Customization Meets 2-Hop Labeling: Efficient Routing in Road Networks

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

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

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

Jiawen Wang
Australian National University
Canberra, Australia
jiawen.wang@anu.edu.au

Moritz Laupichler
Karlsruhe Institute of Technology
Karlsruhe, Germany
moritz.laupichler@kit.edu

Peter Sanders
Karlsruhe Institute of Technology
Karlsruhe, Germany
sanders@kit.edu

ABSTRACT

Efficient route planning is crucial for modern navigation systems, yet traditional methods face challenges in scenarios with unknown or frequently changing traffic dynamics. This paper introduces a general labeling framework based on the 2-hop cover property, enabling robust, metric-independent preprocessing. Using this framework, we propose *Customizable Tree Labeling* (CTL), a tree-based method combining three key components: metric-independent preprocessing with tree hierarchies, metric customization for dynamic updates, and efficient query algorithms for fast route computation. To allow trade-offs between customization time, labeling size, and query performance, we further develop a parameterized customization technique by dynamically combining tree labels and shortcut graphs. Our key contributions include the introduction of a customizable labeling framework, a novel tree hierarchy for compact and scalable representation, and a hybrid query algorithm that integrates labels and shortcuts for fast and accurate route computation. We conduct extensive experiments on ten large-scale real-world road networks and a case study on the traffic assignment problem. Our algorithms achieve query response times significantly faster than the state-of-the-art methods, while maintaining competitive customization times and labeling size, making it well-suited for real-time and dynamic routing applications.

PVLDB Reference Format:
Muhammad Farhan, Henning Koehler, Qing Wang, Jiawen Wang, Moritz Laupichler, and Peter Sanders. Customization Meets 2-Hop Labeling: Efficient Routing in Road Networks. PVLDB, 18(10): 3326 - 3338, 2025.
doi:10.14778/3748191.3748198

1 INTRODUCTION

Finding an optimal route between two locations in a road network is an essential building block in modern mobility applications such as navigation, logistics, and traffic management [6, 14]. Given a road network $G = (V, E)$, where V is the set of vertices (e.g., intersections) and E is the set of edges (e.g., road segments), with associated weights $w(e)$ representing costs, such as distance, time or fuel consumption, an *optimal route* between two nodes s (source) and t (target) is a path that minimizes the total cost:

$$p^* = \arg \min_{p \in P(s,t)} \sum_{e \in p} w(e),$$

where $P(s, t)$ is the set of all paths from s to t in G and $\sum_{e \in p} w(e)$ is the total cost of the path p. The notion of "*optimal route*" depends on the metric defining the weights $w(e)$, such as shortest distance, least travel time, or minimum fuel consumption, and the optimal route p^* is the one that achieves the minimum total cost. The choice of metric varies widely depending on individual user needs and contextual factors. For instance, while some users prioritize the fastest route to minimize travel time, others may favor an optimized path that reduces fuel consumption, toll expenses, or environmental impact.

Example 1.1. *Consider the* traffic assignment *(TA) problem, which models how traffic flows are distributed across a transportation network to determine optimal routes that minimize travel time while accounting for congestion. Each edge $e \in E$ has a travel cost determined by current traffic load. Travelers' demands are represented as a set of origin-destination (O-D) pairs $\{(s_i, t_i)\}$, where s_i and t_i denote the origin and destination of a demand, respectively. The goal is to compute optimal routes for all O-D pairs, accounting for congestion caused by all travelers. Solving the TA problem is central to applications such as transportation planning, policy-making, and real-time traffic management, as optimal routes help reduce congestion, delays, and fuel consumption, improving urban transportation [19, 34, 39, 42, 46]. A case study on TA will be discussed in Section 10.*

Related Work. Traditionally, route planning relies on a single static cost metric, modeling the road network as a weighted graph where edge weights represent the cost for traveling along a road segment. Considering distance as a cost metric, optimal routes are then defined as the shortest paths between vertices. While Dijkstra's algorithm [40] can compute these paths, it becomes inefficient for large-scale road networks, requiring more than one second per query [43]. Such latency is unsuitable for applications demanding responses within microseconds or nanoseconds. To address this inefficiency, a wide range of techniques have been developed to accelerate route planning in road networks. These include *goal-directed search* methods such as A* and ALT [24, 26], *hierarchical techniques* such as Highway Hierarchies (HH) [36] and Contraction Hierarchies (CH) [21], and *labeling-based techniques* that precompute vertex labels

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:10.14778/3748191.3748198

encoding shortest path information [1, 2, 4, 10, 17, 27, 32]. See [5] for a comprehensive survey. Of particular relevance to this work are Contraction Hierarchies (CH), which reduce the search space by contracting vertices in a specific order and introducing shortcuts, and labeling-based methods, which allow shortest paths to be efficiently retrieved directly from precomputed labels. Despite their success, traditional route planning methods fall short when edge weights are unknown during preprocessing or change frequently due to dynamic user preferences or traffic updates. These limitations stem from *metric-dependent preprocessing*, which assumes fixed edge weights in precomputed structures.

To address the limitations of traditional route planning methods, customizable speed-up techniques were introduced [8, 12, 16]. These techniques divide preprocessing into two stages: a slow, metric-independent phase that constructs an auxiliary structure based on the network topology, and a fast, metric-dependent phase that customizes this structure for specific metrics. This approach enables fast customization to metric changes, producing optimized data for real-time query processing. Two prominent customization methods are Customizable Route Planning (CRP) [12] and Customizable Contraction Hierarchies (CCH) [16]. CRP precomputes multilevel partition-based overlay graphs using separator-based techniques, while CCH employs a nested dissection order [23] to construct and customize CH edges for query answering. Both methods use metric-independent auxiliary structure, avoiding the need for full re-computation and making them suitable for scenarios with frequently changing edge weights. However, their query performance remains suboptimal despite fast customization times. A recent approach, Customizable Hub Labeling (CHL) [8], extends the customization paradigm to Hub Labeling (HL) [1, 2]. In particular, it studies Hierarchical Customizable Hub Labelings (HCuHL), which materialize CCH search spaces as labels. Conceptually this is similar to the non-parameterized ($\theta = 0$) version of our approach, though the paper is theoretical in nature without concrete algorithms, making experimental comparison impossible.

Our Contributions. In this work, we aim to develop an efficient labeling method to accelerate query responses for route planning. Our key observations are: (a) *Metric-independent phase design:* Labeling-based methods require a metric-independent phase to handle scenarios where metrics are unknown during preprocessing. The classical 2-hop cover property [11], however, assumes a single known metric (e.g., distance) and relies on precomputed shortest paths, making it inapplicable for direct use in the metric-independent phase. (b) *Customization vs. query performance*: Customization and query often have competing performance needs. For instance, structures like CCH excel at updating costs (customization) but are less efficient for query responses. Conversely, labeling-based structures may require more customization time but support faster queries. This highlights the need for distinct structures specifically designed for customization and query phases, respectively. (c) *Trade-off control:* Balancing customization and query performance is critical. A controlled trade-off ensures efficient query response times while tailoring customization levels to specific application needs. This balance is essential for real-time applications, where speed depends on achieving an optimal trade-off.

The contributions and structure of the paper are as follows:

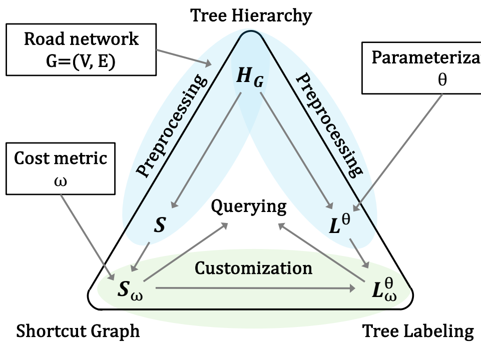

Figure 1: Framework overview: The preprocessing processes the road network G to H_G, S, and L^θ (blue). The customization takes S and L^θ, along with a cost metric ω, as input to produce S_ω and L^θ_ω (green). Queries are then executed on the customized structures S_ω and L^θ_ω.

(1) A labeling framework (Section 4). We present a labeling framework parameterised by θ, which governs the extent of labeling, as depicted in Figure 1. For extreme parameter settings our approach can either be classified in the framework of [8] as a hierarchical customizable hub labeling (for $\theta = 0$), or becomes essentially a customizable contraction hierarchy (for $\theta = \infty$).

(2) Customizable tree labeling (Section 5). Building on this framework, we propose *customizable tree labeling*, which uses a compact and efficient tree hierarchy as the foundation for a tree labeling scheme and a shortcut graph scheme in the metric-independent phase. Shortcut graph customization is then used to efficiently customize the tree labeling, enabling highly efficient query responses.

(3) Parameteric customization (Section 6). To balance customization and query performance, we design a parametric customization method. This approach offers a flexible mechanism to adjust the level of customization in tree labeling, enabling fine-grained control over the trade-off between customization and query efficiency.

(4) Complexity analysis (Section 7). We compare complexity bounds of existing approaches and our method, discussing efficiencies in preprocessing, customization, querying, and storage.

(5) Key variants (Section 8). We introduce several key variants: parallel customization for scalable hub labeling, path queries for efficient route reconstruction, and support for directed road networks with optimized memory usage, enhancing performance and adaptability for diverse routing challenges.

(6) Experimental study (Section 9). We conduct extensive experiments on 10 large real-world road networks, including the entire USA and Western Europe road networks. Results show our method significantly outperforms the state-of-the-art CCH in preprocessing time and is up to 25–35 times faster in finding optimal routes on the two largest networks.

(7) Case study on traffic assignment (Section 10). We study the traffic assignment problem on Stuttgart's road network in Germany. Our method achieves 3–9 times faster query times and overall runtime compared to state-of-the-art methods, demonstrating its efficiency for traffic assignment in large urban networks.

Remark 1.1. Customizable routing differs from scenarios where only a small fraction of edge weights change, such as traffic variations or road closures. In such cases, incremental maintenance [18, 22, 29, 33, 44, 45], also referred to as *partial customizable routing*, is more appropriate. Customizable routing, by contrast, performs full metric customization, offering broader adaptability. Experiments and a detailed discussion can be found in Section 9.3.

