Efficient Shortest Path Counting on Large Road Networks

Yu-Xuan Qiu
AAII, University of Technology Sydney
Australia
yuxuan.qiu@student.uts.edu.au

Dong Wen
The University of New South Wales
Australia
dong.wen@unsw.edu.au

Lu Qin
AAII, University of Technology Sydney
Australia
lu.qin@uts.edu.au

Wentao Li
AAII, University of Technology Sydney
Australia
wentao.li@uts.edu.au

Rong-Hua Li
Beijing Institute of Technology
China
lironghuabit@126.com

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

ABSTRACT

The shortest path distance and related concepts lay the foundations of many real-world applications in road network analysis. The shortest path count has drawn much research attention in academia, not only as a closeness metric accompanying the shorted distance but also serving as a building block of centrality computation. This paper aims to improve the efficiency of counting the shortest paths between two query vertices on a large road network. We propose a novel index solution by organizing all vertices in a tree structure and propose several optimizations to speed up the index construction. We conduct extensive experiments on 14 real-world networks. Compared with the state-of-the-art solution, we achieve much higher efficiency on both query processing and index construction with a more compact index.

PVLDB Reference Format:
Yu-Xuan Qiu, Dong Wen, Lu Qin, Wentao Li, Rong-Hua Li, and Ying Zhang. Efficient Shortest Path Counting on Large Road Networks. PVLDB, 15(10): 2098 - 2110, 2022.
doi:10.14778/3547305.3547315

1 INTRODUCTION

Given the strong expressive power of the graph model, road maps are often abstracted as graphs, aka road networks, in many real-world location-based services and analytical tasks. In these applications, each road is represented by an edge, and each intersection of roads is represented by a graph vertex. The real distance of each road is modeled as a weight value for each edge in the graph. In analyzing road networks, the concept of the shortest path is important and lays the foundation of many complex location-based queries, like the shortest distance [32], kNN [33] and betweenness centrality [6]. The distance or length of a path is the sum of weights of all edges in the path. A path p is the shortest path if there does not exist a path with the same terminal vertices and a distance value smaller than p. The shortest distance between two vertices is the distance of their shortest path. It is a standard metric to evaluate how close (or similar) the two vertices are.

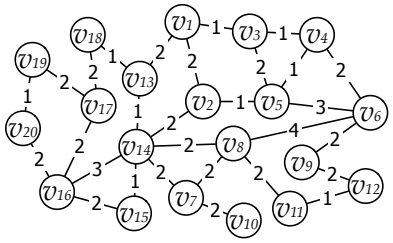


Figure 1: A road network $G(V, E)$.

A great deal of research effort has been contributed to efficiently querying the shortest distance between query vertices in graphs [4, 8, 22, 32, 41, 43]. However, a vertex may reach multiple other vertices with the same shortest distance, which weakens the effectiveness of the shortest distance as a closeness metric. A recent work [42] breaks the tie by formulating the shortest path counting problem, which aims to compute the number of shortest paths between two query vertices in a graph. In this paper, we study the shortest path counting problem on road networks, where the distance is rounded to a specific precision, e.g., meters.

In real road network applications, more shortest paths indicate more traffic options and more flexibility for route planning from the start vertex to the destination. For instance, top-k nearest neighbors search aims at finding k objects close to the query vertex from a candidate set. It is a key operator in taxi-hailing (e.g., Uber), restaurant (e.g., Tripadvisor), and hotel recommendation (e.g., Booking) services. A candidate object can be more desirable than others with the same or similar distance if many shortest paths lead to the object since we have more backup routing plans and a higher probability of avoiding traffic jams. For example, in a movie ticket application, there are two cinemas with the similar shortest distance to the source location. We may prefer the one with more shortest paths considering the traffic options. In addition to serving as a closeness metric, the shortest path count has been used as a building block of betweenness centrality computation [34, 35]. On road networks, the betweenness centrality is widely used as a static predictor of congestion and load, which helps predict the

* Ying Zhang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547315

traffic flow [26]. Given a vertex u, the betweenness centrality of u, denoted by $C_B(u)$, is the fraction of shortest paths passing u, i.e., $C_B(u) = \sum_{s \neq u \neq t \in V} \frac{\text{spc}_u(s,t)}{\text{spc}(s,t)}$, where $\text{spc}(s,t)$ is the number of shortest paths between s and t, and $\text{spc}_u(s,t)$ is the number of paths passing u in $\text{spc}(s,t)$. [35] observes that $\text{spc}_u(s,t) = \text{spc}(s,u) \cdot \text{spc}(u,t)$ if $\text{sd}(s,u) + \text{sd}(u,t) = \text{sd}(s,t)$, where $\text{sd}(s,t)$ is the shortest distance between s and t. Based on this property, several works precompute the shortest distances and the shortest path counts for a set of vertex pairs to approximately compute the betweenness centrality [7, 13, 35, 37]. The techniques in this paper can significantly improve the efficiency of counting shortest paths and boost the efficiency of betweenness centrality computation in practice. Apart from road networks, the proposed method can also be applied to other infrastructure networks, like power grid networks and public transportation networks, which have a small tree height [12, 31].

The State-Of-The-Art Solution. The state-of-the-art algorithm for shortest path counting is proposed in [42]. In [42], the authors devise a labeling-based index by assigning a total order for all vertices. Specifically, for each vertex u, they precompute the shortest distances and the shortest path counts to some vertices with higher ranks than u in the order. To query the number of shortest paths between two vertices u and v, they find every common vertex in the label sets of these two vertices. Each common vertex p acts as a bridge to connect two shortest sub-paths. The sum of two sub-paths' distances is their distance, and the product of two sub-paths' counts is the number of the shortest paths between u and v in terms of p in the index. They sum the counts for all common vertices whose corresponding distances are the shortest between u and v.

Challenges. Their indexing scheme works well as a general method. However, there still exist several challenges and room for improvement. First, based on a total vertex order, a low-ranking vertex may have a large number of labels in the index. More labels imply more index space usage and more comparisons in the query processing. Second, they order the labels for each vertex and perform a merge-sort-like strategy to find common vertices in query processing. Given that the label size for each vertex is not well-bounded, the query strategy needs to access all labels, thus incurring much time overhead. Third, to compute the order-based labels, [42] searches every vertex in the induced subgraph of all vertices with lower ranks. The search space can be the whole graph, which makes the index construction inefficient in large graphs.

Our Approach. In this paper, we propose a new labeling-based index structure that is carefully defined for road networks and other sparse graphs. We adopt the concept of tree decomposition [11] and propose a tree-based labeling structure, given that real-world road networks normally have a low average degree and small treewidth [32, 33]. Specifically, we organize all vertices in the graph into a tree structure such that there is a one-to-one correspondence between the vertices and the tree nodes. By our indexing scheme, we only store a label for each ancestor of a vertex in the tree, which bounds the label size of each vertex well. In query processing, we derive several useful properties which enable us to only consider the common ancestors of two query vertices in the tree. As a result, we only check a small number of labels which significantly improves the query efficiency. Our index is also a labeling-based structure and satisfies the concept of exact shortest path covering defined in [42]. Their hub-pushing-based index construction paradigm can be naturally adapted to construct our tree-based index. To improve the indexing efficiency, we propose a new index construction framework and avoid the costly graph search in [42]. The critical step in index construction is to compute the shortest distance and path count for a vertex u to one of its ancestor v. We propose several rules to reduce all the descendants of u in the tree while preserving the correctness of all shortest paths in the small reduced graph. We compute the result from u to v by utilizing the values derived in previous rounds and avoid searching the graph by only scanning the neighbors of u in the reduced graph.

Contributions. We summarize our main contributions as follows.

- *A novel tree-based index algorithm.* We design a novel index structure called TL-Index. Let n be the number of vertices, h be the treeheight, and w be the treewidth. Our index size is bounded by $O(nh)$, and the query time is bounded by $O(h)$. By contrast, the index size and the query processing time of the state-of-the-art solution are bounded by $O(nw \log n)$ and $O(w \log n)$, respectively [42]. As shown in our experiments (Section 5), h is much smaller than $w \log n$ in real-world graphs. For instance of the New York City map, we have $h = 505$ and $w \log n = 2412$. Therefore, our solution achieves higher query efficiency than the state-of-the-art method with smaller space usage.
- *A new index construction paradigm.* We propose a new paradigm to construct the index and two optimizations to improve efficiency. Compared to the index construction framework proposed in [42], we improve the time complexity of index construction from $O(nh^2 + nh \log n)$ to $O(nhw + n \log n)$, because w is typically several times smaller than h in practice.
- *Extensive experiments and evaluations.* We conduct extensive experiments on 14 real-world networks, including the USA map with 24 million vertices and 58 million edges. The state-of-the-art method cannot finish indexing within 24 hours on the USA map, while our proposed method only takes one hour. On other large real-world maps, our method achieves 20 times faster indexing and seven times faster querying than the state-of-the-art method. The results validate the effectiveness of our index structure and the efficiency of the indexing algorithm.

2 PRELIMINARIES

2.1 Problem Statement

We consider a road network $G = (V, E, \phi)$ which is usually a degree-bounded, connected, and weighted graph. $V(G)$ is a set of vertices, $E(G)$ is a set of edges, and $\phi : e \in E(G) \mapsto N^+$ is a weight function for each edge. We mainly focus on undirected graphs in this paper. Our techniques can be easily extended to directed graphs and other sparse graphs. When it is clear from the context, we use V and E to denote $V(G)$ and $E(G)$, and use $n = |V|$ and $m = |E|$ to denote the numbers of vertices and edges, respectively. We denote all neighbors of a vertex v in G as $N_G(v) = \{u | (u,v) \in E(G)\}$. We use $N(v)$ to denote $N_G(v)$ when the context is obvious. We use $\phi(e)$ to denote the weight of an edge $e \in E$. On road networks, the edge weight may represent the actual length or the travel time of a road segment. A (simple) path[1] $p = (v_1, v_2, \ldots, v_k)$ is a sequence of

[1] In this paper, the term "path" always means a simple path.

distinct vertices where $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$. The length of a path p, denoted by $\phi(p)$, is the sum of weights of all edges on the path, i.e., $\phi(p) = \sum_{i=1}^{k-1} \phi(e(v_i, v_{i+1}))$. Given two vertices s and t, the shortest distance between s and t, denoted by $\mathrm{sd}(s,t)$, is the smallest length of all paths between s and t. A path p between s and t is a shortest path if $\phi(p) = \mathrm{sd}(s,t)$. We denote the set of all the shortest paths between s and t in the graph G by $P_G(s,t)$, and we use $P(s,t)$ when context is obvious. The number of shortest paths is denoted by $\mathrm{spc}(s,t)$, i.e., $\mathrm{spc}(s,t) = |P(s,t)|$.

