# Accelerating Shortest Path Counting on Road Networks

Zebin Chen
*University of New South Wales*
Sydney, Australia
zebin.chen1@unsw.edu.au

Kaiyu Chen
*University of New South Wales*
Sydney, Australia
kaiyu.chen1@unsw.edu.au

Dong Wen
*University of New South Wales*
Sydney, Australia
dong.wen@unsw.edu.au

Zhengyi Yang
*University of New South Wales*
Sydney, Australia
zhengyi.yang@unsw.edu.au

Wentao Li
*University of Leicester*
Leicester, United Kingdom
wl226@leicester.ac.uk

Ying Zhang
*University of Technology Sydney*
Sydney, Australia
ying.zhang@uts.edu.au

***Abstract*—Counting the number of shortest paths between two query vertices on road networks has a wide range of applications and has recently drawn significant research attention. The state-of-the-art solution builds a tree-based index using the concept of tree decomposition. However, its performance deteriorates when the tree decomposition results in an unbalanced tree and may not perform well when the query vertices are close to each other. This paper aims to improve the efficiency of shortest path counting. We propose a novel indexing scheme that combines hub labeling with a balanced tree hierarchy. This approach significantly reduces the number of visited labels compared to the state-of-the-art solution. Furthermore, we introduce several optimizations to enhance the efficiency of index construction and minimize its size. Extensive experiments conducted on real-world road networks demonstrate that our method achieves up to 4.1 times higher query efficiency and reduces the index size by a factor of 2.35 compared to the state-of-the-art solution.**

## I. Introduction

Road networks can be modeled as graphs, where vertices represent locations and weighted edges represent roads. Given any two vertices, a shortest path counting query determines the number of shortest paths between them, enabling various applications. Shortest path counts enhance Point of Interest (POI) recommendations by enabling service providers to identify the top-k nearest neighbors, such as restaurants or taxi-hailing locations [1]. Users often prefer locations with multiple shortest paths when distances are similar, as this provides greater flexibility and accessibility. Another key application is computing betweenness centrality [2], [3], a metric that measures the significance of a vertex within a network [4]. Betweenness centrality measures the significance of a vertex in a network by summing the fractions of the shortest paths that pass through it. Formally, the betweenness centrality of a vertex $u$ is defined as $\sum_{s \neq u \neq t \in V} \frac{spc_u(s,t)}{spc(s,t)}$, where $spc(s,t)$ is the number of shortest paths between vertices $s$ and $t$, and $spc_u(s,t)$ is the number of these paths that pass through $u$. This metric is particularly useful for predicting traffic flow and identifying critical points in the network [5].

**Existing Methods.** To query shortest path counts on road networks, a straightforward solution is to use a modified version of Dijkstra's algorithm that tracks path counts. However, this method often fails to meet real-time requirements, as it can traverse the entire graph in the worst case to compute results. TL-Index [1] is the state-of-the-art solution for shortest path counting queries on road networks. It combines hub labeling [6] with tree decomposition [7]. In this approach, hub labeling assigns a subset of vertices as labels for each vertex and stores both the distance and path counts to these labeled vertices. The labeled subset is determined using a tree hierarchy derived from tree decomposition. Queries are answered by processing the common ancestors of the two query vertices in the tree, where the common ancestors are guaranteed to form a vertex cut of query vertices. While this approach enhances query performance, it is sensitive to the number of common ancestors. Performance can degrade if the tree decomposition yields an unbalanced tree, resulting in a large number of common ancestors between query vertices.

**Our Idea.** Our general idea is still to maintain a vertex cut for any pair of vertices by a tree structure, and the query can be answered via the labels from query vertices to cut vertices. Inspired by a recent work [8] which computes a balanced cut tree to improve the efficiency of shortest distance queries, we extend the tree structure for shortest path counting queries. The tree structure maintains a many-to-one correspondence between graph vertices and tree nodes. We first follow the essential idea of the state-of-the-art solution and modify the tree in [8] to guarantee the set of common ancestors of any two vertices in the tree is their vertex cut. To this end, we assign a ranking scheme for vertices in the same tree node and precompute labels (shortest distance and shortest path count values) from each vertex to its ancestors. The labels are computed by excluding vertices with higher ranks or in ancestor nodes. In this way, we guarantee each shortest path is only counted once in query processing. Compared with TL-Index, our query efficiency is improved due to the relatively balanced tree structure, which indicates the number of common ancestors between two vertices in the tree is relatively smaller.

To further improve the query efficiency, we refine our index to guarantee that the vertices in the lowest common ancestor already form a vertex cut between vertices. The property enables only scanning labels stored by query vertices for the lowest common ancestor in query processing. We derive the index by computing a vertex cut to partition the graph and repeatedly applying this process for each part until no vertex exists. Here, the vertex cut in any round is a global shortest path cut to guarantee the above property, where the global shortest path cut means any shortest path in the original graph between two vertices from different parts must pass through the cut. Our method to compute these global shortest path cuts is to add shortcuts (auxiliary edges) into each part when we partition the graph. The shortcuts guarantee any pair of vertices within each part preserve the shortest distance and the shortest path count of the original graph (called count-preserved graph for short). As a result, we can apply an existing balanced cut method to safely derive a global shortest path count in each round. We propose two optimizations to improve the index construction. The first accelerates shortcut computation for a part by precomputing the count-preserved subgraph at a lower cost. The second optimization adopts a distance threshold to prune certain shortcuts, where the threshold can be efficiently derived from labels computed previously.

**Contributions.** We summarize main contributions as follows.

- *Two new tree-based indexes for shortest path counting.* We propose two new tree-based indexes for shortest path counting on large road networks. The query time by the first index is bounded by the number of common ancestors of two vertices, and the query time by the second index is bounded by the number of vertices in the lowest common ancestor of two query vertices.
- *Optimized index construction algorithms.* We propose two optimizations to speed up the construction of the final index, which achieves higher query efficiency compared with the basic index structure. One optimization speeds up the computation of shortcuts, and the other one prunes shortcuts when the shortcut distance weight is larger than a certain threshold. Note that fewer shortcuts imply lower costs to compute cuts and smaller sizes of cuts (i.e., tree nodes) in the rest rounds.
- *Extensive evaluations.* We conduct extensive experiments on 12 real-world road networks. The results demonstrate that query processing on CTL-Index (the first index) and CTLS-Index (the final index) can be up to 3.5 and 4.1 times faster, respectively, compared to the state-of-the-art method. In short-distance queries, CTLS-Index is up to 16 times faster than the state-of-the-art.

**Paper Organization.** The remainder of this paper is organized as follows: Section II provides the necessary background and reviews state-of-the-art methods. Section III introduces our proposed index. Section IV presents our improved index structure and two optimizations. Section V discusses the results of our experimental evaluation. Finally, Section VI reviews related work, and Section VII concludes the paper.

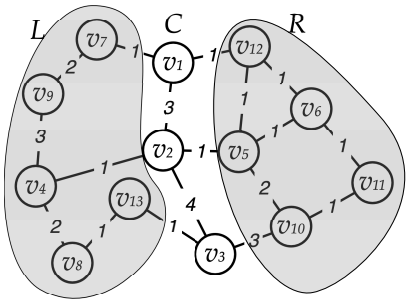


Fig. 1: A road network $G$.

## II. Background

### A. Preliminary

We study an undirected weighted road network $G$. We use $V(G)$ to denote the set of vertices and $E(G)$ to denote the set of edges in $G$. When the context is clear, we use $V$ to represent $V(G)$ and $E$ to represent $E(G)$. Give a vertex set $S \subseteq V$, the induced subgraph $G[S]$ consists of all edges between vertices in $S$. We use $n$ and $m$ to denote the number of vertices and edges in $G$, respectively. The neighbor set of a vertex $v \in V$ in $G$ is denoted as $N_G(v) = \{u|(u,v) \in E(G)\}$. Each edge $(u,v) \in E$ is associated with a positive distance weight $\phi(u,v)$. A path from a vertex $s$ to a vertex $t$ is a sequence of vertices $(s = v_0, v_1, v_2, \ldots, t = v_k)$, where $(v_i, v_{i+1}) \in E$ for each $0 \leq i < k$. The distance weight of a path $p$ is defined as $w(p) = \sum_{i=0}^{k-1} \phi(v_i, v_{i+1})$. Given two vertices $s, t \in V$, the shortest distance between $s$ and $t$ in $G$, denoted as $sd_G(s,t)$, is the minimum $w(p)$ over all paths $p$ from $s$ to $t$. A path $p$ is a shortest path between $s$ and $t$ if $w(p) = sd_G(s,t)$. The shortest path count between $s$ and $t$, denoted by $spc_G(s,t)$, is the number of distinct shortest paths between $s$ and $t$. When the context is clear, we use $sd(s,t)$ and $spc(s,t)$ to represent $sd_G(s,t)$ and $spc_G(s,t)$, respectively.

**Problem Definition.** Given a road network $G(V,E)$ and two arbitrary vertices $s, t \in V$, a shortest path counting query, denoted as $Q(s,t)$, is to answer the number of distinct shortest paths between $s$ and $t$. In this paper, we aim to develop an effective index-based approach to answer $Q(s,t)$ efficiently.

### B. The State of the Art: TL-Index

TL-Index, proposed in [1], is the state-of-the-art index-based solution for querying the number of shortest paths on road networks. The TL-Index constructs a tree hierarchy from the input road network, which enables scanning only a subset of vertices to count the shortest paths between two query vertices. Specifically, they first construct a tree structure by tree decomposition [7]. There is a one-by-one correspondence between tree nodes and the vertices in the original graph. Let $T_G$ be the tree generated by the TL-Index. The tree decomposition method guarantees that the common ancestors[1] (CA) of any pair of vertices form the vertex cut for the two vertices. They compute labels based on the tree structure for

[1]Each tree node in $T_G$ is considered an ancestor of itself.

query processing, and the labels are defined based on the following key concept.

**Definition 2.1:** (CONVEX PATH) Let $\prec$ be a ranking function on $V$, where $u \prec v$ indicates that $u$ has a higher rank than $v$. A path $p = (s, v_1, \ldots, v_k, t)$ is a convex path if each intermediate vertex has a lower rank than at least one of the terminal vertices. That is, for each $v_i$ (where $1 \leq i \leq k$), we have $s \prec v_i$ or $t \prec v_i$.

