Hierarchical Cut Labelling – Scaling Up Distance Queries on Road Networks

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

Answering the shortest-path distance between two arbitrary locations is a fundamental problem in road networks. Labelling-based solutions are the current state-of-the-arts to render fast response time, which can generally be categorised into hub-based labellings, highway-based labellings, and tree decomposition labellings. Hub-based and highway-based labellings exploit hierarchical structures of road networks with the aim to reduce labelling size for improving query efficiency. However, these solutions still result in large search spaces on distance labels at query time, particularly when road networks are large. Tree decomposition labellings leverage a hierarchy of vertices to reduce search spaces over distance labels at query time, but such a hierarchy is generated using tree decomposition techniques, which may yield very large labelling sizes and slow querying. In this paper, we propose a novel solution *hierarchical cut 2-hop labelling (HC2L)* to address the drawbacks of the existing works. Our solution combines the benefits of hierarchical structures from both perspectives - reduce the size of a distance labelling at preprocessing time and further reduce the search space on a distance labelling at query time. At its core, we propose a new hierarchy, *balanced tree hierarchy*, which enables a fast, efficient data structure to reduce the size of distance labelling and to select a very small subset of labels to compute the shortest-path distance at query time. To speed up the construction process of HC2L, we further propose a parallel variant of our method, namely HC2L^p. We have evaluated our solution on 10 large real-world road networks through extensive experiments. The results show that our method is 1.5-4 times faster in terms of query processing while being comparable in terms of labelling construction time and achieving up to 60% smaller labelling size compared to the state-of-the-art approaches.

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

Additional Key Words and Phrases: Shortest-path distance; road network; vertex cut; hierarchical partitioning; distance labelling; 2-hop labelling; balanced tree hierarchy

ACM Reference Format:
Muhammad Farhan, Henning Koehler, Robert Ohms, and Qing Wang. 2023. Hierarchical Cut Labelling – Scaling Up Distance Queries on Road Networks. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 244 (December 2023), 25 pages. https://doi.org/10.1145/3626731

1 INTRODUCTION

A *distance query* in road networks is to find the length of a shortest path between any two given locations. This is a fundamental building block with numerous real-world applications, such

Authors' addresses: 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; Robert Ohms, Australian National University, Canberra, Australia, robert.ohms@anu.edu.au; Qing Wang, Australian National University, Canberra, Australia, qing.wang@anu.edu.au.

2836-6573/2023/12-ART244
https://doi.org/10.1145/3626731

as GPS navigation [22], route planning [18], traffic monitoring [27], and point of interest (POI) recommendation [32]. Nowadays, computational resources such as memory and storage become readily available, e.g., road networks such as USA and EUR with tens of millions of vertices can be easily accommodated by a single server. However, many of these applications have low latency requirements, arising from the need to compute thousands to millions of distance queries per second as part of a more complex problem, which itself needs to be solved frequently, e.g. matching taxi drivers to passengers, optimizing delivery routes with multiple pick up and drop off points that can change dynamically, or providing recommendation on k-nearest POIs to their customers. For example, ride-hailing companies such as Uber are often required to compute millions of shortest-path distances between cars and customers each second [24, 33] (e.g., between the locations of 1k cars and 10k customers) in order to process customers requests, e.g., finding nearest cars to customers. In such cases, even if answering one distance query only takes 1 microsecond, processing 10 million distance queries however would still amount to 10 seconds. Thus, it is very important to improve the performance of computing shortest-path distances for these applications to ensure good service quality to their customers.

Related Work. In what follows, we briefly review the existing work for answering shortest-path distance queries in road networks.

Search-based approaches. A traditional approach to answering a distance query is to use the Dijkstra's algorithm [31] which can compute the length of a shortest-path from a source vertex to a destination vertex in $O(|E| + |V|log|V|)$ time. This is however very slow in reality. Particularly, for pairs of vertices that are far apart from each other, the search space is large. To improve search efficiency, a bidirectional scheme can be used to run two Dijkstra's searches: one from the source vertex and the other from the destination vertex [29]. However, road networks are structurally characterised by high diameters and low node degrees, making search-based approaches such as Dijkstra's algorithm highly inefficient. In general, search-based approaches fail to achieve desired response time performance required by many real-world applications that operate on increasingly large road networks.

To accelerate search by directing the search space towards useful vertices, rather than conducting unrestrained searches, a number of search-based approaches leverage indices. For example, ALT [22] pre-computes a partial distance index to accelerate the A* search. HiTi [26] constructs a hierarchical structure using graph decomposition techniques to accelerate query performance. Highway Hierarchies [30] and Contraction Hierarchies (CH) [21] have shown to prune the search space massively for efficient querying. Moreover, algorithms such as Transit Node Routing (TNR) [7], TRANSIT [9] and Arterial Hierarchy (AH) [36] integrate coordinate information and divide a road network into multiple hierarchical grids. Transit Node Routing and TRANSIT then precompute an index which stores distances between vertices with respect to grids. Arterial Hierarchy is an improved version of the CH algorithm. Despite considerable progress, these algorithms still require exploring a large search space for distance queries when vertices are far apart from each other in a road network.

Labelling-based approaches. To address the limitations of search-based approaches, labelling-based approaches for road networks have been studied with great success [1, 2, 4, 5, 14, 25, 28, 34]. Instead of performing searches on a road network at query time, these approaches precompute distance labels that fully capture distance information between pairs of vertices, and then perform searches on distance labels at query time to compute distances. Labelling-based approaches can answer distance queries significantly faster than search-based approaches, at the cost of requiring additional space for storing precomputed labels. As such, it is still an open problem how to design algorithms which can give both fast query times with small memory costs.

The current state-of-the-art labelling-based approaches exploit hierarchical structures of road networks. Most of these approaches satisfy the 2-hop cover property [15] which requires at least one common vertex in the distance labels $L(s)$ and $L(t)$ to be on a shortest-path between any query pair (s, t), generally falling into three categories: hub-based labellings, highway-based labellings, and tree decomposition labellings. Approaches for hub-based labellings [1, 2] generate distance labels for vertices that contain the distances to hub vertices following a vertex ordering computed by conducting CH searches on a road network. This results in a hierarchical structure among distance labels which is closely relating to the labelling size. Approaches for highway-based labellings [4] decompose a road network into disjoint shortest-paths and then construct distance labels for vertices that contain the distances to a subset of decomposed shortest-paths. Similar to the importance of a vertex ordering to hub-based labellings, the order of shortest-paths is important to highway-based labellings for small labelling sizes. Approaches for tree decomposition labellings [14, 28] first find a hierarchy over vertices in a road network using tree decomposition techniques [12]. Then a distance query (s, t) is answered by searching such a tree-decomposition hierarchy to identify a subset of common vertices in the distance labels $L(s)$ and $L(t)$, and compute the distance between s and t.

Present Work. From the above analysis, we observe that labelling-based approaches leverage hierarchies over vertices in a road network for distance queries in two ways. One way is to reduce the size of distance labellings. In hub-based labellings and highway-based labellings, a distance query (s, t) needs to search through entire labels $L(s)$ and $L(t)$ when computing a shortest-path distance. Therefore, they aim to find a "good ordering" of vertices which can produce distance labels of small sizes, and reduce the search space on $L(s)$ and $L(t)$ in the process. The other way is to assign a hierarchy over vertices, and while querying, use such a hierarchy to reduce the search space to a small subset of entries in a distance labelling. In tree-decomposition labellings, such a hierarchy is obtained by applying existing tree decomposition techniques [12].

Motivated by the above observations, in this work, we aim to design a solution that combines the benefits of using vertex hierarchies from both perspectives - leveraging a vertex hierarchy to reduce the size of a distance labelling and further reduce the search space on such a distance labelling at query time. This leads to the following questions:

- What are the desirable characteristics of such a vertex hierarchy?
- How can a hierarchy be designed to reduce the sizes of distance labellings and find "good hops" to accelerate search on distance labels simultaneously?
- What are the hierarchical properties of distance labellings under this framework?

Contributions. In this paper, we propose a new labelling-based method which addresses the above questions. Our contributions are summarised as follows:

- Upon analysing existing approaches that exploit hierarchical structures of road networks for efficient distance querying, we introduce a new hierarchy called *balanced tree hierarchy* which enables us to reduce the size of a distance labelling significantly. Further, our balanced tree hierarchy allows us to answer a distance query by processing a small subset of labels that are necessary to compute the distance. Compared to the state-of-the-art approaches, our method achieves much faster query time.
- We develop an efficient algorithm to construct our proposed *balanced tree hierarchy*. Our algorithm recursively partitions a road network while preserving two conditions: balanced partitions and small cuts. Specifically, it first finds a balanced and small vertex cut and then creates a minimum number of shortcuts to preserve the distance between border vertices. This results in a tree structured hierarchy among vertices, supported by an efficient data structure for finding hub vertices of any two vertices in a road network.

- We propose a distance labelling called *hierarchical cut 2-hop labelling* (HC2L) which is hierarchical in terms of a vertex quasi-order defined by a balanced tree hierarchy. The labels of any two given vertices must contain a common hub vertex that can be found via their lowest common ancestor in the balanced tree hierarchy. We analyse upper and lower bounds on labelling size of HC2L and devise a novel pruning strategy, called *tail pruning*, that allows our proposed algorithm to produce a HC2L with smaller labelling sizes without compromising query efficiency.
- We conduct extensive experiments to evaluate the performance of the proposed method on 10 real-world large road networks including the whole road network of the USA. The experimental results demonstrate that our method is 1.5-4 times faster than the state-of-the-art approaches in terms of query processing while consuming comparable labelling construction time and up to 60% smaller labelling size.

2 PRELIMINARIES

Let $G = (V, E)$ be a road network where V is a set of vertices, and E is a set of edges. Each edge $(u, v) \in E$ is associated with a positive weight $\omega(u, v) \in \mathbb{R}$. Given a set of vertices $S \subseteq V$, $G[S] = (S, \{(v, v') \in E | v, v' \in S\})$ is an induced subgraph of G formed from S. A path is a sequence of vertices $p = (v_1, v_2, \ldots, v_k)$ where $(v_i, v_{i+1}) \in E$ for each $1 \leq i < k$. The weight of a path p is defined as $\omega(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$. For two arbitrary vertices s and t, a shortest path p between 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 in G, 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 direct neighbors of a vertex $v \in V$, i.e. $N(v) = \{u \in V \mid (u, v) \in E\}$, and $V(G)$ and $E(G)$ to refer to the set of vertices and edges in G, respectively. Each vertex $v \in V$ is associated with a label $L(v)$. The set of labels $L = \{L(v) \mid v \in V\}$ is called a *distance labelling* over G. A *vertex cut* $V_{cut} \subseteq V$ on G is a subset of vertices whose removal from G splits G into multiple connected components. Vertices in V_{cut} are called *cut vertices*.

