PCSP: Efficiently Answering Label-Constrained Shortest Path Queries in Road Networks

Libin Wang
The Hong Kong University of Science and Technology
lwangct@cse.ust.hk

Raymond Chi-Wing Wong
The Hong Kong University of Science and Technology
raywong@cse.ust.hk

ABSTRACT

Shortest path queries are ubiquitous in many spatial applications. Existing solutions assign numerical weights to edges and compute the path with the minimum sum of edge weights. However, in practice, the road categories associated with edges (e.g., toll) can make shortest paths undesirable, e.g., they may use unfavorable toll roads. Augmenting each edge with a label to denote its category, we study the *Label-Constrained Shortest Path* (LCSP) query that finds the shortest path under the constraint that the edge labels along the path should follow a pattern expressed by a formal language. There have been extensive LCSP solutions, but they are either inefficient in query processing or limited to special languages with low expressiveness capacity. In this paper, we propose the index called *Partially Constrained Shortest Path* (PCSP), which answers each query quickly by concatenating two shortest paths that partially satisfy the constraint and support more general regular languages. We also present pruning techniques that further optimize query efficiency. Experimental comparison with the state-of-the-art index demonstrates the superiority of PCSP. It can answer each LCSP query in around 100 microseconds and runs faster than the best-known solution by up to two orders of magnitude.

PVLDB Reference Format:
Libin Wang and Raymond Chi-Wing Wong. PCSP: Efficiently Answering Label-Constrained Shortest Path Queries in Road Networks. PVLDB, 17(11): 3082 - 3094, 2024.
doi:10.14778/3681954.3681985

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lbwang95/PCSP/.

1 INTRODUCTION

The point-to-point shortest path query is one of the fundamental operations in road networks and is widely used in GPS navigation, online car-hailing and urban planning. Based on appropriate numerical weights assigned to each edge (representing road segments), it provides users with the shortest path by minimizing the sum of weights of the traversed edges. However, these shortest paths regard all edges as the same and ignore different road categories behind edges (e.g., highways, toll roads and rural roads). In most cases, users are more concerned with shortest paths that use specific types of roads following a flexible and realistic predefined *pattern*. For example, commercial navigation products often plan to use highways in the continuous middle part of a route. Environmentalists are interested in shortest paths that combine several public transport modes (e.g., shared bikes and subways). Augmenting each edge with an additional "label" (apart from the numerical edge weight) to represent its road category, we study the *Label-Constrained Shortest Path* (LCSP) query that finds the shortest paths of desired patterns.

LCSP is formalized in the context of formal languages [7]. Specifically, all edge labels form the *alphabet* Σ, and the path label concatenated by the edge labels along the path can be seen as a *word*. Given a formal language L (defined on Σ) used to represent the desired patterns, the LCSP query asks for the shortest path such that its path label (or word) belongs to L. For example, web mapping platforms (such as Google Maps) may be interested in the language $B^*A^+B^*$, where A and B stand for highways and secondary roads, respectively. The returned LCSP should first use secondary roads, then highways, and finally secondary roads. Without the language constraint, the recommended shortest paths could interleave highways with some shortcut secondary roads to reduce distances, but it is inconvenient for drivers to follow the paths. In the example of multimodal-trip planning, the pattern can be described by $b^*y^*b^*$, where b and y represent shared bikes and subways, respectively. The LCSP answer satisfying the language constraint should use bikes before and after subways.

Much research effort has been devoted to designing LCSP algorithms [5–7, 24, 37, 43]. However, early index-free solutions were inefficient in query processing [5–7]. Recent indexing approaches focused on the restricted *Kleene languages* that only allow the use of specific labels in an input label set without caring about their order, frequencies, and more complex relationships [24, 37, 43]. This may result in many unreasonable paths, such as the previous example of highways interleaved with shortcut secondary roads. On the other hand, due to the limited expressiveness of Kleene languages, it cannot satisfy users' demand for more various and realistic path patterns. Furthermore, their solutions did not fully harness the indexing power to provide fast query processing. They mainly preprocessed auxiliary data to prune the search space in query time. However, when the data volume and device memory increase, the *indexes* could directly store sufficiently many partial path answers in hash tables to process queries quickly by few table lookups. For example, the state-of-the-art index, called LSD [43], is based on the *tree decomposition* (which is a popular technique for path queries [35]) that allows us to focus on a "small set" of vertices given a source and a destination, with the correctness guaranteed. LSD's query processing then recursively updates the distances from the source (or destination) to the vertices in the "small set". But,

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681985

in fact, these search processes could be done in the preprocessing phase by storing partial path answers for more efficient querying based on table lookups.