**Example 2.1:** Fig. 2 shows the TL-Index corresponding to Fig. 1, with $v_2 \prec v_1 \prec v_4 \prec v_7$. The path $p_1 = \{v_7, v_1, v_2\}$ is a convex path because the end point $v_2$ has a higher rank than $v_1$. In contrast, the path $p_2 = \{v_7, v_1, v_2, v_4\}$ is not a convex path, as neither $v_7$ nor $v_4$ has the highest rank.

Note that the convex path definition here is a generalization of the one presented in [1]. We use it for simplicity, and the essential idea remains unchanged. A convex shortest path is a convex path that minimizes its distance weight. The convex shortest path distance between vertices $s$ and $t$ is denoted as $csd(s,t)$, and the associated convex shortest path count is denoted as $cspc(s,t)$. Based on Definition 2.1, the TL-Index adopts the vertex depth in the tree structure derived from the tree decomposition as the vertex rank. The depth of a vertex $u$, denoted by $depth(u)$, is the shortest distance from the tree node containing $u$ to the tree root. A vertex with a smaller depth ranks higher. The TL-Index precomputes the convex shortest distance and the number of convex shortest path from each vertex to all its ancestors. In such a way, the convex shortest path distance/count from a vertex $u$ to its ancestor $v$ is their shortest path distance/count in the induced subgraph of all vertices in the subtree rooted at $v$. For $u$ and $v$ in $T_G$, let $CA(u,v)$ denote the common ancestors of $u$ and $v$. Given two vertices $s$ and $t$, $Q(s,t)$ can be answered by

$$spc(s,t) = \sum_{\substack{w \in CA(s,t) \\ csd(s,w)+csd(t,w)=sd(s,t)}} cspc(s,w) \cdot cspc(t,w) \tag{1}$$

, where $sd(s,t) = \min_{w \in CA(s,t)} csd(s,w) + csd(t,w)$.

The query time complexity is $O(h)$, where $h$ is the tree height (i.e., the maximum depth of all leaf nodes). The space complexity of the TL-Index is $O(n \cdot h)$.

The first step in index construction involves building a tree structure. Tree decomposition is a technique used to transform a graph into a tree-like structure. The tree decomposition algorithm [9] iteratively removes the vertex $v$ with the smallest degree from $G$ and creates the corresponding tree node $X(v)$. Then, for each pair of neighbors of $v$, the algorithm inserts an edge or updates the edge weights to ensure that the shortest distance and the shortest path count in the remaining graph are consistent with the original graph. After removing $v$, the tree parent of $X(v)$ will be the tree node $X(u)$ where $u$ is the first neighbor of $v$ to be removed. The tree decomposition algorithm terminates after removing all vertices. To compute the convex shortest distance and the convex shortest path count, the TL-Index proposes an upward

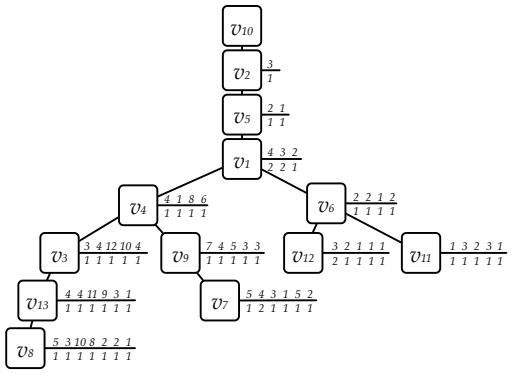



Fig. 2: The TL-Index for $G$.

computing framework, i.e., from high-depth vertices to low-depth vertices. For each vertex, they compute the distance and count to all ancestors, and the framework ensures that they can utilize certain information derived in previous rounds and avoid computing from scratch. The overall time complexity of the TL-Index construction algorithm is $O(n \cdot h \cdot w + n \log n)$. $w$ is the tree width and it is bounded by the tree height $h$.

**Example 2.2:** Fig. 2 contains the distance array and path count array of TL-Index. The first row of label entries represents convex shortest path distance and the second row represents convex shortest path count. For each vertex, the convex shortest path distance to itself is 0, and the convex shortest path count is 1, which are omitted for clarity. For $Q(v_5, v_9)$, $CA(v_5, v_9) = \{v_{10}, v_2, v_5\}$. For $v_5$, the convex shortest path distance to $CA(v_5, v_9)$ are $[2, 1, 0]$. For $v_9$, the convex shortest path distance to $CA(v_5, v_9)$ are $[7, 4, 5]$. The shortest distance between $v_5$ and $v_9$ is calculated as $\min(2+7, 1+4, 0+5) = 5$. Therefore, $Q(v_5, v_9) = 1 \cdot 1 + 1 \cdot 1 = 2$.

## III. CUT TREE BASED INDEX

The state-of-the-art solution TL-Index achieves high efficiency by fully exploiting the tree structure derived by tree decomposition. This tree structure has also been extensively studied for querying shortest distances and shortest paths [10]–[12]. A recent work by [8] proposes an index structure to improve the efficiency of shortest distance queries. Their index is built on a new tree structure called the cut tree, and its high efficiency in answering shortest distance queries motivates us to push the boundaries of efficiency for shortest path counting queries using a similar tree structure. The primary technical challenge lies in designing an appropriate tree structure for counting shortest paths and determining the precomputed values over that structure.

### A. The Cut Tree

**Definition 3.1:** (VERTEX CUT) Given a road network graph $G(V, E)$, a vertex set $C \subset V$ is a vertex cut if the deletion of $C$ from $G$ splits $G$ into multiple connected components. A vertex set $C$ is a vertex cut of two vertices $u$ and $v$ if any path between $u$ and $v$ passes through $C$.

For brevity, we refer to vertex cut as cut. We call a vertex subset $C$ a cut for a partition $L$ and a partition $R$ if $L$ and $R$ in $G$ become disconnected after removing $C$, and $L \cup C \cup R = V$. It follows that any path from a vertex $s \in L$ to a vertex $t \in R$ must pass through at least one vertex $c \in C$. We say that $C$ is a cut of two vertices $s$ and $t$ if they belong to different connected components after deleting $C$. The vertices in $C$ are referred to as cut vertices.

**Example 3.1:** Consider the graph $G$ in Fig. 1, the cut $C = \{v_1, v_2, v_3\}$ splits $G$ into two connected components. $C$ is a cut for $L$ and $R$, as well as for the vertices $v_7$ and $v_{11}$. However, $C$ is not cut for $v_5$ and $v_{10}$, since these two vertices remain in the same connected component after $C$ is deleted.

Our basic indexing idea is to precompute the distance and count labels from each vertex to several other vertices. In query processing, we identify the cut between the two query vertices and use the precomputed labels to count the number of shortest paths. [8] introduces a cut tree structure designed to identify a cut for any pair of vertices. The cut tree is built through multiple rounds of cut computation. In each round, a cut is computed and represented as a tree node (which may contain multiple vertices), and its parent node is set to the cut computed in the previous round. They then repeat this process for each component separated by the cut. Below, we formally define a modified version of the cut tree structure used in our algorithm.

**Definition 3.2:** (CUT TREE) Given a graph $G(V, E)$, a cut tree of $G$, denoted as $T_G$, is a rooted binary tree where each node $X \in T_G$ is a subset of $V$, satisfying the following conditions:

1) $\bigcup_{X \in T_G} X = V$;
2) $\forall X \in T_G$, $X$ is a cut for $LEFT(X)$ and $RIGHT(X)$ in the subgraph induced by all vertices in the subtree of $X$, where $LEFT(X)$ and $RIGHT(X)$ represent the vertices in the left and right subtrees of $X$, respectively.

We use $X(u)$ to denote the tree node containing vertex $u$. The primary difference between our tree structure and the original is the scope of the vertex cut. The original cut tree is designed for shortest distance queries. Their vertex cut is guaranteed to be a global shortest distance cut, i.e., for any two vertices belonging to different subtrees (components) separated by the cut, there exists at least one shortest path between them that passes through the cut. In contrast, our cut tree uses a local vertex cut, i.e., it excludes cut vertices from previous rounds and considers only the remaining subgraph. Thus, our cut tree ignores distance preservation.

**Example 3.2:** Fig. 3 contains a cut tree $T_G$ of the road network $G$ shown in Fig. 1. The ancestor nodes of $X_2$ is $\{X_1, X_2\}$. By Condition (2) in Definition 3.2, vertices in $X_2$ form a cut for the graph $G$ induced by all vertices in the subtree rooted at $X_2$. Specifically, the vertices in $X_2$ form a cut for the components $\{v_7, v_9\}$ and $\{v_8, v_{13}\}$ within the induced subgraph $G[\{v_4, v_7, v_8, v_9, v_{13}\}]$.

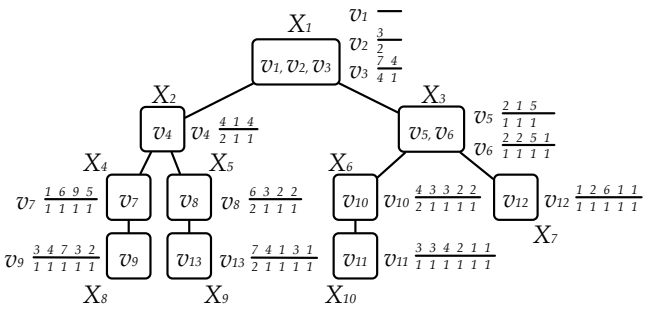



Fig. 3: The CTL-Index for $G$ (Example 3.2).

### B. Labels on the Tree

We first generalize the rationale of the TL-Index in the following lemma, which also provides guidance for determining the indexed labels on the cut tree.

**Lemma 3.1:** *Let $C$ be the vertex cut of $L$ and $R$, where $V = L \cup C \cup R$. Let $\prec$ be a ranking function that the vertices in $C$ are ranked arbitrarily and all vertices in $C$ rank higher than the other vertices in $V$. Given $\prec$ and the corresponding convex shortest path distance/count from each vertex in $C$ to all other vertices in $V$, the shortest path count between an arbitrary vertex $s$ in $L \cup C$ and $t$ in $C \cup R$ can be computed by Eq.* (1) *where $CA(s,t)$ is replaced by $C$.*

**Lemma 3.2:** *Given a cut tree $T_G$ of a graph $G$ and two vertices $u$ and $v$, the set of all vertices in the common ancestors of $X(u)$ and $X(v)$ is a cut of $u$ and $v$ in $G$.*

Lemma 3.2 provides a cut for an arbitrary pair of vertices. To meet the condition in Lemma 3.1, we simply assign ranks for vertices within the same tree node based on their IDs. Specifically, a vertex $u$ ranks higher than a vertex $v$ in two cases: (1) $u, v$ are not in the same tree node, and the tree node of $u$ is an ancestor of $v$; (2) $u$ and $v$ are in the same tree node, and the ID of $u$ is smaller than that of $v$, i.e., $u < v$. Consequently, we have a total rank for all vertices in the common ancestors of two query vertices in the cut tree, and the ranks of all other vertices in the tree are lower. For ease of presentation, we define the ancestor vertices of a vertex $v$, denoted by $A(v)$, as the vertices that rank higher than $v$, i.e., $A(v) = \{u \mid (X(u) = X(v) \wedge u \leq v) \vee X(v)$ is in the subtree of $X(u)\}$. Note that the ancestor vertices of $v$ also include $v$ itself. We define the height $h$ of the cut tree as the maximum number of ancestor vertices any vertex can have, and the width $w$ as the maximum number of vertices any tree node can have. We name our new index CTL-Index, which is formally defined as follows.