2 PROBLEM FORMULATION

Let $G = (V, E)$ represent a road network. A *(cost) metric* is a function $\omega : E \to \mathbb{R}_{>0}$ that assigns a positive cost $\omega(u, v)$ to each edge $(u, v) \in E$, such as travel time. This metric may be initially unknown or dynamically determined based on the road network's properties or external factors. A path is a sequence of distinct vertices $p = (v_1, v_2, \ldots, v_k)$, where $(v_i, v_{i+1}) \in E$ for all $1 \leq i \leq k-1$. The cost of a path p, given a metric ω, is $\omega(p) = \sum_{i=1}^{k-1} \omega(v_i, v_{i+1})$. As previously defined, an *optimal route* between two vertices s and t minimizes the total path cost with respect to a metric ω. We denote the set of all optimal routes between s and t in G under the metric ω by $P^{\omega}_{G}(s, t)$, and the *optimal cost* by $d^{\omega}_{G}(s, t)$.

We formally define the customizable routing problem.

Definition 2.1 (Customizable Routing). *In a road network $G = (V, E)$, for any two vertices $s, t \in V$, the* customziable routing problem *is to efficiently find an optimal route $p \in P^{\omega}_{G}(s, t)$ from s to t with respect to any given cost metric ω.*

To compute an optimal route from one vertex to another, finding the optimal cost is crucial as it restricts the search space and ensures correctness during path restriction. Customizable routing also applies to both directed and undirected graphs. For simplicity, we first focus on computing optimal cost queries in undirected graphs, deferring extensions to directed graphs and path queries to Section 8.

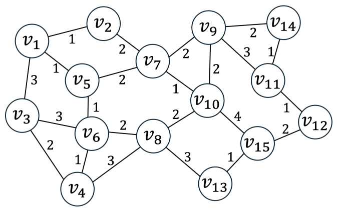

Figure 2: A road network G, customized with a cost metric ω.

Example 2.1. *Fig. 2 shows a road network $G = (V, E)$ with 15 vertices and 23 edges. Each $e \in E$ is assigned a cost by a metric ω. Consider two vertices, v_1 and v_{12}, connected by multiple paths. For instance, $p_1 = \langle v_1, v_2, v_7, v_9, v_{11}, v_{12} \rangle$ with cost $\omega(p_1) = 9$, and $p_2 = \langle v_1, v_5, v_7, v_{10}, v_{15}, v_{12} \rangle$ with cost $\omega(p_2) = 10$. Among these, p_1 is optimal as it has the minimum cost.*

Customizable approaches are ineffective when each query uses a unique metric, as the cost of customization often outweighs that of an index-free search. However, customization works when metrics are frequently reused, such as when the number of distinct metrics is small and multiple customizations can be stored in memory, or when updates reflect periodic changes in costs like travel time.

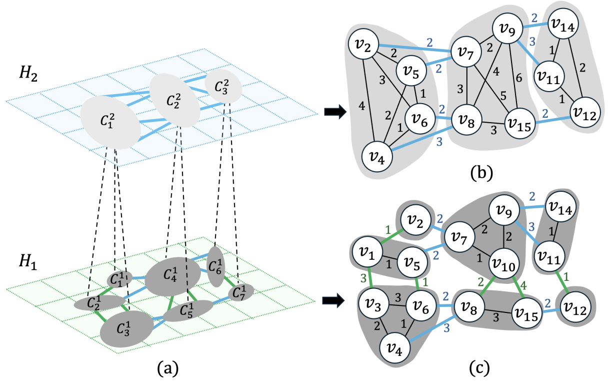

Figure 3: (a) A multilevel overlay graph H with two levels $\{H_1, H_2\}$ for the road network G in Figure 2, (b) the customized overlay graph H_2, and (c) the customized overlay graph H_1.

3 EXISTING SOLUTIONS

In this section, we review existing solutions, Customizable Route Planning (CRP) and Customizable Contraction Hierarchies (CCH).

Customizable Route Planning. Customizable Route Planning (CRP), introduced by Delling et al. [12], is the first solution to the customizable routing problem. It separates the road network's topology from its metric properties through a three-stage process: metric-independent preprocessing, metric-dependent customization, and query execution.

In preprocessing, CRP builds a multi-level overlay graph H by recursively partitioning the road network G. At each level l, cells C^{l}_{i} are nested within supercells C^{l+1}_{j} at the next level $l + 1$. The overlay graph retains only boundary edges, which connect vertices in different cells, and their endpoints, called boundary vertices. Shortcuts are added between boundary vertices within each cell to capture optimal paths. In customization, the overlay graph H is adapted to a specific metric ω by computing shortcuts using Dijkstra's algorithm. At the lowest level, shortcuts are computed within each cell on the original graph. At higher levels, shortcuts in a cell C^{l}_{i} are computed using shortcuts from level $l - 1$. In query execution, shortcuts allow the search to avoid irrelevant areas while maintaining path accuracy.

Example 3.1. *Fig. 3 shows the overlay graph H for the road network G in Fig. 2, consisting of three level-2 cells (top) and seven level-1 cells (bottom). Blue and green edges indicate level-2 and level-1 boundary edges, respectively. Each cell contains only boundary vertices, forming a clique. To answer a query from v_1 to v_{12}, a bidirectional Dijkstra search is performed at level $l(v)$ for each visited vertex v. The search visits v_5, v_{11}, and v_7, where $l(v_5) = 0$, $l(v_{11}) = 1$, and $l(v_7) = 2$. v_5 is searched on the original graph G, v_{11} on H_1, and v_7 on H_2.The optimal path is $p = (v_1, v_2, v_7, v_9, v_{11}, v_{12})$ with $\omega(p) = 9$.*

Customizable Contraction Hierarchy. Dibbelt et al. [16] introduced Customizable Contraction Hierarchies (CCH), extending Contraction Hierarchies (CH) [21] to support customizable cost metrics. CH efficiently answers distance queries in road networks by constructing an augmented graph G^{+} from a given total vertex order σ. Shortcuts are added between vertices v_i and v_k if there exists a shortest valley path between them in G. A valley path $p = (v_1, v_2, \ldots, v_k)$

satisfies $\sigma(v_i) < \min\{\sigma(v_1), \sigma(v_k)\}$ for all intermediate vertices. The cost of a shortcut is the cost of its shortest valley path.

CCH extends CH by decoupling metric-independent preprocessing from metric-dependent customization. It uses a nested dissection order to create a hierarchy of vertex separators that minimize partition size. During preprocessing, an augmented graph G^+ is constructed by adding shortcuts (v_i, v_k) between vertices v_i and v_k if and only if there exists a valley path between them. In customization, actual edge costs are assigned by propagating them through shortcuts in G^+. For a query between s and t, a bidirectional search in G^+ ascends from both vertices to their least common ancestor, the highest-ranked vertex reachable from both, leveraging shortcuts to bypass intermediate vertices of lower rank.

Example 3.2. *Consider the road network in Fig. 2 with nested dissection order* $\sigma = (v_5 > v_6 > v_7 > v_8 > v_{10} > v_{15} > v_{11} > v_{13} > v_2 > v_1 > v_3 > v_4 > v_9 > v_{12} > v_{14})$. *Processing starts with the highest ranked vertex* v_5. *All neighbors have lower ranks, enabling valid valley paths through* v_5. *For example, shortcut* (v_6, v_7) *is added via path* $\langle v_6, v_5, v_7 \rangle$. *During customization, the cost of this shortcut is computed as the sum of edge weights on the valley path, i.e.,* $\omega(v_6, v_7) = \omega(v_6, v_5) + \omega(v_5, v_7) = 1 + 2 = 3$. *For a query from* v_6 *to* v_{11}, *CCH performs a bidirectional search on the augmented graph. The forward search follows* $v_6 \rightarrow v_7 \rightarrow v_9$ *using the shortcut, meeting the backward search* $v_{11} \rightarrow v_9$ *at* v_9. *The total path cost is* $3 + 2 + 3 = 8$.

Discussion. Both CRP and CCH produce compact auxiliary structures for efficient metric customization, making them suitable for scenarios with frequent cost updates. However, their query times remain high due to reliance on bidirectional Dijkstra variants. Despite using shortcuts, queries can be slow in shortcut-dense graphs or when spanning large portion of the network. Query efficiency also depends on preprocessing quality, such as vertex order in CCH or partitioning in CRP. Suboptimal preprocessing can lead to excessive shortcuts or poorly pruned search spaces, further slowing performance. Consequently, while these methods achieve fast customization, their slower queries make them unsuitable for latency-sensitive applications like real-time navigation.

To improve query performance, labeling is a natural candidate. However, a recent study by Blum and Storandt [8] showed that supporting edge cost updates under arbitrary metrics requires hierarchical customizable hub labeling (CHL) whose minimum average and maximum label sizes correspond to the minimum average and maximum search space sizes in CCH. While acceptable for single metrics, storing labeling for multiple metrics reduces memory available per metric. In the following (Sections 4 to 6), we will present an efficient labeling-based solution for customizable routing. Critically our solution can be parameterized to provide a suitable tradeoff between query speed and label size for meeting application needs, combining the benefits (but also some drawbacks) of both CCH and CHL.

4 FRAMEWORK OVERVIEW

This section introduces a customizable labeling framework designed to enhance query performance. The main challenge lies in designing a preprocessing strategy that generalizes across diverse metrics while achieving efficiency in both customization and query phases. To address this, we introduce a parameterization process for labeling that balances customization and query efficiency.

We begin by refining the concept of 2-hop labeling [11] for a specific metric ω. Given a metric ω, a *ω-labeling* of a road network $G = (V, E)$ assigns each vertex $v \in V$ a label $L(v)$ containing cost entries $\{\delta_{vu_1}, \dots, \delta_{vu_k}\}$, where $u_i \in V$ and $\delta_{vu_i} = d_G^{\omega}(v, u_i)$ for $i = 1, \dots, k$.

Definition 4.1 (2-Hop Labeling [11]). *A ω-labeling L over $G = (V, E)$ is a* 2-hop labeling *if, for any $s, t \in V$, the optimal cost $d_G^{\omega}(s, t)$ can be computed using their labels $L(s)$ and $L(t)$. Formally:*

$$d_G^{\omega}(s, t) = \min_{v \in L(s) \cap L(t)} \{\delta_{sv} + \delta_{tv}\},$$

where $L(s) \cap L(t)$ is the set of common vertices in $L(s)$ and $L(t)$.