Formally, the distance query problem on road networks is defined below.

Definition 2.1 (Problem Definition). Given a road network $G = (V, E)$, the *distance query problem* on G is to compute the distance $d_G(s, t)$ between any two arbitrary vertices $s, t \in V$.

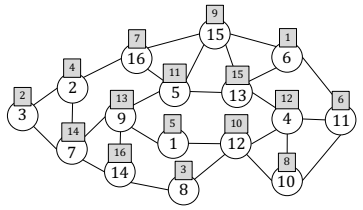
In this work, we study labelling-based techniques to efficiently answer distance queries on road networks.

3 EXISTING SOLUTIONS

The currently fastest known solutions for the shortest-path distance problem on a road network compute a distance labelling L. Such a distance labelling usually satisfies a *2-hop cover property* [15], requiring that the labels of any two vertices must contain at least one common vertex on their shortest-paths, hence called *2-hop labelling*. In the literature, there are three popular labelling techniques that exploit hierarchical structures of road networks for computing 2-hop labellings: 1) hub-based labellings [1, 2], 2) highway-based labellings [4], and 3) tree-decomposition labellings [14, 28]. We now discuss them in detail.

3.1 Hub-Based Labellings

Abraham et al. [1] proposed the first hub-based labelling algorithm, called *Hub-based Labeling* (HL), to construct labels by storing the distances from each vertex to hub vertices (i.e., vertices on shortest paths). The label of each vertex v is thus a set of distance entries $\{(u_1, \delta_{vu_1}), \ldots, (u_k, \delta_{vu_k})\}$ where $\{u_1, \ldots, u_k\} \subseteq V$ and $\delta_{vu_i} = d_G(v, u_i)$. Their work was motivated by the observation that vertices visited by searches of hierarchical and reach-based algorithms [23, 30] form a 2-hop labelling. It turns out that HL produces 2-hop labellings that are much smaller than the worst-case bounds [20].



(a)

Label	Distance Entries
L(14)	(14, 0)
L(13)	(14 3) (13 0)
L(7)	(14 1) (13 3) (7 0)
L(9)	(14 1) (13 2) (7 1) (9 0)
L(4)	(14 3) (13 1) (4 0)
L(5)	(14 2) (13 1) (7 2) (9 1) (5 0)
L(12)	(14 2) (13 2) (9 2) (4 1) (12 0)
L(15)	(14 3) (13 1) (7 3) (9 2) (5 1) (15 0)
L(10)	(14 3) (13 2) (9 3) (4 1) (12 1) (10 0)
L(16)	(14 3) (13 2) (7 2) (9 2) (5 1) (15 1) (16 0)
L(11)	(14 4) (13 2) (4 1) (15 2) (10 1) (11 0)
L(1)	(14 2) (13 3) (7 2) (9 1) (4 2) (12 1) (1 0)
L(2)	(14 2) (13 3) (7 1) (5 2) (15 2) (16 1) (2 0)
L(8)	(14 1) (13 3) (4 2) (12 1) (8 0)
L(3)	(14 2) (13 4) (7 1) (15 3) (16 2) (2 1) (3 0)
L(6)	(14 4) (13 1) (15 1) (10 2) (11 1) (6 0)

(b)

Fig. 1. (a) An example road network, (b) Hub labelling.

Hub-based labellings are not necessarily hierarchical. Later, Abraham et al. [2] studied hierarchical hub labellings with respect to a hierarchy defined by the relationship "vertex v is in the label of vertex u". For any hierarchical hub labelling L, there exists a *canonical labelling*, which is the smallest hierarchical hub labelling with respect to all total vertex orderings that are consistent with the hierarchy of L.

Generally, given any two vertices s and t, hub-based labelling methods compute the distance $d_G(s,t)$ by searching the labels $L(s)$ and $L(t)$ as follows:

$$d_G(s,t) = min\{\delta_{su} + \delta_{tu} : (u, \delta_{su}) \in L(s), (u, \delta_{tu}) \in L(t)\}. \quad (1)$$

Example 3.1. Figure 1(b) shows the canonical labelling for the road network depicted in Figure 1(a) which respects a total vertex ordering 14>13>7>9>4>5>12>15>10>16>11>1>2>8>3>6. The order of each vertex is shown in a box at the top in Figure 1(a). We can see from Table 1 that the labels of any two vertices contain the distance to the highest ranked vertex on their shortest-paths. Consider two vertices 3 and 11, they have one shortest-path $(3, 2, 16, 15, 6, 11)$, among which vertex 15 is of the highest order and thus stored in both labels $L(3)$ and $L(5)$.

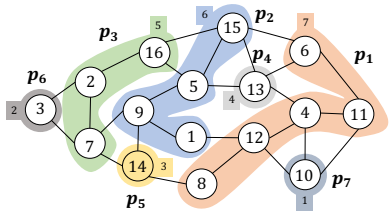
3.2 Highway-Based Labellings

Akiba et al. [4] proposed a highway-based labelling algorithm, namely *Pruned Highway Labelling* (PHL), which generalizes hub-based labellings by explicitly exploiting *highway* structure in road networks. Highways are considered as shortest-paths which are central, i.e., passed through by

many other shortest-paths. Their algorithm first decomposes a road network into a set of disjoint shortest-paths P and then computes a label $L(v)$ for each vertex v which stores the distance from v to the shortest-paths in P. Each label $L(v)$ is a set of triples $\{(p_i, \delta_{u_i u_j}, \delta_{vu_j})\}_{p_i \in P}$ where $p_i \in P$ referring to a shortest-path, $\delta_{u_i u_j} = d_G(u_i, u_j)$ referring to the distance from the starting vertex u_i to another vertex u_j on the shortest path p_i, and $\delta_{vu_j} = d_G(v, u_j)$. To reduce labelling size, a pruned Dijkstra's search is conduced from each shortest-path to construct labels, similar to pruned landmark labelling [5].

Given any two vertices s and t, the distance $d_G(s,t)$ is computed by searching the labels $L(s)$ and $L(t)$ such that

$$d_G(s,t) = min\{\delta_{su_j} + \delta_{tu'_j} + |\delta_{u_i u_j} - \delta_{u_i u'_j}| : (p_i, \delta_{u_i u_j}, \delta_{su_j}) \in L(s), (p_i, \delta_{u_i u'_j}, \delta_{tu'_j}) \in L(t), p_i \in P\}. \quad (2)$$



(a)

Label	Distance Entries
L(6)	(1, 6, 0)
L(11)	(1, 6, 1) (1, 11, 0)
L(4)	(1, 6, 2) (1, 11, 1) (1, 4, 0)
L(12)	(1, 6, 3) (1, 11, 2) (1, 4, 1) (1, 12, 0)
L(8)	(1, 6, 4) (1, 11, 3) (1, 4, 2) (1, 12, 1) (1, 8, 0)
L(1)	(1, 6, 4) (1, 11, 3) (1, 4, 2) (1, 12, 1) (2, 1, 0)
L(9)	(1, 6, 3) (1, 4, 3) (1, 9, 2) (1, 8, 2) (2, 1, 1) (2, 9, 0)
L(5)	(1, 6, 2) (1, 4, 2) (1, 8, 3) (2, 1, 2) (2, 9, 1) (2, 5, 0)
L(15)	(1, 6, 1) (1, 4, 2) (2, 1, 3) (2, 9, 2) (2, 5, 1) (2, 15, 0)
L(7)	(1, 6, 4) (1, 4, 4) (1, 12, 3) (1, 8, 2) (2, 9, 1) (3, 7, 0)
L(2)	(1, 6, 3) (1, 4, 4) (1, 12, 4) (1, 8, 3) (2, 1, 3) (2, 9, 2) (2, 5, 2) (2, 15, 2) (3, 7, 1) (3, 2, 0)
L(16)	(1, 6, 2) (1, 4, 3) (1, 8, 4) (2, 1, 3) (2, 9, 2) (2, 5, 1) (2, 15, 1) (3, 7, 2) (3, 2, 1) (3, 16, 0)
L(13)	(1, 6, 1) (1, 4, 1) (2, 9, 2) (2, 5, 1) (2, 15, 1) (4, 13, 0)
L(14)	(1, 6, 4) (1, 11, 4) (1, 4, 3) (1, 12, 2) (1, 8, 1) (2, 1, 2) (2, 9, 1) (3, 7, 1) (5, 14, 0)
L(3)	(1, 6, 4) (1, 4, 5) (1, 12, 4) (1, 8, 3) (2, 1, 3) (2, 9, 2) (2, 15, 3) (3, 7, 1) (3, 2, 1) (6, 3, 0)
L(10)	(1, 6, 2) (1, 11, 1) (1, 4, 1) (1, 12, 1) (7, 10, 0)

(b)

Fig. 2. (a) Highway decomposition into shortest-paths, (b) Pruned highway labelling.

Example 3.2. Figure 2(a) shows the highway decomposition of the road network into shortest-paths $P = \langle p_1, p_2, p_3, p_4, p_5, p_6, p_7 \rangle$ which are highlighted in different colors. P defines a partial vertex ordering as $p_1 > p_2 > p_3 > p_4 > p_5 > p_6 > p_7$. The order of vertices in each path is shown in a box beside each vertex. Table 2(b) shows the labelling constructed by PHL using the order implied by P. From Figures 2(a) and Table 2(b), we can see that the label of each vertex only stores

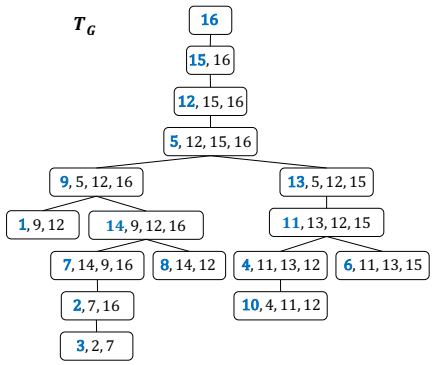
the distances to vertices in the paths with the same or higher orders. Consider two vertices 4 and 8 in p_1, the labels $L(4)$ and $L(8)$ only store the triples from vertices in p_1.