**Definition 3.3:** Given a road network $G(V, E)$, the CTL-Index precomputes:

1) a cut tree of all vertices through graph partitioning;
2) the convex shortest path distance from each vertex to all ancestor vertices;
3) the convex shortest path count from each vertex to all ancestor vertices.

**Algorithm 1:** CTL-Query

**Input:** The CTL-Index and two query vertices $s, t$
**Output:** $sd(s,t)$ and $spc(s,t)$

```
1  CA(s,t) ← the common ancestors of s and t;
2  sd ← ∞, spc ← 0;
3  foreach w ∈ CA(s,t) do
4      d' ← csd(s,w) + csd(t,w);
5      if d' < sd then
6          sd ← d';
7          spc ← cspc(s,w) · cspc(t,w);
8      else if d' = sd then
9          spc ← spc + cspc(s,w) · cspc(t,w);
10 return sd, spc
```

**Example 3.3:** Fig. 3 shows the CTL-Index for the road network $G$ in Fig. 1. The first row of label entries represents convex shortest path distance and the second row represents convex shortest path count. Assume that $v_1 < v_2$. For paths between $v_2$ and $v_7$, the path $p_1 = (v_2, v_4, v_9, v_7)$ is a convex path, whereas $p_2 = (v_2, v_1, v_7)$ is not. The convex shortest path distance between $v_2$ and $v_7$ is 6, and its convex shortest path count is 1.

**Lemma 3.3:** *The space complexity of the CTL-Index is bounded by $O(n \cdot h)$.*

**Proof:** *Each vertex stores cut labels relative to all its ancestor vertices. The space required to store labels for each vertex is $O(h)$. Given that there are $n$ vertices, the overall space complexity is $O(n \cdot h)$.*

### C. Query Processing

Algorithm 1 details query processing using the CTL-Index, which efficiently computes the shortest path count by scanning only the CA of two vertices and processing the labels with Eq. (1). In line 1, the algorithm identifies the common ancestors of the two query vertices. Lines 3–9 iterate over each vertex in the set of common ancestors. Specifically, the algorithm updates the shortest path count if a shorter distance is found, or increases the count if the distance equals the current shortest distance.

**Lemma 3.4:** *The time complexity of Algorithm 1 is $O(h)$.*

**Proof:** *The lowest common ancestor (LCA) tree node of two vertices can be determined in constant time using the method proposed in* [8]*, which encodes tree nodes with bit-strings and counts leading zeros via XOR operation. Once the LCA is identified, the algorithm processes labels from the root to the LCA. The size of the CA is bounded by the maximum label size over all vertices, limiting the time complexity to $O(h)$.*

**Example 3.4:** Given the road network $G$ and its CTL-Index in Fig. 3. Considering $Q(v_4, v_9)$, we first compute $CA(v_4, v_9) = \{v_1, v_2, v_3, v_4\}$. For $v_4$, the convex shortest path distance to each vertex in $CA(v_4, v_9) = [4, 1, 4, 0]$ and the associated path count is $[2, 1, 1, 1]$. For $v_9$, the convex shortest path distance to each vertex in $CA(v_4, v_9) = [3, 4, 7, 3]$ and the associated path count is $[1, 1, 1, 1]$. Therefore, $sd(v_4, v_9) = \min(4+3, 1+4, 4+7, 0+3) = 3$ and $spc(v_4, v_9) = 1 \cdot 1 = 1$.

The correctness of Algorithm 1 is mainly guaranteed by Lemma 3.1, and we omit the detailed proof for brevity.

### D. Index Construction

The framework of index construction runs in multiple rounds, and each round consists of three steps. The first step is to identify the cut of the given graph. The second step is to compute the convex shortest path distance/count labels. The third step is to further partition the graph and assign tree nodes if feasible. Algorithm 2 provides the pseudocode for our index construction. Below, we summarize the core approach. The algorithm takes a graph $G$ and a balance factor $\beta$ (default 0.2) as input, with $\beta$ used to determine a balanced vertex cut. In line 1, a vertex cut of $G$ is identified, and the vertices are divided into three parts. Lines 2–4 compute and store the labels for each vertex $u \in V$. The algorithm recursively partitions $L$ and $R$ further if possible (lines 5–10). If $L$ can be further partitioned (line 6), the resulting cut $C_l$ becomes a child node of $C$ (line 7). We apply the same process to further partition $R$ as well (lines 8–10). Upon the termination of Algorithm 2, the tree structure $T_G$ of $G$ will be fully constructed. Each vertex will also store the label entries corresponding to all its ancestors in $T_G$. We describe the cut tree construction and label computations in detail below.

**Constructing the Cut Tree.** The core idea of building the cut tree is to recursively partition a subgraph $G' \subseteq G$ and build a binary tree based on minimal cuts, with each cut represented as a tree node. In each round, the graph is divided into a left part $L$, a cut $C$, and a right part $R$. The cut $C$ becomes the parent tree node of subsequent tree nodes generated by further partitioning $L$ and $R$.

We adopt the Algorithm BalancedCut proposed in [8] to find the cut. We briefly summarize the idea of the Algorithm BalancedCut in line 1 of Algorithm 2, which consists of three steps. First, the algorithm performs an initial rough partitioning based on two selected distant endpoints, creating two partitions that each contain approximately $\beta \cdot |V(G)|$ vertices, with the remaining vertices forming the cut region. Second, two rough partitions are compressed into two supernodes, and the max-flow min-cut algorithm [13] is applied to find the minimal vertex cut. Third, the algorithm balances the partition sizes by assigning the remaining connected components.

**Lemma 3.5:** *The time complexity of the* BalancedCut *algorithm is $O(m \cdot (\log n + w))$* [8].

**Label Computation.** Given a cut tree $T_G$, each vertex maintains label entries for its ancestor vertices in $T_G$. Specifically, we store the convex shortest path distance and convex shortest path count values in labels. In each recursive iteration, lines 2–4 in Algorithm 2 are executed to compute the labels.

After identifying the cut $C$, we run a modified version of Dijkstra's algorithm (Procedure SSSPC) from each vertex $c \in C$ to explore the convex shortest paths. In each iteration, for each

vertex $u$ in the subgraph, we store the labels corresponding to the convex shortest path distance and convex shortest path count from $u$ to the vertices in $C$ (line 3). We should stress that the vertices in $C$ are ordered in descending order according to the vertex ID before line 2. After computing the labels, line 4 removes $c$ from $V$. This ensures that vertices with higher ranks are excluded from the graph in subsequent iterations when executing line 3, preventing redundant use of cut vertices and ensuring each path is covered exactly once.

The Procedure SSSPC in Algorithm 2 describes the details of the path exploration. Lines 13–14 initializes two arrays $D[\cdot]$ and $PC[\cdot]$ for storing convex shortest path distance and convex shortest path count. This procedure uses a priority queue (min-heap) to manage the exploration order of vertices (line 15). Lines 17–26 traverse the graph to compute shortest convex path counts. If a shorter path to a vertex $w$ via a neighbor $v$ is found, we update the convex shortest path distance to $w$ and set the path count to $w$ to be equal to the path count for $v$ (lines 21–23). The vertex $w$ is then enqueued into $Q$ for further path exploration (line 24). If the newly found convex shortest path distance equals the current shortest distance, we update the path count to $w$ by adding the convex shortest path count to $v$ (lines 25–26).

**Lemma 3.6:** *The time complexity of label computation (lines 2–4 in Algorithm 2) is* $O(w \cdot m \cdot \log n)$.

## IV. Improving Query Efficiency

In this section, we aim to further improve query efficiency. In the query algorithm presented in Section III-C, we scan all common ancestors of the two query vertices, and the query performance is determined by the number of these common ancestors. The rationale is that the set of common ancestors is a cut for the query vertices, which means that any path between them must pass through at least one common ancestor. When two query vertices are close to each other, they may share many common ancestors, which slows down CTL-Index's efficiency. To improve query efficiency, we refine our index structure to identify a smaller vertex cut.

### A. The Improved Index Structure

Our main idea is to construct a new tree structure where the vertices in the lowest common ancestor form a vertex cut between two query vertices, eliminating the need to scan all common ancestors. The cut tree structure in CTL-Index is derived by iteratively computing a vertex cut for the remaining graph. To formalize the new tree structure, we replace the vertex cut with a global shortest path (GSP) vertex cut, which is formally defined as follows.

**Definition 4.1:** (GSP Cut) Given a graph $G(V, E)$, a vertex set $C \subset V$ is a GSP cut if there exist two disjoint vertex sets $L, R$ such that $V = L \cup C \cup R$ and any shortest path between a vertex in $L$ and a vertex in $R$ must pass through $C$.