Parameterization. While 2-hop labelings are highly efficient for query answering, they tend to suffer from large label sizes, compared to shortcut based approaches like CCH or CRP. We therefore will employ only a *partial* 2-hop cover which only maintains part of the label structure based on a parameter θ. At one extreme ($\theta = 0$) we obtain a full 2-hop cover (specifically a hierarchical customizable hub labeling in the terminology of [8]), at the other ($\theta = \infty$) we obtain a customizable contraction hierarchy, while any parameter setting in between provides a tradeoff between these.

Achieving favorable trade-offs can be challenging, particularly when integrating techniques with differing optimization objectives. For instance, labeling techniques might achieve a query time of $1\mu s$ with a label size of 100 GB, while shortcut-based methods might provide a 1 ms query time with a label size of 100 MB. A combined approach averaging 500 μs query time and 50 GB label size would often be deemed both slow and large, representing the worst of both worlds.

Customizable Labeling Framework. Let $\mathcal{G}$ represent the set of all road networks, W the set of possible cost metrics, $\mathcal{D}$ the set of all data structures independent of any metric, and $\mathcal{D}_{\omega}$ the set of all data structures dependent of a metric ω. Our framework consists of three key components:

- Preprocessing algorithm: A function $\mathcal{G} \times \mathbb{R} \rightarrow \mathcal{D}$ that maps a road network G and parameter θ to a metric-independent data structure (L^{θ}, S). Here, L^{θ} is a parameterized labeling and satisfies the customizable cover property [8] when $\theta = 0$, and S is a data structure to accelerate customization.
- Customization algorithm: A function $\mathcal{D} \times W \rightarrow \mathcal{D}_{\omega}$ that customizes a preprocessed data structure (L^{θ}, S) with respect to a metric $\omega \in W$. The result is a customized data structure $(L_{\omega}^{\theta}, S_{\omega})$. By the customizable cover property, L_{ω}^{θ} is guaranteed to be a 2-hop ω-labeling when $\theta = 0$.
- Query algorithm: A function $\mathcal{D}_{\omega} \times V \times V \rightarrow \mathbb{R}$ that takes a customized data structure $(L_{\omega}^{\theta}, S_{\omega})$ and two vertices $s, t \in V$, returning either the optimal cost or an optimal route from s to t with respect to the metric ω.

While conceptually straightforward, designing such a metric-independent data structure while balancing customization efficiency and query performance across varying metrics remains a challenging task. In particular, metric-independent data structure must preserve topological structure of road networks without relying on fixed metrics; customization must efficiently handle large-scale updates; and queries must remain fast despite varying label size. In the following we first present a basic non-parameterized approach which offers fast query times but suffers from large labeling size and customization times (Section 5). This then forms the basis for our parameterized approach, which balances query time and labeling size more favorably (Section 6).

5 TREE-BASED CUSTOMIZABLE ROUTING

This section presents a tree-based solution for customizable routing, enabling efficient preprocessing, customization, and querying. A key challenge in designing tree-based customizable routing solution is to support efficient topology-based preprocessing that yields compact and query-friendly data structures. Traditional labeling-based methods suffer from large label sizes [1, 2, 8], while shortcut-based methods [12, 16] favor customization speed at the cost of slower queries. Our initial tree-based approach presented here falls into the former category, but also maintains an intermediate shortcut based structure which is used during customization.

5.1 Metric-Independent Preprocessing

Tree Hierarchy. To efficiently support customizable routing, we introduce a *tree hierarchy*, which provides a topology-based representation of road networks.

Definition 5.1 (Tree Hierarchy)**.** *Let $\beta \in (0, 0.5)$. A* tree hierarchy *over a road network $G = (V, E)$ is a binary tree $H_G = (\mathcal{N}, \mathcal{E}, f)$, where $\mathcal{N}$ is the set of tree nodes, $\mathcal{E}$ is the set of tree edges, and $f : V \to \mathcal{N}$ is a total surjective function satisfying two conditions:*

(1) Balanced subtrees: *For every internal node $N \in \mathcal{N}$, the left and right subtrees satisfy:*

$$|T_\ell(N)|, |T_r(N)| \leq (1-\beta) \cdot |T_\ell(N) \cup T_r(N)|,$$

where $T_\ell(N)$ and $T_r(N)$ are the sets of vertices in the left and right subtrees of N, respectively.

(2) Ancestor separation: *For any two vertices $s, t \in V$, the set*

$$CA(s,t) = \{v \in V \mid f(v) \in A(f(s)) \cap A(f(t))\}$$

of their common tree ancestors contains at least one vertex on each path between s and t in G, where $\mathsf{A}(\cdot)$ *denotes the set of ancestor nodes of a tree node.*

Fig. 4 shows a tree hierarchy for the road network in Fig. 2, with the root containing vertices $\{v_7, v_8\}$. The balanced subtrees condition keeps the tree compact, ensuring efficiency in label construction and query processing. The ancestor separation condition enables the use of common ancestors as hubs for 2-hop labeling.

Constructing such a tree hierarchy requires effective graph bipartitioning to minimize separator sizes while maintaining balance. Although this problem is NP-hard, recent heuristics have demonstrated excellent scalability and performance on large-scale road networks [13, 17, 38]. In this work, we adopt the recursive bipartitioning method from [17], which iteratively identifies balanced separators to partition the graph.

A hierarchical vertex order $\preceq$ can be derived from the tree hierarchy H_G. Specifically, $\preceq$ is a partial order on the vertices of the road network G: for any vertices $u, v \in V$ with $f(u) \neq f(v)$ we have $v \preceq u$ iff $f(v) \in A(f(u))$. Vertices mapped to the same tree node are totally ordered by $\preceq$, though this order is arbitrary.

We use $anc(v) = \{w \in V \mid w \preceq v\}$ to denote the ancestor vertices of v. The *rank* of a vertex is defined as $\tau(v) = |anc(v)|$. Although different vertices can have the same rank, the ranks of ancestor within $anc(v)$ are distinct, running from 1 to $\tau(v)$, and will be used as indices within labels. The rank of a tree node is the maximum rank of vertices mapped to it.

Example 5.1. *Consider the tree hierarchy shown in Fig. 4. We have $anc(v_3) = \{v_7, v_8, v_3\}$ and $anc(v_5) = \{v_7, v_8, v_3, v_5\}$. Their ranks are $\tau(v_3) = 3$, $\tau(v_5) = 4$ and the rank of tree node $f(v_3)$ is $max(3, 4)$.*

Tree Labeling Scheme. Based on the tree hierarchy H_G, we define the following tree labeling scheme.

Definition 5.2 (Tree Labeling Scheme)**.** *Let H_G be a tree hierarchy over a road network $G = (V, E)$. A* tree labeling scheme *is a tuple $L = (H_G, \mathcal{I}, \mathcal{T}, C)$, where:*

- *$\mathcal{I} = \{\mathcal{I}(v) \mid v \in V\}$: Each vertex $v \in V$ is assigned a bitstring identifying the position of node $f(v)$ in the tree hierarchy.*
- *$\mathcal{T} = \{\mathcal{T}(v) \mid v \in V\}$: Each $\mathcal{T}(v) = [\tau(N_1), \ldots, \tau(N_k), \tau(v)]$ is a* rank array *where $\{N_1, \ldots, N_k, f(v)\} = A(f(v))$ are the ancestor tree nodes of $f(v)$.*
- *$C = \{C(v) \mid v \in V\}$: Each $C(v)$ is a* cost array *$[\delta_{vw_1}, \ldots, \delta_{vw_k}]$, where $anc(v) = \{w_1, \ldots, w_k\}$ and δ_{vw_i} indicates a cost value (not necessarily minimal) for paths between v and w_i.*

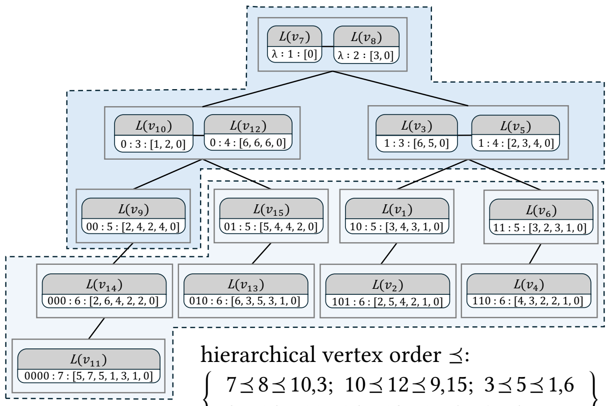

Figure 4: An illustration of a tree hierarchy H_G and tree labeling Scheme L over G, where vertices at the first level are assigned the empty bitstring λ. We only show $\tau(v)$ in place of $\mathcal{T}(v)$ for brevity.

Example 5.2. *Fig. 4 shows a tree hierarchy and tree labeling scheme over an example road network G depicted in Fig. 2. The label information for each vertex v is presented under its label $L(v)$ in the format $\mathcal{I}(v) : \tau(v) : C(v)$. Consider vertex v_{15}: it is assigned a node identifier $\mathcal{I}(v_{15}) = 01$ as its node is the right child of the left child of the root. Its ancestors are $anc(v_{15}) = \{v_7, v_8,\ v_{10}, v_{12},\ v_{15}\}$, resulting in a rank array $\mathcal{T}(v_{15}) = [2, 4, 5]$ and a cost array* $[5, 4, 4, 2, 0]$ *storing distances to its ancestors $\{v_7, v_8, v_{10}, v_{12}, v_{15}\}$.*

The tree labeling scheme leverages the structure of the tree hierarchy H_G to assign labels that encode both topological and cost-related information for efficient query processing. The node identifiers ($\mathcal{I}$) together with the rank arrays ($\mathcal{T}$) enable efficient computation of how many common ancestors two vertices have, and thus which prefix of the cost arrays (C) to use as hub for 2-hop cost computation. In fact, $\mathcal{I}$ and values in $\mathcal{T}$ other than $\tau(v)$ are only used by `GetLcaHeight` in Algorithm 3.

Shortcut Graph Scheme. Fast customization requires efficient path searches. To support this, we utilize a data structure called *shortcut graph scheme*. With the hierarchical vertex order $\preceq$ we extend the road network by adding *shortcuts* between vertices where intermediate vertices have higher ranks than the endpoints.