Motivated by the above challenges, we aim to design an LCSP index that supports faster querying for the more general *regular languages*, which can express many flexible label constraints as in previous examples. Specifically, we propose the index called *Partially Constrained Shortest Path (PCSP)* that tries to find the LCSP by concatenating two shortest paths (stored in the index) that partially satisfy the label constraint (and fully satisfy it after concatenation). We also utilize the tree decomposition to find the small vertex set as in LSD. However, our main novelty lies in PCSPs, which means that any other new techniques that provide this vertex set can be adopted. For Kleene languages, LSD can extend the idea for the unconstrained shortest path by maintaining several distances w.r.t. different label sets (not just a single one in the unconstrained case). However, for regular languages, we cannot simply focus on several label sets as in LSD since there are various regular languages that indicate different *orders*, *frequencies*, and *relationships* of labels. It is nontrivial to preprocess useful PCSPs for query processing. Therefore, we try to provide an index for a given fixed regular language so that we can define some useful PCSPs with the help of the deterministic finite automaton (DFA) behind the fixed regular language. It is useful for applications with clear purposes, e.g., online mapping wants to return paths without unreasonable shortcuts, and a visitor prefers to travel through at most three sightseeing spots (on some roads) in a day. In practice, all popular regular languages can be preprocessed once since the number of patterns of interest is often small. To further improve query efficiency, we design pruning techniques that prune the "small set" without affecting the correctness. The experiments in real networks show that our index can answer LCSP queries faster than the best-known LSD by up to two orders of magnitude. We summarize our contributions as follows.

- We propose the index called PCSP, which is by far the fastest known LCSP solution. It answers LCSP queries with label constraints expressed by regular languages. We also propose several efficient pruning techniques.
- We theoretically analyze the correctness and complexities of the proposed index and pruning techniques.
- Experimental results show that PCSP answers each LCSP query in around 100 microseconds and outperforms the best-known LCSP solution by orders of magnitude.

The remainder of the paper is organized as follows. Section 2 states the problem. Section 3 gives an overview of the index and query processing, detailed in Section 4 and Section 5, respectively. Section 6 shows our experiments. Section 7 reviews the related work. Section 8 concludes our paper.

2 PRELIMINARIES

2.1 Problem Statement

Definition 2.1 (Labeled Road Network). Let $G(V, E, \Sigma, l, d)$ be an undirected graph where V and E are the vertex and edge sets, respectively, and the alphabet Σ is a finite nonempty set of labels. Let $n = |V|$ and $m = |E|$. Each edge $e \in E$ is associated with two attributes: its label $l(e) \in \Sigma$ and distance $d(e) \in \mathbb{R}^+$.

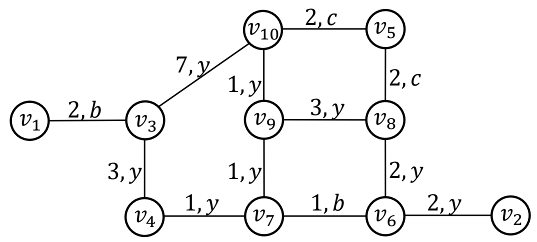


Figure 1: A labeled road network G

Example 2.2. Figure 1 shows a labeled road network G. Let $\Sigma = \{b, y, c\}$ where b, y, c represent bike, subway, and car, respectively. Beside each edge e, we show its distance $d(e)$ and label $l(e)$.

When a road segment is associated with multiple labels or distances, we create multiple edges for it. We will discuss how to handle directed graphs in Section 5. The path pattern constraint is formulated by a regular language [2, 7].

