Shortest-Path Queries on Complex Networks: Experiments, Analyses, and Improvement

Junhua Zhang
AAII, FEIT, University of Technology Sydney, Australia
junhua.zhang@student.uts.edu.au

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

Long Yuan
Nanjing University of Science and Technology, China
longyuan@njust.edu.cn

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

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

Lijun Chang
The University of Sydney, Australia
lijun.chang@sydney.edu.au

ABSTRACT

The shortest-path query, which returns the shortest path between two vertices, is a basic operation on complex networks and has numerous applications. To handle shortest-path queries, one option is to use traversal-based methods (e.g., breadth-first search); another option is to use extension-based methods, i.e., extending existing methods that use indexes to handle shortest-distance queries to support shortest-path queries. These two types of methods make different trade-offs in query time and space cost, but comprehensive studies of their performance on real-world graphs are lacking. Moreover, extension-based methods usually use extra attributes to extend the indexes, resulting in high space costs. To address these issues, we thoroughly compare the two types of methods mentioned above. We also propose a new extension-based approach, Monotonic Landmark Labeling (MLL), to reduce the required space cost while still guaranteeing query time. We compare the performance of different methods on ten large real-world graphs with up to 5.5 billion edges. The experimental results reveal the characteristics of various methods, allowing practitioners to select the appropriate method for a specific application.

PVLDB Reference Format:
Junhua Zhang, Wentao Li, Long Yuan, Lu Qin, Ying Zhang, and Lijun Chang. Shortest-Path Queries on Complex Networks: Experiments, Analyses, and Improvement. PVLDB, 15(11): 2640 - 2652, 2022.
doi:10.14778/3551793.3551820

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://www.dropbox.com/s/c13f03jp94cie8q/MLL_src.zip?dl=0.

1 INTRODUCTION

A graph (or network) is a common structure for describing entities and their relationships [5, 35]. Many real-world graphs, such as social networks, web graphs, and biological networks, are called complex networks because of their complex topology [9, 13]. These complex networks can have millions or even billions of vertices and edges [25, 26], necessitating the development of efficient tools to support operations on these graphs [3, 4].

This paper studies the shortest-path query, a basic operation on complex networks. In a graph G, the shortest-path query $\mathrm{Q_P}(s, t)$ returns the shortest path between two vertices s and t. Shortest-path queries have a wide range of applications [15, 20]. For example, the shortest path between two users in a social network reveals how their relationship is formed [33]. The shortest path also describes how two components interact in a biological network [31, 32] and is critical for resource management in a communication network [9, 30]. Finding one shortest path between two vertices is also a key component in solving problems such as the Steiner tree [23] and the keyword search [37]. Considering the above applications, efficient answering of shortest-path queries is necessary.

One option to handle shortest-path queries is to use **traversal-based methods**, such as breadth-first search (BFS, for unweighted graphs) [7] or Dijkstra's algorithm (for weighted graphs) [22]. However, traversal on large graphs is slow because its runtime is proportional to the graph size [27]. To speed up graph traversal, several preprocessing techniques [38, 41], such as Highway Hierarchies [36], or Contraction Hierarchies [16], have been proposed to limit the traversal scope. However, traversal-based methods still *take a long time to process a query*, even with the preprocessing techniques [27]. Moreover, these preprocessing techniques often rely on the properties of road networks (e.g., planarity and hierarchical structures [1]), rendering them inapplicable to dealing with complex networks [27].

Another option is to use **extension-based methods**, i.e., extending methods designed for shortest-distance query processing to support shortest-path queries. The shortest-distance query is an operation closely related to the shortest-path query, which returns the length $dist(s, t)$ of the shortest path between two vertices s and t [25]. In recent years, many approaches have been proposed to build indexes for shortest-distance query processing in complex networks [3, 4, 25–27], and the well-known approaches are Pruned Landmark Labeling (PLL [3]) and Core-Tree Labeling (CTL [26]). To extend these methods, we can add an extra attribute to each entry in the index for path recovery. Although the extension-based methods can handle shortest-path queries rapidly, *the introduction of extra attributes makes the space cost too high.*

*Wentao Li is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551820

Motivations. Traversal and extension-based methods make different trade-offs in query time and space cost: traversal-based methods do not require high space cost but have no guarantee of query time; extension-based methods provide a quick query response, but their space costs are high. Yet, comprehensive studies of these two types of methods' performance on real-world graphs are lacking. This makes it hard for practitioners to select an appropriate method for applications that use shortest-path queries as a basic component. Also, extension-based methods typically add extra attributes to the original index designed for shortest-distance queries, thus allowing tracking of all vertices on the shortest path. Such an extension often leads to a too large index to support shortest-path queries.

Our Solution. To address the aforementioned issues, we thoroughly compare various methods for handling shortest-path queries. Moreover, we propose a new extension-based approach, Monotonic Landmark Labeling (MLL), to enable the index designed for shortest-distance queries to work for shortest-path queries. MLL non-trivially creates an additional lightweight index as a plug-in to the original index; instead of extending every index entry (which would result in an extended index nearly twice the size of the original index). As a result, MLL can still give rapid query responses but with a low (extra) space cost.

Contributions. The contributions of this paper are summarized as follows.

- *Efficient extension of distance query processing methods (Section 3).* We extend the shortest-distance query processing methods PLL [3] and CTL [26] to allow them to efficiently handle shortest-path queries. Our idea is to add an extra attribute to each index entry so that the path can be found by tracking each vertex on the shortest path using the extra attribute. We discuss the correctness and time complexity of these extension-based methods in processing shortest-path queries.
- *A new extension-based approach (Section 4).* We propose a new extension-based approach, MLL, tailored for shortest-path queries. MLL non-trivially builds an extra lightweight index on top of the index designed for shortest-distance queries. MLL works by decomposing the shortest path between two vertices into several subpaths, which are then indexed. During query processing, the shortest path can be found efficiently by finding and splicing subpaths. We verify that MLL consumes less space than extending each index entry with an extra attribute while still having a fast query speed.
- *Extending* MLL *to handle directed graphs (Section 5).* We describe how to adapt the proposed approach MLL to handle directed graphs. We thus enable our proposed method MLL to work for more general graphs.
- *Comprehensive experimental studies (Section 6).* We select four traversal-based and three extension-based methods for experimental comparisons. We extensively evaluate the performance of various methods on ten real-world graphs. We also examine the impact of directed graphs on the index size of MLL. To the best of our knowledge, this is the first work to empirically compare shortest-path query processing methods on complex networks.

Due to space limitations, the proofs and some examples are omitted in this paper and can be found in our technical report [43].

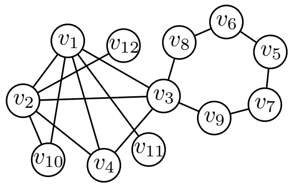


Figure 1: The Example Graph G

2 PRELIMINARY

Let $G(V, E)$ be an undirected unweighted graph with $n = |V(G)|$ vertices and $m = |E(G)|$ edges. The neighbors $N(v, G)$ of each vertex $v \in V(G)$ are defined as $N(v, G) = \{u | (u, v) \in E(G)\}$. The size of $N(v, G)$ is the degree $deg(v, G)$ of v, i.e., $deg(v, G) = |N(v, G)|$. The length of each edge $(u, v) \in E(G)$ is a positive value $\delta(u, v, G)$. In an unweighted graph, the length of all edges is 1.

The path $p(s, t, G)$ between two vertices $s, t \in V(G)$ is a sequence of vertices, i.e., $p(s, t, G) = \{s = v_0, v_1, \cdots, t = v_l\}$. The inner vertices of $p(s, t, G)$ are vertices v_i, where $v_i \neq s, v_i \neq t$, and $i \in [1, l-1]$. The precursor $prev(v_i)$ of a vertex v_i on the path $p(s, t, G)$ is v_{i-1}, where $i \in [1, l]$; the successor $succ(v_i)$ of a vertex v_i on the path $p(s, t, G)$ is v_{i+1}, where $i \in [0, l-1]$. Given two paths $p_1 = \{v_0, v_1, \cdots, v_a\}$ and $p_2 = \{w_0, w_1, \cdots, w_b\}$, when $(v_a, w_0) \in E(G)$, then the splicing operation on p_1 and p_2 is defined as $p_1 + p_2 = \{v_0, v_1, \cdots, v_a, w_0, w_1, \cdots, w_b\}$; or $v_a = w_0$, then $p_1 + p_2 = \{v_0, v_1, \cdots, v_a = w_0, w_1, \cdots, w_b\}$.

The length of $p(s, t, G)$ is defined as $|p(s, t, G)| = \Sigma_{(v_i, v_{i+1})} \delta(v_i, v_{i+1}, G)$, where $i \in [0, l-1]$. The shortest path between s and t is the one with the minimum length among all s-t paths, and the shortest distance $dist(s, t, G)$ is the length of the s-t shortest path. Without loss of generality, we assume that graph G is connected since otherwise, we can work on each connected component separately (as the shortest distance between the vertices of different connected components is positive infinity). If the context is obvious, we remove G from notations for simplicity.

Example 2.1. Fig. 1 gives the example graph G, which contains $n = 12$ vertices and $m = 16$ edges. For the vertex v_5, $N(v_5) = \{v_6, v_7\}$, so $deg(v_5) = 2$. $p(v_5, v_3) = \{v_5, v_6, v_8, v_3\}$ is a v_5-v_3 path. The inner vertices of $p(v_5, v_3)$ are v_6 and v_8. The precursor (resp. successor) of v_6 is v_5 (resp. v_8) on $p(v_5, v_3)$. $p(v_5, v_3)$ is the shortest path between v_5 and v_3, thus $dist(v_5, v_3) = 3$. $p(v_4, v_2) = \{v_4, v_2\}$ is a v_4-v_2 path. As $(v_3, v_4) \in E$, we use the splicing operation to get $p(v_5, v_3) + p(v_4, v_2) = \{v_5, v_6, v_8, v_3, v_4, v_2\}$, which is a v_5-v_2 path.

Problem Definition. Given an undirected unweighted graph $G(V, E)$, a shortest-path query is defined as $\mathrm{Q_P}(s, t)$, where $s, t \in V$. The answer to query $\mathrm{Q_P}(s, t)$ is an s-t shortest path $p(s, t)$. If multiple s-t shortest paths exist, an arbitrary one can be returned. The studied problem is how to process query $\mathrm{Q_P}(s, t)$ efficiently on G.

3 DISTANCE QUERIES AND EXTENSIONS

The shortest-distance query is an operation closely related to the shortest-path query, which returns the shortest path length of two vertices. Many methods have been proposed to create indexes to process shortest-distance queries on complex networks; the well-known methods are PLL [3] and CTL [25]. This section describes how to extend PLL and CTL to support shortest-path queries.