Problem Definition Given a road network G and two query vertices $q = (s,t)$, the shortest path counting problem aims to efficiently compute the number of shortest paths between s and t.

Example 1. *Figure 1 shows an example of a road network $G(V,E,\phi)$ with 20 vertices and 29 edges. The weight is marked on each edge. Considering vertices v_6 and v_{16}, there are many paths between them, such as $p_1 = (v_6, v_8, v_{14}, v_{16})$ and $p_2 = (v_6, v_4, v_3, v_1, v_{13}, v_{14}, v_{16})$. We have $\phi(p_1) = 9$ and $\phi(p_2) = 10$, and p_2 is not a shortest path. Given that there is no other path p with length less than p_1, p_1 is a shortest path, and the shortest distance between v_6 and v_{16} is 9. There are also 5 other paths with the same length 9 between v_6 and v_{16}. Therefore, the number of shortest paths between v_6 and v_{16} is 6 in G.*

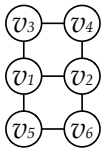
A Basic Online Method. The shortest path count can be straightforwardly derived as a byproduct of the Dijkstra's algorithm in computing the shortest distance. Specifically, we use a queue to maintain all visited vertices with a priority of distance to the source vertex. For each visited vertex during the search, in addition to maintaining the distance from the source vertex, we store the corresponding shortest path count. Given a vertex v, let $D[v]$ and $C[v]$ be intermediate shortest distance and shortest path count, respectively. When exploring v from a neighbor u of v, if a shorter distance (i.e., $D[u] + \phi(u,v) < D[v]$) is found, we replace $D[v]$ and $C[v]$ with $D[u] + \phi(u,v)$ and $C[u]$, respectively. If $D[u] + \phi(u,v) = D[v]$, we add $C[u]$ to $C[v]$. We do not update $C[v]$ and $D[v]$ if $D[u] + \phi(u,v) > D[v]$. The online method works but may suffer from weak scalability in large graphs since all edges in the graph will be scanned in the worst case.

2.2 The State of the Art: Hub Labeling

Zhang and Yu [42] proposed a hub-labeling-based algorithm to count shortest paths efficiently. They first introduce the exact shortest path covering (ESPC) that guarantees to cover all shortest paths without redundancy and then propose a corresponding hub pushing algorithm to build the index. More specifically, for each vertex u, they precompute a collection of labels $L(u)$, and each label is a triplet $(w, \mathrm{sd}(u,w), \delta_{u,w})$, where $\mathrm{sd}(u,w)$ is the shortest distance between u and w, and $\delta_{u,w}$ is the number of a subset of shortest paths between u and w in the graph (i.e., $\delta_{u,w} \leq \mathrm{spc}(u,w)$). Given two query vertices u and v, the shortest path count is computed based on the following equation.

$$\mathrm{spc}(u,v) = \sum_{w \in L(u), w \in L(v), \mathrm{sd}(u,w)+\mathrm{sd}(v,w)=\mathrm{sd}(u,v)} \delta_{u,w} \cdot \delta_{v,w} \quad (1)$$

In Equation (1), the shortest distance $\mathrm{sd}(u,v)$ can be derived by the formula $\min_{w \in L(u), w \in L(v)} \mathrm{sd}(u,w) + \mathrm{sd}(v,w)$.



Vertex	$L(\cdot)$
v_1	$(v_1, 0, 1)$
v_2	$(v_2, 0, 1), (v_1, 1, 1)$
v_3	$(v_3, 0, 1), (v_2, 2, 1), (v_1, 1, 1)$
v_4	$(v_4, 0, 1), (v_3, 1, 1), (v_2, 1, 1), (v_1, 2, 2)$
v_5	$(v_5, 0, 1), (v_2, 2, 1), (v_1, 1, 1)$
v_6	$(v_6, 0, 1), (v_5, 1, 1), (v_2, 1, 1), (v_1, 2, 2)$

Figure 2: A simple graph and its hub-labeling index given the vertex order $v_1 \leq v_2 \leq v_3 \leq v_4 \leq v_5 \leq v_6$.

For index construction, Zhang and Yu [42] assign a total order $\leq$ for all vertices. They process vertices in descending order. For each vertex $w \in V$, they perform a search from w in the induced subgraph of all the vertices whose orders are not higher than w. The shortest distance and shortest path count from w to each vertex v are collected in the search, and a corresponding label is added to $L(v)$. During the search, if a shorter distance between w and v is found based on the existing labels of w and v (i.e., a vertex with a high order than w covers the shortest path of w and v), they prune the search space and stop to search the neighbors of v. The algorithm finishes in n iterations.

Example 2. *We use a simple graph in Figure 2 to illustrate the index structure of [42]. We assume that the total order is $v_1 \leq v_2 \leq v_3 \leq v_4 \leq v_5 \leq v_6$. The labels of all six vertices are presented on the right. Give a pair of query vertices v_4 and v_5, we have the shortest distance $\mathrm{sd}(v_4, v_5) = \min\{\mathrm{sd}(v_4, v_2) + \mathrm{sd}(v_5, v_2), \mathrm{sd}(v_4, v_1) + \mathrm{sd}(v_5, v_1)\} = 3$, and the number of shortest paths $\mathrm{spc}(v_4, v_5) = \delta(v_4, v_2) \cdot \delta(v_5, v_2) + \delta(v_4, v_1) \cdot \delta(v_5, v_1) = 1 \times 1 + 2 \times 1 = 3$.*

Several optimizations are also developed in [42] to reduce the index size. Given that both query efficiency and index size are closely related to the label size, they also analyze the maximum label size for each vertex for several types of graphs. We will compare their theoretical results with our method on road networks in Section 3.2.

2.3 Opportunities

We first analyze the limitations of the state-of-the-art method. In [42], the label size for a vertex can be very large, and we need to scan all the labels of two query vertices in the worst case. To precompute the labels for each vertex, their method scans the induced subgraph of all vertices with lower ranks. When processing the first vertex, the entire graph is scanned, and searching a large graph is costly.

The key to improving the efficiency of counting shortest paths is to reduce the number of label comparisons in query processing. To this end, we organize vertices in a tree structure called tree decomposition [18] and proposed a tree-based index structure. Even though tree decomposition has been used in existing works to compute the shortest distance on road networks, the main technical challenge in our problem is to avoid the redundancy of query results, which is quite different from existing studies.

A straightforward idea to construct our index is to adopt a similar framework in the hub-labeling method, where high-ranking vertices are first processed. We significantly improve the efficiency of index construction by adopting a reverse framework. We first process low-ranking vertices so that all intermediate information can be fully utilized when high-ranking vertices are processed. We propose graph reduction techniques to guarantee the correctness

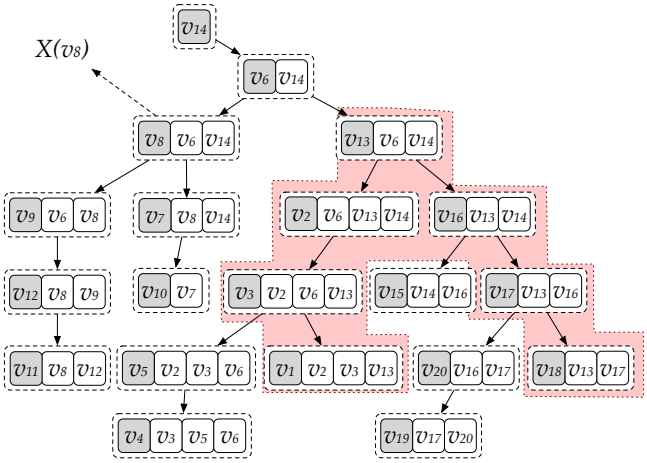


Figure 3: Tree decomposition T_G of G.

of our new computing framework. We also propose a rule to relax the index definition, which reduces the computational cost but still guarantees correctness.

3 TREE-BASED SHORTEST PATH COUNTING

We propose a new labeling-based index carefully defined based on a tree structure of all vertices in the graph. Compared to the state of the art, our solution achieves higher efficiency for both query processing and index construction with a more compact index.

3.1 Tree Decomposition

Tree Decomposition has been used in many applications to speed up certain graph computational problems [18]. We give the formal definition as follows.

Definition 1. *(Tree Decomposition) Given a graph $G(V, E)$, a tree decomposition of G, denoted as T_G, is a tree in which every tree node $X \in T_G$ is a subset of V (i.e., $X \subseteq V$) such that the following conditions hold:*

(1) $\bigcup_{X \in T_G} X = V$;
(2) for every $(u, v) \in E$, there exists $X \in T_G$ such that $u \in X$ and $v \in X$;
(3) for every $u \in V$, $\{X | u \in X\}$ forms a connected subtree of T_G.

Definition 2. *(Treewidth and Treeheight) Given a tree decomposition T_G of a graph G, the* treewidth *of T_G, denoted by $w(T_G)$, is one less than the maximum cardinality of all nodes in T_G, i.e., $w(T_G) = \max_{X \in T_G} |X| - 1$. The* treeheight*, denoted by $h(T_G)$, is the maximum depth of all nodes in T_G where the depth of a node X is the distance from X to the root node in T_G.*

We use w and h to denote treewidth and treeheight, respectively, when it is clear from the context. It is important to note that we can derive a tree decomposition of a road network with low treewidth and low treeheight values. For example, on the road network of New York City with 264,346 vertices and 733,846 edges, we can construct a tree decomposition with a treeheight of 505 and a treewidth of 134. Detailed statistics for other datasets can be found in Table 1.