Definition 2.3 (Regular Expression and Regular Language). Given an alphabet Σ disjoint from $\{\epsilon, \emptyset, (,), \cup, \cdot, *\}$, we can define the collection of regular expressions over Σ recursively. 1) The empty set $\emptyset$, each $a \in \Sigma$, and the empty string ϵ are regular expressions. 2) If A and B are regular expressions, A^* (Kleene Star), $A \cup B$ (union), and $A \cdot B$ (concatenation) are regular expressions. We can similarly define the corresponding regular languages by these notations. Each regular language L can be defined by a regular expression.

Definition 2.4 (Path). A u-v path p_{uv} of length k is a finite sequence of vertices $\langle v_0 = u, v_1, v_2, \ldots, v_k = v\rangle$ such that each $e_i = (v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$. Its distance is defined by $d(p_{uv}) = \sum_{i=1}^{k} d(e_i)$, where $d(e_i)$ is the distance of edge e_i, and its label is defined by $l(p_{uv}) = l(e_1) \cdot l(e_2) \ldots l(e_k)$, which is the concatenation of the edge labels along the path. Let $p_{sv} \oplus p_{vt}$ be the path concatenation of p_{sv} and p_{vt}. We only consider simple paths.

Example 2.5. In Figure 1, the path $p_{v_1 v_2} = \langle v_1, v_3, v_4, v_7, v_6, v_2\rangle$ has its distance $d(p_{v_1 v_2}) = 9$ and label $l(p_{v_1 v_2}) = byyby$.

Definition 2.6 (Label-Constrained Path). Given a regular language L over Σ, a path p is a label-constrained path if its label $l(p)$ satisfies the language L, i.e., $l(p) \in L$.

Example 2.7. Consider a regular language L defined by $b^*y^*b^*$, which means that we can first ride a bike, then take the subway, and finally ride a bike. In Figure 1, the path $p_{v_1 v_2} = \langle v_1, v_3, v_4, v_7, v_6, v_2\rangle$ is not a label-constrained path since its label $l(p_{v_1 v_2}) = byyby \notin L$. We will not consider this path (which uses a bike to transfer between subway stations) by using the regular language L. In fact, using the Kleene language cannot express this requirement to avoid generating this unreasonable path. The path $p^{opt} = \langle v_1, v_3, v_4, v_7, v_9, v_8, v_6, v_2\rangle$ is a label-constrained path since its label $l(p^{opt}) = byyyyyy \in L$.

Definition 2.8 (Label-Constrained Shortest Path (LCSP) Problem). Given a labeled network $G(V, E, \Sigma, l, d)$ and a regular language L, we aim to build an in-memory index to answer the LCSP query with a source s and a destination t by the LCSP p^{opt} such that it has the shortest distance among all the label-constrained s-t paths.

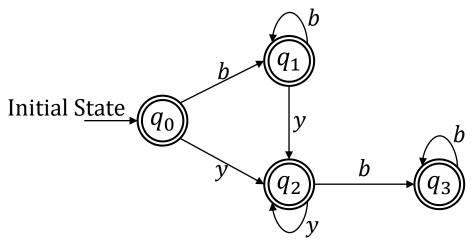


Figure 2: A DFA for $b^*y^*b^*$

Example 2.9. We still use the labeled road network and the regular language in the previous examples. Suppose that we want to answer an LCSP query with $s = v_1$ and $t = v_2$. Without the constraint, the shortest path is $\langle v_1, v_3, v_4, v_7, v_6, v_2\rangle$ with a distance of 9, but its label cannot be accepted by L. Under the constraint, the LCSP is $p^{opt} = \langle v_1, v_3, v_4, v_7, v_9, v_8, v_6, v_2\rangle$ with its label $l(p^{opt}) \in L$ and the minimum distance $d(p^{opt}) = 14$.

2.2 Deterministic Finite Automatons (DFA)

We will use the DFA to judge whether a path label $l(p)$ can be accepted by a regular language L.

Definition 2.10 (Deterministic Finite Automaton (DFA)). A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, consisting of 1) a finite set of states Q, 2) a finite set of input labels, i.e., the alphabet Σ, 3) a transition function $\delta : Q \times \Sigma \to Q$, which defines the next state given the current state and a label, 4) an initial state $q_0 \in Q$, and 5) a set of accepting or final states $F \subseteq Q$.