Table 1: Comparison of Different Approaches

	PLL	CTL	MLL
v_1	$(v_1, 0, -)$	$(v_1, 0, -)$	
v_2	$(v_1, 1, -)$, $(v_2, 0, -)$	$(v_1, 1, -)$, $(v_2, 0, -)$	$(v_1, -)$
v_3	$(v_1, 1, -)$, $(v_2, 1, -)$, $(v_3, 0, -)$	$(v_1, 1, -)$, $(v_2, 1, -)$, $(v_3, 0, -)$	$(v_1, -)$, $(v_2, -)$
v_4	$(v_1, 1, -)$, $(v_2, 1, -)$, $(v_3, 1, -)$, $(v_4, 0, -)$	$(v_1, 1, -)$, $(v_2, 1, -)$, $(v_3, 1, -)$, $(v_4, 0, -)$	$(v_1, -)$, $(v_2, -)$, $(v_3, -)$
v_5	$(v_1, 4, v_6)$, $(v_2, 4, v_6)$, $(v_3, 3, v_6)$, $(v_5, 0, -)$	$(v_3, 3, v_6)$	(v_3, v_6)
v_6	$(v_1, 3, v_8)$, $(v_2, 3, v_8)$, $(v_3, 2, v_8)$, $(v_5, 1, -)$, $(v_6, 0, -)$	$(v_3, 2, v_8)$, $(v_5, 1, -)$	(v_3, v_8), $(v_5, -)$
v_7	$(v_1, 3, v_9)$, $(v_2, 3, v_9)$, $(v_3, 2, v_9)$, $(v_5, 1, -)$, $(v_7, 0, -)$	$(v_3, 2, v_9)$, $(v_5, 1, -)$	(v_3, v_9), $(v_5, -)$
v_8	$(v_1, 2, v_3)$, $(v_2, 2, v_3)$, $(v_3, 1, -)$, $(v_5, 2, v_6)$, $(v_6, 1, -)$, $(v_8, 0, -)$	$(v_3, 1, -)$, $(v_5, 2, v_6)$, $(v_6, 1, -)$	$(v_3, -)$, $(v_6, -)$
v_9	$(v_1, 2, v_3)$, $(v_2, 2, v_3)$, $(v_3, 1, -)$, $(v_5, 2, v_7)$, $(v_7, 1, -)$, $(v_9, 0, -)$	$(v_3, 1, -)$, $(v_5, 2, v_7)$, $(v_7, 1, -)$	$(v_3, -)$, $(v_7, -)$
v_{10}	$(v_1, 1, -)$, $(v_2, 1, -)$, $(v_{10}, 0, -)$	$(v_1, 1, -)$, $(v_2, 1, -)$	$(v_1, -)$, $(v_2, -)$
v_{11}	$(v_1, 1, -)$, $(v_{11}, 0, -)$	$(v_1, 1, -)$	$(v_1, -)$
v_{12}	$(v_1, 2, v_2)$, $(v_2, 1, -)$, $(v_{12}, 0, -)$	$(v_2, 1, -)$	$(v_2, -)$

3.1 PLL and Its Extension

Pruned landmark labeling (PLL) is a classical method for processing shortest-distance queries. PLL supports queries by creating an index $\mathsf{L}^{\mathsf{PLL}}$ offline. The index assigns a label $\mathsf{L}^{\mathsf{PLL}}(u) = \{(v, dist(u, v))\}$ to each vertex $u \in V$ in the graph G. $\mathsf{L}^{\mathsf{PLL}}(u)$ contains some selected **landmarks** v and the corresponding shortest distance $dist(u, v)$ for u. The number of landmarks contained in the label of u is defined as the label size $|\mathsf{L}^{\mathsf{PLL}}(u)|$. The maximum label size Δ^{PLL} of PLL is defined as the value of the largest label size among all vertices. The index size $|\mathsf{L}^{\mathsf{PLL}}|$ is the sum of the label size over each vertex $u \in V$, i.e., $|\mathsf{L}^{\mathsf{PLL}}| = \Sigma_{u \in V} |\mathsf{L}^{\mathsf{PLL}}(u)|$.

Distance Query Processing. To report the shortest distance $dist(s, t)$ between vertices s and t in G, we only need to use the labels of s and t, as given in Equation 1.

$$dist(s, t) = \min_{w \in \mathsf{L}^{\mathsf{PLL}}(s) \cap \mathsf{L}^{\mathsf{PLL}}(t)} dist(s, w) + dist(w, t). \qquad (1)$$

We find common landmarks w in the labels of s and t, and select the smallest distance $dist(s, w) + dist(w, t)$ through w as a result. The time cost to compute $dist(s, t)$ is $O(|\mathsf{L}^{\mathsf{PLL}}(s)| + |\mathsf{L}^{\mathsf{PLL}}(t)|)$.

Example 3.1. Consider the graph G in Fig. 1. The PLL column of Table 1 shows $\mathsf{L}^{\mathsf{PLL}}$ (ignore the third attribute marked blue in each entry) of G. For v_2, its label $\mathsf{L}^{\mathsf{PLL}}(v_2) = \{(v_1, 1), (v_2, 0)\}$ has two landmarks, v_1 and v_2, so $|\mathsf{L}^{\mathsf{PLL}}(v_2)| = 2$. The index size is $|\mathsf{L}^{\mathsf{PLL}}| = 44$. To compute $dist(v_2, v_3)$, we use the labels of v_2 and v_3 and get the common landmarks $\{v_1, v_2\}$. As $dist(v_2, v_1) + dist(v_3, v_1) = 1 + 1 = 2$ and $dist(v_2, v_2) + dist(v_2, v_3) = 0 + 1 = 1$, by Equation 1, we know that $dist(v_2, v_3) = 1$ and v_2 is the landmark on v_2-v_3 shortest path.

Theorem 3.2 presents the condition for determining whether vertex v will join the label of u as a landmark.

THEOREM 3.2 ([25]). *The entry $(v, dist(u, v))$ is in $\mathsf{L}^{\mathsf{PLL}}(u)$ iff $r(v) \geq r(w)$ for $\forall w$ on all v-u shortest paths.*

Extension to Path Queries. We next extend the PLL index to support shortest-path queries. The extended index $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}$ assigns a label $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(u) = \{(v, dist(u, v), succ(u))\}$ to each vertex u. We add an extra attribute $succ(u)$, the successor of u on the shortest path $\{v_0 = u, v_1 = succ(u), \cdots, v_{l-1}, v_l = v\}$ from u to v, in the label $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(u)$ of u. If $dist(u, v) < 2$, then the successor $succ(u)$ is meaningless, and we store "$-$" instead. The successor can be used to track all vertices on the shortest path and thus recover this path.

Example 3.3. Consider the graph G in Fig. 1, where Table 1 lists the extended index $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}$. For v_6, $(v_3, 2, v_8) \in \mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(v_6)$, where v_3 is the landmark and v_8 is the successor of v_6 on v_6-v_3 shortest path.

Algorithm 1: Processing $\mathsf{Q_P}(s, t)$ Based on $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}$

```
Input: $\mathsf{Q_P}(s, t)$, index $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}$
Output: The shortest path $p(s, t)$
1  $w, dist(s, t) \leftarrow$ Equation 1;
2  if $dist(s, t) = 0$ then $p(s, t) \leftarrow \{s\}$; return $p(s, t)$;
3  if $dist(s, t) = 1$ then $p(s, t) \leftarrow \{s, t\}$; return $p(s, t)$;
4  $p_1 \leftarrow \{s\}, p_2 \leftarrow \{t\}$;
5  while $dist(s, w) > 1$ do
6      $s \leftarrow succ(s)$, where $(w, dist(s, w), succ(s)) \in \mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(s)$;
7      $p_1 \leftarrow p_1 + \{s\}$;
8  while $dist(w, t) > 1$ do
9      $t \leftarrow succ(t)$, where $(w, dist(t, w), succ(t)) \in \mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(t)$;
10     $p_2 \leftarrow \{t\} + p_2$;
11 return $p(s, t) = p_1 + \{w\} + p_2$
```

Path Query Processing. We describe how to answer $\mathsf{Q_P}(s, t)$ using index $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}$ in Algorithm 1. First, we use Equation 1 to get $dist(s, t)$ and the landmark w in the labels of s and t, s.t., $dist(s, t) = dist(s, w) + dist(w, t)$ (Line 1). If $dist(s, t)$ is 0, then $s = t$ and we return $\{s\}$ as a path (Line 2); if $dist(s, t)$ is 1, then $\{s, t\}$ is an edge and we return this edge as a path (Line 3). Otherwise, w is used to decompose the s-t shortest path into two subpaths p_1 and p_2. p_1 is first initialized to contain only vertex s (Line 4), then we use $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(s)$ to get the successor $succ(s)$ of s on the s-w subpath and iteratively add the vertices on the s-w subpath to p_1 by assigning $succ(s)$ to s (Line 5-7); similarly, we use $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(t)$ to iteratively add the vertices on the t-w subpath to p_2 (Line 8-10). Finally, p_1 and p_2 are spliced with w to produce the answer (Line 11).

Example 3.4. Consider the graph G in Fig. 1. Given $\mathsf{Q_P}(v_6, v_3)$, we use Equation 1 to get $dist(v_6, v_3) = 2$ and the landmark v_3 on the v_6-v_3 shortest path $p(v_6, v_3)$. The shortest path is divided into two subpaths p_1, p_2 by the landmark v_3. We initialize $p_1 = \{v_6\}$ and inquire $\mathsf{L}_{\mathsf{E}}^{\mathsf{PLL}}(v_6)$ to obtain the successor $succ(v_6) = v_8$ of the starting vertex v_6 on the v_6-v_3 subpath. We add v_8 to p_1 and use v_8 as the new starting vertex for the next round of iterations. The iteration stops here because $dist(v_8, v_3) = 1$. We then initialize $p_2 = \{v_3\}$, but the iteration ends because $dist(v_3, v_3) = 0$. Finally, we return the answer $p_1 + \{v_3\} + p_2 = \{v_6, v_8, v_3\}$.

LEMMA 3.5. *Algorithm 1 correctly answers the query $\mathsf{Q_P}(s, t)$.*

LEMMA 3.6. *Algorithm 1 requires*[1] *$O(dist(s, t) \times \log \Delta^{\mathsf{PLL}})$ to find the s-t shortest path, where Δ^{PLL} is PLL's maximum label size.*

[1]If additional data structures such as hash tables or pointers are used to store the labels for the quick label lookup, the complexity can be reduced to $O(dist(s, t))$. We disregard this optimization due to the oversized PLL index.

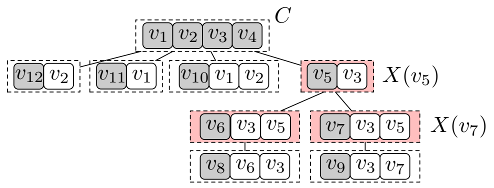


Figure 2: (Core-)Tree Decomposition of G

3.2 CTL and Its Extension

CTL is proposed to avoid the oversized indexes of PLL [25, 26] for shortest-distance queries. CTL relies on the concept of core-tree decomposition, which is a special kind of tree decomposition.

Definition 3.7 (Tree Decomposition). The tree decomposition of a graph $G(V, E)$, denoted as T_G, is a rooted tree, where every node $X \in V(T_G)$ in the tree is a subset of vertices of the graph G, i.e., $X \subseteq V(G)$. T_G meets the following three conditions.

(1) $\bigcup_{X \in V(T_G)} X = V(G)$;
(2) For every edge $(u, v) \in E(G)$ in the graph G, there exists a node X in $V(T_G)$ such that $u \in X$ and $v \in X$;
(3) For every vertex $v \in V(G)$ in the graph G, the set $T(v) = \{X \in V(T_G) | v \in X\}$ is a connected subtree.

The root of subtree $T(v)$ is $X(v)$, for $\forall v \in V(G)$. The **treewidth** of T_G is defined as $tw(T_G) = \max_{X \in V(T_G)} |X| - 1$.

We refer to each $v \in V(G)$ as a vertex and each $X \in V(T_G)$ as a node. The **ancestor nodes** ANC(X) of node X are all the nodes on the shortest path from X to the root in T_G; the **parent node** PAR(X) of X is the ancestor node connecting to X.

Example 3.8. Fig. 2 is the tree decomposition T_G of G in Fig. 1. We verify three conditions for T_G. (1) Each vertex of G (say v_1) appears in some node (say $C = \{v_1, v_2, v_3, v_4\}$) of T_G. (2) For each edge of G, say (v_1, v_2), we find a node C containing both endpoints v_1, v_2. (3) For vertex v_5 of G, all (marked in red) nodes in T_G containing v_5 form a connected subtree $T(v_5)$. The root of subtree $T(v_5)$ is $X(v_5)$. Similarly, $X(v_7)$ is the root of the subtree consisting of the nodes containing v_7. On T_G, the ancestor nodes of $X(v_7)$ are $\text{ANC}(X(v_7)) = \{X(v_7), X(v_5), C\}$, and the parent node of $X(v_7)$ is $\text{PAR}(X(v_7)) = X(v_5)$. The treewidth of T_G is $tw(T_G) = |C| - 1 = 3$.

Definition 3.9 (Core-Tree Decomposition). Given a parameter d, the core-tree decomposition T_G of graph G is a tree decomposition with the fourth condition: there is a special node (defined as the **core part**) $C \in V(T_G)$, s.t., $|C| > d + 1$; for the other nodes (defined as the **tree part**) $X \in V(T_G) \setminus C$, $|X| \leq d + 1$.