**Definition 4.2:** (GSP-Cut Tree) The GSP-Cut tree is the same as the cut tree, with the additional condition that:

**Algorithm 2:** CTL-Construct

**Input:** A graph $G(V, E)$ and a balance factor $\beta$
**Output:** The CTL-Index for $G$

1 $(L, C, R) \leftarrow \mathsf{BalancedCut}(G, \beta)$;
2 **foreach** $c \in C$ **do**
3 $\quad csd, cspc$ from $c$ to $u \in V \leftarrow \mathtt{SSSPC}(G[V], c)$;
4 $\quad V \leftarrow V \setminus \{c\}$;
5 **if** $|L| > 1$ **then**
6 $\quad C_l \leftarrow \mathsf{CTL\text{-}Construct}(G[L], \beta)$;
7 $\quad$ set $C$ as the parent of $C_l$;
8 **if** $|R| > 1$ **then**
9 $\quad C_r \leftarrow \mathsf{CTL\text{-}Construct}(G[R], \beta)$;
10 $\quad$ set $C$ as the parent of $C_r$;
11 **return** $C$
12 **Procedure** $\mathtt{SSSPC}(G, u)$**:**
13 $\quad$ **foreach** $v \in V(G)$ **do** $D[v] \leftarrow \infty$, $PC[v] \leftarrow 0$;
14 $\quad D[u] \leftarrow 0$, $PC[u] \leftarrow 1$;
15 $\quad Q \leftarrow$ an empty priority queue prioritized by $D[\cdot]$;
16 $\quad Q.enqueue(u)$;
17 $\quad$ **while** $Q$ *is not empty* **do**
18 $\quad\quad v \leftarrow Q.dequeue()$;
19 $\quad\quad$ **foreach** $w \in N(v)$ **do**
20 $\quad\quad\quad d \leftarrow D[v] + \phi(v, w)$;
21 $\quad\quad\quad$ **if** $d < D[w]$ **then**
22 $\quad\quad\quad\quad D[w] \leftarrow d$;
23 $\quad\quad\quad\quad PC[w] \leftarrow PC[v]$;
24 $\quad\quad\quad\quad Q.enqueue(w)$;
25 $\quad\quad\quad$ **else if** $d = D[w]$ **then**
26 $\quad\quad\quad\quad PC[w] \leftarrow PC[w] + PC[v]$;
27 $\quad$ **return** $D[\cdot], PC[\cdot]$

$\forall X \in T_G$, $X$ serves as a GSP cut for $LEFT(X)$ and $RIGHT(X)$ in $G$.

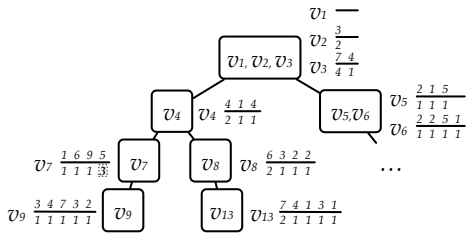



Fig. 4: The CTLS-Index for $G$ (Example 4.1).

**The CTLS-Index.** When the tree is constructed using the GSP cut, any shortest path between two query vertices must pass through a vertex in their lowest common ancestor. This motivates us to define new indexed labels (distance/count values), which enable scanning only the vertices in the LCA of the query vertices during query processing. Based on Lemma 3.1, we compute the convex shortest distance and path count from each vertex to its ancestor, but only apply a ranking method to vertices within the same tree node. In other words,

**Algorithm 3:** CTLS-Query

**Input:** The CTLS-Index and two query vertices $s, t$
**Output:** $sd(s,t)$ and $spc(s,t)$

```
1  LCA(s,t) ← the lowest common ancestor of s and t;
2  sd ← ∞, spc ← 0;
3  foreach w ∈ LCA(s,t) do
4      d' ← scsd(s,w) + scsd(t,w);
5      if d' < sd then
6          sd ← d';
7          spc ← scspc(s,w) · scspc(t,w);
8      else if d' = sd then
9          spc ← spc + scspc(s,w) · scspc(t,w);
10 return sd, spc
```

only vertices within the same tree node as $u$ can rank higher than $u$, while all other vertices rank lower. By contrast, in the previous CTL-Index, all ancestors of $u$ are ranked higher than $u$. For clarity, we refer to the new index as CTLS-Index. Given vertices $u$ and $v$, we refer to the new indexed labels as strong convex shortest path distance and strong convex shortest path count, denoted by $scsd(u,v)$ and $scspc(u,v)$, respectively.

**Example 4.1:** Given a graph $G$ (Fig. 1) and its CTLS-Index (Fig. 4), the set $\{v_4\}$ is a GSP cut for $\{v_8, v_{13}\}$ and $\{v_7, v_9\}$ in $G$, as removing $\{v_4\}$ from $G$ disconnects all shortest paths between $\{v_8, v_{13}\}$ and $\{v_7, v_9\}$. Similarly, the set $\{v_5, v_6\}$ is a GSP cut for $\{v_{12}\}$ and $\{v_{10}, v_{11}\}$. Assume that $v_5 < v_6$, the path $p_1 = (v_5, v_2, v_3, v_{10})$ is a strong convex path, whereas the path $p_2 = (v_6, v_5, v_{10})$ is not a strong convex path.

**Query Processing by CTLS-Index.** Algorithm 3 demonstrates how to answer $Q(s,t)$ using the CTLS-Index. Let $LCA(s,t)$ denote the set of vertices in the lowest common ancestor of the tree nodes for $s$ and $t$. The process in Algorithm 3 is similar to that of Algorithm 1, but only the vertices in $LCA(s,t)$ are processed (line 1). Line 4 compares the strong convex shortest path distance, and the $spc$ value is updated in lines 7 and 9 accordingly. The correctness of Algorithm 3 is guaranteed by Lemma 3.1, so we omit the detailed proof.

**Lemma 4.1:** *The time complexity of Algorithm 3 is $O(w)$, where $w$ is the maximum size of vertices in any tree node.*

**Proof:** *Line 1 of Algorithm 3 takes $O(1)$ time, and the algorithm processes only the vertices in $LCA(s,t)$, which requires $O(w)$ time.*

**Example 4.2:** Given a road network $G$ (Fig. 1), Fig. 4 is the CTLS-Index of $G$. For the query $Q(v_4, v_7)$, $LCA(v_4, v_7) = \{v_4\}$. The shortest distance $sd(v_4, v_7) = \min(0+5) = 5$, and the associated path count $spc(v_4, v_7) = 1 \cdot 3 = 3$.

### *B. Constructing CTLS-Index*

The CTLS-Index is constructed by repeatedly identifying cuts and partitioning the graph, following a process similar to that of CTL-Index construction. The primary challenge lies in computing the GSP cut in each iteration. To address this, we introduce auxiliary edges, referred to as shortcuts, for each subgraph after removing the cut vertices. These shortcuts guarantee that the shortest path distances and counts remain consistent with those in the original graph. With the updated graph and these shortcuts, we can apply the same method to compute the vertex cut, which results in a GSP cut.

For simplicity, we use the notation $(L, C, R)$ to represent a vertex cut $C$ between $L$ and $R$ in a graph $G(V, E)$ where $V = L \cup C \cup R$. We now present the formal definition that preserves both the shortest distance and the shortest path count.

**Definition 4.3:** (SPC-GRAPH) Given a graph $G(V, E)$, a graph $G'(V', E')$ is a shortest path count preserved graph (SPC-Graph) of $G$ if (1) $V' \subseteq V$; and (2) for any pair of vertices $s, t \in V'$, their shortest distance and shortest path count in $G'$ are the same as in $G$.

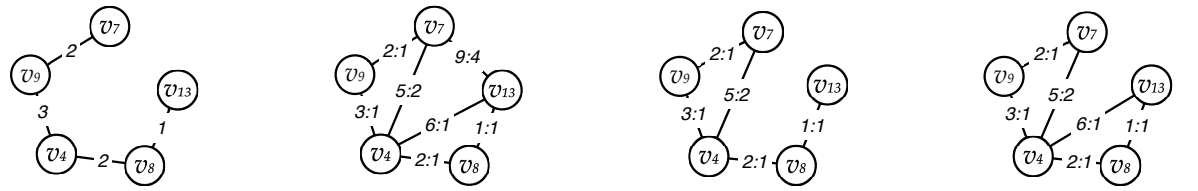
Note that in an SPC-Graph, an additional weight value for each edge, named edge count weight, is introduced to represent the shortest path count between its two terminal vertices. The weight is set to 1 by default and is denoted as $\sigma(s,t)$. The edge count weight may be updated during the index construction. Given the edge count weight, the path count weight of a path $p = (v_0, \ldots, v_k)$ is defined as $\psi(p) = \prod_{i=0}^{k-1} \sigma(v_i, v_{i+1})$. In an SPC-Graph, the total number of shortest paths is the sum of the path count weight of all shortest paths.

**Example 4.3:** Fig. 5a shows the subgraph $G[L]$. $G[L]$ does not preserve shortest path counts within the subgraph. For example, the shortest path count between $v_4$ and $v_7$ is 1 in $G[L]$, whereas it is 3 in $G$. In contrast, Fig. 5b is an SPC-Graph. For any pair of vertices in this graph, both the correct shortest distances and the corresponding path counts can be retrieved.

Assume that the terminals and corresponding weights of all shortcuts to be added to the graph have already been identified. The procedure `addEdge` in Algorithm 4 details how to add each shortcut. We take two terminals $u$ and $v$, along with the distance weight, the count weight, and the current graph $PG$, as input parameters. If the edge $(u, v)$ does not exist or if we find a shorter distance, we create or update it by setting the distance and path count to the new values (lines 10–12). If edge $(u, v)$ already exists with the same distance weight, we increase its count weight by $count$ (lines 13–14). After executing this procedure, the shortcut will be inserted into $PG$ with the correct distance and path count weight. The time complexity of `addEdge` is $O(deg_{max})$, which is the maximum vertex degree. Next, we discuss how to identify the shortcuts.

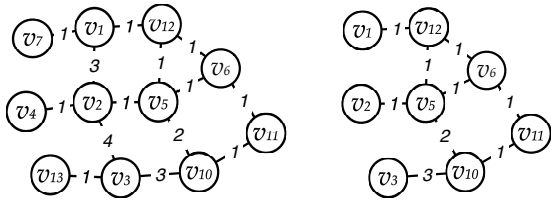
**Shortcut Identification.** Given the cut $C$ between two parts $L$ and $R$. We now discuss how to compute an SPC-Graph including $L$. The same method can be applied to derive the SPC-Graph including $R$. We consider the shortest paths between every pair of vertices in $L$ in the following three types. Given a shortest path $p$ between two vertices in $L$ in the original graph $G$, we say that $p$ is an:

- Inner shortest path if all vertices in $p$ are in $L$;
- Outer shortest path if $p$ contains a vertex not in $L$;
- Outer-Only shortest path if only its terminals are in $L$.



(a) $G[L]$. (b) An SPC-Graph. (c) Using $sd_G(s,t)$. (d) Using $sd_G(s,t,C)$.

Fig. 5: Subgraph of $G$ after removing $C$ and its SPC-Graphs.



(a) Boundary graph of $L$. (b) Boundary graph of $L \cup C$.

Fig. 6: Boundary graphs.