Definition 5.3 (Shortcut Graph Scheme). *Given a road network $G = (V, E)$ and a tree hierarchy H_G, the* shortcut graph scheme *$S = (V, E^*, \preceq)$ consists of the vertex set V, the edge set $E^* = E \cup E'$ that includes all edges E in G and shortcuts E', and a hierarchical vertex order $\preceq$ induced by H_G. A shortcut $(v, u) \in E'$ is added if $(v, u) \notin E$ and there exists a* valley path *p between v and u, meaning that $v, u \preceq w$ for all $w \in V(p) \setminus \{v, u\}$.*

For each vertex $v \in V$, we denote its *upward neighbors* which ranked *lower* than v as $N^+(v) = \{u \mid (v, u) \in E^* \wedge u \preceq v\}$, and *downward neighbors* which are ranked *higher* than v as $N^-(v) = \{u \mid (v, u) \in E^* \wedge v \preceq u\}$ in the shortcut graph scheme S.

We construct S following the method proposed in [33], with two key modifications: (1) *Hierarchical vertex contraction:* Vertices are contracted based on the hierarchical vertex order $\preceq$, starting with the highest-ranked vertices (in decreasing order of rank). This replaces the use of a pre-defined total vertex order. (2) *Shortcut addition:* Shortcuts in S are added without specifying edge costs.

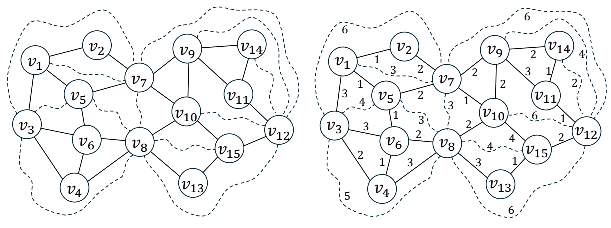

Figure 5: (a) An illustration of shortcut graph scheme; (b) Shortcut graph scheme after customization.

Example 5.3. *Fig. 5(a) shows a shortcut graph scheme S over an example road network G shown in Fig. 2 using the hierarchical vertex order induced by the tree hierarchy H_G depicted in Fig. 4. Dashed edges represent shortcuts. A shortcut edge (v_1, v_7) in S is added because a valley path $\langle v_1, v_2, v_7\rangle$ between v_1 and v_7 exist in G. Note that $\langle v_1, v_5, v_7\rangle$ is not a valley path since $v_5 \preceq v_1$. Consider vertex v_1, we have $N^+(v_1) = \{v_3, v_5, v_7\}$ and $N^-(v_1) = \{v_2\}$.*

5.2 Metric Customization

We describe the customization, which integrates a given cost metric ω into tree labeling scheme L and shortcut graph scheme S through two sequential steps: (1). Customize S to S_ω by incorporating the edge costs of the road network into all shortcuts. (2). Customize L to L_ω by computing the cost array for each $C(v)$ based on S_ω.

Customizing Shortcut Graph Scheme. The *shortcut customization property* is utilized to customize the costs of shortcuts in S.

Algorithm 1: Customizing Shortcut Graph Scheme

```
1 Function CustomizeS($G$, $S$)
2     $S_\omega \leftarrow S$
      // initialize new edge costs for $S_\omega$
3     foreach $(u, v) \in E$ do
4         $\omega(u, v) \leftarrow \omega_G(u, v)$
      // customize shortcut costs
5     foreach $v \in V$ in descending order of $\preceq$ do
6         foreach $u \in N^+(v)$ do
7             foreach $w \in N^-(v) \cap N^-(u)$ do
8                 $\omega(v, u) \leftarrow \min\{\omega(v, u), \omega(w, v) + \omega(w, u)\}$
```

Algorithm 2: Customizing Tree Labeling Scheme

```
1 Function CustomizeL($S_\omega$)
      // initialize
2     foreach $v \in V$ do
3         $C(v) \leftarrow [\infty, \dots, \infty]$, $\delta_{vw} \leftarrow 0$, s.t., $w = \tau(v)$
      // customize label distances
4     foreach $v \in V$ in ascending order of $\preceq$ do
5         foreach $u \in N^+(v)$ do
6             foreach $w$ s.t. $w \preceq u$ do
7                 $\delta_{vw} \leftarrow \min(\delta_{vw},\ \omega(v, u) + \delta_{uw})$
```

Property 5.1 (Shortcut Customization). *Let $S = (V, E^*, \preceq)$. For any $(v, u) \in E^* \setminus E$, the cost $\omega(v, u)$ in $S_\omega = (V, E^*, \omega, \preceq)$ satisfies:*

$$\omega(v, u) = \min\{\omega(w, v) + \omega(w, u) \mid w \in N^-(v) \cap N^-(u)\}.$$

Specifically, the shortcuts in S are customized in descending order of v with respect to $\preceq$, as described in Algorithm 1. For each upward neighbor $u \in N^+(v)$, the algorithm iterates over the common downward neighbors w of v and u. It checks if the path between v and u through w in S has a lower cost than the direct path between v and u. If so, it updates $\omega(v, u)$ to reflect the lower cost between v and u in G (Lines 5–7), based on Property 5.1. By construction, w has a strictly higher rank than v and u, ensuring that the costs $\omega(v, w)$ and $\omega(w, u)$ are already computed before $\omega(v, u)$ is updated.

Example 5.4. *Fig. 5(b) shows shortcut graph scheme S after metric customization, with weights assigned based on the metric in Fig. 2. Consider the following shortcuts in the order processed by Algorithm 1: the cost of (v_{12}, v_{14}) is computed as $\omega(v_{12}, v_{14}) = \omega(v_{12}, v_{11}) + \omega(v_{11}, v_{14}) = 2$ using the triangle $\langle v_{12}, v_{11}, v_{14}\rangle$. This is then used to compute $\omega(v_{12}, v_9) = \omega(v_{12}, v_{14}) + \omega(v_{14}, v_9) = 4$ and finally $\omega(v_{12}, v_7) = \omega(v_{12}, v_9) + \omega(v_9, v_7) = 6$.*

Customizing Tree Labeling Scheme. To customize the cost array $[\delta_{vw_1}, \dots, \delta_{vw_k}]$ for each $C(v)$, we apply the following *label customization property*. This property ensures efficient computation of costs using the hierarchical structure of S_ω.

Property 5.2 (Label Customization). *Let $S_\omega = (V, E^*, \omega, \preceq)$. For any $v, w \in V$ with $w \preceq v$, each cost entry in $C(v)$ satisfies:*

$$\delta_{vw_i} = \min_{u \in N^+(v)} \{\omega(v,u) + \delta_{uw_i}\}, \quad \text{for } 1 \le i \le k.$$

We customize the cost array $C(v)$ for each vertex $v \in V$ in ascending order of $\preceq$ using a top-down approach, as detailed in Algorithm 2. Each cost array $C(v)$ is initialized to ∞, i.e., $\delta_{vw} = \infty$ for all $w \neq v$ and to self as 0 (Lines 2-3). For each $u \in N^+(v)$, we iterate over all $\{w \mid w \preceq u\}$. If a path with lower cost is found, i.e., $\delta_{vw} > \omega(v,u) + \delta_{uw}$, we update δ_{vw} with $\omega(v,u) + \delta_{uw}$. This operation leverages Property 5.2. A key observation is that when processing v, the labels for all vertices $\{w \mid w \preceq v\}$ have already been processed and thus can be used together with its upward neighbors to compute $C(v)$ without recomputating any $C(w)$.

Example 5.5. *Consider Fig. 4 and Fig. 5(b). Suppose the costs for all vertices with ranks lower than 5, namely $\{v_7, v_8, v_3, v_5\}$, have already been customized. We now customize the cost array $C(v_1)$ for vertex v_1. From Fig. 5(b), we know $N^+(v_1) = \{v_7, v_5, v_3\}$. In Algorithm 2, we first consider $v_7 \in N^+(v_1)$ and iterate over $\{v_7\} \preceq v_7$ (Line 6). At Line 7, since $\omega(v_1, v_7) = 3$ and $\delta_{uw} = 0$ with $u = w = v_7$, we compute $\delta_{vw} = 3$, where $v = v_1$ and $w = v_7$. Next, for $v_5 \in N^+(v_1)$, we iterate over $\{v_7, v_8, v_3, v_5\} \preceq v_5$ (Line 6), applying Line 7 to update costs from v_1 to these vertices. Similarly, for $v_3 \in N^+(v_1)$, we iterate over $\{v_7, v_8, v_3\} \preceq v_3$ (Line 6) and again apply Line 7 to update the corresponding costs.*

5.3 Query Processing

We discuss how to efficiently answer route queries using the customized tree labeling L_ω. This process leverages tree labels to estimate costs and restrict the search to relevant paths, thereby significantly enhancing query efficiency. The hierarchical structure of tree labels is central to this optimization.

Let $s, t \in V$ be any two vertices and $CA(s,t)$ their common ancestors as per Definition 5.1. Due to the *ancestor separation* property, the optimal cost between s and t can be computed as:

$$d^\omega_G(s,t) = \min\{d^\omega_G(s,r) + d^\omega_G(t,r) \mid r \in CA(s,t)\}.$$

Note however that our cost arrays do not store distances in G. Instead, it can be shown that $\delta_{vw} = C(v)[\tau(w)]$ is computed as the distance between v and w in the subgraph induced by the vertices in the sub-tree rooted in w. Using the same arguments as in [35] which takes a similar approach, one can show that

$$d^\omega_G(s,t) = \min\{\delta_{sr} + \delta_{tr} \mid r \in anc(s) \cap anc(t)\}.$$

Observe that the cost values used in this are found at the first h positions of $C(s), C(t)$, where $h = |anc(s) \cap anc(t)|$. In Algorithm 3 we compute h as the rank of the lowest common ancestor l_{st} of s, t. First we proceed as in [17] and obtain the level of its tree node $f(l_{st})$ as the length of the common prefix of the identifiers $\mathcal{I}(s), \mathcal{I}(t)$, then look up its rank in either of the rank arrays $\mathcal{T}(s), \mathcal{T}(t)$.