Decomposing a Graph. To obtain a core-tree decomposition T_G (with parameter d) of graph G, we can use minimum degree elimination (MDE) [8]. MDE creates nodes and then edges of T_G.

<u>Node elimination.</u> MDE selects the smallest degree vertex for elimination each time. When the degree of the vertex selected at some time is $\geq d + 1$, elimination stops. We initialize $G_1 = G$, and then every time i, we eliminate vertex v with the smallest degree in G_i.

(1) v: the vertex with the smallest degree in the graph G_i (or any of them if there is a tie), and the order of v is set to $r(v) = i$.
(2) For v's neighbors $N(v, G_i)$ in G_i, we add extra edges to make $N(v, G_i)$ form a clique (i.e., complete graph).
   (a) If $u, w \in N(v, G_i)$ is not an edge in G_i, we add edge (u, w) whose length equals the length of the path $\{u, v, w\}$, i.e., $\delta(u, v, G_i) + \delta(w, v, G_i)$. To record that (u, w) is made by removing v, we set v as the **elimination vertex** of (u, w).
   (b) Otherwise (u, w) is an edge of G_i, then we set its length to the smallest of the current length and the length of the path $\{u, v, w\}$, i.e. $\min\{\delta(u, v, G_i), \delta(u, v, G_i) + \delta(w, v, G_i)\}$. If the edge length is updated with a smaller value, we set v as the elimination vertex of (u, w).
(3) Form a node $X(v) = v \cup N(v, G_i)$.
(4) Delete v from G_i to form G_{i+1} for the next round ($i \leftarrow i + 1$).

<u>Edge generation.</u> We next describe how to generate edges by imposing parent relations for the nodes in T_G. When the vertex elimination stops, suppose the value of i is λ, then we get a graph G_λ. We get all the nodes $X(v)$ in the tree part of T_G, where $r(v) \in [1, \lambda - 1]$.

(1) Take all vertices $V(G_\lambda)$ in G_λ as the core part C of T_G.
(2) For each node $X(v)$ in the tree part,
   (a) if the vertices of $X(v) \setminus \{v\}$ all belong to C, then make C the parent of $X(v)$;
   (b) otherwise, find the vertex $u \notin C$ with the lowest order $r(u)$ in $X(v) \setminus \{v\}$, and make $X(u)$ the parent of $X(v)$.

After the whole process, we get the core-tree decomposition T_G.

CTL Index. Given a core-tree decomposition T_G of G, we create separate indexes for the core part C and the tree part $V(T_G) \setminus C$. For vertices v in the tree part, the order $r(v)$ is set to the moment i when v is eliminated; For vertices v in the core part, the order $r(v)$ is set according to the degree (as in PLL). The order of vertices in the core part is forced to be set higher than the vertices in the tree part. The indexes of the two parts together form the index $\mathsf{L}^{\mathsf{CTL}}$.

<u>Core index.</u> For the core part C, we create the index in the graph G_λ using PLL, thus assigning a core label for each vertex in C.

<u>Tree index.</u> For each node $X(v) \in V(T_G) \setminus C$ in the tree part, we assign a tree label for the corresponding vertex v of $X(v)$. The label of v contains the distances between v and its landmarks u where $u \in \bigcup_{X \in \text{ANC}(X(v)) \setminus C} X$, and $u \neq v$.

When using the index $\mathsf{L}^{\mathsf{CTL}}$ to obtain the distance $dist(s, t)$ between two vertices s and t, if both s and t are in the core part, then we can just use the core index to complete the distance query according to Equation 1; otherwise, if one of the vertices is not in the core part, then we need to use both the core index and the tree index to complete the distance query[2]. The time cost of shortest-distance queries using $\mathsf{L}^{\mathsf{CTL}}$ is $O(d \cdot \log |C| \cdot tw(T_G))$, where $tw(T_G)$ is the treewidth of T_G.

3.2.1 *Extension to Path Queries.*

To handle shortest-distance queries, CTL assigns a (core or tree) label $\mathsf{L}^{\mathsf{CTL}}(u) = \{(v, dist(u, v))\}$ to each vertex $u \in V$ in the graph, where v is the landmark of u. To make CTL support shortest-path queries, like PLL, we need to add an extra attribute $auxi(u)$ to the label entry $(v, dist(u, v))$ of each vertex u, thus obtaining the extended label $\mathsf{L}_{\mathsf{E}}^{\mathsf{CTL}}(u) = \{(v, dist(u, v), auxi(u))\}$. The extra attribute $auxi(u)$ is an inner vertex on the u-v shortest path; it is used to help find the shortest path between two vertices. If $dist(u, v) \leq 1$, then $auxi(u)$ is unnecessary, and we store "-" instead. Because CTL separates the vertices into two parts, we discuss two different extended labels.

[2]For a more detailed distance query process, please refer to the literature [26].

Extended Core Label. For a vertex $u \in C$ in the core part, we extend the label of u in a similar way as we extend PLL, i.e., for any landmark v of u, we set $auxi(u)$ to $succ(u)$, the successor of u on the u-v shortest path in G_λ. The only difference is that for an edge (u, v) in G_λ, its weight $\delta(u, v, G_\lambda)$ may be greater than 1. In this case, we need to further find the corresponding u-v shortest path in G for this edge (u, v) in G_λ. To do so, instead of assigning $auxi(u)$ the value "-", we assign the elimination vertex w of (u, v) — eliminating w forms the edge (u, v) — to $auxi(u)$.

Extended Tree Label. For each node $X(u) \in V(T_G) \setminus C$ in the tree part, we extend the tree label of the vertex u corresponding to that node. Specifically, for each landmark v of u, if $dist(u, v) < 2$, $auxi(u)$ is set to "-"; otherwise, we choose some vertex on the u-v shortest path to be assigned to $auxi(u)$. (1) If $v \notin X(u)$, an inner vertex on the u-v shortest path can be picked from $X(u)$ as $auxi(u)$ — According to Lemma 3 of [11], $X(u) \setminus u$ is the cut that separates u and v, so $X(u)$ must contain some inner vertex on the u-v shortest path; (2) Otherwise, an inner vertex on the u-v shortest path can be found from either $X(u)$ or the elimination vertex w of (u, v) as $auxi(u)$ — Lemma 3 of [11] does not necessarily apply to this case, and the u-v shortest path may contain the elimination vertex w, since w must be on the (local) u-v shortest path whose inner vertices all do not belong to $X(u)$ [26].

Example 3.10. Consider the core-tree decomposition in Fig. 2 and the extended CTL index $\mathsf{L}_\mathsf{E}^{\mathsf{CTL}}$ in Table 1. Given the vertex v_8 in the tree part, for landmark v_5 of v_8, as $v_5 \notin X(v_8)$, we pick an inner vertex $v_6 \in X(v_8)$ on the v_8-v_5 shortest path from $X(v_8) = \{v_8, v_6, v_3\}$ as $auxi(v_8)$ to extend the label entry. Given the vertex v_6 in the tree part, for landmark v_3 of v_6, as $v_3 \in X(v_6)$, we find an inner vertex v_8 on the v_6-v_3 shortest path as $auxi(v_6)$ to extend the label entry, where v_8 is the elimination vertex of the edge (v_6, v_3).

3.2.2 Path Query Processing.

We use the extended CTL index $\mathsf{L}_\mathsf{E}^{\mathsf{CTL}}$ to process the path query $\mathsf{Q_P}(s, t)$, thus getting the s-t shortest path $p(s, t)$. There is some complexity in using the extended CTL index for path queries. For the sake of convenience, we first discuss two special cases of queries, and then introduce how to handle the general case of path queries based on these two special cases.

Special Cases. We first give two special cases (sp1-sp2) of queries.

sp1*: $p(s, t)$ contains only the vertices of the tree part.* We find the s-t shortest path $p(s, t)$ using the following steps.

(1) We first use tree index[3] and a similar function to Equation 1 to find $dist(s, t)$ as well as the landmark vertex w on $p(s, t)$; we divide the path $p(s, t)$ into two parts by landmark w: s-w subpath p_1 and w-t subpath p_2. The following steps only explain how to find p_1, and the process of finding p_2 is similar.
(2) If $dist(s, w) = 0$, we return $p_1 = \{s\}$; if $dist(s, w) = 1$, return $p_1 = \{s, w\}$. Otherwise, we find $(w, dist(w, s), auxi(s)) \in \mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(s)$. We use attribute $auxi(w)$ to decompose p_1 into s-$auxi(s)$ and $auxi(s)$-w subpaths. We recursively call Step (2) to find them. Finally, they are spliced together as p_1 to return.
(3) Return $p_1 + \{w\} + p_2$ as the s-t shortest path.

[3]Note that each vertex u of the tree part uses itself as landmark during query processing, i.e., generate a new label entry $(u, 0, \text{"–"})$.

sp2*: $p(s, t)$ contains vertices of both parts, $s \notin C$, $t \in \mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(s) \cap C$.* If $dist(s, t) = 1$, then $p(s, t) = \{s, t\}$. Otherwise, since $t \in \mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(s)$, we can obtain the label entry $(t, dist(s, t), auxi(s))$ to get the extra attribute $w = auxi(s)$. We then use w to find the path $p(s, t)$ recursively (similar to Step (2) of sp1).

Example 3.11. Consider the extended CTL index in Table 1. For the query $\mathsf{Q_P}(v_9, v_5)$, since the shortest path $p(v_9, v_5)$ does not pass through the vertices of the core part, it belongs to sp1. To process $\mathsf{Q_P}(v_9, v_5)$, we use the tree index to find a landmark v_5 on the v_9-v_5 path to divide the path into subpath $p1(v_9, v_5)$ and the trivial subpath $p2(v_5, v_5)$. (Step (1)). Next, to find $p1(v_9, v_5)$, we query $\mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(v_9)$ to get $(v_5, 2, auxi(v_5) = v_7)$, and then use the extra attribute v_7 to query and splice paths recursively until $p1(v_9, v_5) = \{v_9, v_7, v_5\}$ is found (Step (2)). For the query $\mathsf{Q_P}(v_5, v_3)$, since $v_5 \notin C$ and $v_3 \in \mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(v_5) \cap C$, it belongs to sp2. To process $\mathsf{Q_P}(v_5, v_3)$, we first query $\mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(v_5)$ to get $(v_3, 3, auxi(v_5) = v_6)$, and then use the extra attribute v_6 to query and spice paths recursively until $p(v_5, v_3) = \{v_5, v_6, v_8, v_3\}$ is returned.

General Cases. Based on sp1 and sp2, we are now ready to give query processing in general.

Case 1: $s, t \in C$. First, we use a similar method to Algorithm 1 to get the shortest path $p(s, t, G_\lambda)$ between s and t in G_λ. For each edge (u, v) on the path, if $dist(u, v) = 1$, then the edge (u, v) is returned directly; otherwise, the edge (u, v) needs to be unfolded to the u-v shortest path on the original graph — suppose $r(u) > r(v)$, then $(u, dist(u, v), auxi(v) = w) \in \mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(v)$, we divide $p(u, v)$ into u-w subpath p_1 and w-v subpath p_2. Since eliminating $w \notin C$ produces the edge (u, v) in G_λ, then $u, v \in X(w)$ by the core-tree decomposition process and thus u and v are the landmarks of w. Therefore, the subpaths p_1 and p_2 can be both found using sp2, which can be spliced to produce $p(u, v)$. The s-t shortest path in G is obtained by applying the above process to all edges in $p(s, t, G_\lambda)$.

Case 2: $s \notin C, t \in C$ (or vice versa). Using the distance query of CTL, we can get the vertex c on $p(s, t)$, where $c \in \mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(s) \cap C$ (c is the interface [26]). We divide the s-t shortest path $p(s, t)$ into two segments, $p(s, c)$ and $p(c, t)$, where the subpath $p(s, c)$ can be found by sp2 and the subpath $p(c, t)$ can be processed by Case 1. Finally, $p(s, c)$ and $p(c, t)$ can be spliced to obtain $p(s, t)$.