Any shortest path between two vertices in $L$ is either an Inner shortest path or an Outer shortest path. When removing vertices in the cut $C$, it is clear to see that all Inner shortest paths remain unaffected but all Outer shortest paths are disrupted. Therefore, we aim to introduce shortcuts to restore all disrupted Outer shortest paths. To this end, we observe that any Outer shortest path between an arbitrary pair of vertices in $L$ must be a concatenation of a series of Inner shortest paths and Outer-Only shortest paths. Since all Inner shortest paths are unaffected, restoring all Outer-Only shortest paths will preserve the shortest path counts between vertices in $L$.

**Definition 4.4:** (BORDER VERTICES AND BOUNDARY GRAPH) Given a graph $G(V,E)$ and a vertex set $L \subset V$, we define the border vertices of $L$, denoted by $B$, as the set of vertices in $L$ that have direct edges connecting to vertices in $V \setminus L$. We define the boundary graph of $L$, denoted by $BG$, as the subgraph $G \setminus G[L]$ where $G \setminus G[L]$ excludes all edges within $G[L]$ and the resulting isolated vertices.

An Outer-Only shortest path between vertices in $L$ is clearly a shortest path in the boundary graph of $L$. Fig. 6a shows the boundary graph of $L$ derived from $G$ in Fig. 1.

**Lemma 4.2:** *Given a graph $G$ and its $(L,C,R)$, with border vertices $B \subseteq L$ and the boundary graph $BG$ of $L$, let $G^+$ be the graph by applying the following operations to $G[L]$. For each pair of border vertices $u,v \in B$, we add shortcuts to the subgraph $G[L]$ with the following properties:* 1) $\phi(u,v) = sd_{BG}(u,v)$; 2) $\sigma(u,v) = spc_{BG}(u,v)$. *We have that $G+$ is an SPC-Graph of $G$.*

**Example 4.4:** Considering the subgraph of $G$ indcued by $L$, we identify the Outer-Only shortest paths $p_1 = (v_7, v_1, v_2, v_4)$ and $p_2 = (v_7, v_1, v_{12}, v_5, v_2, v_4)$ between $v_7$ and $v_4$, for which shortcuts should be added. Additionally, we find the Outer-Only shortest paths $p_3 = (v_7, v_1, v_2, v_3, v_{13})$, $p_4 = (v_7, v_1, v_{12}, v_5, v_2, v_3, v_{13})$ and $p_4 = (v_4, v_2, v_3, v_{13})$. After adding shortcuts with their corresponding edge values, the resulting graph becomes an SPC-Graph (Fig. 5b).

Based on Lemma 4.2, we count the shortest paths for each pair of border vertices in the boundary graph as shortcuts. Algorithm 4 presents SPC-Graph construction in detail. Line 1 identifies the border vertex set. Line 2 defines the boundary graph of $L$. Lines 3–4 run the `SSSPC` procedure from each border vertex to compute values for the Outer-Only shortest path distance array $oo\_dist$ and the Outer-Only shortest path count array $oo\_cnt$. In lines 5–6, shortcuts are inserted into $G[L]$ using the `addEdge` procedure. The condition $u < v$ uses vertex IDs to ensure that the shortcut is added only once.

We should stress that since graph traversal is performed in an SPC-Graph, the procedure `SSSPC` should be modified to account for the edge count weight. Specifically, we update the `SSSPC` procedure by replacing line 23 in Algorithm 2 with $PC[w] \leftarrow PC[v] \cdot \sigma(v,w)$ and line 26 with $PC[w] \leftarrow PC[w] + PC[v] \cdot \sigma(v,w)$ when updating the path count array. In the subsequent context, we use this updated version of `SSSPC` to perform graph traversing, unless otherwise specified.

**Lemma 4.3:** *The time complexity of Algorithm 4 is $O(w \cdot deg_{max} \cdot (m \log n + w \cdot deg_{max}^2))$.*

**Proof:** *Line 1 takes $O(m)$ to find border vertices. Line 3 iterates over $O(w \cdot deg_{max})$ border vertices. For each border vertex, line 4 runs `SSSPC` in $O(m \log n)$, and lines 5–6 check and add edges. Each `addEdge` costs $O(deg_{max})$ and is performed $O(w \cdot deg_{max})$ times. The total complexity is $O(w \cdot deg_{max} \cdot (m \log n + w \cdot deg_{max}^2))$.*

**The Overall CTLS-Construct.** The construction algorithm for CTLS-Index is the same as Algorithm 2 except the following key differences: 1) Line 3 computes the $scsd$ and $scspc$ using the updated `SSSPC` procedure, 2) We execute Algorithm 4 to construct the SPC-Graph including $L$ and the SPC-Graph including $R$ before line 6 and line 9, respectively, and perform partitioning using the SPC-Graph at those lines.

### C. Accelerating SPC-Graph Construction

We propose two optimizations in this subsection. The first one speeds up the process of computing the shortest path distance/count between border vertices in the boundary graph. The second one prunes certain shortcuts in order to reduce the number of edges in the SPC-Graph.

**Searching From Cut Vertices.** Algorithm 4 could incur significant computational overhead given the potentially large number of border vertices, as we must run the procedure `SSSPC` from each border vertex to count the shortest paths in the boundary graph. To improve the efficiency of computing Outer-Only shortest paths, we observe that the cut vertices $C$

**Algorithm 4:** SPC-Graph

**Input:** $G(V, E), L$
**Output:** SPC-Graph including $L$

```
1  B ← {v ∈ L | (v, w) ∈ E, w ∈ V \ L};
2  BG ← G \ G[L];
3  foreach u ∈ B do
4      oo_dist, oo_cnt ← SSSPC(BG, u);
5      foreach v ∈ B ∧ u < v do
6          addEdge(G[L], u, v, oo_dist(u, v), oo_cnt(u, v));
7  return G[L]
8  Procedure addEdge(PG, u, v, dist, count):
9      if (u, v) ∉ E(PG) ∨ dist < φ(u, v) then
10         E(PG) ← E(PG) ∪ {(u, v)};
11         φ(u, v) ← dist;
12         σ(u, v) ← count;
13     else if dist = φ(u, v) then
14         σ(u, v) ← σ(u, v) + count;
```

are often much fewer than the border vertices. Additionally, we observe that the cut vertices $C$ are border vertices of $L \cup C$. This enables computing the SPC-Graph including $L$ by searching from $C$ rather than from border vertices of $L$. Specifically, our method first derives the SPC-Graph including $L \cup C$ by adding shortcuts between vertices in $C$, then derives the SPC-Graph including $L$ from it. Fig. 6b shows the boundary graph of $L \cup C$.

To compute the SPC-Graph including $L \cup C$, Algorithm 5 uses the same process as in Algorithm 4. In Algorithm 5, lines 4–9 search Outer-Only shortest paths within the boundary graph of $L \cup C$ instead. In line 9, we omit the vertices in $L$ when adding shortcuts, as those are limited to $C$ and do not involve vertices from $L$.

Next, we demonstrate how to derive the SPC-Graph including $L$ from the SPC-Graph including $L \cup C$. Given the smaller size of the cut vertex set, instead of restoring paths from the boundary graph of $L$, we iteratively remove vertices in $C$ from the SPC-Graph including $L \cup C$ and add shortcuts between the border vertices of $L$. Lines 14–19 in Algorithm 5 explain the details. We iterate each cut vertex to process its neighbors (lines 15–16). Each time we check a cut vertex, we connect its neighboring pairs to preserve path information (line 18). Line 19 removes $c \in C$ once its path counts have been restored.

**Example 4.5:** Given the graph $G$ in Fig. 1, we briefly demonstrate how to construct the SPC-Graph including $L$. First, we iterate over the vertices in $C$, searching for paths in the boundary graph of $L \cup C$. For $v_1$, the Outer-Only shortest path distances to $v_2$ and $v_3$ are 3 and 7, respectively. We increase the count weight of the edge $(v_1, v_2)$ by 1, and we add edge $(v_1, v_3)$ with $\phi(v_1, v_3) = 7$ and $\sigma(v_1, v_3) = 2$. We also add shortcuts for $v_2$ and $v_3$. Second, we derive the SPC-Graph including $L$ by removing $C$ from the SPC-Graph including $L \cup C$. We start by removing $v_1$ and adding the shortcut $(v_2, v_7)$ with $\phi(v_2, v_7) = 4$ and $\sigma(v_2, v_7) = 2$. We then remove $v_2$ and $v_3$ sequentially. After removing all vertices in $C$, we obtain the Fig.5b.

**Pruning Shortcuts.** Not all Outer-Only shortest paths need to be restored. For example, the Outer-Only shortest path $p = (v_7, v_1, v_2, v_3, v_{13})$ does not contribute to the construction of the SPC-Graph. Adding unnecessary shortcuts increases the graph density, making further partitioning more difficult. We observe that restoring the paths in Outer-Only shortest paths satisfying a specific condition is sufficient to ensure the correctness of the SPC-Graph construction. The condition is that the path distance in the boundary graph must equal the global shortest distance between the terminals. If the condition does not hold, the shortcut is redundant, since no shortest path in the SPC-Graph passes through it. Fig. 5c shows the pruning of redundant shortcuts in the SPC-Graph from Fig. 5b.

Computing the global shortest distance between border vertex pairs introduces computational overhead, as we must run SSSPC from each border vertex to all others in the input graph. Below, we introduce Lemma 4.4 to relax the distance requirement for the necessary shortcuts.

**Lemma 4.4:** *Given a graph $G$, a partition $(L, C, R)$, $BG$ of $L$ and two vertices $s, t \in L$, we have:*

$$sd_G(s,t) \leq sd_G(s,t,C) \leq sd_{BG}(s,t)$$

*, where $sd_G(s,t,C)$ denotes the shortest distance between $s$ and $t$ along paths passing through at least one vertex in $C$, and $sd_{BG}(s,t)$ is the shortest path distance in $BG$.*

**Proof:** *Given two vertices $s$ and $t$, $sd_G(s,t)$ represents the shortest distance considering three types of paths. $sd_G(s,t,C)$ stores the shortest distance considering only the Outer shortest path type. $sd_{BG}(s,t)$ determines the shortest distance restricted to the Outer-Only shortest path type, which is a subset of Outer shortest path type. This implies that $sd_G(s,t) \leq sd_G(s,t,C) \leq sd_{BG}(s,t)$, as each subsequent distance considers fewer path types.*

According to Lemma 4.4, we can use $sd_G(s,t,C)$ as a distance threshold to prune redundant shortcuts, as it is easier to compute compared to the global shortest distance. Based on Lemma 3.1, we can efficiently obtain $sd_G(s,t,C)$ values using Eq. (1) with precomputed labels (replacing $CA$ with $C$ in Eq. (1)). This is because removing $C$ splits each border vertex into separate connected components, and $C$ is a vertex cut for any pair of border vertices in the boundary graph. When constructing the SPC-Graph including $L$, we prune redundant shortcuts using $sd_G(s,t,C)$. In Algorithm 5, line 1 initializes *out_dist* to store the $sd_G(s,t,C)$ values. Lines 11–13 efficiently obtain these distances using the previously computed labels. When deriving the SPC-Graph including $L$ from the SPC-Graph including $L \cup C$ , line 17 ensures shortcuts are added between $L$'s border vertices only if the precomputed distance value in *out_dist* for a neighboring pair matches the sum of the distance weights of the two connecting edges.