6 PARAMETRIC CUSTOMIZATION

Balancing customization and query performance is crucial for routing systems to support a wide range of applications, from real-time navigation to logistics planning. The tree labeling method introduced in Section 5 enables efficient queries, but computing full labels for every vertex can be unnecessarily expensive. We observe that vertices higher in the hierarchy tend to be more central and are involved in queries more frequently, making full labeling beneficial. In contrast, lower-level vertices contribute less and require only minimal labeling. To leverage this distinction, we introduce a parameterized labeling approach that adjusts the level of labeling based on the hierarchical importance of each vertex and enables controllable trade-offs between customization and query efficiency.

The labels in L_ω serve as pre-computed optimal routes, avoiding the need for upward searches in S_ω. The core idea is to combine *partial* tree labels in the customized tree labeling L_ω and shortcuts in the customized shortcut graph S_ω to answer route queries. A higher degree of labeling in L_ω improves query performance but increases customization time and label size. Conversely, relying more on searching within S_ω enables faster customization and smaller label sizes but slows query responses. This interplay between L_ω and S_ω adds design complexity, requiring careful consideration to ensure accurate routing with minimal computational overhead.

Algorithm 3: Query optimal routes

```
Function FindRoute($s, t$)
    $h \leftarrow$ GetLcaHeight($s, t$)
    $C(s) \leftarrow$ GetCost($s, h$); $C(t) \leftarrow$ GetCost($t, h$)
    return $\min_{i=1}^{h} C(s)[i] + C(t)[i]$
```

Algorithm 4: Cost computation

```
1  Function GetCost($v, h$)
2      if $C(v)$ is not truncated then
3          return $C(v)$
       // initialize cost and ancestor arrays of length $\tau(v)$
4      $c \leftarrow [\infty, \dots, \infty]$; $a \leftarrow [\perp, \dots, \perp]$
       // follow $S_\omega$ shortcuts upwards while labels are truncated
5      $c[\tau(v)] \leftarrow 0$; $a[\tau(v)] \leftarrow v$
6      for $\tau(w) = \tau(v)$ down to 1 do
7          $w \leftarrow a[\tau(w)]$
8          if $w = \perp$ then
9              continue
10         if $C(w)$ is not truncated then
               // update c using $C(w)$
11             for $i \in [1, \min(\tau(w) - 1, h)]$ do
12                 $d \leftarrow c[\tau(w)] + C(w)[i]$
13                 if $d \le c[i]$ then
14                     $c[i] \leftarrow d$; $a[i] \leftarrow \perp$
15         else
               // follow $S_\omega$ shortcuts
16             foreach $(n, \omega) \in N^+(w)$ do
17                 $d \leftarrow c[\tau(w)] + \omega$
18                 if $d < c[\tau(n)]$ then
19                     $c[\tau(n)] \leftarrow d$; $a[\tau(n)] \leftarrow n$
20     return $C(v)$
```

A key challenge here is how exactly to utilize preserved labels when answering queries for lower-level vertices whose labels have been truncated. Although it is possible to simply use our shortcut graph scheme as a fall-back whenever labels are unavailable, such an approach would result in an unacceptable degradation of query performance – close to that of plain contraction hierarchies, while incurring additional space and customization overheads.

Parameterized Labeling. A *parameterized labeling*, denoted as L_ω^θ where $\theta \in \mathbb{N}_0$ is a non-negative integer, modifies L_ω by altering its cost arrays, truncating those near the bottom of the hierarchy. Formally, let $\pi(v)$ denote the maximum rank of v's successors in H_G, i.e., $\pi(v) = max\{\tau(u) \mid v \preceq u\}$. The cost arrays in L_ω^θ are defined as $C^\theta = \{C^\theta(v) \mid v \in V, \pi(v) - \tau(v) \geq \theta\}$. Essentially, labels in L_ω^θ are retained unchanged ($C^\theta(v) = C(v)$) at the upper levels of the tree hierarchy, but *truncated* ($C^\theta(v) = \emptyset$) at the lower levels.

This design of parameterized labeling is further supported by several observations: (1) *Local searches:* Vertices at lower levels tend to have few upward neighbors in S_ω, limiting the effort required for local searches in S_ω. (2) *Vertex distribution:* Most vertices appear in lower levels, so truncating them reduces label size significantly. (3) *Label sizes:* Cost arrays make up the bulk of label sizes. Vertices at lower levels have more ancestors, leading to larger labels.

Example 6.1. *Consider Fig. 4, where vertices in dark blue represent untruncated vertices, while the vertices in light blue are truncated for* $\theta = 2$*. Consider for example vertex* v_{12}*: we have* $\pi(v_{12}) - \tau(v_{12}) = 7 - 3 = 4 \geq \theta$*, so its full label information consisting of* $\mathcal{I}(v_{12}) = 0, \mathcal{T}(v_{12}) = [2, 4]$ *and* $C(v_{12}) = [6, 6, 6, 0]$ *is preserved. In contrast, for vertex* v_1 *we have* $\pi(v_1) - \tau(v_1) = 6 - 5 = 1 < \theta$*, so its cost array is not stored. We still maintain* $\mathcal{I}(v_1) = 10$ *and* $\mathcal{T}(v_1) = [2, 4, 5]$*.*

Integrated Querying. Using a parameterized labeling L_ω^θ, we propose an integrated query algorithm that combines shortcut exploration and label-based cost lookup. For cases where costs $C(s)$ and/or $C(t)$ have been truncated due to parameterization, the algorithm computes these costs using both S_ω and L_ω^θ. Once computed, $C(s)$ and $C(t)$ can be used to find an optimal route as before. The high-level description is provided in Algorithm 3. At Line 3, we pass the height h of the lowest common ancestor to Algorithm 4, since we only require the first h cost values and will use this fact to limit computational work.

At the core of integrated querying lies cost computation, as described in Algorithm 4. For a query vertex v whose cost has been truncated (i.e., $C^\theta(v) = \emptyset$), we perform a combination of online and offline searches with online search being conducted on the shortcut graph S_ω and offline search on the truncated hub labeling L_ω^θ. We start by initializing a cost array c and an ancestor array a (Lines 4–5). We then perform an upward search on S_ω until we reach a vertex whose cost has not been truncated (Lines 15–19). From here, we replace the remaining steps of the upward search with more efficient cost lookups in L_ω^θ (Lines 10–14). Observe that in Lines 11–14 we do not update costs to all ancestors, but only for those up to height h, as these are the only ones used by Algorithm 3. Note further that we may skip distance updates for an ancestor $w = \perp$ in Line 9 since in those cases w was not reached through upward search in S_ω, and while the ancestor at rank $\tau(w)$ (whose identity is unknown) may well appear on a shortest path p from v to another ancestor, this path must have been considered previously when processing $C(w')$, where w' is the first untruncated ancestor of v in p. The deletion of ancestors in Line 14, which is done to prevent needless processing, can be justified in the same way.

If $\theta = 0$ or the costs of a query pair (s, t) have not been truncated (i.e., $C^\theta(s) \neq \emptyset$ and $C^\theta(t) \neq \emptyset$), integrated querying is reduced to querying over the full labeling L_ω. Conversely, for $\theta = \infty$, integrated querying reduces to bidirectional search over S_ω as in CCH.

Example 6.2. *Consider a query* (v_{14}, v_{13})*, which is answered using the parameterized labeling* L_ω^θ *shown in Fig. 4. Since both* $C^\theta(v_{14})$ *and* $C^\theta(v_{13})$ *are truncated, their costs will be computed up to height* $h = 3$*, the rank of their lowest common ancestor* v_{12}*. For* $v = v_{14}$ *in Algorithm 4, the computation begins by following the upwards neighbors* $\{v_9, v_{12}\}$ *of* v_{14} *in* S_ω *as shown in Fig. 5(b). The cost and ancestor arrays are then updated to* $c = [\infty, \infty, \infty, 2, 2, 0]$ *and* $a = [\perp, \perp, \perp, v_{12}, v_9, v_{14}]$ *in Lines 15–19. In the next iteration (*$w = v_9$*) we find that* $C(v_9)$ *is not truncated, and update the distance array to* $c = [4, 6, 4, 2, 2, 0]$ *in Lines 11–14. This is repeated for* $w = v_{12}$ *but does not cause any changes to* c*. For ancestors* $w \in \{v_7, v_8, v_{10}\}$*,* $a[\tau(w)]$ *is unknown (*$\perp$*), so no further updates are performed. The process of cost computation for* $C(v_{13})$ *is similar, with the key difference being that* $C^\theta(v_{15})$ *is truncated. As a result, upward search in* S_ω *is performed during the second iteration. This leads to* $a = [\perp, v_8, v_{10}, v_{12}, v_{15}, v_{13}]$ *being almost completely known, with no iterations skipped.*

7 COMPLEXITY ANALYSIS

Table 1 summarizes the complexity bounds of existing approaches and our method. Here, n and m denote the number of vertices and edges, respectively, and k represents the partitioning depth. We assume $n \in O(m)$.

CRP. We denote by n_p, m_p, and $\hat{m}_p$ the maximum number of nodes, edges and shortcuts within any partition, and by n_b, m_b, and $\hat{m}$ the total number of boundary nodes, cross-partition edges and shortcuts. During preprocessing, CRP constructs the multi-level graph partition by repeatedly applying the techniques of [13], with a combined complexity of $O(m \cdot m_b)$. During customization, costs in the overlay graph are updated by running Dijkstra's algorithm within partitions from each boundary node, with a total complexity of $O(n_b \cdot m_p \cdot \log n)$. Shortest-path queries use a hierarchical structure to explore the overlay graph efficiently, with a complexity of $O((m_p + k \cdot \hat{m}_p) \cdot \log n)$. Finally, the space complexity includes storage for the base graph $O(m)$ and overlay graph $O(\hat{m})$, for a total of $O(m + \hat{m})$.

CCH. For CCH, preprocessing involves constructing vertex ordering and the CCH graph G^+, with a complexity of $O(m \cdot k)$. Customization updates shortcut costs in $O(\hat{m} \cdot \hat{d})$, where $\hat{m}$ is the number of shortcut edges and $\hat{d}$ is the maximum upward neighbors in G^+. Queries run in $O(k \cdot \hat{d})$, and space complexity is $O(\hat{m})$.

CTL. For CTL, l is the maximum length of cost arrays in L, while $\hat{m}$ and $\hat{d}$ denote the number of edges and the maximum number of upward neighbors in S, respectively. Although parameter definitions differ across approaches, they are conceptually equivalent and similar in size (identical when using the same hierarchies). During preprocessing, we use the algorithm from [17] to compute balanced separators with a complexity of $O(m \cdot c)$, where c is the

Table 1: Comparison of time and space complexity.