Case 3: $s, t \notin C$. Using the distance query of CTL, we can get the vertex c (resp. d) on $p(s, t)$, where $c \in \mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(s) \cap C$ (resp. $d \in \mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(s) \cap C$). If both c and d do not exist, this indicates that s and t do not pass through the core part, then $p(s, t)$ is obtained directly using sp1; otherwise, since $c \in C$, the subpath $p(s, c)$ can be handled by Case 2; since both $c, d \in C$, the subpath $p(c, d)$ can be handled by Case 1; since $d \in C$, the subpath $p(d, t)$ can be processed by Case 2. $p(s, t)$ can be obtained by splicing $p(s, c)$, $p(c, d)$, and $p(d, t)$.

Example 3.12. Consider the extended index $\mathsf{L}_\mathsf{E}^{\mathsf{CTL}}$ in Table 1. We show how to process the query $\mathsf{Q_P}(v_5, v_{10})$ (Case 3). (1) We first find $c = v_3 \in \mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(v_5)$ on the v_5-v_{10} path using the distance query of CTL. We can find the subpath $p(v_5, v_3) = \{v_5, v_6, v_8, v_3\}$ (Case 2). (2) We then find $d = v_1 \in \mathsf{L}_\mathsf{E}^{\mathsf{CTL}}(v_{10})$ on the v_5-v_{10} path by querying the CTL index. We can find the subpath $p(v_1, v_{10}) = \{v_1, v_{10}\}$ (Case 2). (3) We find the subpath $p(v_3, v_1) = \{v_3, v_1\}$ via the extended core index (Case 1). Hence, $p(v_5, v_{10}) = p(v_5, v_3) + p(v_3, v_1) + p(v_1, v_{10}) = \{v_5, v_6, v_8, v_3, v_1, v_{10}\}$.

LEMMA 3.13. *Using $\mathsf{L}^{\mathsf{CTL}}_{\mathsf{E}}$ can correctly answer the query $\mathrm{Q_P}(s,t)$.*

LEMMA 3.14. *Using $\mathsf{L}^{\mathsf{CTL}}_{\mathsf{E}}$ requires $O(dist(s,t) \times \log \Delta^{\mathsf{CTL}})$ to answer the query $\mathrm{Q_P}(s,t)$, where Δ^{CTL} is CTL's maximum label size.*

4 MONOTONIC LANDMARK LABELING

Section 3 describes how to extend PLL and CTL to support shortest-path queries. Implementing the extensions requires adding an extra attribute to each index entry to enable fast pathfinding. However, adding the extra attributes makes the extended PLL and CTL indexes too large: both the extended PLL and CTL indexes occupy about twice the size of the original index. On the other hand, although using the traversal-based approach does not require high space cost, there is no way to guarantee query time.

In this section, we propose a new extension-based approach, Monotonic Landmark Labeling (MLL), to further balance the space cost and query time. Considering that CTL can handle large graphs that PLL cannot [26], we choose to extend CTL. Instead of adding extra attributes to each entry of the CTL index, our approach MLL is to non-trivially create an additional lightweight index (i.e., the MLL index $\mathsf{L}^{\mathsf{MLL}}$) on top of the CTL index as a plug-in to facilitate shortest-path queries. This lightweight index not only avoids the excessive space cost caused by the extra attributes but also guarantees query time. We will introduce the new MLL index in Section 4.1, followed by a description of how to use both the CTL index and the MLL index to support queries in Section 4.2, and finally, we will introduce how to create the MLL index in Section 4.3.

4.1 Index Structure

The concept of the monotonic shortest path underpins our method MLL. We set the vertex order of MLL using the same order as CTL.

Definition 4.1 (Monotonic Shortest Path). Given two vertices s, t of G, the s-t shortest path $p(s,t) = \{v_0 = s, v_1, \cdots, v_l = t\}$ is monotonic iff $r(v_i) < \min\{r(s), r(t)\}$ for $\forall v_i \in p(s,t), i \in [1, l-1]$.

The shortest path $p(s,t)$ is monotonic if the order of inner vertices (i.e., vertices excluding s and t) is lower than both s and t. Any edge in the graph is a trivial monotonic shortest path. We show any shortest path can be split into several monotonic shortest paths.

LEMMA 4.2. *The shortest path $p(s,t)$ between any two vertices s and t can be split into a set of monotonic shortest paths $\{\tilde{p_1}, \tilde{p_2}, \cdots, \tilde{p_l}\}$, s.t., $p(s,t) = \tilde{p_1} + \tilde{p_2} \cdots \tilde{p_l}$.*

Example 4.3. Consider the graph G in Fig. 1. We assume that $r(v_1) > r(v_2) > \cdots > r(v_{12})$. The v_3-v_5 shortest path $p(v_3, v_5) = \{v_3, v_9, v_7, v_5\}$ is monotonic as the order of inner vertices $\{v_9, v_7\}$ is lower than v_3 and v_5. The v_5-v_4 shortest path $p(v_5, v_4) = \{v_5, v_7, v_9, v_3, v_4\}$ is not monotonic as the order of the inner vertex v_3 is higher than v_5. We describe how to decompose $p(v_5, v_4)$ into several monotonic shortest paths. First, starting from v_5, we find a vertex v_3 of order higher than v_4 and stop, forming the first monotonic path $\tilde{p_1}(v_5, v_3) = \{v_5, v_7, v_9, v_3\}$. Then starting from v_4 until meeting v_3, we get the second monotonic path $\tilde{p_2}(v_3, v_4) = \{v_3, v_4\}$. It follows that $p(v_5, v_4) = \tilde{p_1}(v_5, v_3) + \tilde{p_2}(v_3, v_4)$.

MLL **Index.** Lemma 4.2 shows that any shortest path can be split into several monotonic shortest paths. Our basic idea is to index monotonic shortest paths, and these indexed shortest paths can be stitched together to answer any shortest-path query. Based on this, we create a new kind of index $\mathsf{L}^{\mathsf{MLL}}$. The index $\mathsf{L}^{\mathsf{MLL}}$ assigns a label $\mathsf{L}^{\mathsf{MLL}}(u)$ to each vertex $u \in V$ in the graph, which includes some landmark v selected for u and the auxiliary vertex $h(u)$.

Definition 4.4. The entry $(v, h(u))$ is in $\mathsf{L}^{\mathsf{MLL}}(u)$ iff

(1) $r(v) \geq r(w)$ for $\forall w$ on all u-v shortest paths, and $v \neq u$;
(2) All u-v shortest paths are monotonic.

$h(u)$ is the highest-order *inner* vertex on all u-v shortest paths.

For MLL, if v is a landmark of u, then v needs to satisfy two conditions. The first condition is that $r(v)$ is the highest over all vertices in u-v shortest paths. Note that PLL only requires this condition to add v as the landmark of u (see Theorem 3.2). Also, we require $v \neq u$ to forbid v to become its own landmark. The second condition is that all u-v shortest paths must be monotonic. Without this condition, MLL will index the shortest paths (between any two vertices) as in PLL. Because of this condition, MLL only indexes the monotonic shortest paths, causing MLL to have a significantly smaller index size than PLL.

Also, we record the inner vertex $h(u)$ that has the highest order among all u-v shortest paths. If $dist(u,v) < 2$, then there is no inner vertex on the u-v shortest path, so we store "-" instead. The role of $h(u)$ is similar to that of the precursor used to extend PLL for shortest-path queries (see Algorithm 1), i.e., using $h(u)$, we can track all vertices on a shortest path and thus recover the path.

Example 4.5. Consider the graph G in Fig. 1, where the MLL column of Table 1 gives the MLL index. For vertex v_6, $(v_3, v_8) \in \mathsf{L}^{\mathsf{MLL}}(v_6)$ because all v_3-v_6 shortest paths are monotonic and v_3 has the highest order on all paths; $h(v_6) = v_8$ as v_8 is the inner vertex with the highest order on all v_3-v_6 paths. Similarly, for v_2, $(v_1, -) \in \mathsf{L}^{\mathsf{MLL}}(v_2)$ because all v_1-v_2 shortest paths are monotonic and v_1 has the highest order; $h(v_2) = -$ because $dist(v_1, v_2) = 1$.

A careful reading of Definition 4.4 reveals redundancy. The second condition (all u-v shortest paths are monotonic) implies that all inner vertices w on u-v shortest paths are of a lower order than v, i.e., $r(v) > r(w)$. We give a more intuitive condition to decide if v is a landmark of u.

THEOREM 4.6. *The entry $(v, h(u))$ is in $\mathsf{L}^{\mathsf{MLL}}(u)$ iff all u-v shortest paths are monotonic, and $r(v) > r(u)$.*

Index Size. MLL supports path queries by building an additional MLL index on top of the CTL index, so the total index size of MLL includes the CTL index size and the MLL index size (as the extra space cost). On the other hand, extending CTL (and PLL) by extra attributes also introduces an extra space cost. According to empirical results, the extra space required for the extended CTL (and PLL) method occupies almost the same size as the original index. To intuitively compare the *extra space cost* required by MLL and the extended CTL (and PLL) method, we compare the MLL index size with the original CTL (and PLL) index size.

We define the label size $|\mathsf{L}^{\mathsf{MLL}}(u)|$ of u as the number of landmarks contained in $\mathsf{L}^{\mathsf{MLL}}(u)$. Then the MLL index size is defined as $|\mathsf{L}^{\mathsf{MLL}}| = \Sigma_{u \in V} |\mathsf{L}^{\mathsf{MLL}}(u)|$. We show the PLL index size exceeds the MLL index size (suppose PLL and MLL use the same vertex order.).

Theorem 4.7. $|\mathsf{L}^{\mathsf{MLL}}| < |\mathsf{L}^{\mathsf{PLL}}|$.

We show that the CTL index size exceeds the MLL index size. We define the label size of each vertex u as the number of u's landmarks in u's label for CTL. So the CTL index size $|\mathsf{L}^{\mathsf{CTL}}|$ is the total label size of all vertices.

Theorem 4.8. $|\mathsf{L}^{\mathsf{MLL}}| < |\mathsf{L}^{\mathsf{CTL}}|$.

Remark. In practice, the MLL index size is small. From the experimental results in Section 6, the MLL index size on all graphs does not exceed 23 GB. Compared to the index sizes of PLL and CTL, the size of the MLL index is on average 22 times and 5.19 times smaller than that of the PLL and CTL indexes (before the extension).

4.2 Query Processing

We first describe how to process queries using $\mathsf{L}^{\mathsf{MLL}}$ (and $\mathsf{L}^{\mathsf{CTL}}$) if all shortest paths between two vertices are monotonic, and then show how to process queries in general. The entire query process is given in Algorithm 2.

Case 1: All Paths Are Monotonic. For MLL, by Definition 4.4, if v is a landmark of u, i.e., $(v, x = h(u)) \in \mathsf{L}^{\mathsf{MLL}}(u)$, then all v-u shortest paths are monotonic. To find the v-u shortest path $p(u, v)$ in this case, we use an idea similar to the one used in the PLL extension, i.e., to employ an auxiliary vertex $h(u)$ (similar to $succ(u)$ in Algorithm 1) to track all vertices on a shortest path. Specifically, we define Procedure Unfold(u, v, x) in Algorithm 2 (Line 12-18).

We use $x = h(u)$ to decompose $p(u, v)$ into subpaths $p(u, x)$ and $p(x, v)$ to find them separately. For $p(u, x)$, if $dist(u, x) = 1$, then (u, x) is an edge and is returned as $p(u, x)$ (Line 14). Otherwise, we take out $(u, h(x))$ from $\mathsf{L}^{\mathsf{MLL}}(x)$, and call Unfold$(u, x, h(x))$ recursively to find $p(u, x)$ (Line 15). Similarly, we find $p(x, v)$ (Line 16-17) and return $p(u, x) + p(x, v)$ as a result (Line 18).

