCL and CH-GSP for a selection of input graphs, when $|R| = 3200$.

Importantly, this excellent performance in updating HCL indexes does not come at the cost of compromising index quality. Lemmas 3.2 and 3.6 guarantee that dynamically updated indexes have the same space usage as those rebuilt by BUILDHCL, and that the compactness of the index –crucial to achieving

Graph	$\|R\| = 800$				$\|R\| = 1600$				$\|R\| = 3200$			
	Cumulative (s)		Amortized (s)		Cumulative (s)		Amortized (s)		Cumulative (s)		Amortized (s)	
	CMT_{FDYN}	$CMT_{CH\text{-}GSP}$	AMR_{FDYN}	$AMR_{CH\text{-}GSP}$	CMT_{FDYN}	$CMT_{CH\text{-}GSP}$	AMR_{FDYN}	$AMR_{CH\text{-}GSP}$	CMT_{FDYN}	$CMT_{CH\text{-}GSP}$	AMR_{FDYN}	$AMR_{CH\text{-}GSP}$
LUX	90.22	58 285.67	9.0×10^{-6}	5.8×10^{-3}	129.91	121 814.27	1.2×10^{-5}	1.2×10^{-2}	110.64	249 240.28	1.1×10^{-5}	2.4×10^{-2}
CAI	36.92	22 789.63	3.6×10^{-6}	2.2×10^{-3}	40.37	47 898.93	4.0×10^{-6}	4.7×10^{-3}	49.64	99 008.76	4.9×10^{-6}	9.9×10^{-3}
NW	943.50	223 760.51	9.4×10^{-5}	2.2×10^{-2}	2 560.12	466 466.28	2.5×10^{-4}	4.6×10^{-2}	6 931.59	915 736.92	6.9×10^{-4}	9.1×10^{-2}
NE	4 464.98	428 357.02	4.4×10^{-4}	4.2×10^{-2}	20 855.86	867 067.34	2.0×10^{-3}	8.6×10^{-2}	25 718.51	1 753 043.30	2.5×10^{-3}	1.7×10^{-1}
ITA	950.73	379 140.34	9.5×10^{-5}	3.7×10^{-2}	2 968.71	727 541.90	2.9×10^{-4}	7.2×10^{-2}	20 791.94	1 515 120.43	2.0×10^{-3}	1.5×10^{-1}
DEU	5 097.15	1 019 891.28	5.0×10^{-4}	1.0×10^{-1}	14 471.19	1 984 033.10	1.4×10^{-3}	1.9×10^{-1}	26 189.12	3 957 331.85	2.6×10^{-3}	3.9×10^{-1}
USA	22 610.78	1 339 209.99	2.2×10^{-3}	1.3×10^{-1}	109 943.56	2 647 513.43	1.0×10^{-2}	2.6×10^{-1}	163 851.92	5 301 780.44	1.6×10^{-2}	5.3×10^{-1}

Table 3: Cumulative and amortized runtimes of DYN-HCL and CH-GSP to retrieve landmark-constrained distances.

millisecond/microsecond-scale query times – is preserved by DYN-HCL. This is achieved at the cost of only a few seconds of overhead per operation, compared to thousands of seconds for full recomputation. Measurements of space occupancy and query times are consistent with those reported in the literature for HCL indexes [13, 30] and are therefore omitted for brevity.

Turning to (G2), in Table 3 we compare our dynamic HCL framework with CH-GSP from both cumulative and amortized cost perspectives. The *cumulative runtime* aggregates the time required to perform all relevant operations (index construction, landmark updates, and queries) while the *amortized cost* is obtained by dividing the cumulative runtime by the total number of query operations, thereby estimating the average computational cost per query for each method. Specifically, we report the following performance indicators:

- CMT_{FDYN}: cumulative runtime for DYN-HCL, defined as the sum of the time spent by BUILDHCL to construct the initial index, the time to update it via UPGRADE-LMK and DOWNGRADE-LMK for each of the σ changes, and the time to answer all queries using QUERY;
- $CMT_{CH\text{-}GSP}$: cumulative runtime for CH-GSP, given by the sum of its preprocessing time and the time to answer all queries using its query routine;
- AMR_{FDYN}, $AMR_{CH\text{-}GSP}$: amortized time per query for the two methods, obtained by dividing the respective cumulative runtime by the number of query operations. In this classical amortized analysis, the time DYN-HCL spends updating the index σ times is charged to the queries, whereas for CH-GSP the only operations considered are the queries themselves (preprocessing is distributed on queries for both methods).