Example 2.11. Figure 2 shows a DFA for the regular language L defined by $b^*y^*b^*$, where $Q = \{q_0, q_1, q_2, q_3\}$, q_0 is the initial state, and all states are final states, i.e., $F = Q$. The state transition function δ is shown by the arrows.

Each regular expression or regular language can be represented by a DFA [3]. A path label is accepted by a regular language if and only if on its corresponding DFA, we can start with the initial state, use the edge labels one by one for transition, and halt on a final state. We define an indicator function for a path label as follows.

Definition 2.12 (Indicator Function). Given a path label $l(p)$, we define an indicator function $\mathbb{1}_{q,q'}(l(p))$ for state q and q' as 1 if $l(p)$ can be accepted by the DFA from state q to q' (which uses the edge labels of $l(p)$ sequentially from state q to q') and 0 otherwise.

Clearly, for each label-constrained path p with its label $l(p)$, $\mathbb{1}_{q_0,q_f}(l(p)) = 1$ hold for at least one $q_f \in F$.

Example 2.13. Back to previous Example 2.9, $l(p^{opt}) = byyyyyy$ and $\mathbb{1}_{q_0,q_2}(l(p^{opt})) = 1$ since $l(p^{opt})$ can be accepted by the DFA in Figure 2 from state q_0 to state q_2 by using the labels sequentially.

Table 1 lists the main notations used throughout the paper.

3 OVERVIEW

Given two vertices s and t, we utilize a small set of vertices that lie in each s-t path, called an s-t separator, to improve query efficiency.

Table 1: List of notations

Symbol	Meaning
V, E	vertex and edge sets
Σ, L	the alphabet of all labels and the regular language
Q, F	the set of states and the set of final states
$\mathbb{1}_{q,q'}(l(p))$	the indicator function for state q and q'
$l(e), d(e)$	the label and distance of an edge e
$l(p), d(p)$	the label and distance of a path p
$X(v), H$	the tree node for v and a separator
$\mathcal{P}^1_{vw}, \mathcal{P}^2_{wv}$	the two PCSP sets for $X(v)$ and its ancestor $X(w)$
α, K	the ratio α and the the top-K separators
$H^{\text{sour}}(H, v)$ $H^{\text{des}}(H, v)$	pruned separators for H and v as the source and the destination

Definition 3.1 (Separator). A u-v separator H is a set of vertices such that u and v are disconnected after we remove the vertices in H. In other words, any u-v path must visit at least one vertex in H.

Given an s-t separator H, a natural idea is to divide the problem into two subproblems of finding two subpaths. For example, for an unconstrained shortest s-t path p^{opt}_{st} (where $L = (\bigcup_{a\in\Sigma} a)^*$ that allows any label), we can compute the shortest distance by $d(p^{opt}_{st}) = \min_{h\in H}\{d(p^{opt}_{sh}) + d(p^{opt}_{ht})\}$ (where p^{opt}_{sh} and p^{opt}_{ht} are the shortest s-h and h-t paths, respectively) since the shortest path p^{opt}_{st} must visit at least one vertex $h \in H$ and any subpath of p^{opt}_{st} is also a shortest path [35]. The query processing can be fast if we can obtain a small s-t separator H quickly and have stored $d(p^{opt}_{sh})$ and $d(p^{opt}_{ht})$ for each $h \in H$ in the index.

However, given a regular language L, for the LCSP p^{opt}_{st}, we cannot use the same idea because the label concatenation may not satisfy the language L (i.e., $l(p^{opt}_{sh}) \cdot l(p^{opt}_{ht}) \notin L$ though $l(p^{opt}_{sh}) \in L$ and $l(p^{opt}_{ht}) \in L$). Instead of storing just one shortest s-h (or h-t) path, our Partially Constrained Shortest Path (PCSP) index stores a set of partially constrained shortest s-h paths (and h-t paths) that satisfy the language constraint partially. Specifically, the path concatenated by an s-h PCSP p with $\mathbb{1}_{q_0,q}(l(p)) = 1$ and an h-t PCSP p' with $\mathbb{1}_{q,q_f}(l(p')) = 1$ for any $q \in Q$ and $q_f \in F$ will satisfy the language constraint completely since the label concatenation $l(p) \cdot l(p')$ (or $l(p \oplus p')$) can be accepted by the DFA from the initial state q_0 to a state q and from q to a final state $q_f \in F$. The LCSP p^{opt}_{st} is the one with the shortest distance among all the concatenated paths for all $h \in H$. To obtain a small s-t separator H quickly, we will use the tree decomposition, which is a commonly used data structure for separators [35, 38]. We will also call the vertices in H as "hoplinks" since each vertex $h \in H$ links one "hop" (i.e., a path) from s to h and the other hop from h to t.