3.3 Tree-Decomposition Labellings

Recently, Ouyang et al. [28] proposed a tree-decomposition labelling method, called *Hierarchical 2-Hop Index* (H2H), to exploit *tree decomposition* structures in road networks. The general idea is to not only construct a 2-hop labelling but also generate a hierarchy among all vertices in a road network using tree decomposition techniques [11]. Based on such a hierarchy, a subset of vertices in labels are visited when answering distance queries. In a later work [14], pruning techniques were proposed to further reduce the number of vertices in labels being visited at query time.

One important property leveraged by tree-decomposition labelling methods is that, for any graph G and any tree decomposition T_G, each internal node of T_G represents a vertex cut on G. This allows to leverage the lowest common ancestor $LCA(s,t)$ of any two vertices s and t in T_G to find a vertex cut that separates s and t. Accordingly, a 2-hop labelling is constructed in H2H such that the label $L(v)$ of each vertex $v \in V(G)$ consists of two arrays: a *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_G, and a *position array* $(p_1, \ldots, p_k)$ that stores positions to $\{w_1, \ldots, w_k\}$ in T_G. Let $L(v).dist$ and $L(v).post$ denote the distance and position arrays in $L(v)$, respectively. The distance between any two given vertices $s, t \in V(G)$ is computed as

$$d_G(s,t) = min\{\delta_{su_i} + \delta_{tu_i} : \delta_{su_i} = L(s).dist(i), \quad \delta_{tu_i} = L(t).dist(i), i \in L(w).post, w = LCA(s,t)\}. \tag{3}$$



(a)

Label	Position	Distance
L(1)	[3 5 6]	[3 3 1 2 1 0]
L(2)	[1 7 8]	[1 2 4 2 2 2 1 0]
L(3)	[7 8 9]	[2 3 4 3 2 2 1 1 0]
L(4)	[3 5 6 7]	[3 2 1 2 1 1 0]
L(5)	[1 2 3 4]	[1 1 3 0]
L(6)	[2 5 6 7]	[2 1 3 2 1 1 0]
L(7)	[1 5 6 7]	[2 3 3 2 1 1 0]
L(8)	[3 6 7]	[4 4 1 3 2 1 0]
L(9)	[1 3 4 5]	[2 2 2 1 0]
L(10)	[3 6 7 8]	[4 3 1 3 2 1 1 0]
L(11)	[2 3 5 6]	[3 2 2 3 2 0]
L(12)	[1 2 3]	[3 3 0]
L(13)	[2 3 4 5]	[2 1 2 1 0]
L(14)	[1 3 5 6]	[3 3 2 2 1 0]
L(15)	[1 2]	[1 0]
L(16)	[1]	[0]

(b)

Fig. 3. Tree decomposition labelling: (a) a tree decomposition T_G of the example road network G shown in Figure 1(a), (b) H2H-Index.

Example 3.3. Figure 3(a) illustrates a tree decomposition T_G of the road network shown in Figure 1(a) while Figure 3(b) shows the H2H index. The label of each vertex stores a position array and a distance array. Consider two vertices 7 and 13, the label $L(7)$ stores a position array $[1, 5, 7]$ which represents the positions (depths) of the ancestors $\{14, 9, 16\}$ inside the node associated with the vertex 7 in T_G shown in Figure 3(a), and a distance array $[2, 3, 3, 2, 1, 1, 0]$ which represents the distances from 7 to all the ancestors of vertex 7 in T_G. For a distance query between vertices 7 and

13, $LCA(7, 13) = 5$ is first obtained, and then using the distances in $L(7)$ and $L(13)$ at positions $[1, 2, 3] \in L(5)$, $d_G(7, 13) = 3$ is obtained according to Equation 3.

3.4 Discussion

Existing hub-based and highway-based labelling solutions search over all the distance entries in labels $L(s)$ and $L(t)$ for a query pair (s, t). Thus, in order to accelerate querying, they exploit a vertex ordering to reduce size of their 2-hop labellings so that they can process smaller labels at query time. It is known that a vertex ordering is crucial for label construction. However, finding a 2-hop labelling of minimum size is NP-hard and finding a vertex ordering that minimizes labelling size is difficult [5, 8, 15]. Thus, hub-based and highway-based labelling solutions still suffer from scalability issues resulting in slow querying when a road network is large.

Existing tree-decomposition labelling solutions assume that a tree decomposition of a road network is given [14, 28]. However, obtaining a tree decomposition with the minimal tree width is a very difficult problem. Even for determining whether a graph G has a tree width of at most a given value, it is known to be NP-complete [6]. Sub-optimal algorithms exist [12], which can compute tree decomposition with a time complexity of $O(|V| \cdot (w^2 + \log(|V|)))$ but may yield very large tree width w and height h. Larger tree width and height take longer in constructing 2-hop labellings and produce larger labelling sizes ($|V| \cdot h$), thus resulting in slow querying. In addition to this, existing tree-decomposition labelling solutions use the Range Minimum Query (RMQ) based algorithms [10] to compute LCA for any two given vertices at query time. Although these algorithms can compute LCA in a tree in time $O(1)$, there are some hidden constant factors and additional computational requirements to achieve this. For instance, they need to precompute a data structure in order to store the information about LCA of all pairs of vertices in a tree. This data structure incurs significant computational overheads as shown in our experiments.

The following example demonstrates the drawbacks of these existing solutions.

Example 3.4. Let us consider a distance query $(3, 10)$ on the road network depicted in Figure 1(a). Based on the hub labelling described in Figure 1(b), HL processes 7 distance entries in $L(3)$ and 6 distance entries in $L(10)$ when computing the distance $d_G(3, 10)$. By contrast, based on the pruned highway labelling depicted in Figure 2(b), PHL processes 10 distance entries in $L(3)$ and 5 distance entries in $L(10)$ when computing the distance $d_G(3, 10)$. However, it is unnecessary to process all of these distance entries because only a small subset of distance entries in labels $L(3)$ and $L(10)$ would be sufficient for computing $d_G(3, 10)$. For instance, only distance entries $(14, 2) \in L(3)$ and $(14, 3) \in L(10)$ in the hub labelling and distance entries $(1, 12, 4) \in L(3)$ and $(1, 12, 1) \in L(10)$ in the highway labelling are needed when computing the distance.

In Table 3(b), H2H stores distances to all ancestors of each vertex along with positions of the ancestors appearing in their associated nodes in T_G. This results in a large labelling size because vertices may appear in multiple nodes in T_G. Further, to compute LCA in constant time, this would require an extra space overhead. For instance, this extra space overhead is 4.64 GB for the whole USA (USA) dataset and 3.69 GB for Western Europe (EUR) dataset as shown in Table 3.

4 OUR SOLUTION

In this section, we present a novel labelling-based solution, referred to as Hierarchical Cut 2-Hop Labelling (HC2L) framework, for shortest-path distance queries on road networks. Let G be a road network. Our solution is to build an efficient *distance scheme* $S = (H_G, L_G, Q)$ over G, where H_G is a vertex hierarchy on G, L_G is a distance labelling on G satisfying the 2-hop cover property, and Q is a query function which, given any two vertices $s, t \in V(G)$, computes $d_G(s, t)$ based on their labels in L and their hierarchical positions in H_G.

4.1 Hierarchy Construction

To design an efficient distance scheme, the choice of a hierarchy is crucial. We observe that binary tree structure may serve as an efficient indexing scheme for road networks. This is because vertices can be indexed in a way that the LCA of any two indexed vertices contains at least one hub vertex on their shortest paths, enabling us to efficiently find them. It is also desirable to keep the height of such a binary tree small, thereby being as balanced as possible. This motivates us to explore *balanced tree hierarchy* defined as follows.

Definition 4.1 (Balanced Tree Hierarchy). Let β be a balancing-parameter with $0 < \beta \leq 0.5$. A *balanced tree hierarchy* is a binary tree $H_G = (\mathcal{N}, \mathcal{E}, \ell)$, where $\mathcal{N}$ is a set of tree nodes, $\mathcal{E}$ is a set of tree edges, and $\ell : V(G) \rightarrow \mathcal{N}$ is a total surjective function. Further, H_G must satisfy the following conditions:

(1) For any internal tree node $N_i \in \mathcal{N}$, its subtrees are balanced:

$$|\textsc{Left}(N_i)|, |\textsc{Right}(N_i)| \leq (1-\beta) \cdot |\textsc{Subtree}(N_i)|$$

where $\textsc{Right}(\cdot)$ and $\textsc{Left}(\cdot)$ refer to the vertices mapped into the right and left subtrees of a tree node, respectively, and $\textsc{Subtree}(\cdot)$ to vertices mapped into the whole subtree.

(2) For any two vertices $s, t \in V(G)$, the lowest common ancestor of their tree nodes $LCA(s,t)$ contains at least one vertex on a shortest-path between s and t.

Thus, for each tree node at the k-th level of H_G, its subtree has at most $V \cdot (1-\beta)^k$ vertices. This leads to the following lemma.

Lemma 4.2. *Let $\alpha = 1/(1-\beta)$. The height of H_G is bounded by $\log_\alpha(n)$, where $n = |V(G)|$.*

In the following we present our algorithm for constructing a balanced tree hierarchy H_G. At its core, the algorithm recursively computes a balanced cut to partition a road network into smaller components, collectively named *hierarchical balanced cuts*. Note that this is closely related to the minimum balanced vertex separator problem, which is known to be NP-hard [19].

4.1.1 Hierarchical Balanced Cuts. Let G be a road network. The algorithm recursively bisects a graph G in two steps: (1) *Balanced partitioning*: partition an input graph $G' \subseteq G$ into two initial partitions connected via another partition referred to as a *cut region*; (2) *Minimal vertex cuts*: find a minimal vertex cut within the cut region. Each iteration of the algorithm splits G' into two smaller but balanced components that contain the initial partitions, while a minimal vertex cut within the cut region ensures that vertices in such a cut are central in G', i.e., passed through by many shortest-paths between vertices. This is illustrated in Figure 5(a). In the following, we discuss these steps in detail.

Balanced Partitioning. Algorithm 1 provides the pseudo-code for this approach. We start with two vertices v_A and v_B as far apart as possible (Lines 11-12), and assign to each vertex v a *partition weight* (Line 13):

$$pw(v) := d_{G'}(v_A, v) - d_{G'}(v_B, v).$$

Partition weights split vertices into a number of equivalence classes.

Definition 4.3 (Equivalence Class). Given $v \in V(G')$, the equivalence class of v in G' is $\{v' \in V(G') \mid pw(v) = pw(v')\}$.