Example 4.9. Consider the graph G in Fig. 1. v_3 is a landmark of v_5: $(v_3, v_6) \in \mathsf{L}^{\mathsf{MLL}}(v_5)$. To find the v_3-v_5 shortest path, we call Procedure Unfold(v_3, v_5, v_6). (1) Since $dist(v_3, v_6) > 1$ and $(v_3, v_8) \in \mathsf{L}^{\mathsf{MLL}}(v_6)$, we recursively call Unfold(v_3, v_6, v_8) to get path $p(v_3, v_6) = \{v_3, v_8, v_6\}$. (2) Since $dist(v_6, v_5) = 1$, we directly return $\{v_6, v_5\}$ as the v_6-v_5 shortest path $p(v_6, v_5)$. $p(v_3, v_6) + p(v_6, v_5) = \{v_3, v_8, v_6, v_5\}$ is returned as $p(v_3, v_5)$.

Lemma 4.10. *Procedure* Unfold$(u, v, x = h(u))$ *correctly returns the u-v shortest path* $p(u, v)$ *when* $(v, h(u)) \in \mathsf{L}^{\mathsf{MLL}}(u)$.

Lemma 4.11. *Procedure* Unfold$(u, v, x = h(u))$ *requires* $O(dist(u, v))$ *shortest-distance queries to return the u-v shortest path* $p(u, v)$ *when* $(v, h(u)) \in \mathsf{L}^{\mathsf{MLL}}(u)$.

Case 2: General Case. If not all shortest paths between two vertices s and t are monotonic, then we cannot use Procedure Unfold to find the s-t shortest path $p(s, t)$: t is not a landmark of s (assume $r(s) \le r(t)$). To handle this case, we resort to Lemma 4.2, which states that any shortest path can be decomposed into several monotonic shortest (sub)paths. When $p(s, t)$ is broken down into monotonic shortest subpaths, Procedure Unfold can find each of them. By splicing these subpaths, we can find out $p(s, t)$.

Algorithm 2 describes how to answer $\mathrm{Q_P}(s, t)$ in a general case. We swap s and t to make $r(s) \le r(t)$ (Line 1). Then we run a distance query using $\mathsf{L}^{\mathsf{CTL}}$ to get $dist(s, t)$ (Line 2). If $dist(s, t)$ is 0 or 1, we can return the path $p(s, t)$ directly (Line 3-4). Otherwise, we enumerate all landmarks in $\mathsf{L}^{\mathsf{MLL}}(s)$ and find the one w that is on $p(s, t)$ (Line 5-7). If $dist(s, w) = 1$, we set the edge $\{s, w\}$ as $p(s, w)$ directly (Line 8); otherwise, since w is a landmark of s, the s-w shortest path $p(s, w)$ can be discovered by Procedure Unfold (because all s-w shortest paths are monotonic) (Line 9). Here, $p(s, w)$ is the first monotonic shortest subpath of $p(s, t)$, we then set w as s to continue with Algorithm 2, until all monotonic shortest subpaths of $p(s, t)$ are found (Line 10-11).

Algorithm 2: Processing $\mathrm{Q_P}(s, t)$ for MLL

```
Input: $\mathrm{Q_P}(s,t)$, index $\mathsf{L}^{\mathsf{CTL}}$, index $\mathsf{L}^{\mathsf{MLL}}$
Output: The shortest path $p(s,t)$
1  if $r(s) > r(t)$ then swap $s$ and $t$;
2  $dist(s,t) \leftarrow$ query by $\mathsf{L}^{\mathsf{CTL}}$;
3  if $dist(s,t) = 0$ then $p(s,t) \leftarrow \{s\}$, return $p(s,t)$;
4  if $dist(s,t) = 1$ then $p(s,t) \leftarrow \{s,t\}$, return $p(s,t)$;
5  for $(w, h(s)) \in \mathsf{L}^{\mathsf{MLL}}(s)$ do
6      $dist(s,w), dist(t,w) \leftarrow$ query by $\mathsf{L}^{\mathsf{CTL}}$;
7      if $dist(s,w) + dist(t,w) = dist(s,t)$ then break;
8  if $dist(s,w) = 1$ then $p(s,w) \leftarrow \{s,w\}$;
9    else $p(s,w) \leftarrow$ Unfold($s$, $w$, $h(s)$);
10 $p(w,t) \leftarrow$ Algorithm 2($w,t$);
11 return $p(s,w) + p(w,t)$

12 Procedure Unfold($u$, $v$, $x$)
13     $dist(u,x), dist(x,v) \leftarrow$ query by $\mathsf{L}^{\mathsf{CTL}}$;
14     if $dist(u,x) = 1$ then $p(u,x) \leftarrow \{u,x\}$;
15     else $p(u,x) \leftarrow$ Unfold($u$, $x$, $h(x)$), where $(u, h(x)) \in \mathsf{L}^{\mathsf{MLL}}(x)$;
16     if $dist(x,v) = 1$ then $p(x,v) \leftarrow \{x,v\}$;
17     else $p(x,v) \leftarrow$ Unfold($x$, $v$, $h(x)$), where $(v, h(x)) \in \mathsf{L}^{\mathsf{MLL}}(x)$;
18 return $p(u,x) + p(x,v)$
```

Example 4.12. Consider the graph G in Fig. 1. When answering $\mathrm{Q_P}(v_6, v_4)$ that finds the v_6-v_4 shortest path $p(v_6, v_4)$, we first find the landmark v_3 from the label of $s = v_6$, which is on $p(v_6, v_4)$. Since $dist(v_6, v_3) > 1$, we call Unfold$(v_6, v_3, h(v_6) = v_8)$ to find the v_6-v_3 shortest path $p(v_6, v_3) = \{v_6, v_8, v_3\}$. Then we set $s = v_3$ and since $dist(v_3, v_4) = 1$, we find $p(v_3, v_4) = \{v_3, v_4\}$ directly. Splicing $p(v_6, v_3)$ with $p(v_3, v_4)$ yields $p(v_6, v_4) = \{v_6, v_8, v_3, v_4\}$.

Theorem 4.13. *Algorithm 2 correctly answers the query* $\mathrm{Q_P}(s, t)$.

Lemma 4.14. *Algorithm 2 requires* $O(\Sigma_{v \in p(s,t)} |\mathsf{L}^{\mathsf{MLL}}(v)|)$ *shortest-distance queries to answer the query* $\mathrm{Q_P}(s, t)$.

Lemma 4.15. $\mathsf{L}^{\mathsf{MLL}}$ *is minimal for correct query processing.*

Remark. The best-case scenario for MLL's query time is similar to extending PLL and CTL using extra attributes, so MLL's query speed will be slightly inferior. However, MLL's sacrifices in query time allow us to use less space cost to support queries. Moreover, the query time of MLL (Algorithm 2) is related to the length of the shortest path (bounded by the graph diameter D) and the label size of MLL. For complex networks (used in this paper), since both the diameter D (mostly under 50) and the average label size (all less than 150) are small, the query time for MLL is still fast in practice — shortest-path queries using MLL on all graphs can be completed in less than 2 milliseconds.

Algorithm 3: MLL Index Construction

```
   Input: Graph $G(V, E)$, Index $\mathsf{L}^{\mathsf{CTL}}$
   Output: The index $\mathsf{L}^{\mathsf{MLL}}$
1  for each vertex $v \in V$ in parallel do
2      $Q \leftarrow$ a queue with only vertex $v$;
3      $dist(v, v) \leftarrow 0$ and $dist(v, u) \leftarrow \infty, \forall u \in V \setminus \{v\}$;
4      $h(u) \leftarrow -, \forall u \in V$;
5      while $Q \neq \emptyset$ do
6          $u \leftarrow Q.pop()$;
7          if $r(u) > r(v)$ then continue;
           // Check if all $u$-$v$ shortest paths are monotonic
8          if Check($u$, $v$, $dist(v,u)$) then
9              Insert $(v, h(u))$ into $\mathsf{L}^{\mathsf{MLL}}(u)$;
10         for $w \in N(u)$ do
11             if $dist(v, w) = \infty$ then
12                 $dist(v, w) \leftarrow dist(v, u) + 1$; $Q.push(w)$;
13             if $dist(v, w) = dist(v, u) + 1$ and $dist(v, w) > 1$ then
14                 $h(w) \leftarrow \text{argmax}_{x \in \{u, h(u), h(w)\}} r(x)$;
15 return $\mathsf{L}^{\mathsf{MLL}}$

16 Procedure Check($u$, $v$, $d$)
17     if $u \in C$ then $\mathsf{L}(u) \leftarrow$ the core label of $x$ from the CTL index;
18     if $u \notin C$ then $\mathsf{L}(u) \leftarrow X(u)$;
19     if $v \notin \mathsf{L}(u)$ then return False;
20     for $w \in \mathsf{L}(u) \setminus \{u, v\}$ do
21         $dist(u, w), dist(w, v) \leftarrow$ query by $\mathsf{L}^{\mathsf{CTL}}$;
22         if $d = dist(u, w) + dist(w, v)$ return False;
23     return True;
```

4.3 Index Construction

We create the MLL index $\mathsf{L}^{\mathsf{MLL}}$ based on the label condition given by Theorem 4.6, which states that vertex v is added to the label of vertex u as the landmark if all v-u shortest paths are monotonic and $r(v) > r(u)$. If we can check whether all v-u shortest paths are monotonic, then indexing vertex v becomes a process of adding v to lower-order vertices u. This process can be completed using a v-sourced BFS. Since there is no dependency between the BFSs of different vertices, the indexing process of all vertices can be executed in parallel. Once all vertices complete BFS for the indexing process, $\mathsf{L}^{\mathsf{MLL}}$ is created.

Indexing Algorithm. Algorithm 3 describes how to create $\mathsf{L}^{\mathsf{MLL}}$ in parallel for each vertex v. First, each vertex v is inserted into the queue Q, and a v-sourced BFS begins (Line 2). We initialize the distances from v to all vertices, and set the highest-order inner vertex $h(u)$ on all v-u shortest paths as nil (denoted by "–") (Line 3-4). Then, we pop an element u from Q (Line 6). If the order of u is higher than v, then the expansion from u is pruned (Line 7). Otherwise, we need to check if all v-u shortest paths are monotonic. We use Procedure $\mathsf{Check}(u, v, dist(v, u))$ for this purpose, which will be described later. If True, v is added as a landmark to the label of u (Line 8-9). For each unvisited neighbor vertex $w \in N(u)$ of u, we update $dist(v, w)$ and add w to Q (Line 11-12). If $dist(v, w) + 1 = dist(v, u)$, we set $h(w)$ to the one x with the highest order in u, $h(u)$, and $h(w)$, i.e., $\text{argmax}_{x \in \{u, h(u), h(w)\}} r(x)$ (Line 15-16).

Example 4.16. Consider the graph G in Fig. 1. We run in parallel to index each vertex in G. Taking v_3 as an example. v_3 first adds itself to Q. Then, v_3 is popped from Q and we push v_3's neighbors $N(v_3) = \{v_1, v_2, v_4, v_8, v_9\}$ into Q. Next, v_1 and v_2 is popped and pruned. Then, v_4 is popped, and v_3 is added as a landmark of v_4 since $r(v_4) < r(v_3)$ and all v_4-v_3 shortest paths are monotonic. Next, v_8 is popped, and v_3 is added as a landmark of v_8; meanwhile, we insert $v_6 \in N(v_8)$ to Q and update $h(v_6) = \text{argmax}_{x \in \{v_8, h(v_8) = -, h(v_6) = -\}} r(x) = v_8$. After Q becomes empty, we stop.

Procedure $\mathsf{Check}(u, v, d)$. Next, we introduce Procedure $\mathsf{Check}(u, v, d)$ (Line 16-23), where $r(v) > r(u)$, $d = dist(v, u)$. The purpose is to examine if all v-u shortest paths are monotonic. A simple way is to enumerate all u-v shortest paths, and then compare the order of each inner vertex with u. However, this approach is inefficient, and we instead use the CTL index to speed up the checking. Given the core-tree decomposition T_G (under parameter d) of graph G, CTL uses PLL to assign a core label to each vertex of the core part C to form the core index.