We observe that an s-t separator H may use redundant hoplinks that are irrelevant to finding the final LCSP p^{opt}. We prune a hoplink $h \in H$ if whenever the LCSP p^{opt} traverses h, we can always find another hoplink $h' \in H$ such that p^{opt} also traverses h', which means that h' also lies in the LCSP and checking h' is sufficient. To further improve query efficiency, our PCSP index additionally preprocesses some *pruned separators* for specific types of queries and directly uses them in query processing.

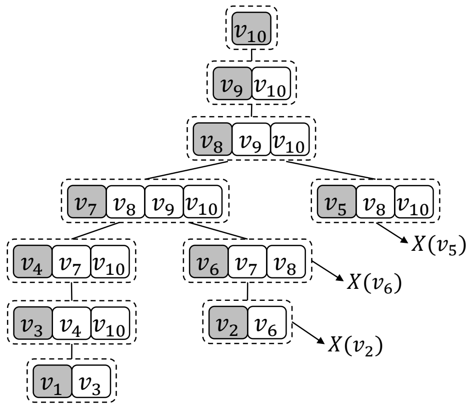


Figure 3: A tree decomposition of G

4 INDEX CONSTRUCTION

We first give some basics of the tree decomposition since we utilize it and then explain how to preprocess PCSPs and pruned separators.

4.1 Tree Decomposition

Tree decomposition embeds the structural information of a graph into a tree so that we can solve graph problems efficiently. Many state-of-the-art solutions for path queries are based on it [13, 32, 35, 41, 43, 47]. Though it is irrelevant to edge labels, we can use it to get a small separator to improve query efficiency.

Definition 4.1 (Tree Decomposition [38]). A tree decomposition of a graph $G(V, E)$ is a rooted tree with its tree nodes $\{X(v)|v \in V\}$. Each vertex v is mapped to a tree node $X(v)$ that represents a vertex set that includes v (i.e., $X(v) \subseteq V$ and $v \in X(v)$). The tree decomposition satisfies the following three conditions:

(1) $\cup_{v \in V} X(v) = V$;

(2) For each edge $e = (u, v) \in E$, there is a tree node $X(v')$ such that $u \in X(v')$ and $v \in X(v')$;

(3) For each $v \in V$, the tree nodes that contain v (i.e., $\{X(v')|v \in X(v')\}$) form a connected subtree.

Example 4.2. We plot the tree decomposition for the network G of Figure 1 in Figure 3. Each tree node $X(v)$ is a set of vertices including v, which we mark in a grey block. For example, $X(v_5) = \{v_5, v_8, v_{10}\}$.

For better clarity, we will use "node" for the tree decomposition and "vertex" for the graph. We define its treewidth as $\omega = \max_{v \in V} |X(v)|$ and its treeheight η as the maximum depth of a tree node (i.e., the length of the simple tree path to the root node).

LEMMA 4.3 ([13]). *For any two vertices s and t such that $X(s)$ and $X(t)$ have no ancestor-descendant relationship, the least common ancestor (LCA) of $X(s)$ and $X(t)$, denoted by $X(c)$, is a separator. Let $X(c_s)$ and $X(c_t)$ be the two child nodes of $X(c)$ in the two branches containing $X(s)$ and $X(t)$, respectively. Then, both $X(c_s)\backslash\{c_s\}$ and $X(c_t)\backslash\{c_t\}$ are separators.*

Example 4.4. For v_1 and v_2, $X(v_7)$ is the LCA of $X(v_1)$ and $X(v_2)$. Thus, $X(v_7) = \{v_7, v_8, v_9, v_{10}\}$ is a separator for v_1 and v_2. Since $X(c_s) = X(v_4)$ and $X(c_t) = X(v_6)$, $X(c_s)\backslash\{c_s\} = \{v_7, v_{10}\}$ and $X(c_t)\backslash\{c_t\} = \{v_7, v_8\}$ are two separators.