As previously discussed, we improve the shortcut computation by first deriving the SPC-Graph including $L \cup C$. Lemma 4.4 can also be applied to prune redundant shortcuts in

**Algorithm 5:** SPC-Graph*

**Input:** $G(V, E), L, C$
**Output:** SPC-Graph $G[L]$

1 $out\_dist, oo\_dist, oo\_cnt \leftarrow \emptyset$;
2 **foreach** $u, v \in C$ **do**
3 $\quad out\_dist(u, v) \leftarrow \min_{w \in C} scsd(u, w) + scsd(w, v)$;
4 $BG \leftarrow G \setminus G[L \cup C]$;
5 **foreach** $u \in C$ **do**
6 $\quad (oo\_dist, oo\_cnt) \leftarrow \texttt{SSSPC}(BG, u)$;
7 $\quad$ **foreach** $v \in C \wedge u < v$ **do**
8 $\quad\quad$ **if** $oo\_dist(u, v) \neq out\_dist(u, v)$ **continue**;
9 $\quad\quad \texttt{addEdge}(G[C], u, v, oo\_dist(u, v), oo\_cnt(u, v))$;
10 $B \leftarrow \{v \in L \mid (v, c) \in E, c \in C\}$;
11 **foreach** $u \in B$ **do**
12 $\quad$ **foreach** $v \in B \cup C$ **do**
13 $\quad\quad out\_dist(u, v) \leftarrow \min_{w \in C} scsd(u, w) + scsd(w, v)$;
14 $Z \leftarrow L \cup C$;
15 **foreach** $c \in C$ **do**
16 $\quad$ **foreach** $u, v \in N_{G[Z]}(c)$ **do**
17 $\quad\quad$ **if** $out\_dist(u, v) \neq \phi(u, c) + \phi(c, v)$ **continue**;
18 $\quad\quad \texttt{addEdge}(G[Z], u, v, \phi(u, c) + \phi(c, v), \sigma(u, c) \cdot \sigma(c, v))$;
19 $\quad Z \leftarrow Z \setminus \{c\}$;
20 **return** $G[Z]$

that SPC-Graph. Using Lemma 3.1, we compute the distance threshold for vertices in $C$ efficiently. The precomputed labels capture the shortest distance information for vertices in $C$ within the original graph. Thus, $sd_G(s, t, C)$ for $s, t \in C$ directly represents the global shortest distance $sd_G(s, t)$. The distance values are then used to prune redundant shortcuts during the construction of SPC-Graph including $L \cup C$. In Algorithm 5, lines 2–3 use precomputed labels to determine the global shortest distances between vertices in $C$. Line 8 ensures that shortcuts between vertices in $C$ are added only if a global shortest path exists within the boundary graph of $L \cup C$.

**Example 4.6:** Compared to the SPC-Graph in Fig. 5b, Fig. 5c prunes two shortcuts, $(v_7, v_{13})$ and $(v_4, v_{13})$, using the gloabl shortest distance $sd_G(s, t)$. In $G$, the shortest path between $v_7$ and $v_{13}$ is of type Outer-Only shortest path, and the shortest path between $v_4$ and $v_{13}$ is of type Inner shortest path. Fig. 5d shows that using $sd_G(s, t, C)$ cannot prune $(v_4, v_{13})$ because it does not account for the Inner shortest path type.

**The Improved CTLS-Index Construction.** We continue to use the construction framework presented in Algorithm 2 and update line 3 to compute the *scsd* and *scspc*. We execute Algorithm 5 instead to construct the SPC-Graph including $L$ and the SPC-Graph including $R$ before line 6 and line 9, respectively, and perform partitioning using the SPC-Graph.

**Lemma 4.5:** *The time complexity of Algorithm 5 is* $O(w \cdot m \log n + w^3 \cdot deg_{max}^2 + w \cdot deg_{max}^3)$.

**Proof:** *Lines 1–9 derive the SPC-Graph including* $L \cup C$*, which takes* $O(w^3 + w \cdot (m \log n + w \cdot deg_{max}))$ *time. Lines 10–13 compute the distance thresholds, which costs* $O(w \cdot deg_{max} \cdot (w \cdot deg_{max} + w) \cdot w)$*. It simplifies to* $O(w^3 \cdot deg_{max}^2)$*. Lines 14–19 derive the SPC-Graph including* $L$ *by iteratively removing vertices in* $C$*. Line 16 processes* $O(w)$ *vertices, and checks their neighbor pairs. Each* `addEdge` *operation costs* $O(deg_{max})$*, and it is performed* $O(deg_{max}^2)$ *times for each cut vertex. This results in a time complexity of* $O(w \cdot deg_{max}^3)$*. Thus, the overall time complexity is* $O(w \cdot m \log n + w^3 \cdot deg_{max}^2 + w \cdot deg_{max}^3)$.

### D. Extensions

*1) Parallelization:* We briefly discuss the parallelization of CTLS-Index construction. For label computation in lines 2–4 of Algorithm 2, the labels for each cut vertex can be computed concurrently with the help of the ranking function. The ranking function assigns higher ranks to vertices with lower depth. For each cut vertex, we execute Procedure `SSSPC` in parallel and prune higher-ranking vertices during the computation. Additionally, we can parallelize the construction of SPC-Graph including $L$ and $R$. We can use two threads to run Algorithm 5 in parallel to perform SPC-Graph construction and split the workloads.

*2) Dynamic Graphs:* In real road networks, road construction or closure is rare, but road conditions (edge weights) frequently change due to traffic congestion [14], [15]. We focus on handling edge weight updates, and provide some potential ideas to maintain the CTLS-Index. When the weight of an edge $\phi(a, b)$ changes, the shortest path distances/counts of vertices within a certain subgraph may need to be updated. We denote the affected vertex set as $V_{\text{Aff}}$. Vertices $u, v \in V_{\text{Aff}}$ if a new shortest path between them passes through edge $(a, b)$ after its weight decreases, or if a previous shortest path passes through it before its weight increases. We assume $a \prec b$ in the following discussion.

CTLS-Index uses GSP cut as tree nodes. If the edge weight decreases, it might introduce a new shortest path, and the original GSP cut might not cover that path. To check if the vertex cut tree is still valid, we process the ancestor tree node of $a$ from the root. For each tree node $X$, assuming that $LEFT(X)$ contains $(a, b)$, we check whether it introduces a new shortcut for border vertices in $RIGHT(X)$ using Dijkstra's Algorithm. If a new shortcut is created in $RIGHT(X)$, we further check the subtree rooted at $RIGHT(X)$ to see if the nodes remain GSP cuts and if new shortcuts are created within the subtree. We reconstruct the vertex cut tree if a tree node is not a GSP cut anymore. If no new shortcut is created for $RIGHT(X)$, we skip checking for its descendant. For the descendants of $X(a)$, we need to perform the same GSP cut and shortcut checking on both the left child and the right child. We reconstruct the subtree using Algorithm 2 if the tree node is not a valid GSP cut.

Next, we can update distances and path counts on labels. Note that the shortest distance from a vertex $w$ to $a$ or $b$ has been computed. Thus, we skip path exploration between

TABLE I: Statistics of Datasets

| Name | Description | # Vertices | # Edges |
|---|---|---|---|
| PWR | Power Network | 5,300 | 8,271 |
| NY | New York City | 264,346 | 733,846 |
| BAY | San Francisco Bay Area | 321,270 | 800,172 |
| COL | Colorado | 435,666 | 1,057,066 |
| FLA | Florida | 1,070,376 | 2,712,798 |
| NW | Northwest USA | 1,207,945 | 2,840,208 |
| NE | Northeast USA | 1,524,453 | 3,897,636 |
| CAL | California | 1,890,815 | 4,657,742 |
| E | Eastern USA | 3,598,623 | 8,778,114 |
| W | Western USA | 6,262,104 | 15,248,146 |
| CTR | Central USA | 14,081,816 | 34,292,496 |
| USA | United States | 23,947,347 | 58,333,344 |

$w$ and the terminals of the updated edge. For each ancestor of $X(a)$, we directly explore paths that pass through the updated edge via $a$ or $b$ within the subgraph and update our labels if we discover a new shortest path. For any subtree that contains a new shortcut, we adopt a similar approach to reduce graph traversal costs between its cut vertices and the shortcut endpoints.

Increasing the edge weight does not change the tree structure since it does not introduce new shortest paths. However, it may invalidate the label entries. Precomputed labels may cover old paths, which will be distorted by the updated edge. First, we need to identify the $V_{\text{Aff}}$ by running Dijkstra's algorithm from $a$ and $b$ respectively and search for paths passing through $(a, b)$ in the original graph. If the shortest distance is the same as the precomputed distance, we add that vertex into $V_{\text{Aff}}$. Second, for vertices in $V_{\text{Aff}}$, we need to update the labels accordingly. We iterate through the vertices in $V_{\text{Aff}}$ in ascending order of their ranking, compute labels by running Procedure `SSSPC` in the updated graph, and update the label entries.

## V. Experiments

We conduct extensive experiments to evaluate our proposed methods against the state-of-the-art algorithm. All algorithms are implemented in C++ and compiled with g++ 11.4.0 using the -O3 optimization flag. The experiments are conducted on a Linux machine with dual Intel(R) Xeon(R) Gold 6342 2.8 GHz CPUs and 500 GB of RAM.

**Datasets.** We use 12 publicly available datasets, including a power network [16] and 11 real United States road networks [17]. Table I provides statistics.