Aspect	CRP	CCH	CTL
Preprocessing	$O(m \cdot m_b)$	$O(m \cdot k)$	$O(m \cdot k)$
Customization	$O(n_b \cdot m_p \cdot \log n)$	$O(\hat{m} \cdot \hat{d})$	$O(l \cdot k \cdot \hat{d})$
Querying	$O((m_p + k \cdot \hat{m}_p) \log n)$	$O(k \cdot \hat{d})$	$O(k \cdot \hat{d})$
Memory Size	$O(m + \hat{m})$	$O(\hat{m})$	$O(l \cdot k)$

separator size. Omitting separators between boundary vertices does not affect the overall complexity. The total size of all separators for any edge is bounded by k, resulting in a preprocessing complexity of $O(m \cdot k)$ for constructing the hierarchy H_G. Customization involves two steps: customizing S and L. Customizing S compares pairs of upward neighbors for each vertex, with a complexity of $O(\hat{m} \cdot \hat{d})$. Customizing L, as described in Algorithm 2, processes cost arrays C in $O(l \cdot \hat{d} \cdot k)$. Querying an optimal route has two parts. The first completes in $O(\theta \cdot \hat{d})$, where θ is the number of ancestor vertices with truncated labels reached. The second part completes in $O(k \cdot |V_{ua}|)$, where V_{ua} is the set of untruncated ancestors. Among truncated ancestors, let w be the one with minimal depth $\tau(w)$. Every vertex in V_{ua} connects to w via a valley path through the starting vertex (denoted v in Algorithm 4), making V_{ua} a subset of w's upward neighbors, with $|V_{ua}| \leq \hat{d}$. Thus, total query complexity is $O(k \cdot \hat{m})$. Finally, space complexity is determined by $O(l \cdot k)$ for storing the label structure L (reducible with truncation), $O(n + \hat{m})$ for S, and $O(n \cdot \log n)$ for H_G.

8 KEY VARIANTS

Parallel Customization. The customization of tree labeling scheme can be parallelized as follows. Vertices $v \in V$ are divided into groups $G_1, \ldots, G_k$ based on their rank $i = \tau(v)$. These groups are processed in descending order of i for customizing the shortcut graph and in ascending order of i for customizing the labeling, enabling parallel execution. Synchronization between groups is achieved using a barrier to align threads. Each thread writes only to the shortcut and label it is currently processing and reads only from shortcuts and labels in strictly lower (or upper) groups, avoiding read/write conflicts and eliminating the need for locks or atomic operations. This approach parallelizes Line 4 of Algorithms 1 and 2, processing vertices in increasing order of levels i.

Path Queries. To query optimal routes, we track the endpoints of shortcuts in S_ω that lie on an optimal route while computing costs in Algorithm 4. These endpoints form a sequence of ancestors with decreasing ranks τ. Rather than storing entire paths, we only store the preceding shortcut endpoint for each ancestor and construct the endpoint sequence once a hub vertex on the optimal route is identified. Tracking shortcut endpoints is trivial for the truncated part of the search (Lines 15-19), where shortcuts in S_ω are explicitly followed. For the untruncated part (Lines 10-14), where costs in L^θ_ω are processed, we store an additional path array $P(w) = [\rho_0, \ldots, \rho_k]$ for each untruncated cost $C(w)$. Here ρ_i denotes the first shortcut endpoint on an optimal route from w to its ancestor with rank i. Since path arrays can be customized by following shortcuts in Algorithm 2, customization time is barely affected, but labeling size doubles.

To construct the tail of the shortcut endpoint sequence, we begin at a vertex w_0 with an untruncated label and iteratively find the next endpoint $w_{i+1} = P(w_i)[\tau(x)]$ until reaching the hub vertex x. The decreasing rank of endpoints w_i ensures their labels remain untruncated. Note that we do not store the identities of ancestor vertices in L^θ_ω, so x is unknown until it appears as a shortcut endpoint; only its rank $\tau(x)$ is available beforehand. To construct an optimal route from the sequence of shortcut endpoints, we proceed as in [16]. For each shortcut (v, w) in S_ω, we store the *downward triangle* node z used to form (v, w) by concatenating (z, v) and (z, w), unpacking the path recursively.

Directed Road Networks. Our method can be easily adapted for directed road networks by modifying the tree labeling scheme L, specifically the cost arrays C. For each vertex $v \in V$, we create *forward* and *reverse* cost arrays to store costs for both directions during customization with respect to a given metric.

9 EXPERIMENTS

We conducted experiments on a Linux server Intel Xeon W-2175 with 2.50GHz CPU, 28 cores, and 512GB of memory. We implemented the proposed method in C++20 and compiled with the GNU C++ compiler 11.4.0 using *-O3* and *-march=native* optimizations.

Datasets. We use 10 real-world road networks summarized in Table 2. Nine are from the US, sourced from the 9th DIMACS Implementation Challenge [15], and one is from Western Europe, managed by PTV AG [3].

Table 2: Summary of 10 real-world road networks.

Network	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
EUS	Eastern USA	3 598,623	8 778 114	4 461	201 MB
WUS	Western USA	6 262 104	15 248 146	4 420	349 MB
CUS	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	3 175	974 MB

Baselines. We compare our method, CTL, against two baseline methods: *Customizable Contraction Hierarchies (CCH)* [16], a fully customizable method, and *Incremental Hierarchical 2-Hop Labeling (IncH2H)* [45], a partially customizable method which incrementally maintains 2-hop labeling for a small set of edge weight increases and decreases to support efficient queries. We exclude CRP [12] from our comparisons, as CCH [7, 16] has been shown to provide better query performance. We also do not compare against non-customizable approaches, including goal-directed search techniques [24, 26, 31] and hierarchical methods [28, 37], as these have been significantly outperformed in terms of query time by recent non-customizable techniques such as Contraction Hierarchies (CH) [21, 22] and labeling-based methods [1, 30]. Furthermore, these techniques are considerably slower in terms of customization time when compared to fully or partially customizable approaches, such as CCH or IncH2H [7, 16, 45], which are included in our experimental evaluation. All algorithms were implemented in C++.

Table 3: Comparison of preprocessing time (PT), customization time (CT), query time (QT), and labeling size (LS).

Network	PT [s]		CT [s]		QT [μs]		LS [MB]	
	CTL	CCH	CTL	CCH	CTL	CCH	CTL	CCH
NY	2.161	2.732	0.242	0.134	3.535	22.77	49	16
BAY	2.093	3.272	0.176	0.099	3.148	11.07	40	14
COL	3.141	5.074	0.218	0.121	3.356	19.31	57	18
FLA	9.470	14.06	0.636	0.399	4.304	18.11	148	48
CAL	22.03	30.25	1.544	0.820	5.558	36.62	307	86
EUS	58.25	76.91	2.688	1.785	10.62	84.11	474	162
WUS	104.4	149.8	5.041	3.213	11.00	81.37	812	276
CUS	418.3	554.3	22.04	10.58	15.64	318.9	2 388	657
USA	716.6	964.1	33.44	16.37	14.08	401.2	4 135	1 101
EUR	697.4	901.4	30.30	12.84	18.82	643.4	3 720	869

9.1 Performance Comparison

We first compare the performance of our method CTL with CCH across four metrics: *preprocessing time (PT)* – the time to construct metric-independent data structures; *customization time (CT)* – the time to customize the preprocessed data structures to a given metric; *query time (QT)* – the time to retrieve the optimal cost; and *labeling size (LS)* – the memory consumed by the precomputed data structures. For CT, QT, and LS evaluations, we set $\theta = 20$ for the five smaller networks and $\theta = 100$ for the five larger datasets, as taller tree hierarchies in larger datasets benefit from a higher θ value. Table 3 summarizes the comparison results.

Preprocessing Time (PT). The preprocessing time of CTL represent the total time spent on constructing the tree hierarchy H_G and the shortcut graph S_G. Before constructing H_G, we contract the road network by repeatedly removing degree-one vertices, following a similar approach described in [17]. In comparison, the preprocessing time for CCH includes the time required to obtain the vertex ordering and to construct the CCH graph G^+.

As shown in Table 3, the preprocessing time of CTL is smaller than that of CCH. This is because contracting degree-one vertices reduces the graph size by up to 30%, significantly improving the partitioning performance for constructing H_G, particularly on large networks. In contrast, CCH directly partitions the original graph to obtain a vertex ordering, which is inherently more time-consuming.

Customization Time (CT). To evaluate customization performance, we randomly generated five sets of weights based on travel times and calculated the average customization time for CTL and CCH. Compared to CCH, CTL is approximately twice as slow due to the additional step required to customize the tree labeling L. However, this performance gap can be narrowed by selecting larger θ values. As shown in Fig. 6(c)–(d), the customization time decreases consistently as θ increases. Moreover, employing the parallel variant significantly boosts performance, achieving up to a 5 times improvement over the sequential version, as illustrated in Fig. 9.

Query Time (QT). After customizing each weight metric, we randomly sampled 1 million query pairs and calculated the average query time. As shown in Table 3, CTL significantly outperforms CCH, particularly on large datasets where CTL is up to 34 times faster. We can also see in Fig. 6(a)–(b), the query time increases with larger θ values due to the additional search effort; however the increase is sublinear. Even with larger θ values, CTL maintains

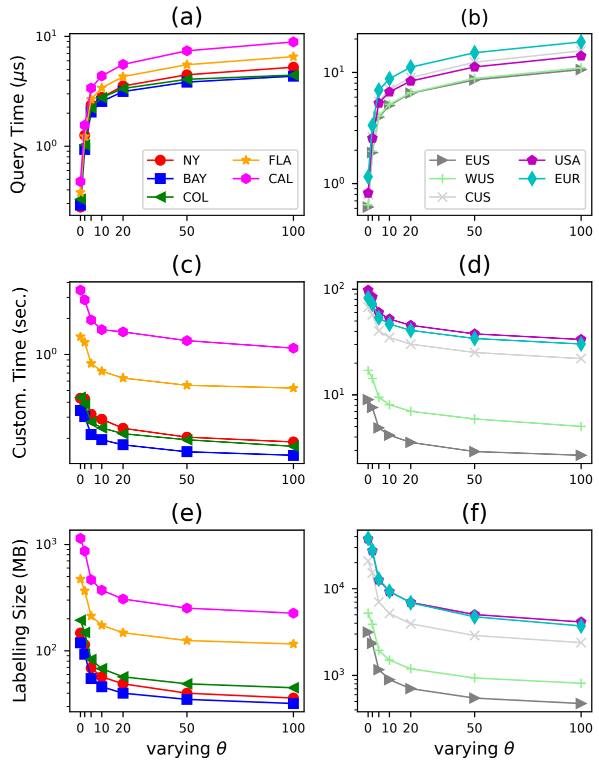

Figure 6: Query time, customization time, and labeling size under varying parameter θ.

a significant advantage over CCH. For instance, as illustrated in Table 3, for the largest datasets, USA and EUR, CTL achieves an average query time of 14μs and 19μs, respectively, compared to 401μs and 643μs for CCH.