Tree Decomposition Construction. Given a graph G, there could be multiple tree decompositions, and it is NP-Complete to determine the minimized treewidth of all tree decompositions of G [5]. In this paper, we adopt a sub-optimal tree decomposition method proposed in [27] with a time complexity of $O(n \cdot (w^2 + \log n))$. The method is relatively efficient in practice and has been used in several research works on road networks [16, 32, 33]. It processes each vertex in a greedy way. Specifically, in each iteration, the algorithm picks the vertex v with the smallest degree and creates a corresponding tree node $X(v)$ with all neighbors of v in the graph. Then, it removes v and updates the graph by adding an edge between every pair of unconnected neighbors of v. Assume u is the first removed neighbor of v after removing v, we set $X(u)$ as the parent of $X(v)$ in the tree decomposition. The algorithm terminates after removing all vertices, and we get the tree decomposition of the given graph.

It is straightforward to see that the derived tree decomposition contains n nodes, and there is a one-to-one correspondence from graph vertices to tree nodes. In the rest, we assume this property holds, and tree decomposition is computed using the method in [27].

We always refer to each $v \in V$ in graphs as a vertex and refer to each $X \in T_G$ as a (tree) node. We use the vertex v instead of the tree node $X(v)$ for simplicity when it is clear from the context. The depth of a vertex v, denoted by $\mathsf{Depth}(v)$, is the number of edges from the tree node $X(v)$ to the root node. The ancestor set of a vertex v, denoted by $\mathsf{A}(v)$ is the set of vertices u such that $X(u)$ is an ancestor of $X(v)$ in the tree decomposition. The subtree set of a vertex v, denoted by $\mathsf{T}(v)$, is the set of vertices u such that $X(u)$ is in the subtree rooted by $X(v)$ in the tree decomposition.

Example 3. *Figure 3 shows a tree decomposition T_G for the road network in Figure 1. To construct such a tree decomposition, we first pick the vertex with the lowest degree. Suppose we pick v_{19} and create a tree node $X(v_{19})$ with its neighbors v_{20} and v_{17}. We then remove v_{19} and add an edge between v_{17} and v_{20}. We repeat the above process until all the vertices are removed. Assume v_{20} is the first removed neighbor of v_{19}, we set $X(v_{20})$ as the parent of $X(v_{19})$. We repeat the process and get a tree with 20 tree nodes. The corresponding tree node of the vertex v_3 is $X(v_3) = \{v_3, v_2, v_6, v_{13}\}$. The ancestors of v_3 are $\mathsf{A}(v_3) = \{v_{14}, v_6, v_{13}, v_2\}$, and the subtree set of v_3 is $\mathsf{T}(v_3) = \{v_3, v_5, v_1, v_4\}$. For the vertex v_{13}, all tree nodes containing v_{13} form a connected subtree and are marked in the red area in Figure 3.*

3.2 TL-Index

We propose a new index structure, named TL-Index, to count shortest paths based on tree decomposition. Compared to the state-of-the-art hub-labeling-based index, the only similar part in TL-Index is, conceptually, to store a set of labeling vertices with corresponding distance and count values to each vertex v. We carefully pick the labeling values by utilizing tree decomposition. We organize the labels in a structure that can avoid the merge-sort-like style query mechanism in [27] and achieve higher query efficiency. The details of query processing can be found in Section 3.3. Below, we introduce an important definition which is crucial to guarantee the correctness of the index.

Definition 3. *(Convex Path) Given a tree decomposition T_G of graph $G(V, E)$, a path $p = (s, v_1, v_2, \ldots, v_k, t)$ between two vertices s and t is a convex path if for every $1 \leq i \leq k$, the depth of $X(v_i)$ is larger than the smaller one of $X(s)$ and $X(t)$, i.e., $\forall 1 \leq i \leq k, \mathsf{Depth}(v_i) > \min(\mathsf{Depth}(s), \mathsf{Depth}(t))$.*

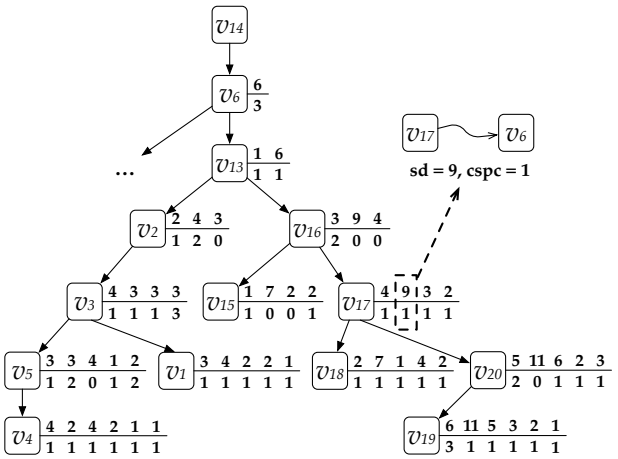


Figure 4: The TL-Index for G.

Example 4. *Given the graph G in Figure 1 and its tree decomposition T_G in Figure 3, the path (v_6, v_8, v_7, v_{10}) is a convex path. This is because* $\text{Depth}(v_7) > \text{Depth}(v_8) > \min(\text{Depth}(v_6), \text{Depth}(v_{10}))$. *The path $(v_6, v_8, v_{14}, v_{16})$ is not a convex path, as* $\text{Depth}(v_{14}) < \min(\text{Depth}(v_6), \text{Depth}(v_{16}))$.

Let $\odot$ be the concatenation of two paths. We provide a support lemma followed by the key theorem motivating our index below.

Lemma 1. *Given a tree decomposition T_G and an arbitrary path p in G, there exists only one vertex in p with the lowest depth.*

Proof. We prove Lemma 1 by contradiction. Given a path p, suppose we have $u, v \in p$, and both u and v have the lowest depth in p. Thus, $T(u)$ and $T(v)$ must be two disjoint sub-trees in T_G. We denote the sub-path between u and v by $(u, w_1, w_2, \ldots, w_{i-1}, w_i, v)$. We first consider the u side. As $(u, w_1) \in E$, based on Definition 1 (2), we know $\exists X \in T_G, u \in X, w_1 \in X$. Based on Definition 1 (3), $X(u)$ and X should be in the same sub-tree, $X(w_1)$ and X should also be in the same sub-tree, i.e., $X(u)$ and $X(w_1)$ should be in the same sub-tree. As u has the minimum depth, we have $w_1 \in T(u)$. We similarly have $w_2 \in T(u), \ldots, w_i \in T(u)$. On the v side, we also have $w_i \in T(v)$ where the contradiction exists. □

Theorem 1. *Given an arbitrary path $p(v_1, v_2, \ldots, v_k)$, either p is a convex path or there exists one and only one pair of convex paths p_1 and p_2 such that $p = p_1 \odot p_2$.*

Proof. Based on Lemma 1, given a non-convex path p between two vertices s and t, assume that v is the vertex with the smallest depth in p. We have $v \neq s$ and $v \neq t$. Otherwise, p is a convex path. Therefore, p can be divided to two sub-paths from v, and both sub-paths are convex paths. □

Based on Theorem 1, each shortest path is either a convex shortest path or a concatenation of two convex shortest paths, given that any sub-path of a shortest path is also a shortest path. Here, a convex shortest path is a convex path that has the same length as the shortest path between two terminal vertices in the graph. To compute the shortest path count for any pair of vertices, our idea is to precompute the distance and the count of all convex shortest paths between each possible pair of vertices. Note that the number of convex shortest paths is significantly smaller than that of all shortest paths, which is supported by the following lemma.

Lemma 2. *Given a tree decomposition T_G of a graph G and an arbitrary path p, let v be the vertex with the smallest depth in p. v is the ancestor of all other vertices in p.*

The shortest paths between two vertices s and t can be divided into two types. The first is the convex shortest path between s and t, and the other is the concatenation of two convex shortest paths from s and t, respectively. It is easy to see that for a non-convex shortest path p, two convex sub-paths join at the vertex with the smallest depth in p. Therefore, our index stores the count of each precomputed convex shortest path as a label of the terminal vertex with a larger depth. We formally define the index as follows.

Definition 4. *Given a road network G,TL-Index precomputes:*

(1) a tree structure of all vertices by tree decomposition;
(2) the shortest distance from each vertex to all its ancestors;
(3) the convex shortest path count from each vertex to all its ancestors.

Note that by a tree structure in Definition 4, we discard all vertices except v in each tree node $X(v)$ and use v as a tree node instead of the original vertex set $X(v)$.

Example 5. *We show the TL-Index for the road network G (Figure 1) in Figure 4 based on the tree decomposition T_G in Figure 3. For simplicity, we only show a part of the index. The index is based on the tree structure of the tree decomposition. For each vertex (index tree node), we store the shortest distance and the corresponding convex shortest path count to each ancestor. The label for the ancestor close to the root is arranged in the front. We take the vertex v_{17} as an example. As we mark in Figure 4, the shortest distance between v_{17} and v_6 is 9, and there is only 1 convex shortest path. The shortest distance between v_{17} and v_{14} is 4, and its corresponding convex shortest path count is 1. Note that in the labels of v_{20}, the count value for v_6 is 0 since there is no convex path between them whose length is less than or equal to 11.*

Theorem 2. *The space complexity of TL-Index is $O(n \cdot h)$.*

Zhang and Yu [42] also analyze the space usage of their proposed hub-labeling index for graphs with small treewidth. They show that their index size can be bounded by $O(wn \log n)$ if the vertex order is carefully arranged, where w is the treewidth, and n is the number of vertices. Our index size is much smaller given that h is much smaller than $w \log n$. The statistics (e.g., w, h and n) of each dataset evaluated in experiments is provided in Table 1. We also compare the practical index size in Figure 12.