The results in Table 3 provide strong evidence of the practicality of UPGRADE-LMK and DOWNGRADE-LMK: across all datasets and landmark sizes, CMT_{FDYN} is up to three orders of magnitude lower than $CMT_{CH\text{-}GSP}$. Even in the most challenging cases (e.g., DEU or USA with $|R| = 3200$), DYN-HCL remains highly efficient and substantially faster than CH-GSP in retrieving landmark-constrained distances. Although DYN-HCL must spend time updating the HCL index after each landmark change to ensure correct query answers, this overhead is negligible compared to the much larger query times of CH-GSP. Overall, DYN-HCL proves to be the most practical solution for computing landmark-constrained distances under varying landmark sets. This advantage is further highlighted by the amortized measures: AMR_{FDYN} ranges from just a few microseconds (e.g., CAI with $|R| = 800$) to a few tens of milliseconds (e.g., USA with $|R| = 3200$), depending on graph size and landmark count. In contrast, CH-GSP incurs significantly higher per-query costs – often orders of magnitude greater than DYN-HCL. For example, in ITA with $|R| = 3200$, AMR_{FDYN} is 2 ms, whereas $AMR_{CH\text{-}GSP}$ is approximately 0.2 s. The superior scalability of DYN-HCL is also evident in Figure 2, which plots CMT_{FDYN} and $CMT_{CH\text{-}GSP}$ for a selection of datasets with $|R| = 3200$ (results for other $|R|$ values are similar and omitted for brevity). Both frameworks exhibit roughly linear scaling with graph size – a desirable property – but DYN-HCL achieves far lower constants and is at least an order of magnitude faster than CH-GSP in every tested configuration, making it the more practical choice in all scenarios considered. In summary, DYN-HCL proves to be a highly scalable, space-efficient, and query-accurate solution for computing landmark-constrained distances under dynamic landmark sets. Compared to static rebuilding or alternative approaches such as CH-GSP, DYN-HCL offers: a) consistently lower runtimes for both updates and queries; b) negligible overhead for handling landmark dynamics; c) robust performance even at scale and under heavy query loads, while preserving index compactness and achieving millisecond-level query latencies. These results establish DYN-HCL as both a practical and theoretically sound framework for dynamic landmark-constrained distance queries, and, by extension, for shortest-path or shortest beer path queries with dynamic landmarks in large-scale networks.

5 Conclusion and Future Work

In this paper, we introduced DYN-HCL, a dynamic framework for efficiently maintaining highway cover labelings in the presence of landmark set updates. Our contributions include two novel, non-trivial algorithms for incrementally and decrementally updating HCL indexes as landmark vertices are inserted or removed, which: (i) avoid full recomputation by identifying only the parts of the index affected by modifications via few graph traversals; and (ii) preserve the compactness and query performance of the static HCL structure. Through an extensive experimental evaluation on both real-world and synthetic graphs, we demonstrated that DYN-HCL dramatically outperforms static recomputation, achieving speedups of up to four orders of magnitude, while maintaining identical memory footprints and microsecond-scale query times. Compared to state-of-the-art baselines for landmark-constrained distance queries under dynamically evolving landmarks, DYN-HCL is substantially more efficient in both cumulative and amortized runtime, establishing it as a highly practical solution.

As future work, we plan to: (i) generalize our approach to handle directed graphs and path-reporting queries; while these extensions are straightforward from a design perspective (see [15, 29]), experimental validation is essential to assess their practical performance; (ii) explore the feasibility of adapting DYN-HCL to a batch-dynamic model, where multiple landmark insertions and deletions are processed together – building on recent advances in batch-dynamic indexing [14, 28] to further reduce overhead in large-scale update scenarios; and (iii) conduct experiments in fully dynamic settings, where both the graph topology and

the landmark set evolve over time, by combining DYN-HCL with the techniques in [29] to support more general and realistic scenarios. Finally, we are interested in extending HCL to support multi-category landmark sets, allowing for richer semantic interpretations (e.g., different types of "important" vertices, similarly to what is modeled through generalized shortest paths), which would open the door to more expressive query applications and to further comparisons with methods to solve generalized shortest paths.

Acknowledgments

Work partially supported by: (i) Italian Ministry of University and Research through Project "EXPAND: scalable algorithms for EXPloratory Analyses of heterogeneous and dynamic Networked Data" (PRIN grant n. 2022TS4Y3N), funded by the European Union - Next Generation EU; (ii) Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM); (iii) European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (program RESTART), project MoVeOver/SCHEDULE ("Smart interseCtions witH connEcteD and aUtonomous vehicLEs", CUP J33C22002880001); (iv) French government, through the UCA$^{\text{JEDI}}$ Investments in the Future project managed by the National Research Agency (ANR, reference number ANR-15-IDEX-01).