Labeling Size (LS). In Table 3, we compare the space requirements of CTL and CCH. We can see that CCH consumes about 3× less space because it does not compute a labeling, instead relies on expensive searches at query time over a preprocessed CCH graph G^+. We can also see in Fig. 6(e)–(f) that the increased θ drastically reduces the labeling size for CTL, e.g., with $\theta = 100$, it consumes only about 4 gigabytes for two largest networks USA and EUR as shown in Table 3 and provide the best trade-offs between query time, customization time and labeling size.

9.2 Performance Discussion and Analysis

Varying Parameter θ. We evaluate the performance of CTL with varying $\theta \in \{0, 2, 5, 10, 20, 50, 100\}$, as shown in Fig. 6. Fig. 6(c)-(d) show a linear decrease in customization time with increasing θ, reducing to half at $\theta = 20$. Similarly, labeling size follows the same trend, as seen in Fig. 6(e)–(f). In contrast, query time in Fig. 6(a)–(b) increases with higher θ due to greater dependence on the shortcut graph S_G for cost computation. However, this increase is sublinear, and even at $\theta = 100$, query time remains about an order of magnitude faster than CCH on large networks. For large datasets like USA and EUR, choosing $\theta > 100$ can further optimize the trade-off between customization time, query time, and labeling size, depending on application requirements.

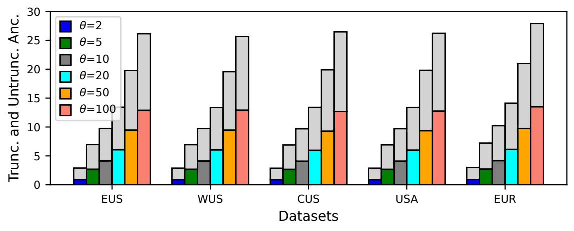

Figure 7: Truncated ancestors (colored) using S_ω, and untruncated ancestors (grey) using L_ω^θ for cost computation.

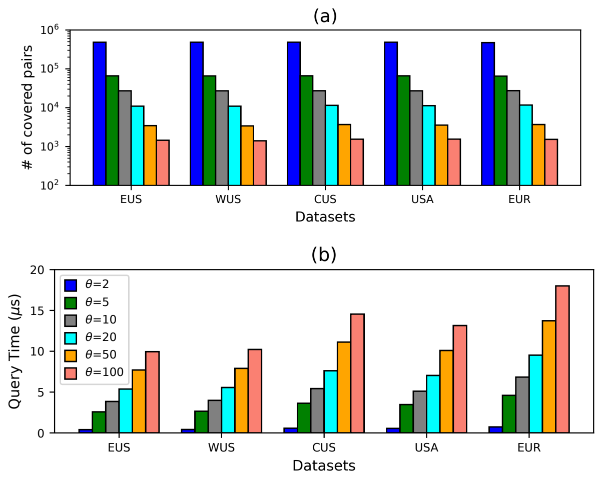

Figure 8: (a) shows the distribution of query pairs answered using L_ω^θ and their performance in (b) with increasing θ.

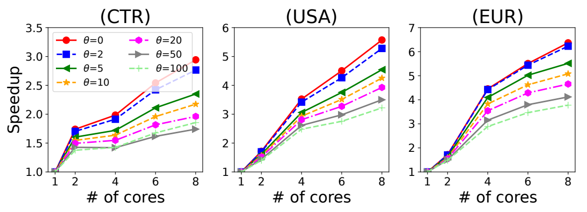

Figure 9: Customization performance with increasing the # of cores on largest three networks.

Pairs Coverage. Fig. 8(a) shows the total number of query pairs whose labels are untruncated under varying θ, while Fig. 8(b) illustrates query performance for pairs with truncated labels for either or both vertices. As θ increases, the fraction of untruncated query pairs decreases. For smaller θ, query performance improves because a higher proportion of untruncated label costs are used which are processed more efficiently. Conversely, larger θ values result in increased query times, as shown in Fig. 8(b), because the number of truncated pairs grows, requiring more computations on the shortcut graph S_ω during queries.

Truncated/Untruncated Ancestors. Fig. 7 presents the average number of ancestors whose costs are computed using S_ω for truncated vertices (colored parts of bars) and precomputed costs using L_ω^θ for untruncated vertices (grey parts of bars) during queries. The results highlight a balance between the use of the shortcut graph and labeling, with labeling being the dominant factor as precomputed costs are utilized most of the time. This dominance of labeling keeps query times increasing sublinearly with higher θ values, making CTL much more efficient than CCH for querying.

9.3 Scalability

Scalability w.r.t. # of Cores. Fig. 9 shows the performance of CTL on three largest networks, CUS, USA, and EUR, by varying the number of cores from 1 to 8. These networks are well-suited for parallel processing due to the large number of vertices at each tree hierarchy level, enabling efficient workload distribution across cores. CTL achieves near-linear performance improvement with increasing cores, demonstrating its scalability and effectiveness in leveraging parallel processing for large-scale networks.

Scalability w.r.t. # of Updates. In Fig. 10, we compare CTL with the state-of-the-art partial customizable method IncH2H, which incrementally maintains the H2H-Index [32] to reflect edge cost updates. We randomly sampled update batches of size 200−1800. For each batch i, we first double the edge costs (a, b, ω), i.e., $2\times\omega$, to test IncH2H$^+$ (cost increases), then revert them to ω for IncH2H$^-$ (cost decreases). We compare these update times with the customization time of CTL ($\theta = 20$ for small networks, $\theta = 100$ for large), and with preprocessing time of IncH2H. As shown in Fig. 10, the update time of IncH2H grows with batch size, exceeding customization time of CTL at just 200 updates on most large networks and struggling even on many small ones. On the EUR dataset, this threshold is reached at 700 updates. Overall, CTL is better suited for scenarios with frequent or large-scale updates, such as minute-by-minute changes. Preprocessing time of IncH2H is over an order of magnitude slower than the customization time of CTL. Table 5 further shows that while IncH2H offers fast queries, it suffers from high update times and large label sizes, making it unsuitable for customizable routing applications requiring frequent updates and scalable performance.

10 CASE STUDY: TRAFFIC ASSIGNMENT

This section applies the proposed method to the traffic assignment (TA) problem, a well-known challenge in traffic science and operations research [19, 34, 39, 46]. TA determines traffic flow patterns in a road network (Example 1.1) and aims to achieve *user equilibrium*, where no traveler can switch routes to reduce travel time without impacting others. Based on Wardrop's first principle [42], this equilibrium models realistic route choices, making TA essential for analyzing and optimizing road networks. We solve the TA problem using the Frank-Wolfe method [20]. Each iteration updates edge costs, computes shortest paths for all O-D pairs $\{(s_i, t_i)\}$, and adjusts costs based on the congestion function derived from previous iterations' edge loads. The algorithm ensures convergence to equilibrium and stops when improvements are negligible [39].

Dataset. Following [9], we use the road network of Stuttgart, Germany, along with four demand sets representing morning peak, evening peak, full-day, and week-long travel demands. The Stuttgart network is treated as undirected, and each demand set is made symmetrical by including the inverse pair (t_i, s_i) for every O-D pair (s_i, t_i). The network comprises $|V| = 134\,663$ vertices and $|E| = 324\,110$ edges. Detailed information on the number of O-D pairs (including inverse pairs) and time windows for each demand set is provided in Table 6.

Table 4: Average running time per iteration of traffic assignment using CCH or CTL for the *S-morn*, *S-even*, *S-day*, and *S-week* demand sets. Reported times (in seconds) include customization (once per iteration), queries (sum over all O-D pairs), and total runtime.

		S-morn			*S-even*			*S-day*			*S-week*		
Method	θ	cust.	queries	total	cust.	queries	total	cust.	queries	total	cust.	queries	total
CCH	–	0.047	11.28	11.33	0.048	12.78	12.82	0.048	143.2	143.2	0.048	878.5	878.6
CTL	0	0.192	1.806	1.999	0.190	2.019	2.209	0.191	17.18	17.38	0.190	89.94	90.13
CTL	2	0.162	2.154	2.316	0.163	2.430	2.594	0.164	21.68	21.85	0.163	115.5	115.7
CTL	5	0.104	2.586	2.691	0.104	2.915	3.019	0.103	26.97	27.07	0.103	148.2	148.3
CTL	10	0.092	2.705	2.796	0.092	3.062	3.154	0.091	28.69	28.78	0.091	159.1	159.2
CTL	20	0.083	2.826	2.909	0.082	3.173	3.255	0.082	30.32	30.41	0.083	169.2	169.3
CTL	50	0.075	2.998	3.073	0.075	3.384	3.459	0.072	34.56	34.63	0.076	183.8	183.8
CTL	100	0.072	3.140	3.212	0.072	3.523	3.595	0.076	32.72	32.80	0.072	195.7	195.8

Table 5: Query time (QT) and labeling size (LS) for IncH2H.