3.3 Query Processing with TL-Index

We propose the query processing algorithm in this section. Our TL-Index is also a labeling-based index that satisfies the criteria of exact shortest path covering (ESPC) defined in [42]. Given two query vertices s and t, the general idea is to identify all common labeling vertices as shown in Equation (1). We utilize the tree structure to bound the common labels of query vertices. In this way, we avoid the merge-sort-like strategy to process labels of two query vertices

Algorithm 1: TL-Query

Input: the TL-Index and two query vertices s, t
Output: the shortest distance $\mathrm{sd}(s,t)$, and corresponding count $\mathrm{spc}(s,t)$

1 $v \leftarrow$ the LCA of s and t in the tree;
2 $d \leftarrow \infty, c \leftarrow 0$;
3 **foreach** $u \in \mathsf{A}(v) \cup \{v\}$ **do**
4 $d' \leftarrow \mathrm{sd}(s,u) + \mathrm{sd}(u,t)$;
5 **if** $d' < d$ **then**
6 $d \leftarrow d'$;
7 $c \leftarrow \mathrm{cspc}_{s,u} \cdot \mathrm{cspc}_{u,t}$;
8 **else if** $d' = d$ **then**
9 $c \leftarrow c + \mathrm{cspc}_{s,u} \cdot \mathrm{cspc}_{u,t}$;
10 **return** d *and* c

in Equation (1) and speed up the query processing by visiting a limited number of common vertices directly. We reduce the visited labels in query processing by the following lemma.

Lemma 3. *Given two query vertices s and t, let $\mathsf{CA}(s,t)$ be the set of vertices v such that $X(v)$ is a common ancestor[2] of $X(s)$ and $X(t)$. For each shortest path p between s and t, let u be the vertex with the lowest depth in p. We have $u \in \mathsf{CA}(s,t)$.*

Based on Lemma 3, the shortest distance path count between s and t can be computed using the following equation.

$$\mathrm{spc}(s,t) = \sum_{v \in \mathsf{CA}(s,t), \mathrm{sd}(s,v)+\mathrm{sd}(v,t)=\mathrm{sd}(s,t)} \mathrm{cspc}_{s,v} \cdot \mathrm{cspc}_{v,t}, \quad (2)$$

where $\mathrm{cspc}_{s,v}$ denotes the number of all convex shortest paths between vertices s and v. The shortest distance between s and t can be computed as follows.

$$\mathrm{sd}(s,t) = \min_{v \in \mathsf{CA}(s,t)} \mathrm{sd}(s,v) + \mathrm{sd}(v,t) \quad (3)$$

We provide the query processing algorithm based on TL-Index in Algorithm 1. We first compute the LCA of s and t in line 1. Then, for each common ancestor of s and t, we initialize the shortest path count if a shorter distance is found (lines 5–7). We increase the existing count if the same distance value is found (lines 8–9).

Theorem 3. *The time complexity of Algorithm 1 is $O(h)$, where h is the* treeheight *of the tree decomposition T_G.*

Proof. In line 1 of Algorithm 1, it takes $O(1)$ time to find the LCA [9]. In line 3, the size of $\mathsf{A}(v) \cup \{v\}$ is bounded by the treeheight h of T_G. □

4 INDEX CONSTRUCTION

We propose algorithms for index construction in this section. Section 4.1 extends the framework in the state of the art and presents a non-trivial indexing algorithm. Sections 4.2, 4.3, 4.4, and 4.5 propose optimizations and our improved algorithm for index construction.

[2]Each tree node is also regarded as an ancestor of itself in Lemma 3.

Algorithm 2: TL-Construct

Input: A road network $G(V, E, \phi)$
Output: The TL-Index of G

1 $T_G \leftarrow \mathsf{TreeDecomposition}(G)$;
2 **foreach** $X(u) \in T_G$ *in a top-down manner* **do**
3 **foreach** $v \in \mathsf{T}(u)$ **do** $D[v] = \infty$;
4 $D[u] = 0, C[u] = 1$;
5 $Q \leftarrow$ an empty queue prioritized by $D[\cdot]$;
6 $Q.enqueue(u)$;
7 **while** Q *is not empty* **do**
8 $v \leftarrow Q.dequeue()$;
9 $d \leftarrow \min_{p \in \mathsf{A}(u)} \mathrm{sd}(u,p) + \mathrm{sd}(p,v)$;
10 **if** $d < D[v]$ **then**
11 $\mathrm{sd}(u,v) \leftarrow d$, $\mathrm{cspc}(u,v) \leftarrow 0$;
12 **continue**
13 **else** $\mathrm{sd}(u,v) \leftarrow D[v], \mathrm{cspc}(u,v) \leftarrow C[v]$;
14 **foreach** $v' \in N(v)$ **do**
15 $nd \leftarrow D[v] + \phi(v,v')$;
16 **if** $D[v'] > nd \wedge \mathrm{Depth}(v') > \mathrm{Depth}(u)$ **then**
17 $D[v'] \leftarrow nd, C[v'] \leftarrow C[v]$;
18 $Q.enqueue(v')$
19 **else if** $D[v'] = nd$ **then**
20 $C[v'] \leftarrow C[v'] + C[v]$;

4.1 Basic Index Construction by Hub Pushing

In [42], the authors propose a hub-pushing algorithm to construct an order-based labeling index. We introduce a non-trivial baseline named TL-Construct by extending their framework.

The pseudocode of TL-Construct is provided in Algorithm 2. Given the tree decomposition, TL-Construct processes vertices in a top-down manner in the tree. The algorithm runs in n rounds. In each round (lines 2–20), we pick the vertex u with the smallest depth in the tree and break the tie by picking an arbitrary one. We search from u with a distance priority. The arrays $D[\cdot]$ and $C[\cdot]$ store the distance and the shortest path count, respectively, from u to each vertex during the search.

The distance and the shortest path count to the source vertex u itself are initialized by 0 and 1 respectively in line 4. In each iteration, we pop the top vertex v from the priority queue in line 8, and v is the nearest unprocessed vertex to u.

Line 9 computes the distance between u and v based on the information computed in earlier rounds. For each common ancestor p of u and v, we compute the distance by using p as a bridging vertex given that the shortest distances from p to u and from p to v have been computed. If the distance based on earlier information is shorter than the current distance, we store the shorter distance and terminate further exploration from v in lines 10–12. Otherwise, we store the shortest distance $\mathrm{sd}(u,v)$ and the convex shortest path count $\mathrm{cspc}(u,v)$ for the TL-Index.

We extend the search space from v to each neighbor of v. In line 16, $D[v'] > nd$ means we find shorter distance from u to v'. In this case, we replace the existing distance and the convex shortest path count to v'. Note that $D[v']$ can be ∞ if v' is not visited in the search. If the distance is the same as that computed in earlier iterations

Algorithm 3: Operator $\ominus$

Input: A road network G and a vertex $u \in V(G)$
Output: The graph $G \ominus u$

```
1 foreach v, w ∈ N(u) do
2     if (v, w) ∉ E ∨ φ(v, w) > φ(v, u) + φ(u, w) then
3         E ← E ∪ {(v, w)};
4         φ(v, w) ← φ(v, u) + φ(u, w);
5         ς(v, w) ← ς(v, u) × ς(u, w);
6     else if φ(v, w) = φ(v, u) + φ(u, w) then
7         ς(v, w) ← ς(v, w) + ς(v, u) × ς(u, w);
8 remove u and all incident edges from G;
```

(line 19), we increase the number of convex shortest paths in line 20. Lines 16 and 19 also guarantee that we only process vertices with a larger depth than the source vertex u.

Example 6. *We show a running example for Algorithm 2. Given a graph G (Figure 1) and its tree decomposition T_G (Figure 3), let us consider the shortest distance and the shortest path count between the vertices v_{14} and v_{16}. Algorithm 2 starts from v_{14} and adds all its neighbors into Q in lines 14–18. It also updates $D[\cdot]$ and $C[\cdot]$ in line 17. For instance, we have $D[v_{15}] = 1$, $C[v_{15}] = 1$, $D[v_{16}] = 3$, and $C[v_{16}] = 1$ after scanning all the neighbors of v_{14}. In the next iteration, suppose Q pops v_{15} in line 8, and Algorithm 2 expands its neighbor, i.e., v_{16} in line 15-20. We have $nd = D[v_{15}] + \phi(v_{16}, v_{15}) = 1 + 2 = 3$ which equals $D[v_{16}]$. Thus, we add $C[v_{16}]$ and $C[v_{15}]$ in line 20. When Q pops v_{16}, we store $D[v_{16}]$ and $C[v_{16}]$ to $\mathrm{sd}(v_{14}, v_{16})$ and $\mathrm{cspc}(v_{14}, v_{16})$, respectively. When Q is empty in line 7, we have computed the shortest distance and the shortest path count from v_{14} to each other vertex. The next vertex is v_6 in line 2. Consider a case that $u = v_6$ when Q pops v_{16} in line 8. We find $D[v_{16}] = 11$ which is greater than $\mathrm{sd}(v_6, v_{14}) + \mathrm{sd}(v_{14}, v_{16}) = 9$ (lines 9-10). Therefore, we set $\mathrm{sd}(v_6, v_{16}) = 9$, $\mathrm{cspc}(v_6, v_{16}) = 0$ (line 11) and terminate the exploration in line 12.*

Theorem 4. *The time complexity of Algorithm 2 is $O(nh^2 + nh \log n)$.*

Proof. For each vertex $u \in G$, we only visit the vertices $v \in T(u)$. Thus, we visit $O(n \cdot h)$ times in total. In each visit, we query $O(h)$ times in line 9. The maintenance of the priority queue Q costs $O(h \cdot m + h \cdot n \log n)$ using Fibonacci heap. Thus, the time complexity of Algorithm 2 is $O(nh^2 + nh \log n)$. □

4.2 A New Upward Computing Framework

The index construction algorithm proposed in Section 4.1 essentially computes a distance value and a count value for each pair of vertices with an ancestor-descendant relationship. For each vertex u in each round, Algorithm 2 performs a priority-queue-based search for all vertices in the subtree rooted from u and computes values between u and all its descendants. However, the search space can be the whole graph in the worse case. In addition, for each visited vertex v, the algorithm scans all the ancestors of u to check if a shorter distance value exists, which incurs significant extra cost.

To improve the efficiency of index construction, we propose a novel index construction algorithm, which is called TL-Construct*. TL-Construct* adopts an upward computing framework. Specifically, for each vertex u in each round, we compute the distance value and the count value between u and all its ancestors in the tree. We propose a graph reduction technique in Section 4.3 to support the correctness of the upward computing framework. We relax the index definition in Section 4.4 to speed up the upward computation of the shortest distance and the shortest path count while guaranteeing the correctness simultaneously. We show the final index construction algorithm in Section 4.5.