Then, two initial partitions P'_A and P'_B are created by picking vertices with lowest and highest partition weights, respectively, until the desired balancing condition (e.g. $\beta = 0.3$) is satisfied (Lines 14-15). The remaining vertices form a region, called a *cut region*.

However, there is a complication that needs to be addressed. Suppose that we have $v_A = 2$ and $v_B = 3$ in the partition P_A shown in Figure 5(a), the vertex 7 constitutes a bottleneck that causes all vertices whose shortest-paths to 2 and 3 pass through it to have the same equivalence class because of same partition weight as 7. As a result, nodes from this equivalence class are added to P'_A and P'_B arbitrarily, leading to large cuts. To address this issue, we detect and remove the bottleneck from the graph temporarily and repeat the process for finding a rough partition on the remaining graph (Lines 16-21). The bottleneck itself is then also added to the cut region (Line 22). As this bottleneck removal can lead to the graph being disconnected, we first compute the connected components of G (Line 3). If the largest component $C_{\max}$ contains no more than $(1 - \beta) \cdot |V|$ nodes, the empty cut is already balanced (Lines 9-10). Otherwise we find our initial partition within $C_{\max}$ (Lines 5-7).

It is worth noting that the choice of initial vertices only determines the initial partition and the real optimization happens during the vertex cut step. Picking distant vertices is mainly important to ensure that initial partitions are well separated, thus leaving enough room for a vertex cut to avoid dense clusters. We ran additional experiments (not reported) where we made multiple random choices for the arbitrary node in line 11, then computed a partition for each choice, and picked the one with the smallest cut. This resulted in only minor reductions to labelling size (less than 5% for the graphs examined), and we concluded that further optimization of our initial vertex choices (beyond being far apart) would be unlikely to justify the resulting increase in construction time.

Minimal Vertex Cuts. Algorithm 2 shows the pseudo-code for this approach. To find a minimal vertex cut within a cut region, we case this as a minimal (S, T)-vertex-cut problem by contracting vertices in the initial partitions P'_A and P'_B into single vertices S and T, respectively (Lines 3-11). Here S is adjacent to a vertex v in the cut region iff any vertex in P'_A is, and similar for T. This allows us to reduce this problem to a maximum flow problem using the well-known graph transformation technique [13] and solve it using a variant of Dinitz's algorithm [17] (Line 12).

The flow graph for the example graph in Figure 1(a), resulting from the transformation in [13], is shown in Figure 4(b). *Inner* edges (shown in red) connect the incoming and outgoing copies of a node, and have capacity one. *Outer* edges connect copies of different nodes, and have infinite capacity. Since flow paths alternate between inner and outer edges, and each vertex is incident to only one inner edge, outer edges are effectively limited to capacity one as well. For unit capacity graphs, Dinitz's algorithm requires at most $O(\sqrt{|V|})$ phases [17], and each phase runs in $O(|E|)$. As each phase increases the flow by at least one, the number of phases is also bounded by $|V_{cut}|$. This results in a complexity bound of $O(|E| \cdot \min(\sqrt{|V|}, |V_{cut}|))$.

When extracting a minimal vertex cut from the maximal flow, we have two options: pick for each flow path the node closest to S (amongst those that cannot be reached from S in the residual graph), or the node closest to T (amongst those that cannot reach T in the residual graph). We evaluate both options and pick the more balanced one. For the flow graph in Figure 4(b), this means we pick $\{16, 5, 12\}$ over $\{15, 13, 12\}$. Once a vertex cut V_{cut} has been found and removed from the graph, we assign connected components to either P_A or P_B while maximizing balance (Lines 14-15).

However, if S and T end up with an edge between them, due to an edge between $a \in S$ and $b \in T$, there exists no vertex cut in the cut region. This case occurs mostly at the lower levels of a tree hierarchy. To maintain balance guarantees, we move a and b to the cut region, and connect them to S and T, respectively. This ensures that one of them becomes a cut vertex and the other remains in its original partition.

Lemma 4.4. *Algorithm 1 runs in $O(|E| \cdot \log |V| \cdot k)$, where k is the number of times the check in line 18 succeeds. Algorithm 2 runs in $O(|E| \cdot (\log |V| \cdot k + \min(|V_{cut}|, \sqrt{|V|})))$.*

Algorithm 1: Balanced Partition

Function `BalancedPartition`(G, β)
  if G *is disconnected* **then**
    $CC \leftarrow$ connected components of G
    $C_{\max} \leftarrow$ largest connected component
    if $|C_{\max}| > (1-\beta) \cdot |V|$ **then**
      $(P'_A, C, P'_B) \leftarrow$ `BalancedPartition`$(C_{\max}, \beta)$
      return $(P'_A,\ C \cup (V \setminus C_{\max}),\ P'_B)$
    else
      $C_{2nd} \leftarrow$ 2nd largest connected component
      return $(C_{\max},\ V \setminus (C_{\max} \cup C_{2nd}),\ C_{2nd})$
  // find distant nodes using BFS or Dijkstra
  $v_A \leftarrow$ node furthest away from arbitrary node
  $v_B \leftarrow$ node furthest away from A
  // find balanced partitions
  sort nodes by $pw(v) = d_G(v_A, v) - d_G(v_B, v)$
  $P'_A \leftarrow$ first $\beta \cdot |V|$ nodes
  $P'_B \leftarrow$ last $\beta \cdot |V|$ nodes
  // handle bottlenecks
  $w_A \leftarrow \max\{pw(v) \mid v \in P'_A\}$
  $w_B \leftarrow \min\{pw(v) \mid v \in P'_B\}$
  if $w_A = w_B$ **then**
    $EQ \leftarrow \{v \in V \mid pw(v) = w_A\}$
    $\mathcal{B} \leftarrow \arg\min_{v \in EQ} d_G(v_A, v)$
    $(P'_A, C, P'_B) \leftarrow$ `BalancedPartition`$(G \setminus \mathcal{B}, \beta)$
    return $(P'_A,\ C \cup \mathcal{B},\ P'_B)$
  // ensure P'_A, P'_B are connected
  $P'_A \leftarrow \{v \in V \mid pw(v) \le w_A\}$
  $P'_B \leftarrow \{v \in V \mid pw(v) \ge w_B\}$
  return $(P'_A,\ V \setminus (P'_A \cup P'_B),\ P'_B)$

In practice we found that $k \le 1$ and $\log|V| \le |V_{cut}| \le \sqrt{|V|}$, except for very small subgraphs, resulting in effective time complexities of $O(|E| \cdot \log|V|)$ and $O(|E| \cdot |V_{cut}|)$.

4.1.2 Distance Preservation. There is one issue with applying Algorithm 2. Although this algorithm can partition a graph G into two balanced components P_A and P_B via a vertex cut V_{cut}, the induced subgraphs of G by P_A and P_B are not necessarily distance-preserving.

Definition 4.5 (Distance-Preserving Property). Let $G[P]$ denote an induced subgraph of G by the vertices in a partition P. We say that P is *distance-preserving* iff the following condition is satisfied:

$$\forall x, y \in P. d_{G[P]}(x, y) = d_G(x, y). \tag{4}$$

The example below illustrates the distance-preserving property.

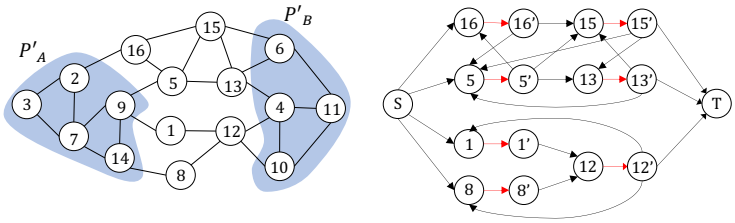


Fig. 4. (a) Initial partitions, (b) Flow graph.

Algorithm 2: Balanced Cut

```
1  Function BalancedCut(G, β)
2      (P'_A, C, P'_B) ← BalancedPartition(G, β)
       // find vertices in cross-partition edges
3      C_A ← P'_A ∩ N(P'_B)
4      C_B ← P'_B ∩ N(P'_A)
       // construct s-t flow graph
5      G_ST ← G[C ∪ C_A ∪ C_B] ∪ {s, t}
6      N_S ← C_A ∪ (C ∩ N(P'_A \ C_A))
7      N_T ← C_B ∪ (C ∩ N(P'_B \ C_B))
8      foreach v ∈ N_S do
9          add edge (s, v) to G_ST
10     foreach v ∈ N_T do
11         add edge (t, v) to G_ST
       // find minimum cut (Dinitz's algorithm)
12     V_cut ← minimum s-t vertex cut on G_ST
       // construct final partitions
13     CC ← connected components of G \ V_cut
14     foreach cc ∈ CC in order of decreasing size do
15         add cc to smaller of P_A, P_B
16     return (P_A, V_cut, P_B)
```

Example 4.6. Consider the road network in Figure 5(a), which is partitioned into P_A and P_B by $V_{cut} = \{5, 12, 16\}$. P_A is not distance-preserving since $d_{G[P_A]}(1, 8) = 3$ but $d_G(1, 8) = 2$. In this case, P_B is distance-preserving because Equation 4 is satisfied.

To characterise how the violation of this distance-preserving property by a partition relates to cut vertices, we introduce the notion of *border vertex*.

Definition 4.7 (Border Vertex). Let (P_A, P_B, V_{cut}) be a balanced cut on a graph G and $P \in \{P_A, P_B\}$. Then a vertex $v \in P$ is a *border vertex* w.r.t. V_{cut} iff there exists an edge $(v, w) \in E(G)$ such that $w \in V_{cut}$.

Let $V_{border}(P) = \{v \in P \mid (v, w) \in E(G), w \in V_{cut}\}$ referring to the set of all border vertices in P on a graph G. We have the following Lemma.

Lemma 4.8. *Given two vertices $x, y \in P$, the following statements are equivalent:*

(1) $d_{G[P]}(x, y) \neq d_G(x, y)$;
(2) All shortest paths between x and y must pass through at least one vertex in V_{cut};
(3) All shortest paths between x and y must pass through at least two vertices in $V_{border}(P)$.

If P is not distance-preserving, according to Lemma 4.8, it would suffice to add *shortcuts* between border vertices in P. This then leads to a distance-preserving subgraph as defined below.