**Algorithms.** We compare the following algorithms:

- TL-Index: The tree decomposition index structure in [1].
- TL-Query: The query processing algorithm in [1].
- TL-Construct: The TL-Index construction algorithm in [1].
- CTL-Index: Index structure in Definition 3.3.
- CTL-Query: Algorithm 1.
- CTL-Construct: Algorithm 2.
- CTLS-Index: Index structure in Section IV-A.
- CTLS-Query: Algorithm 3.
- CTLS-Construct: Construction algorithm in Section IV-B.
- CTLS$^+$-Construct: CTLS-Construct with shortcut pruning.
- CTLS*-Construct: CTLS-Construct with shortcut pruning and search from cut vertices (Algorithm 5).

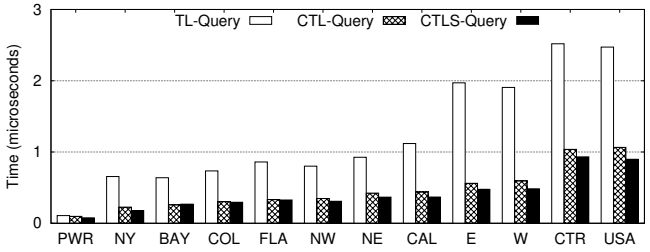



Fig. 7: Average query time.

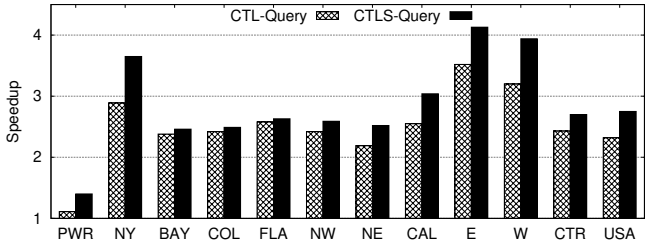



Fig. 8: Speedup over TL-Query.

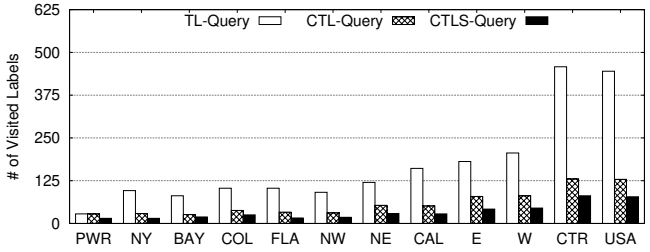
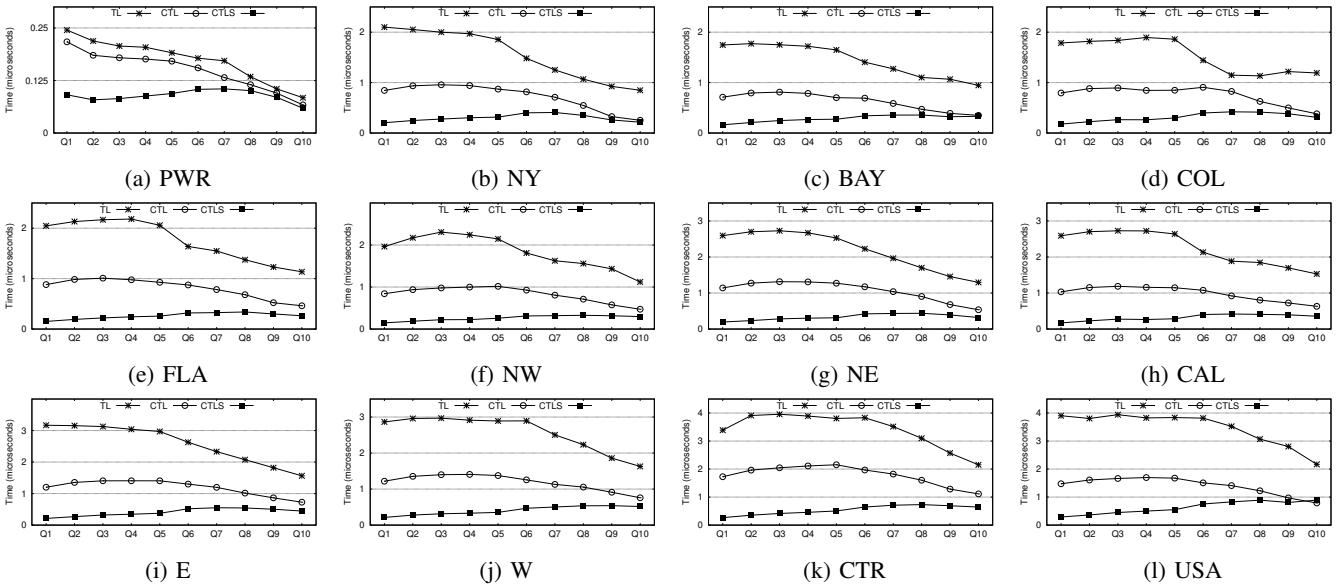



Fig. 9: Number of visited labels in query processing.

We obtained the source code from the authors of [1] for comparison in our experiments. In the experiments, we set the balance factor $\beta$ to 0.20, as recommended in [8] for both the CTL-Index and CTLS-Index. Additionally, each element in the label entry is encoded using 32-bit integer.

**Exp-1: Query Time.** Fig. 7 shows the average query time for TL-Query, CTL-Query, and CTLS-Query over one million random queries across various datasets. The query processing time is measured in microseconds ($\mu s$). The results demonstrate that both of our proposed algorithms outperform TL-Query significantly. For example, in the W dataset, the average running time is 0.60 $\mu s$ for CTL-Query and 0.48 $\mu s$ for CTLS-Query, whereas TL-Query takes 1.91 $\mu s$. CTLS-Query is faster than CTL-Query, as it only scans the LCA of query vertices, which is generally smaller than the CA set. We compare the speedup of our proposed algorithms over TL-Query in Fig. 8. CTL-Query achieves speedups ranging from 1.1 to 3.5 times and CTLS-Query from 1.4 to 4.1 times faster, respectively.




(a) PWR (b) NY (c) BAY (d) COL

(e) FLA (f) NW (g) NE (h) CAL

(i) E (j) W (k) CTR (l) USA

Fig. 10: Query processing time by varying query distance.

**Exp-2: Visited Label Number.** Query time closely depends on the number of labels visited. We report the average number of labels visited in one million random queries across different datasets, as shown in Fig. 9. Both of our proposed algorithms require processing significantly fewer labels than TL-Query. For example, in the NE dataset, the average number of labels visited by CTL-Query and CTLS-Query is 53 and 29, respectively, while TL-Query visits 120 labels on average.

**Exp-3: Varying Query Distance.** To evaluate the query efficiency of different algorithms at varying query distances, we generate ten groups of queries, $Q_1, ...Q_{10}$, for each dataset. Let $x = (l_{max}/l_{min})^{1/10}$, where $l_{min}$ is 1 kilometer and $l_{max}$ is the maximum distance between any pair of vertices in the dataset. For each group $Q_i$ (where $1 \leq i \leq 10$), we randomly generate 10,000 queries with source-target shortest distance in the range $(l_{min} \times x^{i-1}, l_{min} \times x^i)$. Fig. 10 presents the average query processing time for each query set from $Q_1$ to $Q_{10}$.

The results indicate a trend where, for both TL-Query and CTL-Query, query processing times generally decrease as the query distance increases across all datasets. For example, in the CAL dataset, CTL-Query has an average runtime of 1.03 $\mu s$ and 0.63 $\mu s$ on $Q_1$ and $Q_{10}$, respectively, and TL-Query has an average runtime of 2.59 $\mu s$ and 1.53 $\mu s$, respectively. This is because both TL-Query and CTL-Query need to scan all labels from the LCA to the root in the index structure. Intuitively, two distant vertices are more likely to have the LCA with lower depth. An increase in the query distance leads to a smaller CA set to be processed by both algorithms. On the other hand, the average query time of CTLS-Query tends to increase across the ten query sets. In the CTR dataset, CTLS-Query takes 0.27 $\mu s$ on $Q_1$ and increase to 0.64 $\mu s$ on $Q_6$. CTLS-Query significantly outperforms the other algorithms on short-distance query sets, as short-distance queries typically involve smaller vertex cuts.

Overall, our proposed query algorithms consistently outperform TL-Query. Specifically, across different query distances, CTL-Query achieves speedups ranging from 1.1 to 3.36 times and CTLS-Query from 1.2 to 16.4 times faster. For short-distance queries, CTLS-Query is much faster than CTL-Query.

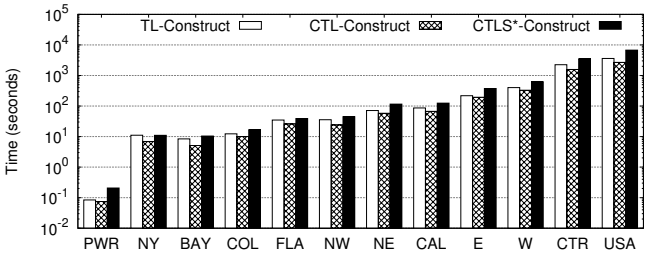



Fig. 11: Index construction time.

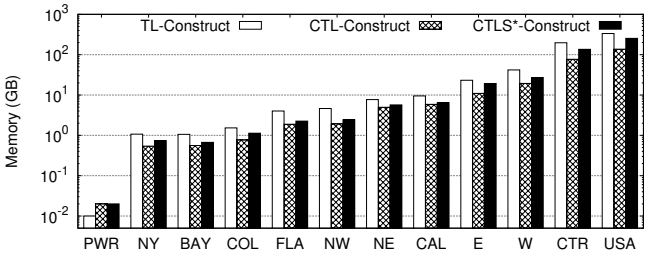



Fig. 12: Memory usage.

**Exp-4: Indexing Time.** This experiment evaluates the index construction time of different algorithms, as shown in Fig. 11. In terms of efficiency, TL-Construct is on average 1.34 times slower than CTL-Construct and 1.52 times faster than CTLS*-Construct. CTLS*-Construct requires additional graph traversal to preserve shortest path counts. We include the memory

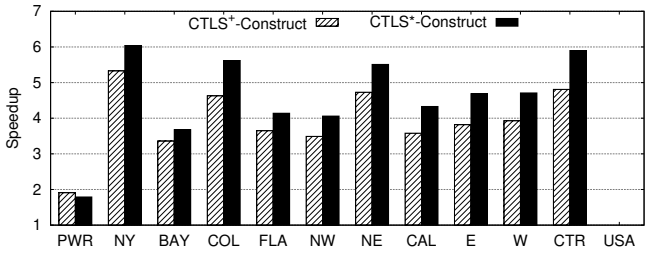



Fig. 13: Speedup over CTLS-Construct.