4.3 Graph Reduction

Preserving Shortest Path Count. For simplicity, we first introduce the DC-Graph, which is denoted by $G(V, E, \phi, \varsigma)$. Compared with the conventional road network, the DC-Graph contains an additional function ς to assign a count weight for each edge. Given a path p in a DC-Graph, we define the count value of p, denoted by $\varsigma(p)$, as the following equation.

$$\varsigma(p) = \prod_{e \in p} \varsigma(e) \tag{4}$$

Given two vertices u and v in a DC-Graph G', to count the number of shortest paths $\mathrm{spc}_{G'}(u, v)$, we sum the count values of all their shortest paths instead of just calculating the number of paths. The DC-Graph is a generalized version of conventional road networks. Given $\varsigma(e) = 1$ for all edges e, counting paths in a DC-Graph is equivalent to counting those in the corresponding conventional graph with the same vertices and edges. Next, we define the DCP-Graph as follows.

Definition 5. *(DCP-Graph) Given a road network $G(V, E, \phi)$, a DC-Graph $G'(V', E', \phi', \varsigma)$ is a DCP-Graph if $V' \subseteq V$ and for every pair $u, v \in V'$, $\mathrm{sd}_G(u, v) = \mathrm{sd}_{G'}(u, v)$ and $\mathrm{spc}_G(u, v) = \mathrm{spc}_{G'}(u, v)$.*

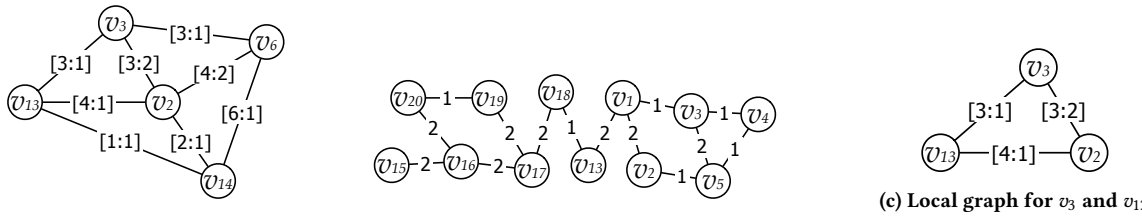
Example 7. *Figure 5 (a) shows a DCP-graph with five vertices from G in Figure 1. All other vertices are reduced. Each edge has two weights namely ϕ and ς, we denote them by $[\phi : \varsigma]$. The reduced graph G' preserves the shortest distance and the shortest path count of all pairs of vertices in G'. For example, we have $\mathrm{sd}_G(v_2, v_6) = \mathrm{sd}_{G'}(v_2, v_6) = 4$ and $\mathrm{spc}_G(v_2, v_6) = \mathrm{spc}_{G'}(v_2, v_6) = 2$. We also have $\mathrm{sd}_G(v_{14}, v_6) = \mathrm{sd}_{G'}(v_{14}, v_6) = 6$ and $\mathrm{spc}_G(v_{14}, v_6) = \mathrm{spc}_{G'}(v_{14}, v_6) = 3$.*

Based on Definition 5, we propose a reduction operation for each vertex in a graph G and transform G to a DCP-Graph as shown in Algorithm 3. Note that for any edge e whose count weight $\varsigma(e)$ is not defined, we initialize it as 1 in the algorithm. For every pair of neighbors v, w of u in G, if 1) the edge (v, w) does not exist earlier, or 2) the edge (v, w) exists but a shorter edge appears (line 2), we create the edge and replace the edge distance weight and the edge count weight by $\phi(v, u) + \phi(u, w)$ and $\varsigma(v, u) \times \varsigma(u, w)$, respectively. If there is an existing edge (v, w) and the distances are the same (line 6), we increase the corresponding edge count weight in line 7.

Lemma 4. *Algorithm 3 returns a DCP-Graph of G.*

Proof. Let G' be $G \ominus w$. For any vertices $u \in V(G')$ and $v \in V(G')$, we consider the following two cases:

- Case 1: the shortest path from u to v in G does not pass through w. Thus, we know $\mathrm{sd}_G(u, v) = \mathrm{sd}_{G'}(u, v)$ and $\mathrm{spc}_G(u, v) = \mathrm{spc}_{G'}(u, v)$ as the shortest path in G is also the shortest path in G'.



(a) A reduced graph G' of G (b) Local graph for v_3 and v_{13} in the original graph G. (c) Local graph for v_3 and v_{13} in the reduced graph G'.

Figure 5: Examples of optimizations in index construction based on the graph G in Figure 1 and its tree decomposition in Figure 3. For each edge in the subfigures (a) and (c), the label means [the distance weight ϕ: the count weight ς]. For the subfigure (b), the label means the distance of the edge.

Algorithm 4: DCP-TreeDecomposition

Input: $G(V, E, \phi)$
Output: Tree decomposition T_G

1 $T_G \leftarrow \emptyset$
2 **foreach** $e \in E$ **do** $\varsigma(e) = 1$;
3 $V' \leftarrow V, i \leftarrow 1$;
4 **while** $V \neq \emptyset$ **do**
5 $u \leftarrow$ the vertex with the smallest degree in V;
6 $X(u) \leftarrow \{u\} \cup N(u)$;
7 create a tree node $X(u)$ in T_G;
8 $G \leftarrow G \ominus u$;
9 $\pi(u) = i$;
10 $i \leftarrow i + 1$;
11 **foreach** $u \in V'$ **do**
12 **if** $|X(u)| > 1$ **then**
13 $v \leftarrow \arg\min_{v \in X(u) \setminus \{u\}} \pi(v)$;
14 set $X(v)$ be the parent of $X(u)$ in T_G;
15 **return** T_G

- Case 2: the shortest path from u to v in G passes through w. In this case, suppose that the shortest path between u and v is $(u, \ldots, w_i, w, w_j, \ldots, v)$. As Algorithm 3 eliminates w in G', and inserts a new edge (w_i, w_j) with $\phi(w_i, w_j) = \phi(w_i, w) + \phi(w, w_j)$ and $\varsigma(w_i, w_j) = \varsigma(w_i, w) \times \varsigma(w, w_j)$ (if there is no edge $(w_i, w_j) \in G$ or $\phi(w_i, w_j) > \phi(w_i, w) + \phi(w, w_j)$ in G) or $\varsigma(w_i, w_j) \leftarrow \varsigma(w_i, w_j) + \varsigma(w_i, w) \times \varsigma(w, w_j)$ (if there is already $\phi(w_i, w_j) = \phi(w_i, w) + \phi(w, w_j)$ in G).

Hence, both the distance and the corresponding edge count are preserved, and Algorithm 3 returns a DCP-Graph of G. □

Graph Reduction in Tree Decomposition. We can reduce the graph in tree decomposition in a natural way, given that we need to connect every pair of neighbors when eliminating each vertex in tree decomposition. The revised tree decomposition is called DCP-Tree Decomposition and is shown in Algorithm 4. Lines 4–10 iteratively reduce each vertex from the graph. The vertex u is removed from G in line 8. $\pi(\cdot)$ is an array to record the removing order of all vertices. Performing the operator $\ominus$ for all vertices would not increase the time complexity of tree decomposition.

Lemma 5. *The time complexity of Algorithm 4 is $O(n \cdot w^2 + n \log n)$.*

Proof. In Algorithm 4, the dominant cost for each vertex is maintaining and selecting the vertex with the smallest degree in V', which costs $O(n \cdot \log n)$ time. In line 8, the $\ominus$ operator costs $O(w^2)$ time. Thus, the overall time complexity of Algorithm 4 is $O(n \cdot w^2 + n \log n)$. □

4.4 Relaxing Convex Shortest Path

In this section, we focus on the phase of computing convex shortest path count. We simplify the logic of index construction by relaxing the convex shortest path count in the index definition. Given a graph G, its tree decomposition T_G and two vertices u, v with $\mathsf{Depth}(u) \neq \mathsf{Depth}(v)$, the local graph of u and v is the induced subgraph of $\mathsf{T}(u) \cup \mathsf{T}(v)$ in G. That is to say, the local graph contains all the vertices who have tree depths no smaller than both u and v.

Example 8. *Given graph G in Figure 1 and the tree decomposition in Figure 3, the local graph for v_2 and v_{13} in $V(G)$ is shown in Figure 5 (b). As we can see from the figure, all the vertices are from the subtree rooted by v_{13}, given that $\mathsf{T}(v_2) \subset \mathsf{T}(v_{13})$.*

Based on the concept of the local graph, we relax the definition of the convex shortest path as follows.