Definition 4.9 (Shortcut-Enhanced Subgraph). A *shortcut-enhanced subgraph* with respect to a partition P, denoted as $G\langle P\rangle = (P, E^* \cup E^\Delta)$, consists of

- the set of vertices in P;
- the set of edges $E^* = \{(v_i, v_j) \in E(G) \mid v_i, v_j \in P\}$;
- the set of shortcuts $E^\Delta = \{(b_i, b_j) \mid \{b_i, b_j\} \in V_{border}(P)\}$ with $\omega(b_i, b_j) = d_G(b_i, b_j)$.

Example 4.10. Let us consider the hierarchical balanced cut presented in Figure 5(a), in which the partition P_A is not distance-preserving. A shortcut set $E^\Delta = \{(1, 8)\}$ with $\omega(1, 8) = 2$ is added, which leads to a shortcut-enhanced subgraph $G\langle P_A\rangle$.

However, not all of the shortcuts in E^Δ are actually needed. The following Lemma characterises redundant shortcuts.

Lemma 4.11. *Let $b_1, b_2 \in V_{border}(P)$. A shortcut between b_1 and b_2 is* redundant *iff one of the following conditions is satisfied:*

(1) $d_{G[P]}(b_1, b_2) = d_G(b_1, b_2)$, or
(2) $d_G(b_1, b_2) = d_G(b_1, b_3) + d_G(b_3, b_2)$ for some border vertex $b_3 \in V_{border}(P)/\{b_1, b_2\}$.

Algorithm 3 describes the pseudo-code for adding shortcuts. We first compute $d_{cut}(b, b')$ in P that is the minimal length of paths between two border vertices b and b' passing through at least one cut vertex in V_{cut} (Line 7). The distance $d_G(b, b')$ between border vertices b and b' is the minimum of the lengths $d_{cut}(b, b')$ and $d_{G[P]}(b, b')$ (Line 8). After obtaining distances between all pairs of border vertices in $G[P]$ and P, we can check the conditions (i) and (ii) in Lemma 4.11 to eliminate redundant shortcuts.

Lemma 4.12. *The time complexity of Algorithm 3 for adding shortcuts between border vertices B is*

$$O(|B| \cdot (|E| \cdot \log |V| + |B| \cdot (|B| + |V_{cut}|))).$$

In practice, the $|B| \cdot (|B| + |V_{cut}|)$ component is negligible, leaving us with an effective complexity of $O(|B| \cdot |E| \cdot \log |V|)$.

4.2 Labelling Construction

In the following, we present our algorithm of constructing distance labellings, namely *Hierarchical Cut 2-Hop Labelling* (HC2L). This algorithm constructs distance labellings upon a vertex quasi-order defined by a balanced tree hierarchy H_G over G. Thus, such distance labellings are said to be *hierarchical*.

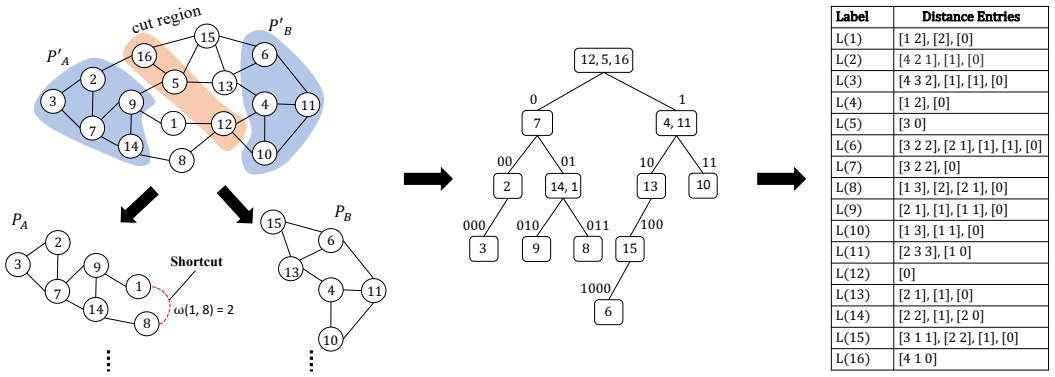
Definition 4.13 (Vertex Quasi-Order). A balanced tree hierarchy H_G defines a *vertex quasi-order* $\preceq$ on $V(G)$ such that $v_i \preceq v_j$ iff $\ell(v_i)$ is an ancestor of $\ell(v_j)$ in H_G, including $\ell(v_j)$ itself.

Definition 4.14 (Hierarchical Cut 2-Hop Labelling). A distance labelling L_G over G is a *hierarchical cut 2-hop labelling* (HC2L) w.r.t. H_G if,

(1) for any label $L(v)$, $v \preceq u$ holds for any vertex u in $L(v)$;

Algorithm 3: Add Shortcuts

```
1  Function AddShortcuts(G, V_cut, P)
2      V_border ← {v ∈ G[P] | (v, w) ∈ E, w ∈ V_cut}
3      foreach b ∈ V_border do
4          d_G[P] ← Dijkstra's search in G[P] starting from b
5          foreach b' ∈ V_border \ {b} do
6              store d_G[P](b, b')
               // distances to cut vertices already known
7              d_cut(b, b') ← min{d_G(b, c) + d_G(c, b') | c ∈ V_cut}
8              d_G(b, b') ← min(d_G[P](b, b'), d_cut)
9      foreach b1, b2 ∈ V_border do
10         if d_G(b1, b2) < d_G[P](b1, b2) then
11             redundant ← False
12             foreach b3 ∈ V_border \ {b1, b2} do
13                 if d_G(b1, b3) + d_G(b3, b2) = d_G(b1, b2) then
14                     redundant ← True
15             if not redundant then
16                 add shortcut (b1, b2, d_G(b1, b2))
```



Label	Distance Entries
L(1)	[1 2], [2], [0]
L(2)	[4 2 1], [1], [0]
L(3)	[4 3 2], [1], [1], [0]
L(4)	[1 2], [0]
L(5)	[3 0]
L(6)	[3 2 2], [2 1], [1], [1], [0]
L(7)	[3 2 2], [0]
L(8)	[1 3], [2], [2 1], [0]
L(9)	[2 1], [1], [1 1], [0]
L(10)	[1 3], [1 1], [0]
L(11)	[2 3 3], [1 0]
L(12)	[0]
L(13)	[2 1], [1], [0]
L(14)	[2 2], [1], [2 0]
L(15)	[3 1 1], [2 2], [1], [0]
L(16)	[4 1 0]

Fig. 5. An illustration of: (a) Hierarchical balanced cut, (b) Balanced tree hierarchy, and (c) Hierarchical cut 2-hop labelling, for the road network illustrated in Figure 1(a).

(2) for any two vertices $\{s, t\} \subseteq V$ there exists $r \in LCA(s, t)$ with $(r, \delta_{sr}) \in L(s)$, $(r, \delta_{tr}) \in L(t)$ and $\delta_{sr} + \delta_{tr} = d_G(s, t)$.

Condition (1) states that L_G is *hierarchical* in terms of the vertex quasi-order $\preceq$ defined by H_G. Condition (2) stipulates that L_G is a 2-hop labelling for which the distance between s and t can be computed from distance entries corresponding to $LCA(s, t)$ in their labels alone.

4.2.1 Upper and Lower bounds. Assume that we are given a balanced tree hierarchy $H_G = (\mathcal{N}, \mathcal{E}, \ell)$. We start with a naive labelling algorithm to construct labels. Let $L(v) = \emptyset$ for each vertex $v \in$

$V(G)$. Then for each tree node $N_i \in \mathcal{N}$, we conduct Dijkstra's search from every $u \in N_i$ to compute the shortest-path distance $d_G(u,v)$ and add it to $L(v)$ for all $v \in \textsc{Subtree}(N_i)$, i.e., $L(v) = L(v) \cup \{(u, d_G(u,v))\}$. Thus, for each vertex $v \in V(G)$, $L(v)$ stores $|A_v|$ distance entries, where $A_v = \{u \in V(G) | u \preceq v\}$.

Lemma 4.15. *The space complexity of the naive labelling is bounded by* $O(\sum_{v \in V} |A_v|)$.

Indeed, the naive labelling approach provides the upper bound of the labelling size for hierarchical cut 2-hop labelling. It suffers from the following drawbacks: 1) it can produce very large labelling sizes for large road networks such as the whole USA and Central Europe road networks, 2) queries may perform unnecessary computations. The question arises: *can we prune distance entries to reduce labelling size and accelerate queries?* To answer this question, we first exploit a new labelling property, called *cut cover* property.

Definition 4.16 (Cut Cover). Let $V_{cut} \subseteq V(G)$ be a vertex cut on G. Then, for any vertex $v \in G(V) \backslash V_{cut}$ and any cut vertex $u \in V_{cut}$, $(u, \delta_{uv}) \in L(v)$ iff there is no other cut vertex $u' \in V_{cut}$ satisfying:

$$d_G(u,v) = d_G(u,u') + d_G(u',v). \tag{5}$$

Assume that distances among two cut vertices in any vertex cut of T_G are known, denoted as δ_{cut}. The cut cover property ensures that the label of every non-cut vertex contains the distance information to cut vertices - either directly or indirectly through another cut vertex. Thus, the cut cover property relates to the lower bound of the labelling size for hierarchical cut labelling.

Lemma 4.17. *Let L^* be a hierarchical cut 2-hop labelling satisfying the cut cover property. Then the intersection of all hierarchical cut 2-hop labellings of H_G is L^*, i.e., $u \in L^*(v)$ iff $u \in L(v)$ for all hierarchical cut 2-hop labellings L of H_G.*

If a hierarchical cut 2-hop labelling L satisfies the cut cover property, then L is minimal, i.e., if removing any distance entry from $\{L(v)\}_{v \in V \backslash V_{cut}}$, then there must exist two vertices $s, t \in V(G)$ whose distance $d_G(s,t)$ cannot be computed from $\{L(v)\}_{v \in V \backslash V_{cut}}$ and δ_{cut}. Note that not all hierarchical cut 2-hop labellings can satisfy the cut cover property because it does not guarantee 2-hop labelling. Below we introduce a new labelling method, called *tail pruned labelling*, which guarantees that: 1) the resultant labelling is still 2-hop, and 2) the labelling size is small and within the upper and lower bounds, thereby enabling efficient querying.

4.2.2 Tail Pruned Labelling. Given a balanced cut (P_A, P_B, V_{cut}) on a graph $G = (V, E)$, we rank each cut vertex $v \in V_{cut}$ based on the following equation:

$$r(v) = |\{u \in V \mid \exists v' \in V_{cut} \setminus \{v, u\} \text{ with } sp(v, v', u)\}|, \tag{6}$$

where $sp(v, v', u)$ denotes that v' lies on a shortest path between v and u, i.e., $d_G(v,u) = d_G(v,v') + d_G(v',u)$. Ties are broken arbitrarily to obtain a total ordering. Intuitively, vertices in a vertex cut are assigned a rank in terms of how frequently they can be hit by other cut vertices in their shortest-paths to other vertices. We then prune as follows.