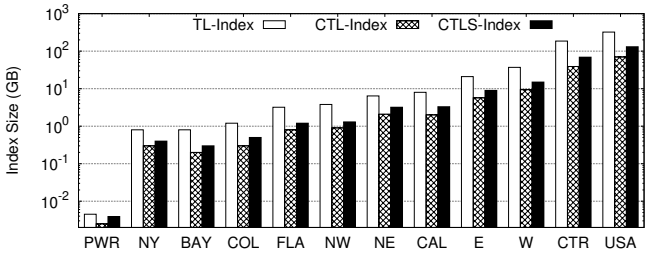



Fig. 14: Index size.

usage during index construction in Fig. 12 to evaluate the scalability on large graphs.

Fig. 13 compares the construction efficiency relative to CTLS-Construct. Note that CTLS-Construct runs out of memory on the USA dataset, hence the speedup result is omitted from Fig. 13. CTLS-Construct requires more time because it includes all shortcuts during the SPC-Graph construction. This leads to a denser graph, making it harder to find balanced vertex cuts for subsequent partitioning. On average, CTLS$^+$-Construct and CTLS*-Construct are 3.4 and 4.6 times faster than CTLS-Construct, respectively. CTLS*-Construct is faster than CTLS$^+$-Construct, mainly due to exploring paths from a smaller vertex set.

**Exp-5: Index Size.** Fig. 14 compares the sizes of different index structures. Both of our proposed index structures are consistently smaller than TL-Index across all datasets. On average, the size of TL-Index is 3.7 times larger than CTL-Index, and 2.35 times larger than CTLS-Index. In particular, the size of CTL-Index is significantly smaller than the size of TL-Index in each dataset, ranging from 1.8 to 4.8 times. This size reduction is attributed to CTL-Index being a more balanced binary tree, which minimizes the number of labels needed for each vertex. In contrast, CTLS-Index consumes more memory than the CTL-Index due to additional shortcut insertions. These insertions complicate the identification of a balanced cut. As a result, the vertex cut becomes larger, leading to an increase in label size.

## VI. Related Work

**Shortest Distance.** Classic algorithms like breath first search and Dijkstra's algorithm [18] do not meet the low-latency requirements. Several studies have explored indexing techniques that apply specific constraints to guide the search process. [19] introduces the concept of vertex reach and integrates it with Dijkstra's algorithm to reduce the search space. The ALT algorithm [20] accelerates A$^*$ search by using a partial distance index. Highway hierarchy [21] and contraction hierarchy [22] leverage the hierarchical structure of road networks to improve search efficiency. Another important category of algorithms in this area is hub labeling [6]. [23], [24] focus on efficient hub labeling techniques for road networks. Pruned landmark labeling (PLL) [25] is a state-of-the-art solution for various types of large graphs. To accelerate index construction, [26] revises PLL and introduces a parallel version. [10], [11] propose integrating hub labeling with vertex hierarchy derived from tree decomposition [7]. This method is further improved by reducing the number of vertices in the labels [27]. HC2L [8] is the state-of-the-art solution for shortest distance queries on road networks. [28], [29] evaluate the performance of various shortest-path query algorithms.

**Shortest Path Counting.** The work by [30] introduces the concept of an exact shortest path cover to ensure the correctness of the hub labeling method and presents a hub-labeling algorithm for the shortest path counting problem. To address scalability issues in this approach, [31] proposes a parallel algorithm. To accommodate machines with limited memory, [32] proposes a size-tunable index-based approach that balances query time and index space. [33] introduces a data structure specifically designed for categorical path counting queries. Several studies have also explored shortest path counting on various types of graphs. To handle frequent updates in dynamic graphs, [34] investigates the maintenance of hub labeling-based index labels. [35] studies planar graphs, and [36] explores probabilistic biological networks.

## VII. Conclusion

We address the problem of counting shortest paths on road networks by introducing two new tree-based indexes.‘ In addition, we introduce several optimizations to enhance index construction efficiency. Extensive experiments on real-world road networks demonstrate the effectiveness of our proposed approaches.

## REFERENCES


[1] Y. Qiu, D. Wen, L. Qin, W. Li, R. Li, and Y. Zhang, "Efficient shortest path counting on large road networks," *Proc. VLDB Endow.*, vol. 15, no. 10, pp. 2098–2110, 2022.

[2] U. Brandes, "A faster algorithm for betweenness centrality*," *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[3] R. Puzis, Y. Elovici, and S. Dolev, "Fast algorithm for successive computation of group betweenness centrality," *Phys. Rev. E*, vol. 76, p. 056709, Nov 2007.

[4] M. Barthelemy, "Betweenness centrality in large complex networks," *The European Physical Journal B - Condensed Matter*, vol. 38, no. 2, p. 163–168, Mar. 2004.

[5] A. Kirkley, H. Barbosa, M. Barthelemy, and G. Ghoshal, "From the betweenness centrality in street networks to structural invariants in random planar graphs," *Nature Communications*, vol. 9, no. 1, Jun. 2018.

[6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM J. Comput.*, vol. 32, no. 5, pp. 1338–1355, 2003.

[7] H. L. Bodlaender, "Treewidth: Characterizations, applications, and computations," in *Graph-Theoretic Concepts in Computer Science, 32nd International Workshop, WG 2006, Bergen, Norway, June 22-24, 2006, Revised Papers*, ser. Lecture Notes in Computer Science, vol. 4271, 2006, pp. 1–14.

[8] M. Farhan, H. Koehler, R. Ohms, and Q. Wang, "Hierarchical cut labelling - scaling up distance queries on road networks," *Proceedings of the ACM on Management of Data*, vol. 1, pp. 1 – 25, 2023.

[9] A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel, "Treewidth: Computational experiments," *Electron. Notes Discret. Math.*, vol. 8, pp. 54–57, 2001.

[10] D. Ouyang, D. Wen, L. Qin, L. Chang, X. Lin, and Y. Zhang, "When hierarchy meets 2-hop-labeling: efficient shortest distance and path queries on road networks," *VLDB J.*, vol. 32, no. 6, pp. 1263–1287, 2023.

[11] D. Ouyang, L. Qin, L. Chang, X. Lin, Y. Zhang, and Q. Zhu, "When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks," in *Proceedings of the 2018 International Conference on Management of Data*, New York, NY, USA, 2018, p. 709–724.

[12] J. Zhang, L. Yuan, W. Li, L. Qin, Y. Zhang, and W. Zhang, "Label-constrained shortest path query processing on road networks," *VLDB J.*, vol. 33, no. 3, pp. 569–593, 2024. [Online]. Available: https://doi.org/10.1007/s00778-023-00825-w

[13] Y. Dinitz, "Dinitz' algorithm: The original version and even's version," in *Theoretical Computer Science, Essays in Memory of Shimon Even*, ser. Lecture Notes in Computer Science, vol. 3895, 2006, pp. 218–240.

[14] D. Ouyang, L. Yuan, L. Qin, L. Chang, Y. Zhang, and X. Lin, "Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees," *Proc. VLDB Endow.*, vol. 13, no. 5, pp. 602–615, 2020.

[15] Y. Zhang and J. X. Yu, "Relative subboundedness of contraction hierarchy and hierarchical 2-hop index in dynamic road networks," in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Z. G. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 1992–2005.

[16] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

[17] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, "The shortest path problem : ninth dimacs implementation challenge," 2009.

[18] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[19] R. J. Gutman, "Reach-based routing: A new approach to shortest path algorithms optimized for road networks," in *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics, New Orleans, LA, USA, January 10, 2004*, 2004, pp. 100–111.

[20] A. V. Goldberg and C. Harrelson, "Computing the shortest path: *A* search meets graph theory," in *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, 2005, pp. 156–165.

[21] P. Sanders and D. Schultes, "Highway hierarchies hasten exact shortest path queries," in *Algorithms - ESA 2005, 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3669, 2005, pp. 568–579.

[22] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, ser. Lecture Notes in Computer Science, vol. 5038, 2008, pp. 319–333.

[23] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck, "A hub-based labeling algorithm for shortest paths in road networks," in *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6630, 2011, pp. 230–241.

[24] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Hierarchical hub labelings for shortest paths," in *Algorithms – ESA 2012*, Berlin, Heidelberg, 2012, pp. 24–35.

[25] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," 2013.

[26] W. Li, M. Qiao, L. Qin, Y. Zhang, L. Chang, and X. Lin, "Scaling distance labeling on small-world networks," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, 2019, pp. 1060–1077.

[27] Z. Chen, A. W. Fu, M. Jiang, E. Lo, and P. Zhang, "P2H: efficient distance querying on road networks by projected vertex separators," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, 2021, pp. 313–325.

[28] Y. Li, L. H. U, M. L. Yiu, and N. M. Kou, "An experimental study on hub labeling based shortest path algorithms," *Proc. VLDB Endow.*, vol. 11, no. 4, pp. 445–457, 2017.

[29] J. Zhang, W. Li, L. Yuan, L. Qin, Y. Zhang, and L. Chang, "Shortest-path queries on complex networks: Experiments, analyses, and improvement," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2640–2652, 2022.

[30] Y. Zhang and J. X. Yu, "Hub labeling for shortest path counting," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, p. 1813–1828.

[31] Y. Peng, J. X. Yu, and S. Wang, "Pspc: Efficient parallel shortest path counting on large-scale graphs," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 896–908.

[32] Y. Wang, L. Yuan, Z. Chen, W. Zhang, X. Lin, and Q. Liu, "Towards efficient shortest path counting on billion-scale graphs," in *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, 2023, pp. 2579–2592.

[33] M. He and S. Kazi, "Data structures for categorical path counting queries," in *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, ser. LIPIcs, vol. 191, 2021, pp. 15:1–15:17.

[34] Q. Feng, Y. Peng, W. Zhang, X. Lin, and Y. Zhang, "DSPC: efficiently answering shortest path counting on dynamic graphs," in *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, 2024, pp. 116–128.

[35] I. Bezáková and A. Searns, "On counting oracles for path problems," in *29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16-19, 2018, Jiaoxi, Yilan, Taiwan*, ser. LIPIcs, vol. 123, 2018, pp. 56:1–56:12.

[36] Y. Ren, A. Ay, and T. Kahveci, "Shortest path counting in probabilistic biological networks," *BMC Bioinform.*, vol. 19, no. 1, pp. 465:1–465:19, 2018.