Network	QT [μs]	LS [MB]
NY	0.913	849
BAY	0.841	889
COL	1.018	1 444
FLA	1.019	2 605
CAL	1.333	8 613
EUS	1.683	21 610
WUS	1.702	37 791
CUS	2.483	183 887
USA	3.428	317 961
EUR	3.888	331 354

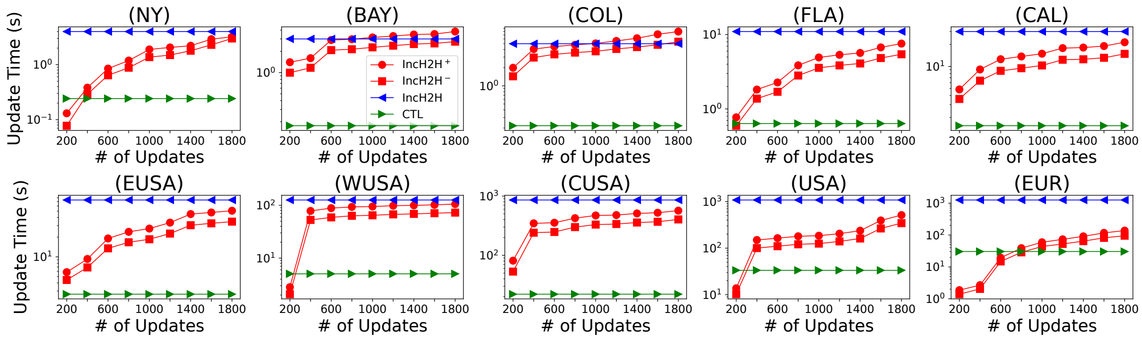

Figure 10: Update time analysis of IncH2H with varying numbers of updates for both edge cost increases and decreases, compared to the reconstruction time of IncH2H and the customization time of CTL.

Table 6: Demand sets for traffic assignment (TA).

Demand set	Time window	O-D pairs
S-morn	Tue, 07:30-08:30am	496 862
S-even	Tue, 04:30-05:30pm	560 728
S-day	Tue, whole day	6 710 884
S-week	whole week	42 496 556

Baselines. Buchhold et al. [9] present the state-of-the-art approach for TA using CCH. As shown in Section 9, CTL achieves faster query times than CCH, albeit with slightly higher customization times. Thus, we expect CTL to be better suited for TA than CCH for sufficiently large demand sets. We integrate our implementation of CTL into a TA framework[1] and compare the total time of customizing and querying with the state-of-the-art CCH. Since TA requires path queries, we extend CTL to support path computation as detailed in Section 8. To evaluate typical running times per iteration,we limited the TA algorithm to 50 iterations per configuration instead of running it to full convergence. This approach ensures a fair comparison, as running times per iteration stabilize quickly.

Results. Table 4 demonstrates that CTL consistently outperforms CCH across all demand sets in total runtime, with query times up to 8 times faster for $\theta = 0$. While CTL's customization times are higher than CCH's at lower θ, the dominance of query efficiency results in up to 4 times faster total runtimes for smaller sets (*S-morn*, *S-even*) and up to 8 times for larger sets (*S-day*, *S-week*). As θ increases, CTL achieves reduced customization times at a slight cost to query performance, offering flexible trade-offs that allow tuning based on specific application needs. For traffic assignment problems involving larger demand sets, moderate θ values (e.g., 20 or 50) effectively balance customization and query efficiency, enabling significant reductions in total runtime, with query times as low as 30.32 seconds for *S-day* and 169.2 seconds for *S-week*, facilitating faster iterations in large-scale traffic simulations.

[1]https://github.com/vbuchhold/routing-framework

11 CONCLUSION

This paper addresses customizable routing in dynamic road networks for efficient real-time route computation. We analyze limitations of existing methods and propose a three-phase approach, *Customizable Tree Labeling (CTL)*, which offers significantly faster metric-independent preprocessing and query performance, while remaining competitive in customization time and label size. CTL provides two variants: a basic one optimized for query speed, and a parameterized version that balances preprocessing, customization, and querying through a novel integrated querying technique. Experiments on 10 large real-world road networks show that our algorithms significantly outperform existing methods in preprocessing and querying, demonstrating their effectiveness for real-time and dynamic routing. For future work, we aim to optimize CTL through parallelization. As customization and query scan cost arrays, they are well suited for SIMD or GPU acceleration. We also plan to explore scalable multi-threading, building on recent advances in parallel CH preprocessing [41] and graph partitioning [25].

ACKNOWLEDGMENTS

Muhammad Farhan was supported by the Helmholtz Information & Data Science Academy (HIDA) through a Visiting Researcher grant, which enabled a short-term stay at the Karlsruhe Institute of Technology (KIT) in Karlsruhe, Germany.

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 *2014 Proceedings of the sixteenth workshop on algorithm engineering and experiments (ALENEX)*. 147–154.

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

[6] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. 2007. In transit to constant time shortest-path queries in road networks. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 46–59.

[7] Thomas Bläsius, Valentin Buchhold, Dorothea Wagner, Tim Zeitz, and Michael Zündorf. 2025. Customizable Contraction Hierarchies–A Survey. *arXiv preprint arXiv:2502.10519* (2025).

[8] Johannes Blum and Sabine Storandt. 2022. Customizable Hub Labeling: Properties and Algorithms. In *International Computing and Combinatorics Conference*. Springer, 345–356.

[9] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. 2019. Real-time Traffic Assignment Using Engineered Customizable Contraction Hierarchies. *Jorunal of Experimental Algorithmics* 24, Article 2.4 (Dec. 2019), 28 pages. https://doi.org/10.1145/3362693

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

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

[12] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. 2017. Customizable route planning in road networks. *Transportation Science* 51, 2 (2017), 566–591.

[13] Daniel Delling, Andrew V Goldberg, Ilya Razenshteyn, and Renato F Werneck. 2011. Graph partitioning with natural cuts. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 1135–1146.

[14] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. 2009. Engineering route planning algorithms. In *Algorithmics of large and complex networks: design, analysis, and simulation*. Springer, 117–139.

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

[16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2016. Customizable contraction hierarchies. *Journal of Experimental Algorithmics (JEA)* 21 (2016), 1–49.

[17] Muhammad Farhan, Henning Koehler, Robert Ohms, and Qing Wang. 2024. Hierarchical Cut Labelling–Scaling Up Distance Queries on Road Networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[18] Muhammad Farhan, Henning Koehler, and Qing Wang. 2025. Dual-Hierarchy Labelling: Scaling Up Distance Queries on Dynamic Road Networks. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–25.

[19] Michael Florian and Donald Hearn. 1995. Network Equilibrium Models and Algorithms. In *Network Routing*. Handbooks in Operations Research and Management Science, Vol. 8. Elsevier, Chapter 6, 485–550. https://doi.org/10.1016/S0927-0507(05)80110-0

[20] Marguerite Frank and Philip Wolfe. 1956. An Algorithm for Quadratic Programming. *Naval research logistics quarterly* 3, 1-2 (1956), 95–110.

[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] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact routing in large road networks using contraction hierarchies. *Transportation Science* 46, 3 (2012), 388–404.

[23] Alan George. 1973. Nested dissection of a regular finite element mesh. *SIAM journal on numerical analysis* 10, 2 (1973), 345–363.

[24] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory.. In *SODA*, Vol. 5. 156–165.

[25] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. 2024. Scalable High-Quality Hypergraph Partitioning. *ACM Trans. Algorithms* 20, 1 (2024), 9:1–9:54. https://doi.org/10.1145/3626527

[26] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[27] 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 ACM SIGMOD International Conference on Management of Data*. 445–456.

[28] 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.

[29] Henning Koehler, Muhammad Farhan, and Qing Wang. 2025. Stable Tree Labelling for Accelerating Distance Queries on Dynamic Road Networks. *arXiv preprint arXiv:2501.17379* (2025).

[30] Dennis Luxen and Peter Sanders. 2011. Hierarchy Decomposition for Faster User Equilibria on Road Networks. In *Experimental Algorithms*, Panos M. Pardalos and Steffen Rebennack (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 242–253.

[31] Jens Maue, Peter Sanders, and Domagoj Matijevic. 2010. Goal-directed shortest-path queries using precomputed cluster distances. *Journal of Experimental Algorithmics (JEA)* 14 (2010), 3–2.

[32] 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 ACM SIGMOD International Conference on Management of Data*. 709–724.

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

[34] Olga Perederieieva, Matthias Ehrgott, Andrea Raith, and Judith Y.T. Wang. 2015. A Framework for and Empirical Study of Algorithms for Traffic Assignment. *Computers & Operations Research* 54 (2015), 90–107. https://doi.org/10.1016/j.cor.2014.08.024

[35] Yu-Xuan Qiu, Dong Wen, Lu Qin, Wentao Li, Rong-Hua Li, Zhang Ying, et al. 2022. Efficient shortest path counting on large road networks. *Proceedings of the VLDB Endowment* (2022).

[36] Peter Sanders and Dominik Schultes. 2005. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Conference on Algorithms*.

[37] 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.

[38] Christian Schulz. 2013. *High Quality Graph Partitioning*. Ph.D. Dissertation. Karlsruhe Institute of Technology. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000035713

[39] Yosef Sheffi. 1985. *Urban Transportation Networks*. Prentice-Hall, Englewood Cliffs, N.J.

[40] Robert Endre Tarjan. 1983. *Data structures and network algorithms*. SIAM.

[41] Zijin Wan, Xiaojun Dong, Letong Wang, Enzuo Zhu, Yan Gu, and Yihan Sun. 2024. Parallel Contraction Hierarchies Can Be Efficient and Scalable. arXiv:2412.18008 [cs.DS] https://arxiv.org/abs/2412.18008

[42] John Glen Wardrop. 1952. Some Theoretical Aspects of Road Traffic Research. *Proceedings of the Institution of Civil Engineers* 1, 3 (1952), 325–362. https://doi.org/10.1680/ipeds.1952.11259

[43] Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, Andy Diwen Zhu, and Shuigeng Zhou. 2012. Shortest path and distance queries on road networks: an experimental evaluation. *Proceedings of the VLDB Endowment* 5, 5 (2012), 406–417.

[44] Mengxuan Zhang. 2021. *Efficient shortest path query processing in dynamic road networks*. Ph.D. Dissertation.

[45] Yikai Zhang and Jeffrey Xu Yu. 2022. Relative Subboundedness of Contraction Hierarchy and Hierarchical 2-Hop Index in Dynamic Road Networks. In *Proceedings of the 2022 International Conference on Management of Data*. 1992–2005.

[46] Zhong Zhou and Matthew Martimo. 2010. Computational Study of Alternative Methods for Static Traffic Equilibrium Assignment. In *Proceedings of the 12th world conference on transport research (WCTRS), Lisbon, Portugal*. 1–15.