Minimum Degree Elimination (MDE). The tree decomposition can be built by the MDE algorithm (Algorithm 6 in [35] adapted from [10]). It first creates tree nodes iteratively and then links them by a tree. In the first phase, in each iteration of creating a tree node, it first finds the vertex v with the minimum degree in the graph G' (which is initially G). Then, $X(v)$ is formed by v and its current neighbors in G'. It next removes v and its incident edges and adds an edge between each pair of v's neighbors in G' if the edge does not exist. In this way, any two neighbors of v are still connected after we remove v and all v's incident edges (since they are connected via v before the removal). In the second phase, the parent of each node $X(v)$ is set to the node $X(v')$ where v' is the first removed vertex in $X(v)\backslash\{v\}$.

Example 4.5. MDE first creates $X(v_1) = \{v_1, v_3\}$ since v_1's degree is currently the minimum. It removes the edge (v_1, v_3) and adds no new edge. It similarly creates $X(v_2)$. For $X(v_3) = \{v_3, v_4, v_{10}\}$, it removes the two edges (v_3, v_4) and (v_3, v_{10}) and adds (v_4, v_{10}). Similarly, it creates $X(v_4), X(v_5), \ldots, X(v_{10})$. Next, MDE links all nodes. The parent of $X(v_1)$ is $X(v_3)$ because v_3 is the first removed vertex in $X(v_1)\backslash\{v_1\}$. We can similarly do the rest steps and finally get the tree decomposition in Figure 3.

4.2 Partially Constrained Shortest Path

4.2.1 Two PCSP Sets. Formally, for each node $X(v)$, whenever $X(w)$ is an ancestor of $X(v)$, our PCSP preprocesses the following two sets of PCSPs:

(1) The first set $\mathcal{P}^1_{vw} = \{(q, p^{1q}_{vw})|q \in Q, S^{1q}_{vw} \neq \emptyset\}$, where $S^{1q}_{vw} = \{p_{vw}|\mathbb{1}_{q_0,q}(l(p_{vw})) = 1\}$ is the set of all v-w paths p_{vw} whose labels $l(p_{vw})$ can be accepted by the DFA from the initial state q_0 to the state q, and $p^{1q}_{vw} = \arg\min_{p \in S^{1q}_{vw}} d(p)$ is the one in S^{1q}_{vw} with the shortest distance.

(2) The second set $\mathcal{P}^2_{wv} = \{(q, p^{2q}_{wv})|q \in Q, S^{2q}_{wv} \neq \emptyset\}$, where $S^{2q}_{wv} = \{p_{wv}|\exists q_f(q_f \in F \wedge \mathbb{1}_{q,q_f}(l(p_{wv})) = 1)\}$ is the set of all w-v paths p_{wv} whose labels $l(p_{wv})$ can be accepted by the DFA from the state q to a final state $q_f \in F$, and $p^{2q}_{wv} = \arg\min_{p \in S^{2q}_{wv}} d(p)$ is the one in S^{2q}_{wv} with the shortest distance.

The two sets $\mathcal{P}^1_{vw}$ and $\mathcal{P}^2_{wv}$ store a set of shortest v-w paths p^{1q}_{vw} from the initial state q_0 to some states (which will be used in the "first" s-h subpath) and a set of shortest w-v paths p^{2q}_{wv} from some states to final states (which will be used in the "second" h-t subpath), respectively. They can be fetched by looking up a hash table w.r.t. the node $X(v)$ and its ancestor $X(w)$. Note that the order of v and w in the subscript is important because the labels of v-w paths are different from the labels of w-v paths. Also, note that "1" and "2" in the superscript indicate whether the path label is accepted by the DFA from an initial state to intermediate states or from intermediate states to final states. We do not need to store $\mathcal{P}^2_{vw}$ and $\mathcal{P}^1_{wv}$ since they will never be used in query processing.