Definition 6. *(Local Shortest Distance and Local Shortest Path Count) Given a tree decomposition T_G of graph $G(V, E)$, the local shortest distance (resp. shortest path count) between two vertices u and v, denoted by $\mathsf{sd}(u, v)^-$ (resp. $\mathsf{cspc}(u, v)^-$), is the shortest distance (resp. shortest path count) of u and v in their local graph.*

Example 9. *Let us continue Example 8. In the local graph (Figure 5(b)), for vertices v_2 and v_{13}, we can find the shortest path $p_1 = (v_2, v_1, v_{13})$. The shortest distance $\mathsf{sd}(v_2, v_{13})^- = 4$, and the local shortest path count is $\mathsf{cspc}(v_2, v_{13})^- = 1$. By contrast, When we look at the full graph G in Figure 1, we find the shortest path $p_2 = (v_2, v_{14}, v_{13})$ which has a smaller distance than the local shortest path p_1. We can also see that $\mathsf{Depth}(v_{14}) < \min(\mathsf{Depth}(v_2), \mathsf{Depth}(v_{13}))$ in T_G and p_2 is not a convex shortest path. Therefore, the shortest distance between v_2 and v_{13} in the full graph G is $\mathsf{sd}(v_2, v_{13}) = 3$, and the convex shortest path count is $\mathsf{cspc}(v_2, v_{13}) = 0$,*

Lemma 6. *Given a convex shortest path p between two vertices u and v, p is a local shortest path in the local graph of u and v.*

Lemma 7. *Algorithm 1 is correct if we replace the shortest distance and the convex shortest path count in TL-Index (Definition 4) by the local shortest distance and the local shortest path count, respectively.*

Algorithm 5: TL-Construct*

Input: A road network $G(V, E, \phi)$
Output: The TL-Index of G

1 $T_G \leftarrow$ DCP-TreeDecomposition(G);
2 **foreach** $X(u) \in T_G$ *in a top-down manner* **do**
3 **foreach** $v \in \mathsf{A}(u)$ **do**
4 **foreach** $u' \in X(u) \setminus \{u\}$ **do**
5 **if** $\mathsf{Depth}(u') < \mathsf{Depth}(v)$ **then continue**;
6 $d \leftarrow \phi(u, u') + \mathsf{sd}(u', v)^-$;
7 $c \leftarrow \varsigma(u, u') \cdot \mathsf{cspc}(u', v)^-$;
8 **if** $d < \mathsf{sd}(u, v)^-$ **then**
9 $\mathsf{sd}(u, v)^- \leftarrow d$;
10 $\mathsf{cspc}(u, v)^- \leftarrow c$;
11 **else if** $d = \mathsf{sd}(u, v)^-$ **then**
12 $\mathsf{cspc}(u, v)^- \leftarrow \mathsf{cspc}(u, v)^- + c$;

Proof. Given vertices u and v, $w \in \mathsf{CA}(u, v)$. We first consider the shortest paths between u and w. If there is any convex shortest path between u and w, we have $\mathsf{sd}(u, w) = \mathsf{sd}(u, w)^-$ and $\mathsf{cspc}(u, w) = \mathsf{cspc}(u, w)^-$ based on Lemma 6. If there is no convex shortest paths between u and w, we have $\mathsf{sd}(u, w) < \mathsf{sd}(u, w)^-$. Based on Theorem 1, there must be at least one vertex $w' \in \mathsf{A}(w)$ that satisfies $\mathsf{sd}(u, w) = \mathsf{sd}(u, w')^- + \mathsf{sd}(w', w)^-$ and $\mathsf{spc}(u, w) = \sum_{w'} \mathsf{cspc}(u, w')^- \cdot \mathsf{cspc}(w', w)^-$. We have the same result for the shortest paths between w and v. Thus, Algorithm 1 is correct. □

Based on Lemma 7, we compute the local shortest distance and the local shortest path count from each vertex to its ancestors. Compared to computing the exact shortest distance and convex shortest path count, the local values reduce significant search space in the index construction. Recall that based on the graph reduction techniques in Section 4.3, we have a reduced graph preserving the shortest distance and the shortest path count during the index construction. By combing the reduction and the local computation ideas, computing the local shortest distance and local shortest path count is conducted in a local reduced subgraph.

Example 10. *An example to compute values from v_3 to v_{13} is provided in Figure 5 (c). When processing v_3, many other vertices have been reduced as shown in G' of Figure 5 (a). Based on Lemma 7, we only care the induced subgraph of all vertices in G' without a depth smaller than v_{13}. The final search space is shown in Figure 5 (c).*

4.5 The Final Algorithm

As shown in Algorithm 5, we first compute the tree decomposition and the count weight for all new edges (shortcuts) inserted to the graph. Then, for each vertex u, we compute the local shortest distance and the local shortest path count from u to its ancestors in lines 3–12. Specifically, $X(u) \setminus \{u\}$ in line 4 are neighbors of u in the DCP-Graph after performing the reduction operation $\ominus$ for all subtree vertices of u. Note that for every vertex u' in $X(u) \setminus \{u\}$, the values have been computed in earlier rounds. Therefore, we use each neighbor u' to compute the distance and the count from u to each v (line 6 and line 7), since the shortest path from u to v must pass at least one of the neighbors. We do not need to consider the neighbor u' with a smaller depth than the target vertex v in line 5

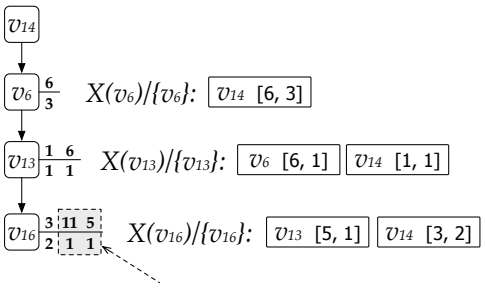


Figure 6: An example of TL-Construct*.

since u' is not in the local graph. During computing distance via neighbors, we replace the shortest distance and shortest path count if a shorter distance is found (lines 8–10). We increase the count if the same shortest distance is found (lines 11–12).

Example 11. *We show a running example with a partial TL-Index in Figure 6. On the right side, we list the distance weight and the corresponding count weight derived by the* DCP-TreeDecomposition *(Algorithm 4). Given $u = v_6$ in line 2, we need to check its ancestor v_{14}. In line 6 and 7, we have $d = \phi(v_6, v_{14}) + \mathsf{sd}(v_{14}, v_{14})^- = 6 + 0 = 6$ and $c = \varsigma(v_6, v_{14}) \cdot \mathsf{cspc}(v_{14}, v_{14})^- = 1 \times 1 = 1$. Given that $\mathsf{sd}(v_6, v_{14})^-$ is initialized as ∞, we update $\mathsf{sd}(v_6, v_{14})^- = 6$ and $\mathsf{cspc}(v_6, v_{14})^- = 1$. For the iteration of $u = v_{16}$ in line 2, when $v = v_{14}$ in line 3, we update $\mathsf{sd}(v_{16}, v_{14})^- = 3$ and $\mathsf{cspc}(v_{16}, v_{14})^- = 2$, which is straightforwardly derived by* DCP-TreeDecomposition. *Then, for the labels from v_{16} to v_6 ($v = v_6$ in line 3), we calculate $d = \phi(v_{16}, v_{13}) + \mathsf{sd}(v_{13}, v_6)^- = 5 + 6 = 11$ and $c = \varsigma(v_{16}, v_{13}) \cdot \mathsf{cspc}(v_{13}, v_6)^- = 1$ in lines 6–7. Then, we update the labels correspondingly in line 9-10. Note that the labels from v_{16} to v_6 and v_{13} differ from those in the TL-index in Figure 4. This is because we only store the local shortest distance and the local shortest path count by our final algorithm.*

Theorem 5. *The time complexity of Algorithm 5 is $O(n \log n + nhw)$, where n is the number of vertices, h is the* treeheight, *and w is the* treewidth.

Proof. In line 1, Algorithm 4 takes $O(n \cdot w^2 + n \log n)$. From line 2 to line 12, there are three loops and the time complexity is $O(nhw)$. The overall time complexity is $O(n \log n + nhw)$.

□

5 EXPERIMENTS

We conduct extensive experiments to evaluate our methods against the state-of-the-art approach. All the algorithms are implemented in C++ with -O3 optimization, and the experiments are conducted on a Linux machine with an Intel Xeon Gold 6248 2.5GHz CPU and 768GB RAM. We evaluate all the algorithms on fourteen real-world graphs as shown in Table 1. GRD is the power grid network of the western states in the USA[3]. SYD is a public transportation network containing all the public transportation stops in Sydney [28]. All the rest are road networks from DIMACS [4].

Compared Algorithms. In our experiments, we compare our algorithms with the state-of-the-art solution HL-Index [42] for the

[3]http://konect.cc/
[4]http://www.diag.uniroma1.it//challenge9/download.shtml

Table 1: Statistics of road networks.

Name	Description	n	m	h	w
GRD	US Power Grid	4,941	6,594	72	25
SYD	Public Transport	24,063	28,695	194	79
NY	NYC	264,346	733,846	505	134
BAY	Bay Area	321,270	800,172	403	108
COL	Colorado	435,666	1,057,066	465	146
FLA	Florida	1,070,376	2,712,798	520	136
NW	Northwest US	1,207,945	2,840,208	548	146
NE	Northeast US	1,524,453	3,897,636	828	219
CAL	CA and NV	1,890,815	4,657,742	713	215
LKS	Great Lakes	2,758,119	6,885,658	1325	370
EUS	Eastern US	3,598,623	8,778,114	1022	272
WUS	Western US	6,262,104	15,248,146	1041	326
CUS	Central US	14,081,816	34,292,496	2433	660
USA	Full US	23,947,347	58,333,344	2564	693

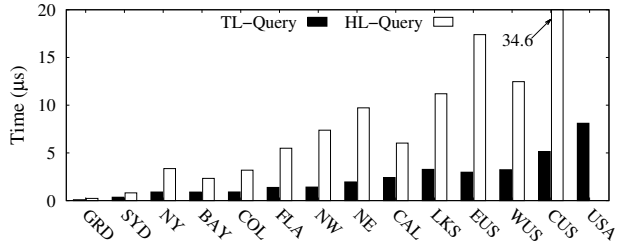


Figure 7: Query time.

shortest path counting query processing on real-world networks. We obtain the C++ code from the authors and revise their index construction algorithm to handle weighted graphs by replacing the Breadth-First Search with the Dijkstra's Search, given that only unweighted graphs are considered in their implementation. We compare the following algorithms in experiments.

- HL-Index: The hub-labeling index in [42].
- HL-Query: The query processing algorithm in [42].
- HL-Construct: The index construction algorithm in [42].
- TL-Index: Our index structure (Definition 4).
- TL-Query: Our query processing algorithm (Algorithm 1).
- TL-Construct: Our basic indexing algorithm (Algorithm 2).
- TL-Construct*: Our optimized indexing algorithm (Algorithm 5).

Note that the hub-labeling method cannot finish indexing the USA dataset within 24 hours, thus we do not report the results.

Exp-1: Query Time. We compare the average query time between TL-Query and HL-Query. For each dataset, we randomly generate one million queries. We record the query time of each algorithm and report the average time in Figure 7. We also show the speedup of TL-Query over HL-Query in Figure 8. As we can see from Figure 7 and Figure 8, TL-Query is significantly faster than HL-Query on all datasets. This is mainly because TL-Query only needs to visit a small number of labels compared with HL-Query in counting shortest paths. For example, in the CUS dataset, TL-Query takes 5.14 µs on average while HL-Query requires 34.59 µs. TL-Query is 6.73 times faster than HL-Query.