Definition 4.18 (Tail Pruning). A cut vertex $v_i \in V_{cut}$ is tail-pruned from $L(u)$ iff

(1) $\exists v_j \in V_{cut}$ with $r(v_j) < r(v_i)$ and $sp(v_i, v_j, u)$, and
(2) $\forall v_k \in V_{cut}$ with $r(v_i) < r(v_k)$, v_k is tail-pruned from $L(u)$.

Informally speaking, in the above definition, Condition (1) ensures that a 2-hop labelling remains after pruning, and Condition (2) allows us to omit vertex identifiers in labels. Algorithm 5 describes the pseudo-code for the tail pruned labelling approach. A modified version of the Dijkstra's algorithm is shown in Algorithm 4 which computes the distances from a given (cut) vertex *root*

Algorithm 4: Dijkstra with pruneability tracking

```
1  Function DistAndPrune(G, root, P)
2      dist(v ∈ G) ← ∞
3      add (root, 0, False) to Q
4      while Q is not empty do
           // Q is ordered by (d, p) with True < False
5          (v, d, p) ← pop_min(Q)
6          if dist(v) = ∞ then
7              dist(v) ← (d, p)
8              for (v, w, l) ∈ G do
9                  if v ∈ P then
10                     add (w, d + l, True) to Q
11                 else
12                     add (w, d + l, p) to Q
13     return dist
```

and tracks whether a shortest path from *root* to a vertex $v \in G$ passes through another (cut) vertex in a given set P. In Algorithm 5, we compute the ranks of cut vertices as per Equation 6 (Lines 2-5). Afterwards, Algorithm 4 is invoked with P containing only cut-vertices of lower ranks, allowing condition (1) of Definition 4.18 to be checked (Line 7). Finally, the list of distance values is tail-pruned in accordance with Definition 4.18 (Lines 8-10).

Example 4.19. Let us consider $V_{cut} = \{5, 12, 16\}$ on the road network in Figure 5. Evaluating Equation (6) gives us the ranking $r(12) < r(5) < r(16)$. As a result, $L(1) = \{(12, 1), (5, 2), (16, 3)\}$ is represented as $[1, 2, 3]$ and can be tail-pruned to $[1, 2]$. For $L(2) = \{(12, 4), (5, 2), (16, 1)\}$, represented as $[4, 2, 1]$, no tail-pruning is possible. Although vertex 16 lies on a shortest path between vertices 5 and 2, it has a greater rank than 5, so $5 \in L(2)$ does not meet Condition (1) of Definition 4.18. Nor does it meet Condition (2), since $16 \in L(2)$ does not meet Condition (1).

We may construct a hierarchical cut 2-hop labelling by applying the pruned landmark labelling (PLL) method [5], which can conduct a pruned breadth-first search (BFS) from each cut vertex. However, PLL does not suit our data structure design for labelling because the way PLL reduces distance entries still requires a full scan on distance arrays (will discuss in the next paragraph) in the labels of vertices in a distance query. As a result, the query performance deteriorates in comparison with our tail pruned labelling method.

We implement an efficient data structure to leverage the advantages of our balanced tree hierarchy H_G. Since H_G is a binary tree, we represent each tree node using a bitstring of length equal to its level (its distance to the root) in H_G. When $\beta = 1/3$ and G contains no more than $2/3^{-58} \approx 16.3$ billion vertices, binary strings (including their 6-bit length) can be stored as 64-bit integers. Furthermore, unlike other approaches, we only store distance values in labels, which reduces storage requirements by half (for vertex identifiers and distance values of equal size). Thus, the label of each vertex is a list of distance arrays, each corresponding to a vertex cut. This enables us to search only a reduced set of hub vertices from exactly one vertex cut in the labels of any query pair.

Before constructing labels, we contract the graph by repeatedly removing degree-one vertices. Most shortest-paths from a degree-one vertex to other vertices pass through its closest vertex in

Algorithm 5: Labelling with tail pruning

```
1  Function Labelling(G, V_cut)
       // compute cut cover pruneability
2      foreach v ∈ V_cut do
3          dist ← DistAndPrune(G, v, V_cut \ {v})
4          P_#(v) ← |{v ∈ G | dist(v).p = True}|
       // rank cut vertices
5      [v_0, ..., v_n] ← order V_cut by P_#
       // compute pruneability
6      for i = 0 ... n do
7          dist_i ← DistAndPrune(G, v_i, {v_0, ..., v_{i-1}})
       // create labels with tail pruning
8      foreach v ∈ G do
9          k ← max{i ∈ 0 ... n | dist_i(v).p = False}
10         L(v) ← [dist_0(v).d, ..., dist_k(v).d]
11     return L
```

the contracted graph, called its *root*. For each degree-one vertex we store its distance to its root, as well as a reference to the root. This allows us to compute distances between two vertices v and w by using the references and then adding the distances stored.

However, this approach only works when the roots of v and w lie on all (shortest) paths between them, which may not be the case when v and w have the same root. To deal with this case efficiently, we observe that contracted vertices with the same root form a tree (with their common root designated as its root), and store for each contracted vertex its tree parent. This allows us to follow the paths from v and w to their lowest common ancestor u in the tree using reference information alone, and compute their distance as

$$d_G(v, w) = d_G(v, root) + d_G(w, root) - 2 \cdot d_G(u, root).$$

A similar approach is taken by PHL in [4]. The difference is that they only prune vertices that have degree one in the original graph. This ensures that all paths between different vertices must pass through their roots, but reduces contraction effectiveness (from ~30% to ~20% for the graphs we experimented on).

4.3 Query Processing

Now we describe how our query function Q efficiently answers distance queries on a road network G, given a balanced tree hierarchy H_G and a hierarchical cut 2-hop labelling L_G.

Let (s, t) be a query pair with $s, t \in V(G)$. We process the query for (s, t) in two steps: (1) In the first step, we compute the lowest common ancestor $LCA(s, t)$ of the tree nodes $\ell(s)$ and $\ell(t)$ in H_G (Line 2); (2) In the second step, we compute the distance between s and t using the information in LCA and the label sets $L(s)$ and $L(t)$ in L_G as follows:

$$\begin{aligned} d_G(s, t) = & min\{\delta_{sr} + \delta_{tr} : \\ & \delta_{sr} \in L(s), \delta_{tr} \in L(t), r \in LCA(s, t)\}. \end{aligned} \tag{7}$$

As distances to cut vertices are organized by level, we only need to find the level of $LCA(s, t)$, i.e., the length of its identifying bitstring. This can be computed as the number of leading zeros of

the XOR of the bitstrings of s and t – operations that are naively supported by most CPUs and thus extremely fast.

Example 4.20. Consider the balanced tree hierarchy H_G and the hierarchical cut 2-hop labelling L_G in Figure 5(b) and 5(c). The distance query $(14, 15)$ first finds $LCA(14, 15) = \{12, 5, 16\}$, the first cut in H_G, by comparing the bitstrings 01 and 100. The distances between 14 and $\{12, 5, 16\}$ are $[2, 2, 3]$, which are found as the first distance array in $L(14)$, with the last distance value 3 removed by tail-pruning. Similarly, the distances between 15 and $\{12, 5, 16\}$ are found to be $[3, 1, 1]$. We then compute $d_G(14, 6)$ as $\min(2+3, 2+1)$, ignoring the last distance value 1 in $[3, 1, 1]$ as its counterpart was pruned.

Lemma 4.21. *Given a balanced tree hierarchy H_G, and a query pair (s, t), $LCA(s, t)$ in H_G can be obtained in time $O(1)$.*

Lemma 4.22. *Given a distance query (s, t), the distance $d_G(s, t)$ can be computed using Equation 7 in $O(V_{cut})$, where V_{cut} is the largest cut in the balanced tree hierarchy H_G.*

4.4 Parallelization

To speed up construction, we can use multi-threading to parallelise certain tasks. Whenever we partition a (sufficiently large) subgraph, we create a new thread in order to process two partitions in parallel. In each thread, we further parallelise computing distances for labels, shortcuts, and pruning in each thread, i.e., we perform a Dijkstra search from each cut (or border) node. These searches can easily be carried out in parallel, with workloads being practically identical. Together these tasks account for the majority of labelling construction process. We must note however that effective parallelization is limited to a handful of physical threads, as a significant part of the work (>10%) is not or not fully parallelised – mainly due to early cut computations.

5 EXPERIMENTS

Hardware and Platform All the experiments are 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++11 and compiled using g++ 9.4.0 with the -O3 option.

Datasets We use 10 undirected real road networks, nine of them are from the US and publicly available at the webpage of the 9th DIMACS Implementation Challenge [16] and one is from Western Europe managed by PTV AG [3]. Table 1 summarizes these datasets in which the largest dataset is the whole road network in the USA. Further, we consider two versions for these datasets which have different units of edge weights i.e., distances and travel times.

Baseline Methods We compared our proposed algorithm HC2L with four state-of-the-art algorithms for shortest-path distance queries in road networks as follows, 1) Hub Labelling (HL) [2], 2) Pruned Highway Labelling (PHL) [4], 3) Hierarchical 2-Hop Labelling (H2H) [28], and 4) Projected Vertex Separator Based 2-Hop Labelling (P2H) [14].

The code for H2H, PHL and HL was publicly available and implemented in C++. Unfortunately, we could not obtain the implementation of P2H; we reported only the part of experimental results presented in [14]. They implemented P2H in C++ and their experiments were conducted on a machine with Quad Intel(R) Xeon(R) Platinum 8160 24-core @ 2.10GHz CPU and 768 GB RAM, running CentOS Linux 7. We use the same parameter settings as suggested by the authors of these methods, unless otherwise stated. We select balance partition threshold $\beta = 0.2$ for HC2L. Furthermore, we set the number of threads equal to the available 28 cores.

Benchmark Generation To evaluate the query performance, we randomly sampled 1,000,000 pairs of vertices from all pairs of vertices in each road network, i.e., $V \times V$. We also evaluate the

Table 1. Summary of datasets.