Artifacts

Our experiments can be reproduced using the code available at https://github.com/D-hash/DynamicHighwayLabelling. It is written in C++ and can be easily compiled via the provided *Makefile*, after installing the *NetworKit* library for graph processing (https://networkit.github.io/). Input datasets can be downloaded from the repositories mentioned in the references.

References

[1] Takuya Akiba, Takanori Hayashi, Nozomi Nori, Yoichi Iwata, and Yuichi Yoshida. 2015. Efficient Top-k Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In *Proc. of 29th AAAI Conference on Artificial Intelligence*. AAAI Press, , 2–8. doi:10.1609/AAAI.V29I1.9154

[2] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast Exact Shortest-path Distance Queries on Large Networks by Pruned Landmark Labeling. In *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD 2013)*. ACM, , 349–360. doi:10.1145/2463676.2465315

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2014. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *23rd International World Wide Web Conference, WWW '14, Seoul, 2014*. ACM, , 237–248. doi:10.1145/2566486.2568007

[4] Julian Arz, Dennis Luxen, and Peter Sanders. 2013. Transit Node Routing Reconsidered. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7933)*, Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela (Eds.). Springer, , 55–66. doi:10.1007/978-3-642-38527-8_7

[5] Joyce Bacic, Saeed Mehrabi, and Michiel Smid. 2021. Shortest Beer Path Queries in Outerplanar Graphs. In *Proc. of 32nd International Symposium on Algorithms and Computation (ISAAC 2021) (LIPIcs, Vol. 212)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, , 62:1–62:16. doi:10.4230/LIPICS.ISAAC.2021.62

[6] Joyce Bacic, Saeed Mehrabi, and Michiel Smid. 2023. Shortest Beer Path Queries in Outerplanar Graphs. *Algorithmica* 85, 6 (2023), 1679–1705. doi:10.1007/S00453-022-01045-4

[7] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2016. Route Planning in Transportation Networks. In *Algorithm Engineering - Selected Results and Surveys*. Lecture Notes in Computer Science, Vol. 9220. , , 19–80. doi:10.1007/978-3-319-49487-6_2

[8] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. 2018. Deterministic Fully Dynamic Data Structures for Vertex Cover and Matching. *SIAM J. Comput.* 47, 3 (2018), 859–887. doi:10.1137/140998925

[9] Davide Bilò, Luciano Gualà, Stefano Leucci, and Alessandro Straziota. 2024. Graph Spanners for Group Steiner Distances. In *Proc. of 32nd Annual European Symposium on Algorithms (ESA 2024) (LIPIcs, Vol. 308)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, , 25:1–25:17. doi:10.4230/LIPICS.ESA.2024.25

[10] Béla Bollobás. 2011. *Random Graphs, Second Edition*. Cambridge Studies in Advanced Mathematics, Vol. 73. Cambridge University Press, . doi:10.1017/CBO9780511814068

[11] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and P. Krishna Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proceedings of the Fourth International Conference on Weblogs and Social Media, ICWSM 2010, Washington, DC, USA, May 23-26, 2010*, William W. Cohen and Samuel Gosling (Eds.). The AAAI Press, , 10–17.

[12] 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. doi:10.1137/S0097539702403098

[13] David Coudert, Andrea D'Ascenzo, and Mattia D'Emidio. 2024. Indexing Graphs for Shortest Beer Path Queries. In *24th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2024) (OASIcs, Vol. 123)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, , 2:1–2:18. doi:10.4230/OASICS.ATMOS.2024.2

[14] Annalisa D'Andrea, Mattia D'Emidio, Daniele Frigioni, Stefano Leucci, and Guido Proietti. 2015. Dynamic Maintenance of a Shortest-Path Tree on Homogeneous Batches of Updates: New Algorithms and Experiments. *ACM J. Exp. Algorithmics* 20 (2015), 1.5:1.1–1.5:1.33. doi:10.1145/2786022

[15] Gianlorenzo D'Angelo, Mattia D'Emidio, and Daniele Frigioni. 2019. Fully Dynamic 2-Hop Cover Labeling. *ACM J. Exp. Algorithmics* 24, 1 (2019), 1.6:1–1.6:36. doi:10.1145/3299901