Exp-2: Visited Label Size in Query Processing. When processing queries, both TL-Query and HL-Query need to check and compare

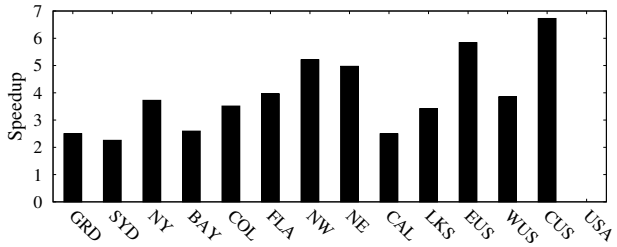


Figure 8: TL-Query **speedup over** HL-Query**.**

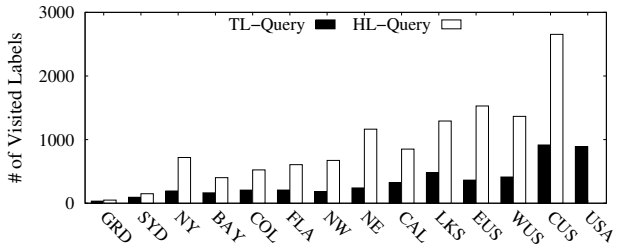


Figure 9: Number of visited labels in query processing.

a number of labels to derive the final result. We evaluate the number of visited labels in query processing in this evaluation. Similar to the query processing time, we report the average value of one million random queries. The results are shown in Figure 9. We can see that the average label size of TL-Query is significantly smaller than HL-Query on all the datasets. For example, on the NE dataset, the average size of TL-Query is 240, and that of HL-Query is 1164. That means, on average, HL-Query needs to visit 1164 labels to finish the query, while TL-Index only needs to visit 240 labels. This result is consistent with our query performance evaluation in Figure 7.

Exp-3: Varying Query Distance. We further test the query efficiency of the algorithms by varying the query distance. We generate ten groups of queries, $Q_1, Q_2, \ldots, Q_{10}$, by distances for each dataset. Specifically, we set l_{min} to be 1,000 (1 kilometer) and l_{max} to be the maximum resulting distance of any pair of vertices in the graph. Let $x = (l_{max}/l_{min})^{1/10}$. A randomly generated query belongs to Q_i if its distance between the source and target vertices falls in the range $(l_{min} \times x^{i-1}, l_{min} \times x^i]$. We randomly generate 10,000 queries for each group and each dataset. We record the average time of TL-Query and HL-Query for the 10,000 queries and report the results by varying query distance from Q_1 to Q_{10} in Figure 10.

We can see that our solution TL-Query outperforms HL-Query in all cases. We also make the following observations. First, when the query distance increases, the time cost of HL-Query increases. For example, on the BAY dataset, HL-Query takes 1.869 µs on average to process Q_1 queries. When handling Q_{10} queries, HL-Query takes 2.256 µs. This is because, in HL-Index, two faraway vertices may have more common labels than two close vertices. Second, in contrast to HL-Query, when the query distance increases, the time cost of TL-Query decreases. On Q_1 queries, TL-Index takes 1.447 µs. While on Q_{10} queries, it only requires 0.776 µs. This is because, in TL-Query, the dominant cost is querying the label from the LCA to the root. Intuitively, when the distance between two vertices is long, their LCA is more likely to have a lower Depth in the tree decomposition. Therefore, our proposed method visits

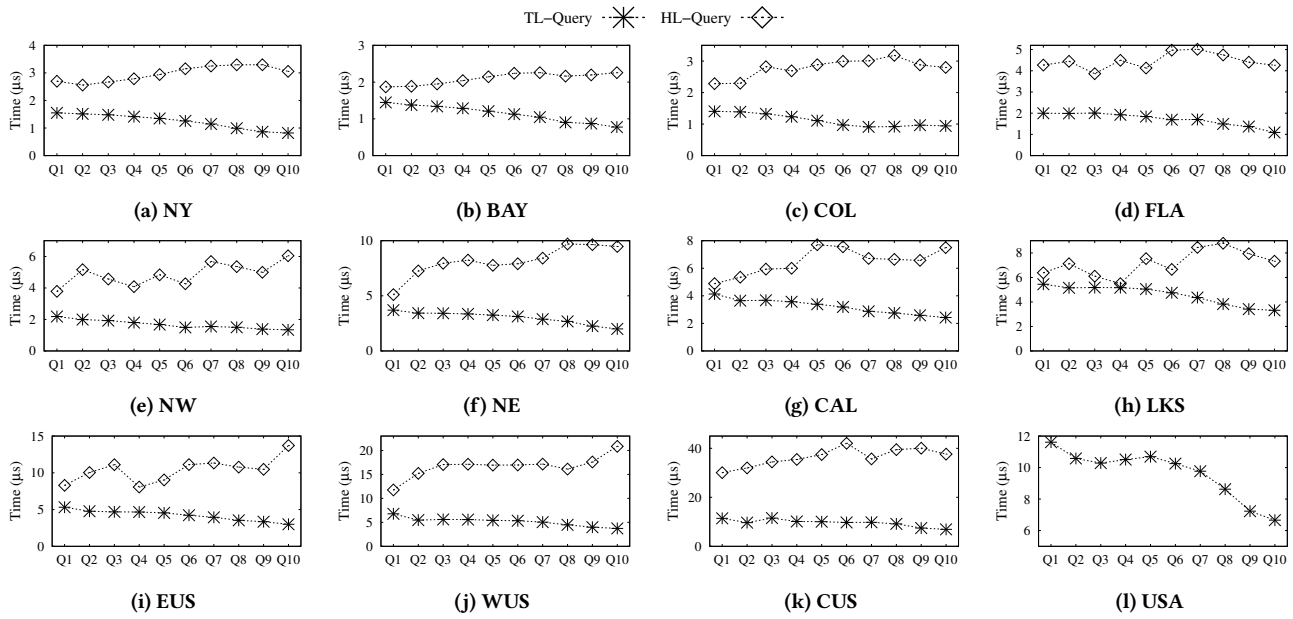


Figure 10: Query processing time varying query distance.

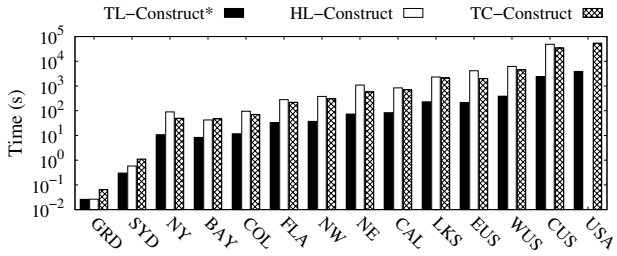


Figure 11: Index construction time.

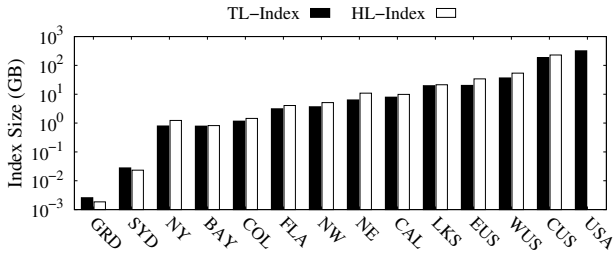


Figure 12: Index size (GB).

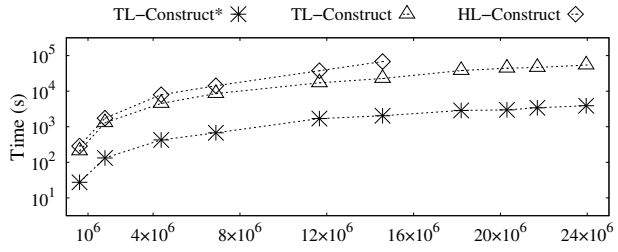
fewer labels when the query distance is longer. Third, when the distance between the source and the target vertices is relatively large, our method TL-Query is significantly faster than HL-Query.

Exp-4: Index Construction. The index construction time for the algorithms HL-Construct, TL-Construct*, and TL-Construct is shown in Figure 11. When the dataset size increases, the indexing time of all the algorithms also increases. Among all three algorithms, our TL-Construct* is the most efficient in indexing. Our solution TL-Construct* is 8–20 times faster than HL-Construct on large road networks. For example, on EUS, WUS, and CUS, our TL-Construct* is 19.41, 15.98, and 20.19 times faster than HL-Construct, respectively. Note that HL-Construct cannot finish indexing USA within 24 hours. Compared to TL-Construct, our proposed TL-Construct* is also several times faster. For example, on EUS, WUS, CUS, and USA, our TL-Construct* is 9.34, 11.61, 14.27, and 13.92 times faster than TL-Construct. Meanwhile, our baseline algorithm is also faster than HL-Construct for most datasets, thanks to the tree structure. The results show the advance of our indexing algorithm.

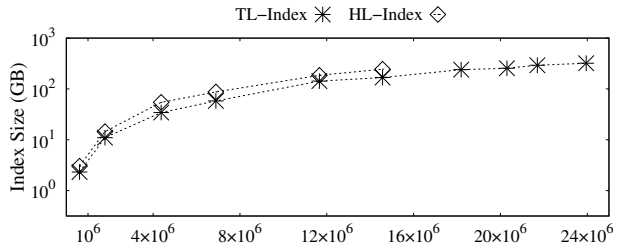
Exp-5: Index Size. We report the index size for the HL-Index and our proposed TL-Index. The results are shown in Figure 12. When the size of the network increases, the size of the index also increases for both algorithms. The index size of our TL-Index is smaller than that of HL-Index on most datasets. Specifically, on most of the datasets, the index size of TL-Index is 20%–40% smaller than that of HL-Index. This is because HL-Index is designed based on a total vertex order. Many labels are precomputed for each vertex.