Dataset	Region	$\|V\|$	$\|E\|$	*diam.*	Memory
NY	New York City	264,346	733,846	720	17 MB
BAY	San Francisco	321,270	800,172	721	18 MB
COL	Colorado	435,666	1,057,066	1,245	24 MB
FLA	Florida	1,070,376	2,712,798	2,058	62 MB
CAL	California	1,890,815	4,657,742	2,315	107 MB
E	Eastern USA	3,598,623	8,778,114	4,461	201 MB
W	Western USA	6,262,104	15,248,146	4,420	349 MB
CTR	Central USA	14,081,816	34,292,496	5,533	785 MB
USA	United States	23,947,347	58,333,344	8,440	1.30 GB
EUR	Western Europe	18,010,173	42,560,279	5,175	974 MB

Table 2. Comparison of query times, labelling sizes and construction times with distances as edge weights between our method, i.e., HC2L, and the baseline methods. HC2L^p represents HC2L with parallelism.

Dataset	Query Time [μs]				Labelling Size				Construction Time [s]				
	HC2L	H2H	PHL	HL	HC2L	H2H	PHL	HL	HC2L	HC2L^p	H2H	PHL	HL
NY	0.225	0.432	0.983	0.765	144 MB	341 MB	320 MB	233 MB	15	6	16	34	32
BAY	0.220	0.563	0.707	0.665	113 MB	339 MB	235 MB	219 MB	12	4	12	18	27
COL	0.351	0.750	0.909	0.720	236 MB	217 MB	403 MB	341 MB	27	12	21	38	45
FLA	0.371	0.754	0.965	0.827	487 MB	1.25 GB	1.14 GB	907 MB	68	23	46	121	137
CAL	0.442	1.125	1.106	0.958	1.24 GB	3.87 GB	2.58 GB	1.78 GB	215	57	146	327	318
E	0.555	1.241	1.671	1.218	3.37 GB	9.81 GB	8.44 GB	4.74 GB	654	163	409	1,578	1,149
W	0.583	1.382	1.661	1.163	5.71 GB	18.3 GB	13.5 GB	7.50 GB	1,197	261	702	2,314	1,654
CTR	0.760	1.630	2.503	1.613	24.4 GB	73.9 GB	55.9 GB	25.5 GB	6,203	1,658	4,029	15,882	7,591
USA	0.737	1.940	2.389	1.663	45.1 GB	155 GB	95.6 GB	44.7 GB	11,203	1,977	7,737	26,515	13,157
EUR	0.922	2.414	2.239	1.673	44.1 GB	160 GB	70.9 GB	34.1 GB	12,242	3,083	9,194	20,466	8,728

query performance of the algorithms by varying the distance between the source and target vertices in a query similar to [28, 29]. Specifically, for each road network, we generate 10 sets of queries $Q_1, Q_2, \ldots, Q_{10}$ as follows: we set l_{min} to be 1000 meters, and set l_{max} to be the maximum distance of any pair of vertices in the map. Let $x = (\frac{l_{max}}{l_{min}})^{1/10}$. For each $1 \leq i \leq 10$, we generate 10,000 queries to form each set Q_i, in which the distance of the source and target vertices for each query falls in the range $(l_{min} \cdot x^{i-1},\ l_{min} \cdot x^i]$. For each algorithm, we report the average query processing time.

5.1 Performance Evaluation

We compare the performance of our proposed algorithm with the baseline methods in terms of the query time, labelling size, and construction time. The experimental results are presented in Table 2, Table 4, and Figure 6.

5.1.1 Querying Time. In Table 2, we report for each method and dataset the average query time over 1 million random distance queries, where edge weights are distances. We confirm that HC2L is the fastest on all the datasets. In most cases, we are 1.5-2.5 times faster than HL, 2-3 times faster than H2H, and 2-4 times faster than PHL. The is because we process a significantly smaller number of distance entries in the labels of a query pair when computing their shortest-path distance. Table 4

Table 3. Comparing LCA storage requirements and average hub size by HC2L and the baseline algorithms for all the datasets with distances as edge weights.

Dataset	LCA Storage		Average Hub Size (AHS)				
	HC2L	H2H	HC2L	P2H	H2H	PHL	HL
NY	2.02 MB	40.3 MB	12	19	34	50	137
BAY	2.45 MB	49.0 MB	9	–	51	22	105
COL	3.32 MB	66.5 MB	20	24	66	39	118
FLA	8.17 MB	180 MB	13	21	50	37	132
CAL	14.4 MB	317 MB	23	36	109	39	154
E	27.5 MB	632 MB	36	42	164	98	213
W	47.8 MB	1.12 GB	37	67	171	99	192
CTR	108 MB	2.62 GB	66	101	225	185	132
USA	183 MB	4.64 GB	62	125	331	160	138
EUR	137 MB	3.49 GB	142	–	749	154	310

Table 4. Comparison of query times, labelling sizes and construction times with travelling times as edge weights between our method and the baseline methods.

Dataset	Query Time [μs]				Labelling Size				Construction Time [s]				
	HC2L	H2H	PHL	HL	HC2L	H2H	PHL	HL	HC2L	HC2L^p	H2H	PHL	HL
NY	0.210	0.531	0.574	0.593	133 MB	272 MB	137 MB	152 MB	16	6	14	11	19
BAY	0.213	0.537	0.475	0.549	106 MB	258 MB	103 MB	154 MB	13	4	11	7	18
COL	0.327	0.815	0.554	0.560	189 MB	523 MB	166 MB	221 MB	26	14	21	12	27
FLA	0.361	0.776	0.625	0.577	466 MB	1.07 GB	478 MB	606 MB	74	23	46	40	84
CAL	0.427	1.033	0.638	0.697	1.16 GB	3.11 GB	905 MB	1.07 GB	231	51	130	78	173
E	0.479	1.203	0.820	0.819	3.01 GB	8.47 GB	2.41 GB	2.43 GB	706	176	360	256	503
W	0.551	1.281	0.837	0.821	5.29 GB	16.7 GB	3.85 GB	4.10 GB	1,277	315	693	401	770
CTR	0.701	2.054	1.037	0.977	23.2 GB	84.4 GB	11.3 GB	11.4 GB	7,296	1,728	4,127	1,513	2,954
USA	0.723	1.864	1.141	1.018	38.4 GB	135 GB	22.4 GB	20.3 GB	11,249	2,039	6,988	3,124	4,752
EUR	0.872	2.720	1.297	1.106	38.3 GB	169 GB	20.9 GB	17.9 GB	13,292	3,116	10,649	3,488	4,502

reports the results when we consider travel times, rather than distances, as edge weights for each dataset. We notice that the average query times for almost all the methods become faster compared to their corresponding results in Table 4. The reason for this speedup lies in the reduced labelling sizes which will be further discussed in Section 5.1.2. Evidently, HC2L is still the fastest on all the datasets.

Table 3 shows the average number of hubs for which the sum of distances is computed when evaluating Eq. (1) or Eq. (7). Compared to H2H, HC2L produces minimal cuts which are very small in practice as shown in Table 5, and thus the search space of HC2L is significantly reduced for distance queries. The query time of PHL and HL depends on the label sizes of a query pair. We can see from Table 3 that the average number of hubs of PHL and HL are much bigger than HC2L, which make them significantly underperform HC2L. Furthermore, PHL is generally slower than HL as it employs highways instead of vertices as hubs, making distance computation more complex than simply adding two distance values.

Varying Distance Querying In Figure 6, we report results for distance query sets containing query pairs with varying distances to test the performance of all the algorithms. We can clearly see

Table 5. Comparing Tree Height and Max Cut Size on all the datasets with distances as edge weights.

Dataset	Tree Height		Max Cut Size/Width	
	HC2L	H2H/P2H	HC2L	H2H/P2H
NY	24	399	40	171
BAY	25	297	41	120
COL	27	376	42	192
FLA	29	345	34	140
CAL	30	603	56	266
E	30	869	75	328
W	32	895	89	342
CTR	35	1,771	153	667
USA	35	2,135	178	855
EUR	36	2,892	280	1,181

that HC2L significantly outperforms all the baseline methods in every query set. Particularly, our method HC2L shows good performance for both short and long distance query sets. Regardless of distance, only cut vertices of the LCA of the query vertices need to be considered as hubs. Vertex cuts tend to be smaller at lower levels of the hierarchy, thus making local queries generally faster. Exceptions to this exist though – the NY dataset has a top-level cut of size 5, making distant queries very fast as well. Additionally, queries involving top-level cuts enjoy better caching, due to the memory layout of labels. H2H and HL show similar behaviour. In contrast, PHL has rather poor query performance for local queries – this happens because it optimizes its index for high-speed highways, which are less likely to intersect with shortest paths between local vertex pairs. Optimizing indexes for real-world workloads is an interesting problem that often requires specific algorithmic designs as discussed in [35].

5.1.2 Labelling Size. Table 2 shows that the labelling sizes produced by our proposed method HC2L is significantly smaller than the baseline methods when indexing for distance queries. In most cases, the labelling size of HC2L is about 2-4 times smaller than H2H, 2-3 times smaller than PHL and 1-2 times smaller than HL. The only exception is the EUR dataset, for which HL produces a smaller labelling. Despite this, HC2L still beats HL in terms of query time on EUR. This is because HC2L only uses a fraction of labels during query answering, although the gap does become smaller. We have also analysed the extra overhead of labelling sizes by H2H and HC2L to find LCA in constant time in Table 3. It shows that the overheads incurred by H2H are about 20 times greater – this happens because the tree hierarchies of H2H are neither binary nor balanced, thus requiring a different indexing approach. We found that without tail pruning index sizes grow by 10-15%, but construction time is reduced by around 20%. Table 4 shows labelling sizes for datasets using travel times as edge weights. Here HC2L produces slightly smaller labelling sizes than the corresponding labelling sizes in Table 2, H2H is roughly the same, but for PHL and HL labelling sizes are reduced significantly when changing edge weights from distances to travelling times. The reason for this is that PHL and HL can exploit better orderings using travel times as edge weights, which leads to better pruning. Further, we notice that HL pruning is more effective than the tail pruning approach employed by HC2L. This is because HL pruning affects the whole vertex ordering, while the tail pruning by H2CL only affects the vertex ordering within each cut; thus, one can expect HL to perform better in the cases where a large percentage of labels can be pruned.