[16] Rathish Das, Meng He, Eitan Kondratovsky, J. Ian Munro, Anurag Murty Naredla, and Kaiyu Wu. 2022. Shortest Beer Path Queries in Interval Graphs. In *Proc. of 33rd International Symposium on Algorithms and Computation (ISAAC 2022) (LIPIcs, Vol. 248)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, , 59:1–59:17. doi:10.4230/LIPICS.ISAAC.2022.59

[17] Andrea D'Ascenzo and Mattia D'Emidio. 2023. Top-k Distance Queries on Large Time-Evolving Graphs. *IEEE Access* 11 (2023), 102228–102242. doi:10.1109/ACCESS.2023.3316602

[18] Andrea D'Ascenzo and Mattia D'Emidio. 2025. On Mining Dynamic Graphs for k Shortest Paths. In *Proc. of 16th International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2024)*. Springer Nature Switzerland, , 320–336. doi:10.1007/978-3-031-78541-2_20

[19] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. 2015. Public Transit Labeling. In *Experimental Algorithms - 14th International Symposium, SEA 2015, Paris, France, June 29 - July 1, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9125)*, Evripidis Bampis (Ed.). Springer, , 273–285. doi:10.1007/978-3-319-20086-6_21

[20] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. 2011. Customizable Route Planning. In *Experimental Algorithms*, Panos M. Pardalos and Steffen Rebennack (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 376–387.

[21] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. 2014. Robust Distance Queries on Massive Networks. In *Proc. of 22th Annual European Symposium on Algorithms (ESA 2014) (Lecture Notes in Computer Science, Vol. 8737)*. Springer, , 321–333. doi:10.1007/978-3-662-44777-2_27

[22] Daniel Delling, Andrew V. Goldberg, Ruslan Savchenko, and Renato F. Werneck. 2014. Hub Labels: Theory and Practice. In *Proc. of 13th International Symposium on Experimental Algorithms (SEA 2014) (Lecture Notes in Computer Science, Vol. 8504)*. Springer, , 259–270. doi:10.1007/978-3-319-07959-2_22

[23] Daniel Delling and Renato F. Werneck. 2015. Customizable Point-of-Interest Queries in Road Networks. *IEEE Trans. Knowl. Data Eng.* 27, 3 (2015), 686–698. doi:10.1109/TKDE.2014.2345386

[24] Camil Demetrescu, Andrew V Goldberg, and David S Johnson. 2009. *The shortest path problem: 9th DIMACS implementation challenge*. Vol. 74. American Mathematical Soc., .

[25] Mattia D'Emidio and Imran Khan. 2019. Dynamic Public Transit Labeling. In *Computational Science and Its Applications - ICCSA 2019 - 19th International Conference, Saint Petersburg, Russia, July 1-4, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11619)*, Sanjay Misra, Osvaldo Gervasi, Beniamino Murgante, Elena N. Stankova, Vladimir Korkhov, Carmelo Maria Torre, Ana Maria A. C. Rocha, David Taniar, Bernady O. Apduhan, and Eufemia Tarantino (Eds.). Springer, , 103–117. doi:10.1007/978-3-030-24289-3_9

[26] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2016. Customizable Contraction Hierarchies. *ACM J. Exp. Algorithmics* 21, 1 (2016), 1.5:1–1.5:49. doi:10.1145/2886843

[27] Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. 2012. The Yahoo! Music Dataset and KDD-Cup '11. In *Proceedings of KDD Cup 2011 competition, San Diego, CA, USA, 2011 (JMLR Proceedings, Vol. 18)*, Gideon Dror, Yehuda Koren, and Markus Weimer (Eds.). JMLR.org, , 8–18. http://proceedings.mlr.press/v18/dror12a.html

[28] Muhammad Farhan, Henning Koehler, and Qing Wang. 2024. BatchHL$^+$: batch dynamic labelling for distance queries on large-scale networks. *VLDB J.* 33, 1 (2024), 101–129. doi:10.1007/S00778-023-00799-9

[29] Muhammad Farhan and Qing Wang. 2023. Efficient Maintenance of Highway Cover Labelling for Distance Queries on Large Dynamic Graphs. *World Wide Web (WWW)* 26, 5 (2023), 2427–2452. doi:10.1007/S11280-023-01146-2

[30] Muhammad Farhan, Qing Wang, Yu Lin, and Brendan D. McKay. 2019. A Highly Scalable Labelling Approach for Exact Distance Queries in Complex Networks. In *Proc. of 22nd International Conference on Extending Database Technology, EDBT 2019*. OpenProceedings.org, , 13–24. doi:10.5441/002/EDBT.2019.03