Exp-6: Indexing Scalability. We test the indexing time and the index size when varying the dataset size (the number of vertices in the road networks) from 10^6 to 24×10^6. We divide the map of the whole US into 10×10 grids and generate ten road networks using the $1 \times 1, 2 \times 2, \ldots, 10 \times 10$ grids in the middle of the map. We denote them as $G_1, G_2, \ldots, G_{10}$, respectively. We report the indexing time and the index size in Figure 13 (a) and Figure 13 (b), respectively. When the dataset increases from 10^6 to 24×10^6, the indexing time increases stably for all algorithms. Our TL-Construct* always outperforms the other two algorithms. HL-Construct cannot finish the datasets whose sizes are greater than 18×10^6 within 24 hours. Therefore, they are not reported in the figure.

Exp-7: The Number of Shortest Paths. We report the average and the maximum shortest path count in Figure 14 and Figure 15. Figure 14 shows that the larger the road network, the greater the shortest paths count. This is because the shortest paths on larger



(a) Indexing time.



(b) Index size.

Figure 13: Scalability testing.

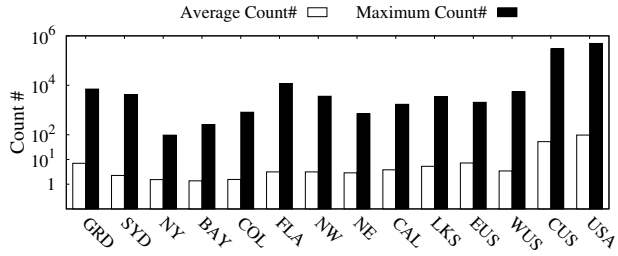


Figure 14: Shortest path count on different graphs.

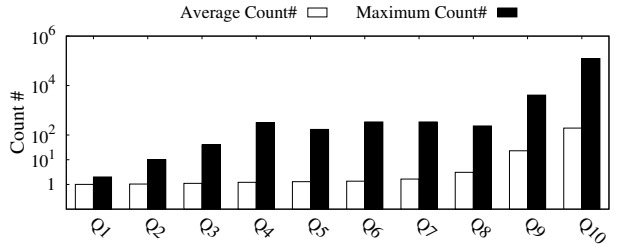


Figure 15: Shortest path count varying distance.

road networks may have more hops, and we are more likely to get the shortest paths with the same shortest distance. Generally, for small networks, like NY, BAY, and COL, the average shortest path count is around 1.5. For medium networks, like FLA and WUS, the average shortest path count is about 3–7. For large networks, like CUS and USA, the average shortest path count is 52 and 97, respectively. Figure 15 shows the average and maximum shortest path count number varying the shortest distance on the USA graph. The results on other graphs show a similar trend. The ten groups of queries are the same as those in Exp-3. The shortest path count number increases with the increase of the shortest distance, which is in accordance with the results above.

6 RELATED WORKS

Shortest Distance Query in Networks. Querying the shortest distance is a critical problem in graph data analysis. The Dijkstra algorithm [19] is one of the most renowned algorithms for this problem. However, for large networks, such online algorithms may be inefficient. Thus, existing research works mainly focus on pre-computing an effective index to accelerate query processing. For example, Goldberg et al. proposed an A* search method accelerated by precomputed shortest distances [23]. Gavoille et al. studied the labeling methods for undirected graphs [21]. Sanders et al. designed the Highway Hierarchies, which imitates the natural hierarchies of road network [38]. Geisberger et al. proposed another hierarchy-based algorithm named Contraction Hierarchies [22] which relies on a pre-assigned total order. Another important class of methods for shortest distance query is hub-labeling-based algorithms [17]. Abraham et al. studied efficient hub-labeling algorithms for road networks [1, 2]. Ouyang et al. combined the advantages of both hub labeling and hierarchy and proposed an H2H labeling scheme for road networks [32]. Chen et al. [16] proposed the P2H method which improves the H2H labeling scheme by reducing the label size. Akiba et al. presented the pruned highway labeling [3] and the pruned landmark labeling [4] for road networks and scale-free networks, respectively.

Network Substructure Counting. In the literature, many research works count small subgraphs like motifs [15, 30] or graphlets [14]. Jain et al. proposed an elegant clique counting algorithm based on classic pivoting techniques [25]. Shi et al. developed a parallel clique counting algorithm [39]. A comparison between different k-clique counting or listing algorithms can be found in [29]. There are also many works study the counting of paths in the literature. Flum et al. proved that counting the cycles and paths of length k in both directed and undirected graphs, parameterized by k, is #W[1]-complete [20]. Valiant proved that the $s-t$ simple path counting problem is #P-complete [40]. Bezakova et al. provided a shortest paths counting query method for planar graphs [10]. Zhang et al. devised a hub-labeling-based method for shortest path counting on large graphs [42]. Ren et al. studied shortest path counting in probabilistic biological networks[36]. He et al. proposed a data structure for categorical path counting queries [24].

7 CONCLUSION

In this paper, we study the problem of counting the shortest paths between two given vertices on large road networks. We propose a novel index structure based on tree decomposition.Our algorithm can achieve $O(h)$ query processing time, and the index size is $O(n \cdot h)$, where n is the number of vertices and h is the treeheight of the tree decomposition of the road network, which is small in practice. We propose an efficient algorithm and several optimizations to speed up the index construction. The experimental results show our algorithm significantly outperforms the state-of-the-art method.

ACKNOWLEDGMENTS

Lu Qin is supported by ARC FT200100787 and DP210101347. Rong-Hua Li is supported by NSFC Grants 62072034 and U1809206. Ying Zhang is supported by ARC FT170100128.

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 *International Symposium 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 *European Symposium on Algorithms*. 24–35.

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

[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*. 349–360.

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

[6] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. 2007. Approximating Betweenness Centrality. In *Algorithms and Models for the Web-Graph*, Vol. 4863. 124–137.

[7] David A. Bader and Kamesh Madduri. 2006. Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks. In *ICPP*. 539–550.

[8] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2016. Route Planning in Transportation Networks. In *Algorithm Engineering - Selected Results and Surveys*. 19–80.

[9] Michael A Bender and Martin Farach-Colton. 2000. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*. 88–94.

[10] Ivona Bezáková and Andrew Searns. 2018. On counting oracles for path problems. In *International Symposium on Algorithms and Computation*.

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

[12] Hans L Bodlaender, John R Gilbert, Hjálmtýr Hafsteinsson, and Ton Kloks. 1991. Approximating treewidth, pathwidth, and minimum elimination tree height. In *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 1–12.

[13] Ulrik Brandes. 2008. On variants of shortest-path betweenness centrality and their generic computation. *Soc. Networks* 30, 2 (2008), 136–145.

[14] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2017. Counting graphlets: Space vs time. In *WSDM*. 557–566.

[15] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. 2019. Motivo: Fast Motif Counting via Succinct Color Coding and Adaptive Sampling. *PVLDB* 12, 11 (2019), 1651–1663.

[16] 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 *SIGMOD*. 313–325.

[17] 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.

[18] Reinhard Diestel. 2016. *Graph theory*. 351–355 pages.

[19] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[20] Jörg Flum and Martin Grohe. 2004. The parameterized complexity of counting problems. *SIAM J. Comput.* 33, 4 (2004), 892–922.

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

[22] 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.

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

[24] Meng He and Serikzhan Kazi. 2021. Data structures for categorical path counting queries. In *32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*.

[25] Shweta Jain and C Seshadhri. 2020. The power of pivoting for exact clique counting. In *WSDM*. 268–276.

[26] Alec Kirkley, Hugo Barbosa, Marc Barthelemy, and Gourab Ghoshal. 2018. From the betweenness centrality in street networks to structural invariants in random planar graphs. *Nature communications* 9, 1 (2018), 1–12.

[27] Arie M. C. A. Koster, Hans L. Bodlaender, and Stan P. M. van Hoesel. 2001. Treewidth: Computational Experiments. *Electron. Notes Discret. Math.* 8 (2001), 54–57.

[28] Rainer Kujala, Christoffer Weckström, Richard K Darst, Miloš N Mladenović, and Jari Saramäki. 2018. A collection of public transport network data sets for 25 cities. *Scientific data* 5, 1 (2018), 1–14.

[29] Ronghua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. 2020. Ordering Heuristics for k-clique Listing. *PVLDB* 13, 11 (2020), 2536–2548.

[30] Chenhao Ma, Reynold Cheng, Laks VS Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. 2019. Linc: a motif counting algorithm for uncertain graphs. *PVLDB* 13, 2 (2019), 155–168.

[31] Silviu Maniu, Pierre Senellart, and Suraj Jog. 2019. An experimental study of the treewidth of real-world graph data. In *ICDT*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[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 *SIGMOD*. 709–724.

[33] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Progressive Top-K Nearest Neighbors Search in Large Road Networks. In *SIGMOD*. 1781–1795.

[34] Matteo Pontecorvi and Vijaya Ramachandran. 2015. A Faster Algorithm for Fully Dynamic Betweenness Centrality. *CoRR* abs/1506.05783 (2015).

[35] Rami Puzis, Yuval Elovici, and Shlomi Dolev. 2007. Fast algorithm for successive computation of group betweenness centrality. *Physical Review E* 76, 5 (2007), 056709.

[36] Yuanfang Ren, Ahmet Ay, and Tamer Kahveci. 2018. Shortest path counting in probabilistic biological networks. *BMC bioinformatics* 19, 1 (2018), 1–19.

[37] Matteo Riondato and Evgenios M. Kornaropoulos. 2014. Fast approximation of betweenness centrality through sampling. In *WSDM*. 413–422.

[38] Peter Sanders and Dominik Schultes. 2005. Highway hierarchies hasten exact shortest path queries. In *European Symposium on Algorithms*. 568–579.

[39] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Parallel clique counting and peeling algorithms. In *SIAM Conference on Applied and Computational Discrete Algorithms*. 135–146.

[40] Leslie G Valiant. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput.* 8, 3 (1979), 410–421.

[41] 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. *PVLDB* 5, 5 (2012), 406–417.

[42] Yikai Zhang and Jeffrey Xu Yu. 2020. Hub Labeling for Shortest Path Counting. In *SIGMOD*. 1813–1828.

[43] Andy Diwen Zhu, Hui Ma, Xiaokui Xiao, Siqiang Luo, Youze Tang, and Shuigeng Zhou. 2013. Shortest path and distance queries on road networks: towards bridging theory and practice. In *SIGMOD*. 857–868.