The CTL index is sufficient to check if all v-u shortest paths are monotonic. We first determine whether vertex u belongs to the core part C, and if so, we set $\mathsf{L}(u)$ as the core label of u from the CTL index (Line 17). If u does not belong to C, we find the corresponding tree node $X(u)$ in T_G and assign $X(u)$ to $\mathsf{L}(u)$ (Line 18). If v is not in $\mathsf{L}(u)$, we return False, i.e., not all v-u shortest paths are monotonic (Line 19). Otherwise, we enumerate the vertices w in $\mathsf{L}(u)$, where $w \neq u, w \neq v$, and obtain distances $dist(u, w)$ and $dist(w, v)$ by querying the CTL index (Line 20-21). If there is a vertex $w \in \mathsf{L}(u) \setminus \{u, v\}$ on the u-v shortest path (i.e., $dist(u, w) + dist(w, v) = dist(u, v)$), then False is returned (Line 22); otherwise True is returned (Line 23).

Example 4.17. Consider the graph G in Fig. 1, where the core-tree decomposition ($d = 2$) is in Fig. 2. For $\mathsf{Check}(v_8, v_4, 2)$, since $v_8 \notin C$, $X(v_8) = \{v_6, v_3\}$ is assigned to $\mathsf{L}(v_8)$. As $v_4 \notin \mathsf{L}(v_8)$, we return False. For $\mathsf{Check}(v_2, v_1, 1)$, since $v_2 \in C$, the label $\{(v_1, 0), (v_2, 0)\}$ of v_2 from the core index is assigned to $\mathsf{L}(v_2)$. As $v_1 \in \mathsf{L}(v_2)$ and $\nexists w \in \mathsf{L}(v_2) \setminus \{v_1, v_2\}$ on the v_1-v_2 shortest path, True is returned.

Lemma 4.18. *$\mathsf{Check}(u, v, d)$ correctly checks if all u-v shortest paths are monotonic ($r(v) > r(u)$).*

Lemma 4.19. *Algorithm 3 correctly creates the* MLL *index.*

Lemma 4.20. *Algorithm 3 requires $O(\Delta^{\mathsf{CTL}} \times |\mathsf{L}^{\mathsf{CTL}}|)$ shortest-distance queries, where Δ^{CTL} is* CTL*'s maximum label size.*

5 EXTENSION OF MLL

In this section, we generalize our method MLL to directed graphs. In Section 4, we assumed that the complex network is undirected for MLL. MLL can also be extended to support shortest-path queries on directed graphs. Given a directed graph $G(V, E)$, if $(u, v) \in E$, then u is an in-neighbor of v and v is an out-neighbor of u.

Index construction. MLL relies on the CTL index, but building the CTL index on directed graphs is challenging due to performing core-tree decomposition on directed graphs. There are two differences when *decomposing a directed graph* compared to an undirected graph. (1) We perform the decomposition using minimum degree elimination (MDE) [8], which iteratively finds the vertex v with the smallest degree (the total number of in-/out-neighbors) in a graph. In eliminating the vertex v, we connect *any* two neighbors u, w of v by directed edges, set the edge weight, and remove v using similar methods to that of undirected graphs; the other parts are the same. (2) For each vertex u in the tree node $X(v)$, we need to store two shortest distances: the forward distance $dist_+(u, v)$ from u to v and the backward distance $dist_-(u, v)$ from v to u.

When the graph is decomposed, we get a directed graph G_λ, and we can use PLL to create a forward label $\mathsf{L}_+^{\mathsf{CTL}}(u)$ for each vertex $u \in C$. Then, the direction of the edges of G_λ is reversed to get a **reverse graph**, and we create a backward label $\mathsf{L}_-^{\mathsf{CTL}}(u)$ for each $u \in C$ by using PLL on the reverse graph of G_λ. As for the tree index, for each landmark v of u, we calculate the distance from v to u as a forward label and the distance from u to v as a backward label. Similarly, for our MLL index, we use Algorithm 3 to create a forward label $\mathsf{L}_+^{\mathsf{MLL}}(u)$ for each vertex $u \in V$ on the original graph G and a backward label $\mathsf{L}_-^{\mathsf{MLL}}(u)$ on the reverse graph of G.

Query processing. To perform shortest-path queries on directed graphs, we can use a similar way as on undirected graphs (by Algorithm 2). One thing to note is that we need to consider whether to use forward or backward labels for each vertex. For example, when we need to obtain the shortest path from s to t, we should use the forward label $\mathsf{L}_+^{\mathsf{MLL}}(s)$ for s and the backward label $\mathsf{L}_-^{\mathsf{MLL}}(t)$ for t. Algorithm 2 also needs the support of shortest-distance queries using CTL, which can be adapted similarly on directed graphs.

6 EXPERIMENTS

Algorithms. We intend to conduct thorough experimental comparisons of traversal-based and extension-based approaches for dealing with shortest-path queries on complex networks.

- Traversal-based: we select and implement four representative methods, all of which need graph traversal to process queries.
  - BFS. We start a BFS from s until t is met to process $\mathsf{Q_P}(s, t)$.
  - BiBFS. Bidirectional BFS, i.e., search the path from both the s and t sides to process $\mathsf{Q_P}(s, t)$.
  - $\mathsf{PLL_B}$. This method uses both the index and graph traversal. To reduce the PLL index size, we construct a partial PLL by only constructing labels within a specific distance (≤ 5 in this paper) and ignoring others with larger distance values. This partial PLL index is extended to support shortest-path queries. When dealing with $\mathsf{Q_P}(s, t)$, if the partial PLL index finds $dist(s, t) \leq 5$, then the s-t shortest path can be returned using Algorithm 1; otherwise, the query is handled by BiBFS.
  - $\mathsf{CTL_B}$. We use CTL as preprocessing to speed up BFS, that is, when processing $\mathsf{Q_P}(s, t)$, the CTL index can provide distance information to determine whether a vertex w is on the s-t shortest path, i.e., whether $dist(s, t) = dist(s, w) + dist(w, t)$. Vertices not on the s-t shortest path can be pruned directly.
- Extension-based: we select the extended PLL and CTL introduced in Section 3 and MLL introduced in Section 4, all of which use only indexes to process queries.
  - $\mathsf{PLL_E}$. We use the extended PLL index, i.e., add an extra attribute to each index entry, for query processing. The query method is in Algorithm 1. We use the parallel version of PLL to speed up index creation, whose source code is provided by the authors of [25].
  - $\mathsf{CTL_E}$. We use the extended CTL index, i.e., add an extra attribute to each index entry, for query processing. The query processing method is introduced in Section 3.2.2. The index construction of CTL can be accelerated using multiple cores. The source code of CTL is provided by the authors of [26].
  - MLL. MLL uses both the CTL index and the MLL index for query processing (see Algorithm 2). The index of MLL can be constructed in parallel, and the construction algorithm is presented in Algorithm 3.

Table 2: Dataset Description

	Datasets	n	m	Type	Diameter	$dist_{avg}$	Deg_{avg}
DELI	Delicious[4]	536,109	1,365,961	Social	14	5.16	5.10
DIGT	DIGT[4]	4,000,151	8,649,016	Social	15	7.81	4.32
FRIE	Friendster[4]	8,658,745	55,170,227	Social	25	5.37	12.74
STAC	Stack[4]	6,024,271	63,497,050	Interaction	11	3.86	21.08
LIVE	LiveJournal[5]	5,363,260	79,023,142	Social	20	5.45	29.47
FACE	Facebook[4]	58,790,783	92,208,195	Social	24	7.25	3.14
TWIT	Soc-Twitter[4]	21,297,772	265,025,809	Social	26	4.87	24.89
SK05	SK-2005[5]	50,636,154	1,949,412,601	Web	40	5.20	77.00
UK06	UK-2006[5]	77,741,046	2,965,197,340	Web	42	6.16	76.28
UK07	UK-2007[5]	133,633,040	5,507,679,822	Web	257	6.22	82.43

We implemented all the algorithms using C++ and compiled them using GNU GCC 4.8.5 and -O3 level optimizations. We use OpenMP to support the implementation of the parallel algorithms. All experiments were conducted on a machine with 64 CPU cores and 500 GB main memory running Linux (Red Hat Linux 4.8.5, 64bit). Each CPU core is Intel Xeon 2.4GHz.

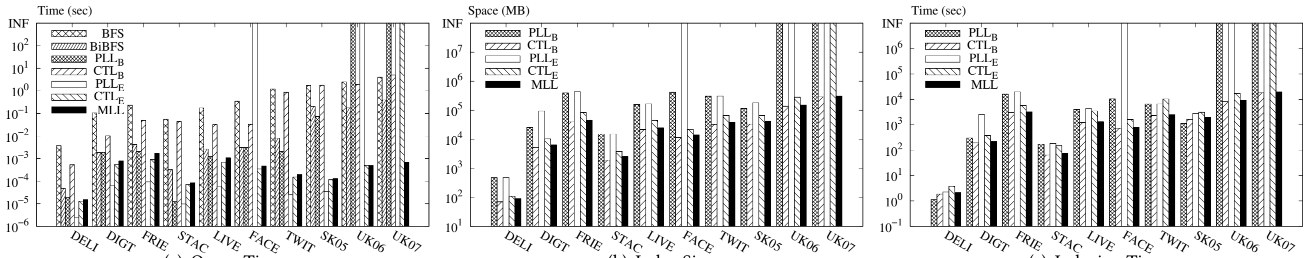
Datasets. We ran experiments on 10 real-world graphs, whose details are given in Table 2. The largest graph has over 5.5 billion edges. These graphs are small-world graphs, most of which have diameters (longest shortest distances) less than 50, and the average distance between two vertices of all graphs, i.e., $dist_{avg}$, is less than 10. The average degree of these graphs, Deg_{avg}, varies between 3.14 and 82.43. The dataset comes from various complex networks, including social networks, web graphs, and interaction networks. All graphs were downloaded from Network Repository[4][34] and Laboratory for Web Algorithms[5][10].

Summary of Findings. For traversal-based methods (i.e., BFS, BiBFS, $\mathsf{PLL_B}$, and $\mathsf{CTL_B}$), BFS and BiBFS can process queries without building indexes, but their query speed is slow. $\mathsf{CTL_B}$ tries to use the CTL index to accelerate BFS, but it cannot guarantee query time. $\mathsf{PLL_B}$ can speed up the query process by creating a partial PLL index, but it still cannot guarantee query time for shortest-path queries with long distances. Thus, the traversal-based approaches are only applicable when the query speed is not so demanding while the space budget is low.

On the other hand, extension-based methods (i.e., $\mathsf{PLL_E}$, $\mathsf{CTL_E}$, and MLL) use the pre-computed index for query processing, and their query speed is much faster than traversal-based methods since they avoid graph traversal at query time. Moreover, the three extension-based methods make a different trade-off between query time and space cost: among them, MLL has the smallest index size and $\mathsf{PLL^E}$ has the largest index size, while the index size of CTL is in between; MLL has the slowest query speed and $\mathsf{PLL^E}$ has the fastest query speed, while the query speed of CTL is in between.

Ex-1: Query Time Comparison. We compare the query time of all methods. For the approach using indexes for query processing, we set the query time to "INF" if the index cannot be built. We

[4]https://networkrepository.com/networks.php
[5]https://vigna.di.unimi.it



(a) Query Time (b) Index Size (c) Indexing Time

Figure 3: The Comparison of Different Methods

generated 1000 random queries and obtained the average query processing time. We show the results in Fig. 3(a).

Comparison among extension-based methods. Since the extension-based methods (i.e., $\mathsf{PLL_E}$, $\mathsf{CTL_E}$, and MLL) do not rely on graph traversal, they can process queries very quickly: all of them handle shortest-path queries within two milliseconds. Among them, $\mathsf{PLL_E}$ has the fastest query speed, and the query time of $\mathsf{PLL_E}$ is on average 10.53 times shorter than that of MLL. The query speed of MLL is comparable to that of $\mathsf{CTL_E}$, and the query time of $\mathsf{CTL_E}$ is on average 1.94 times shorter than that of MLL. Considering the performance of extension-based methods in query processing, they are suitable for applications requiring high query speed.

Comparison with traversal-based methods. Due to the inevitable need for graph traversal, the query speed of traversal-based methods (i.e., BFS, BiBFS, $\mathsf{PLL_B}$, and $\mathsf{CTL_B}$) is much slower than that of extension-based methods. We take MLL as a representative to compare extension-based methods with traversal-based methods.