5.1.3 Construction Time. In Table 2 we compare the construction time of our method HC2L with the baseline methods when indexing for distance. We observe that the single-threaded implementation of HC2L is slower than H2H, faster than PHL and comparable to HL, but the parallel variant of our method, denoted by HC2L^p, significantly outperforms the baseline methods on all the datasets. The construction time of our algorithms includes both the time for obtaining a balanced tree hierarchy and the time for constructing a hierarchical cut labelling. When considering travel times as edge weights, as shown in Table 4, PHL and HL perform significantly faster in construction, compared to considering distances as edge weights in Table 2; nonetheless, these methods are still slower than our parallel implementation. This increase in construction time is consistent with the decrease in labelling size due to better pruning, as discussed earlier.

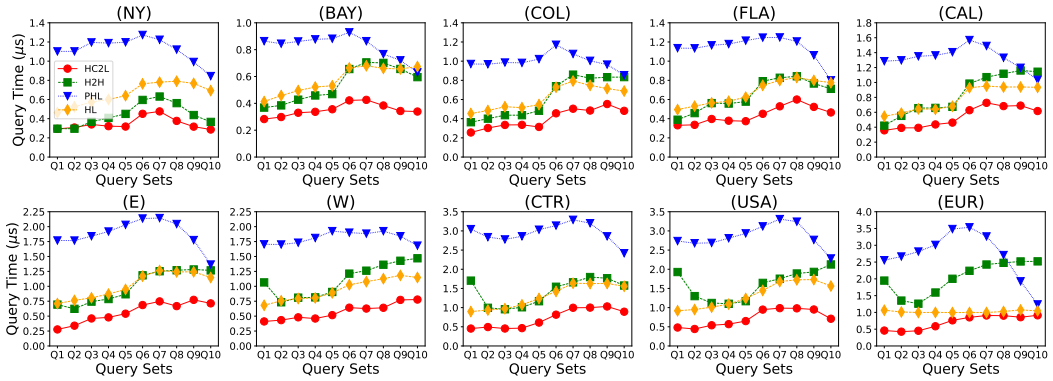


Fig. 6. Distance query performance under varying distances for all the datasets with distances as edge weights.

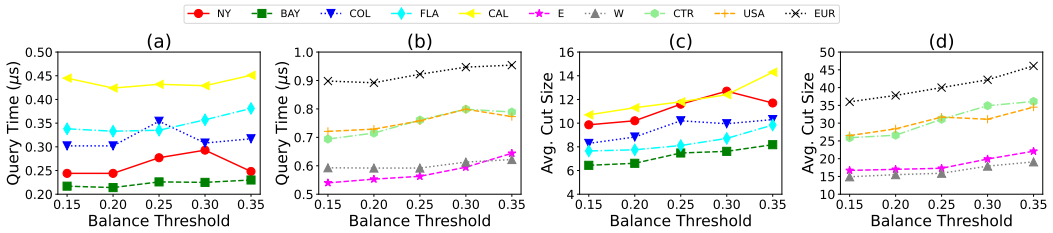


Fig. 7. Query performance with respect to cut size under varying balance partition thresholds for all the datasets with distances as edge weights.

5.2 Analysis of the Proposed Approaches

We evaluate how the balance threshold β, cut size and tree height affect the performance of our method. Figure 7 shows the average query times and cut sizes under varying balance thresholds β on all the datasets. We can observe that whenever a cut size increases/decreases under a particular threshold, a similar trend reflects in query time of that threshold. The average query times decrease or remain the same on the majority of datasets when we increase β from 0.15 to 0.20 and for thresholds over 0.20 seem increasing. This also aligns well with the cut sizes, and using the larger balance threshold 0.20 leads to more balanced trees.

In Table 5, we also report the height and maximum cut size/width of our method HC2L with a balance threshold 0.20 against the baseline methods H2H and P2H. It is clear that HC2L produces much smaller cuts and has significantly smaller heights for all datasets compared to H2H and P2H. This is because our method is based on novel techniques to find balanced partitioning and small cuts which result in a balanced binary tree with a very small height. In contrast, H2H and P2H use heuristic techniques to decompose a road network into a tree which may produce arbitrary height and width of a very large number in practice as shown in Table 5.

5.3 Extension to Directed Graphs

Our approach can be extended to directed graphs by storing distances from both directions in each label. We can compute them by performing searches on both directions when adding shortcuts as well as conducting label construction. To compute vertex cuts, we can treat all edges as undirected to ensure that they separate paths in both directions. However, road networks tend to be *almost* undirected (with some famous exceptions such as Stockholm), in which case the two distances stored within each label will frequently be identical – this can be exploited for optimizations.

5.4 Remarks

Before finishing this section, we comment on some possible extensions and open problems related to our method proposed in this work:

- Although our parallel implementation reduces construction time to less than one hour for all graphs, further increasing the number of cores has a limited impact on performance. This is because most cores would remain poorly utilized. Particularly, better utilization requires the vertex cut computation to be parallelizable, which seems tricky to do and is outside the scope of this paper.
- In dynamic settings, e.g., due to roads being temporarily closed or suffering from slower travel speeds, we need to update our labellings, preferably without recomputing them from scratch. Our balanced tree hierarchy construction does not depend on edge weights, except for shortcuts. This should enable us to preserve a balanced tree hierarchy (with some adjustments for shortcuts) and limit updates only to distance values. This is in contrast to existing approaches such as HL or PHL which rely on edge weights for node or highway ordering.
- Existing labelling-based methods for distance queries on road networks still require labellings of sizes larger than the original graph. Thus, how to reduce labelling sizes needed for distance queries on road networks without compromising query performance still remains an open problem.

6 CONCLUSION

In this paper, we analyse the drawbacks of the current state-of-the-art labelling-based solutions in order to exploit hierarchical structures of road networks for efficiently answering distance queries. We propose a novel solution called hierarchical cut labelling (HC2L) to overcome the drawbacks of existing solutions. Our proposed solution uses a novel *balanced tree hierarchy* to find a partial vertex order which helps in significantly reducing labelling size and selecting a small subset of labels for a distance query pair. Accordingly, query processing is accelerated, regardless of distance between the node pairs queried. As demonstrated experimentally, index size and construction time are competitive as well. For future work, we plan to investigate dynamic updates to our index structure, and efficient algorithms for the balanced minimal S-T cut problem.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2011. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *Proceedings of the 10th International Conference on Experimental Algorithms*. 230–241.

[2] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2012. Hierarchical Hub Labelings for Shortest Paths. In *Proceedings of the 20th Annual European Conference on Algorithms*. 24–35.

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

[4] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. 2014. Fast shortest-path distance queries on road networks by pruned highway labeling. In *Proceedings of the 16th workshop on algorithm engineering and experiments (ALENEX)*. 147–154.

[5] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 349–360.

[6] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. 1987. Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods* 8, 2 (1987), 277–284.

[7] Julian Arz, Dennis Luxen, and Peter Sanders. 2013. Transit node routing reconsidered. In *Proceedings of the 12th International Symposium on Experimental Algorithms*. 55–66.

[8] Maxim Babenko, Andrew V Goldberg, Haim Kaplan, Ruslan Savchenko, and Mathias Weller. 2015. On the complexity of hub labeling. In *40th International Symposium Mathematical Foundations of Computer Science (MFCS)*. 62–74.

[9] Holger Bast, Stefan Funke, and Domagoj Matijevic. 2006. Transit ultrafast shortest-path queries with linear-time preprocessing. *9th DIMACS Implementation Challenge [1]* (2006).

[10] Michael A. Bender and Martín Farach-Colton. 2000. The LCA Problem Revisited. In *LATIN 2000: Theoretical Informatics*, Gaston H. Gonnet and Alfredo Viola (Eds.). Springer Berlin Heidelberg, 88–94.

[11] Hans L. Bodlaender. 1993. A Tourist Guide through Treewidth. *Acta Cybern.* 11, 1-2 (1993), 1–21.

[12] Hans L Bodlaender. 2006. Treewidth: characterizations, applications, and computations. In *Graph-Theoretic Concepts in Computer Science: 32nd International Workshop*. Springer, 1–14.

[13] J. A. Bondy and U. S. R. Murty. 1976. *Graph Theory with Applications*. Elsevier.

[14] 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 ACM SIGMOD International Conference on Management of Data*. 313–325.

[15] 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.

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

[17] Yefim Dinitz. 2006. Dinitz' Algorithm: The Original Version and Even's Version. In *Theoretical Computer Science, Essays in Memory of Shimon Even (Lecture Notes in Computer Science, Vol. 3895)*, Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman (Eds.). 218–240.

[18] DongKai Fan and Ping Shi. 2010. Improvement of Dijkstra's algorithm and its application in route planning. In *Proceedings of the 7th international conference on fuzzy systems and knowledge discovery*, Vol. 4. 1901–1904.

[19] Uriel Feige and Mohammad Mahdian. 2006. Finding small balanced separators. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, Jon M. Kleinberg (Ed.). 375–384.

[20] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. 2004. Distance labeling in graphs. *Journal of Algorithms* 53, 1 (2004), 85–112.

[21] 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*. 319–333.

[22] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Vol. 5. 156–165.

[23] Ronald J Gutman. 2004. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALC)* 4 (2004), 100–111.

[24] Shuai Huang, Yong Wang, Tianyu Zhao, and Guoliang Li. 2021. A learning-based method for computing shortest path distances on road networks. In *Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE)*. 360–371.

[25] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. 2012. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 445–456.

[26] Sungwon Jung and Sakti Pramanik. 2002. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering* 14, 5 (2002), 1029–1046.
[27] 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.
[28] 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 ACM SIGMOD International Conference on Management of Data*. 709–724.
[29] Ira Sheldon Pohl. 1969. *Bi-Directional and Heuristic Search in Path Problems.* Ph. D. Dissertation. Stanford, CA, USA.
[30] Peter Sanders and Dominik Schultes. 2005. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Conference on Algorithms*. 568–579.
[31] Robert Endre Tarjan. 1983. *Data structures and network algorithms.* SIAM.
[32] Pranali Yawalkar and Sayan Ranu. 2019. Route recommendations on road networks for arbitrary user preference functions. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 602–613.
[33] Mengxuan Zhang. 2021. *Efficient shortest path query processing in dynamic road networks.* Ph. D. Dissertation.
[34] 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 ACM SIGMOD International Conference on Management of Data*. 1992–2005.
[35] Bolong Zheng, Jingyi Wan, Yongyong Gao, Yong Ma, Kai Huang, Xiaofang Zhou, and Christian S. Jensen. 2022. Workload-Aware Shortest Path Distance Querying in Road Networks. In *Proceedings of the 38th IEEE International Conference on Data Engineering ICDE*. 2372–2384.
[36] 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*.

Received April 2023; revised July 2023; accepted August 2023