[31] Muhammad Farhan, Qing Wang, Yu Lin, and Brendan D. McKay. 2022. Fast fully dynamic labelling for distance queries. *VLDB J.* 31, 3 (2022), 483–506. doi:10.1007/S00778-021-00707-Z

[32] Qingshuai Feng, You Peng, Wenjie Zhang, Ying Zhang, and Xuemin Lin. 2022. Towards Real-Time Counting Shortest Cycles on Dynamic Graphs: A Hub Labeling Approach. In *Proc. of 38th IEEE International Conference on Data Engineering (ICDE 2022)*. IEEE, , 512–524. doi:10.1109/ICDE53745.2022.00043

[33] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transp. Sci.* 46, 3 (2012), 388–404. doi:10.1287/TRSC.1110.0401

[34] Joachim Gudmundsson and Yuan Sha. 2023. Shortest Beer Path Queries in Digraphs with Bounded Treewidth. In *Proc. of 34th International Symposium on Algorithms and Computation (ISAAC 2023) (LIPIcs, Vol. 283)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, , 35:1–35:17. doi:10.4230/LIPICS.ISAAC.2023.35

[35] Tesshu Hanaka, Hirotaka Ono, Kunihiko Sadakane, and Kosuke Sugiyama. 2023. Shortest Beer Path Queries Based on Graph Decomposition. In *Proc. of 34th International Symposium on Algorithms and Computation (ISAAC 2023) (LIPIcs, Vol. 283)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, , 37:1–37:20. doi:10.4230/LIPICS.ISAAC.2023.37

[36] Takanori Hayashi, Takuya Akiba, and Yuichi Yoshida. 2015. Fully Dynamic Betweenness Centrality Maintenance on Massive Networks. *Proc. VLDB Endow.* 9, 2 (2015), 48–59. doi:10.14778/2850578.2850580

[37] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. 2006. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In *The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 13-14, 2006 (DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 74)*, Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (Eds.). DIMACS/AMS, , 41–72. doi:10.1090/DIMACS/074/03

[38] Vassilis Kaffes, Alexandros Belesiotis, Dimitrios Skoutas, and Spiros Skiadopoulos. 2018. Finding shortest keyword covering routes in road networks. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management, SSDBM 2018, Bozen-Bolzano, Italy, July 09-11, 2018*, Dimitris Sacharidis, Johann Gamper, and Michael H. Böhlen (Eds.). ACM, , 12:1–12:12. doi:10.1145/3221269.3223038

[39] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *22nd International World Wide Web Conference, WWW '13, Companion Volume*, Leslie Carr, Alberto H. F. Laender, Bernadette Farias Lóscio, Irwin King, Marcus Fontoura, Denny Vrandecic, Lora Aroyo, José Palazzo M. de Oliveira, Fernanda Lima, and Erik Wilde (Eds.). International World Wide Web Conferences Steering Committee / ACM, , 1343–1350. doi:10.1145/2487788.2488173

[40] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2023. Multi-constraint shortest path using forest hop labeling. *VLDB J.* 32, 3 (2023), 595–621. doi:10.1007/S00778-022-00760-2

[41] Tiago P Peixoto et al. 2020. The Netzschleuder network catalogue and repository.

[42] You Peng, Xuemin Lin, Ying Zhang, Wenjie Zhang, and Lu Qin. 2022. Answering reachability and K-reach queries on large graphs with label constraints. *VLDB J.* 31, 1 (2022), 101–127. doi:10.1007/S00778-021-00695-0

[43] Michael N. Rice and Vassilis J. Tsotras. 2013. Engineering Generalized Shortest Path Queries. In *Proc. of 29th IEEE International Conference on Data Engineering (ICDE 2013)*. IEEE Computer Society, , 949–960. doi:10.1109/ICDE.2013.6544888

[44] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. http://networkrepository.com

[45] Dominik Schultes. 2008. Routing in Road Networks with Transit Nodes. In *Encyclopedia of Algorithms - 2008 Edition*, Ming-Yang Kao (Ed.). Springer, . doi:10.1007/978-0-387-30162-4_353

[46] Tim Zeitz. 2022. Fast Computation of Shortest Smooth Paths and Uniformly Bounded Stretch with Lazy RPHAST. In *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany (LIPIcs, Vol. 233)*, Christian Schulz and Bora Uçar (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, , 3:1–3:18. doi:10.4230/LIPICS.SEA.2022.3

[47] Yikai Zhang and Jeffrey Xu Yu. 2020. Hub Labeling for Shortest Path Counting. In *Proc. of 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD 2020)*. Association for Computing Machinery, , 1813–1828. doi:10.1145/3318464.3389737