(1) Comparison with BFS and BiBFS. The query time of BFS is, on average, 3265.86 times longer than MLL and up to four orders of magnitude longer than MLL. BiBFS uses bi-directional search to reduce the overhead of graph traversal compared to BFS, but BiBFS still takes a long time to process queries: BiBFS is on average 254 times and up to three orders of magnitude slower than MLL.

(2) Comparison with $\mathsf{PLL_B}$. When processing a query, if the distance between two vertices is short, $\mathsf{PLL_B}$ can use the index to avoid traversing the graph. However, $\mathsf{PLL_B}$ cannot totally avoid graph traversal, which leads to the query time of $\mathsf{PLL_B}$ being 102.46 times longer than that of MLL on average.

(3) Comparison with $\mathsf{CTL_B}$. $\mathsf{CTL_B}$ narrows the search space of BFS by distance queries, so $\mathsf{CTL_B}$ is faster than BFS on some graphs; for example, on DELI, the query time of $\mathsf{CTL_B}$ is 0.14 times that of BFS. But distance queries used by $\mathsf{CTL_B}$ are not free; for example, on UK07, $\mathsf{CTL_B}$ takes 1.25 times longer than BFS. $\mathsf{CTL_B}$ is also much slower than MLL: MLL is on average 3027.45 times and at most four orders of magnitude faster than $\mathsf{CTL_B}$.

Ex-2: Index Size Comparison. There are five methods that require the use of indexes (including $\mathsf{PLL_B}$, $\mathsf{CTL_B}$, $\mathsf{PLL_E}$, $\mathsf{CTL_E}$, and MLL) for query processing. We compare the index size of these five methods and present the results in Fig. 3(b).

Index Size of $\mathsf{PLL_E}$, $\mathsf{CTL_E}$, *and* MLL. Among extension-based methods, $\mathsf{PLL_E}$ has the largest index, while MLL has the smallest index.

(1) The total size of the indexes (including the CTL and the MLL indexes) used by MLL is 6.9 times smaller than the size of indexes used by $\mathsf{PLL_E}$. Because of the oversized indexes, $\mathsf{PLL_E}$ cannot handle large graphs such as FACE, TWIT, and UK07.

(2) Both MLL and $\mathsf{CTL_E}$ are extended based on CTL to support path queries. For $\mathsf{CTL_E}$, the extra space brought by the extension is 0.96 times that of the original CTL index, while the size of the extra space (i.e., the MLL index) required by MLL is 0.2 times that of the original CTL index. Also, due to the difference in the extra space used, $\mathsf{CTL_E}$ cannot handle graph UK07 while MLL can.

Index Size of $\mathsf{PLL_B}$. By limiting the distance in the labels, $\mathsf{PLL_B}$ constructs a partial PLL index. Thus, the index size of $\mathsf{PLL_B}$ is 0.82 times that of $\mathsf{PLL_E}$. However, the index size of $\mathsf{PLL_B}$ is on average 8.05 times[6] that of MLL, and $\mathsf{PLL_B}$ cannot handle large graphs such as FACE and UK07. This shows that even building a partial PLL index still requires a much larger index size than MLL.

Index Size of $\mathsf{CTL_B}$. Instead of extending the CTL index, $\mathsf{CTL_B}$ uses the original CTL index for distance queries to reduce the search range of BFS. $\mathsf{CTL_B}$ has a smaller index size than the extension-based approaches. However, the total index size of MLL is only 1.2 times that of $\mathsf{CTL_B}$, indicating that MLL does not add significantly extra space to the original CTL index.

Ex-3: Indexing Time Comparison. We compare the indexing time of five methods (including $\mathsf{PLL_B}$, $\mathsf{CTL_B}$, $\mathsf{PLL_E}$, $\mathsf{CTL_E}$, and MLL) that require the use of indexes for query processing. We show the results in Fig. 3(c).

Indexing Time of $\mathsf{PLL_E}$, $\mathsf{CTL_E}$, *and* MLL. Among three extension-based methods, MLL has the shortest indexing time: the total indexing time of MLL (including the time to build the CTL index and the MLL index) is on average 4.06 times shorter than that of $\mathsf{PLL^E}$, and also on average 2.15 times shorter than that of $\mathsf{CTL^E}$.

Indexing Time of $\mathsf{PLL_B}$. On the graphs that $\mathsf{PLL_E}$ can index, $\mathsf{PLL_B}$ is 2.44 times faster than $\mathsf{PLL_E}$: $\mathsf{PLL_B}$ only needs to build a partial PLL index while $\mathsf{PLL_E}$ needs a complete one. However, the indexing time of $\mathsf{PLL_B}$ is on average 1.72 times and 3.6 times longer than that of $\mathsf{CTL_E}$ and MLL, respectively.

Indexing Time of $\mathsf{CTL_B}$. $\mathsf{CTL_B}$ only needs to build the CTL index, while $\mathsf{CTL_E}$ needs to add an extra attribute to each entry of the CTL index, which causes the indexing time of $\mathsf{CTL_E}$ to be 2.41 times longer than that of $\mathsf{CTL_B}$. MLL also needs to create the CTL index first. Still, the additional building of lightweight indexes results in the indexing time of MLL being only 1.12 times that of $\mathsf{CTL_B}$, indicating that MLL does not incur much additional indexing time cost to support shortest-path queries.

Ex-4: Test of Query Time at Different Distance Ranges. We test the performance of all methods in handling queries in different distance ranges. We randomly generate five sets of queries Q =

[6]There is no conflict with former results as $\mathsf{PLL_B}$ can index FACE while $\mathsf{PLL_E}$ cannot.

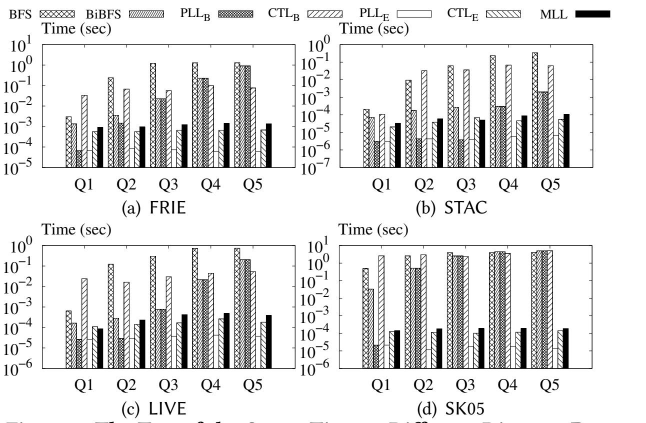


Figure 4: The Test of the Query Time at Different Distance Ranges

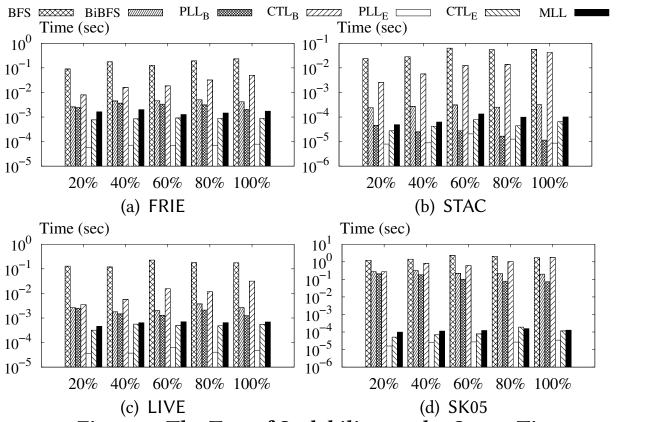


Figure 5: The Test of Scalability on the Query Time

$\{Q_1, Q_2, Q_3, Q_4, Q_5\}$, where each set $Q_i \in Q$, $i \in [1, 5]$, consists of 1000 random queries. For each query $Q_P(s, t) \in Q_i$, we control the shortest distance between s and t located in the range between $\frac{D}{5} \times (i-1)$ and $\frac{D}{5} \times i$, where D is the diameter of the graph. We report the average time for answering queries in each set Q_i. Since various graphs have similar conclusions, we only show the results for FRIE, STAC, LIVE, and SK05 in Fig. 4.

Effect of distance on query time. For all methods, the time to answer queries in Q_1 tends to be shorter, while the time to answer queries in Q_5 tends to be longer. For example, on graph FRIE, MLL takes 1.48 times longer to process queries in Q_5 than in Q_1; BFS takes 433.77 times longer to process queries in Q_5 than in Q_1. One reason for this trend is that a longer path always means examining more labels or visiting more graph vertices to find the path. It is worth noting that the query time for $\mathsf{PLL_B}$ increases dramatically as the query distance increases. This is because $\mathsf{PLL_B}$ can use indexes to answer queries when the distance is less than a certain value (we set it to 5), whereas graph traversal is required for larger distances.

Comparison of traversal-based and extension-based Methods. The extension-based methods are faster than the traversal-based methods in processing queries in any Q_i on all graphs. Taking MLL as an example, MLL is on average 1511.34, 44.95, 44.95 and 91.75 times faster than BFS, BiBFS, $\mathsf{PLL_B}$ and $\mathsf{CTL_B}$ in processing queries in Q_4 of LIVE.

Ex-5: Scalability Test on Query Time. We test the scalability of all methods. To do this, we randomly divide the edges E of graph $G(V, E)$ into five groups with equal size and then generate five test graphs such that the i-th test graph contains the first i groups of edges. Thus, the five test graphs include 20%, 40%, 60%, 80%, 100% of the edges in G, respectively. We conduct experiments on each of the five test graphs.

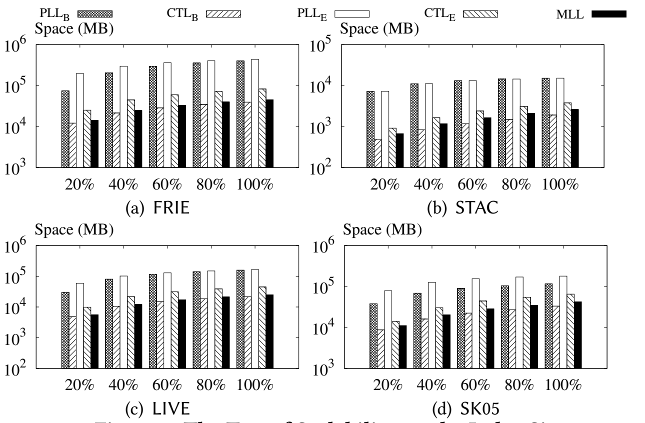


Figure 6: The Test of Scalability on the Index Size

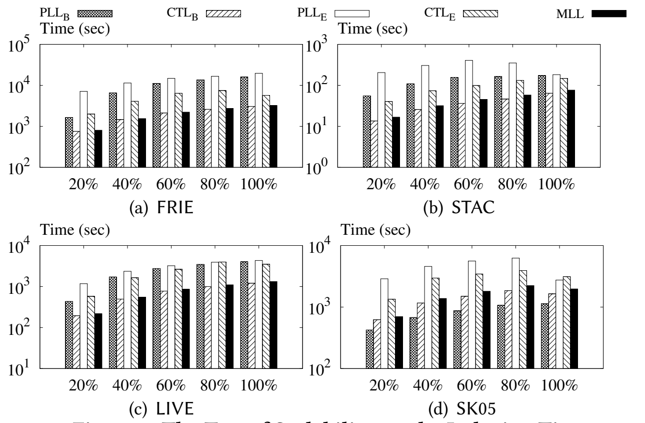


Figure 7: The Test of Scalability on the Indexing Time

We first evaluate the graph size against the query time of all methods. Since various graphs have similar conclusions, we only present the results for FRIE, STAC, LIVE, and SK05 in Fig. 5. We find that, as the graph size increases, the query time of some methods shows an increasing trend. For example, on STAC, for $\mathsf{CTL_B}$, the query time on the test graphs containing 40%, 60%, 80% and 100% edges is 2.2, 4.94, 5.33 and 16.77 times longer than the query time on the test graph with 20% edges. Yet, there is not just an upward trend observed; for example, on STAC, for MLL, the query time on the test graph containing 60% edges is 1.76 times longer than that on the test graph with 80% edges.

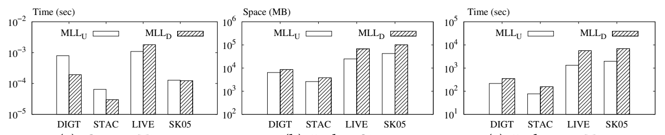
The reasons for the query time fluctuations are manifold. First, most of the query complexity is related to the graph scale, and a large graph generally implies an increase in complexity; however, query processing is also related to other factors, such as graph density and graph diameter. According to [24], the graph diameter decreases as the number of edges increases. Thus, the query time shows a fluctuating trend under the interaction of various factors.

Ex-6: Scalability Test on Index Size. We investigate the effect of the graph size on the index size. We use the same experimental setup as Ex-5 and compare the index size of the five methods (including $\mathsf{PLL_B}$, $\mathsf{CTL_B}$, $\mathsf{PLL_E}$, $\mathsf{CTL_E}$, and MLL) that use indexes for query processing. The results are given in Fig. 5.

We find that the index size of all the five methods increases as the graph size grows. For example, on FRIE, the index built by MLL on the test graph containing 40% edges is 1.75 times larger than the index built on the test graph containing 20% edges, while the index built by MLL on the test graph with 100% edges is 3.23 times larger than the index built on the test graph with 20% edges.

Ex-7: Scalability Test on Indexing Time. We next study the effect of the graph size on the indexing time of the five methods (including $\mathsf{PLL_B}$, $\mathsf{CTL_B}$, $\mathsf{PLL_E}$, $\mathsf{CTL_E}$, and MLL) that rely on indexes for query processing. The experiments use the same settings as Ex-5, and we report the results in Fig. 7.

It can be found from Fig. 7 that the indexing time of all the five methods increases as the graph size increases. Taking MLL as an example, on graph FRIE, the indexing time of MLL is 1.92, 2.77, 3.41, and 4.02 times longer on test graphs containing 40%, 60%, 80%, and 100% edges, respectively, than on the test graph containing 20% edges. Similar phenomena can be observed in other methods.



(a) Query Time (b) Index Size (c) Indexing Time

Figure 8: The Performance of MLL on Directed Graphs

Ex-8: Test of MLL on Directed Graphs. Section 5 introduces how to extend MLL to directed graphs. To test the effectiveness of the extension, we run experiments on four real datasets (DIGT, STAC, LIVE, and SK05). Note that these graphs used are directed, while in the previous experiments, we ignored edge directions to create undirected graphs. The original MLL method is called $\mathsf{MLL_U}$ since it works with undirected graphs; the extended MLL method is called $\mathsf{MLL_D}$ since it works with directed graphs. The results are in Fig. 8.
Query time. The query time of $\mathsf{MLL_D}$ is normally faster than $\mathsf{MLL_U}$, for example, on DIGT, the average query time of $\mathsf{MLL_D}$ is 4.13 times shorter than that of $\mathsf{MLL_U}$. One reason could be that when querying on directed graphs, we only use the labels in one direction (and ignore the labels in the opposite direction). However, due to the randomness of queries and the larger index size, $\mathsf{MLL_D}$ may be slower than $\mathsf{MLL_U}$ in some cases; for example, on LIVE, the average query time of $\mathsf{MLL_D}$ is 1.66 times longer than $\mathsf{MLL_U}$.
Index size. The index size of $\mathsf{MLL_D}$ is generally larger than that of $\mathsf{MLL_U}$: the average index size of $\mathsf{MLL_D}$ is 1.98 times larger than that of $\mathsf{MLL_U}$. One possible explanation for this result is that forming a path in a directed graph is more difficult (a path in a directed graph must be a path in the corresponding version of an undirected graph, and the reverse does not hold). When indexing a directed graph, pruning the index using existing path information is more difficult, resulting in a large index size.
Indexing time. The indexing time for $\mathsf{MLL_D}$ is generally longer than that for $\mathsf{MLL_U}$: on the four graphs used, the indexing time of $\mathsf{MLL_D}$ is on average 2.87 times longer than that of $\mathsf{MLL_U}$.

7 RELATED WORK

Shortest-Path Queries. Traversal-based methods (e.g., BFS [7] on unweighted graphs or Dijkstra [22] on weighted graphs) answer shortest-path queries with a runtime linearly proportional to the graph size. To speed up online traversal, numerous heuristic or preprocessing techniques have been proposed, such as A^* search [17, 19], Highway Hierarchies [36], and Contraction Hierarchies [16]. A survey of the speedup techniques can be found in [38]. However, the above techniques often rely on the properties of road networks, rendering them inapplicable for complex networks. Some studies concentrate on problems close to shortest-path queries, such as shortest-path counting [44] and shortest-path-graph queries [39]. However, to our knowledge, there is no systematic work[7] yet on how to effectively implement (and compare) methods for shortest-path queries on complex networks, which is the focus of this work.
Shortest-Distance Queries. Shortest-distance queries, as an operation closely related to shortest-path queries, have been well studied. Many popular methods create offline indexes to accelerate online query processing. However, since creating indexes with the minimum size is NP-hard [12], subsequent work has focused on designing practical algorithms. [14] created indexes with the help of independent sets. When a graph cannot be loaded into memory, [21] considered using external memory to create the index. PLL was designed to create indexes using the pruned BFS [3]. Li et al. [25] used multi-threading to reduce the indexing time of PLL, while CTL [26] was proposed to reduce the index size of PLL.
(Core-)Tree Decomposition. Tree decomposition was first studied in [18]. [6] proved that determining whether a graph's treewidth exceeds a given value is NP-complete. Several heuristics for tree decomposition are listed in [42], including the widely used MDE heuristic [8]. Tree decomposition was used to handle shortest-path/shortest-distance queries on road networks, e.g., in [40] and [29]. However, for complex networks, tree decomposition may not be applicable due to the presence of large treewidth [2, 26]. Therefore, core-tree decomposition [4, 26, 28] is proposed. Core-tree decomposition has been used to handle shortest-distance queries in complex networks [4, 26]. However, how to efficiently handle shortest-path queries with core-tree decomposition remains open.

8 CONCLUSION

This paper studies shortest-path queries on complex networks. The distance query processing methods PLL and CTL are extended to support shortest-path queries. To reduce the space cost required for extensions, MLL is proposed. MLL is also adapted for directed graphs. Extensive experiments are conducted to investigate the performance of various methods in answering shortest-path queries. The experimental results can help practitioners choose the appropriate method for a specific application.

ACKNOWLEDGMENTS

Long Yuan is supported by NSFC61902184, NSF of Jiangsu Province BK20190453, and Science and Technology on Information Systems Engineering Laboratory WDZC20205250411. Lu Qin is supported by ARC FT200100787 and ARC DP210101347. Ying Zhang is supported by ARC FT170100128 and ARC DP210101393. Lijun Chang is supported by ARC FT180100256 and ARC DP220103731.

[7]The literature [3] briefly mentions how to extend PLL for shortest-path queries, but no specific code and correctness analysis are given.

REFERENCES

[1] Ittai Abraham, Amos Fiat, Andrew V Goldberg, and Renato F Werneck. 2010. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 782–793.

[2] Aaron B Adcock, Blair D Sullivan, and Michael W Mahoney. 2013. Tree-like structure in large social and information networks. In *2013 IEEE 13th International Conference on Data Mining*. IEEE, 1–10.

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

[4] Takuya Akiba, Christian Sommer, and Ken-ichi Kawarabayashi. 2012. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *Proceedings of the 15th International Conference on Extending Database Technology*. 144–155.

[5] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys (CSUR)* 40, 1 (2008), 1–39.

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

[7] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10.

[8] Anne Berry, Pinar Heggernes, and Genevieve Simonet. 2003. The minimum degree heuristic and the minimal triangulation process. In *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 58–70.

[9] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. 2006. Complex networks: Structure and dynamics. *Physics reports* 424, 4-5 (2006), 175–308.

[10] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*. 595–602.

[11] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. 2012. The exact distance to destination in undirected world. *The VLDB Journal* 21, 6 (2012), 869–888.

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

[13] L da F Costa, Francisco A Rodrigues, Gonzalo Travieso, and Paulino Ribeiro Villas Boas. 2007. Characterization of complex networks: A survey of measurements. *Advances in physics* 56, 1 (2007), 167–242.

[14] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. 2013. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *Proceedings of the VLDB Endowment* 6, 6 (2013), 457–468.

[15] Jun Gao, Ruoming Jin, Jiashuai Zhou, Jeffrey Xu Yu, Xiao Jiang, and Tengjiao Wang. 2011. Relational Approach for Shortest Path Discovery over Large Graphs. *Proceedings of the VLDB Endowment* 5, 4 (2011).

[16] 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*. Springer, 319–333.

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

[18] Rudolf Halin. 1976. S-functions for graphs. *Journal of geometry* 8, 1-2 (1976), 171–186.

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

[20] Jihye Hong, Kisung Park, Yongkoo Han, Mostofa Kamal Rasel, Dawanga Vonvou, and Young-Koo Lee. 2017. Disk-based shortest path discovery using distance index over large dynamic graphs. *Information Sciences* 382 (2017), 201–215.

[21] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Yanyan Xu. 2014. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *arXiv preprint arXiv:1403.0779* (2014).

[22] Donald B Johnson. 1973. A note on Dijkstra's shortest path algorithm. *Journal of the ACM (JACM)* 20, 3 (1973), 385–388.

[23] Lawrence Kou, George Markowsky, and Leonard Berman. 1981. A fast algorithm for Steiner trees. *Acta informatica* 15, 2 (1981), 141–145.

[24] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 177–187.

[25] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling distance labeling on small-world networks. In *Proceedings of the 2019 International Conference on Management of Data*. 1060–1077.

[26] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2020. Scaling up distance labeling on graphs with core-periphery properties. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1367–1381.

[27] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. 2017. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment* 11, 4 (2017), 445–457.

[28] Takanori Maehara, Takuya Akiba, Yoichi Iwata, and Ken-ichi Kawarabayashi. 2014. Computing personalized pagerank quickly by exploiting graph structures. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1023–1034.

[29] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data*. 709–724.

[30] Romualdo Pastor-Satorras and Alessandro Vespignani. 2007. *Evolution and structure of the Internet: A statistical physics approach*. Cambridge University Press.

[31] Syed Asad Rahman, P Advani, R Schunk, Rainer Schrader, and Dietmar Schomburg. 2005. Metabolic pathway analysis web service (Pathway Hunter Tool at CUBIC). *Bioinformatics* 21, 7 (2005), 1189–1193.

[32] Syed Asad Rahman and Dietmar Schomburg. 2006. Observing local and global properties of metabolic pathways:'load points' and 'choke points' in the metabolic networks. *Bioinformatics* 22, 14 (2006), 1767–1774.

[33] Royi Ronen and Oded Shmueli. 2009. SoQL: A language for querying and creating data in social networks. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 1595–1602.

[34] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

[35] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.

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

[37] Yuxuan Shi, Gong Cheng, and Evgeny Kharlamov. 2020. Keyword search over knowledge graphs via static and dynamic hub labelings. In *Proceedings of The Web Conference 2020*. 235–245.

[38] Christian Sommer. 2014. Shortest-path queries in static networks. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–31.

[39] Ye Wang, Qing Wang, Henning Koehler, and Yu Lin. 2021. Query-by-Sketch: Scaling Shortest Path Graph Queries on Very Large Networks. In *Proceedings of the 2021 International Conference on Management of Data*. 1946–1958.

[40] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 99–110.

[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. *arXiv preprint arXiv:1201.6564* (2012).

[42] Jinbo Xu, Feng Jiao, and Bonnie Berger. 2005. A tree-decomposition approach to protein structure prediction. In *2005 IEEE Computational Systems Bioinformatics Conference (CSB'05)*. IEEE, 247–256.

[43] Junhua Zhang, Wentao Li, Long Yuan, Lu Qin, Ying Zhang, and Lijun Chang. 2022. *Technical Report*. https://www.dropbox.com/sh/gv96k90xz02fopd/AACYG6d6FftmNiWzzT1JflKya?dl=0

[44] Yikai Zhang and Jeffrey Xu Yu. 2020. Hub Labeling for Shortest Path Counting. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1813–1828.