Example 4.6. For $X(v_1)$, its ancestors are $X(v_{10})$, $X(v_9)$, $X(v_8)$, $X(v_7)$, $X(v_4)$, and $X(v_3)$. For its ancestor $X(v_{10})$, we store $\mathcal{P}^1_{v_1 v_{10}}$ and $\mathcal{P}^2_{v_{10} v_1}$. In $\mathcal{P}^1_{v_1 v_{10}}$, for q_2, $p^{1q_2}_{v_1 v_{10}} = \langle v_1, v_3, v_4, v_7, v_9, v_{10}\rangle$ because

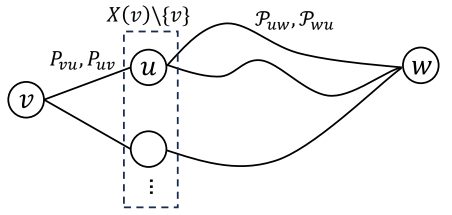


Figure 4: The simplified graph G' in the iteration for $X(v)$

it has the shortest distance of 8 with its label $l(p^{1q_2}_{v_1v_{10}}) = byyyy$ and $\mathbb{1}_{q_0,q_2}(l(p^{1q_2}_{v_1v_{10}})) = 1$. In $\mathcal{P}^2_{v_{10}v_1}$, for q_0, $p^{2q_0}_{v_{10}v_1} = \langle v_{10}, v_9, v_7, v_4, v_3, v_1\rangle$ with its label $l(p^{2q_0}_{v_{10}v_1}) = yyyyb$ and $\mathbb{1}_{q_0,q_3}(l(p^{2q_0}_{v_{10}v_1})) = 1$. For q_1 and q_2, $p^{2q_1}_{v_{10}v_1} = p^{2q_2}_{v_{10}v_1} = p^{2q_0}_{v_{10}v_1}$.

Given s and t, if $X(t)$ (or $X(s)$) is an ancestor of $X(s)$ (or $X(t)$), we first fetch $\mathcal{P}^1_{st}$ (or $\mathcal{P}^2_{st}$) from the hash table. Then, we only need to consider $p^{1q}_{st} \in \mathcal{P}^1_{st}$ with $q \in F$ (or $p^{2q}_{st} \in \mathcal{P}^2_{st}$ with $q = q_0$) and return the one with the shortest distance as the LCSP since its label can be accepted by the DFA from the initial state to a final state.

If $X(s)$ and $X(t)$ have no ancestor-descendant relationship, we set a separator H as the one with a smaller size between $X(c_s)\backslash\{c_s\}$ and $X(c_t)\backslash\{c_t\}$ by Lemma 4.3. By the property of the tree decomposition [38], for each $h \in X(c_s)\backslash\{c_s\}$ and $h \in X(c_t)\backslash\{c_t\}$, $X(h)$ is an ancestor of both $X(s)$ and $X(t)$. Thus, for each $h \in H$, we can fetch $\mathcal{P}^1_{sh}$ and $\mathcal{P}^2_{ht}$ from the hash tables at nodes $X(s)$ and $X(t)$, respectively. Next, for each $h \in H$, we find the path p^{opt}_h with the shortest distance among $\{p^{1q}_{sh} \oplus p^{2q}_{ht} | q \in Q, (q, p^{1q}_{sh}) \in \mathcal{P}^1_{sh}, (q, p^{2q}_{ht}) \in \mathcal{P}^2_{ht}\}$, where the concatenation requires that the two paths share the same hoplink h and the same state q. Finally, we compare the distances of all p^{opt}_h for $h \in H$ and return the LCSP with the shortest distance.

4.2.2 Preprocessing PCSP Sets. For each node $X(v)$ and each $X(v)$'s ancestor $X(w)$, we need to preprocess the two sets $\mathcal{P}^1_{vw}$ and $\mathcal{P}^2_{wv}$ of PCSPs. Instead of directly deriving them, we consider their two supersets so that we can perform concise recursive computations in a top-down manner in the tree. We will finally extract the two sets from their supersets.

Specifically, for each node $X(v)$ and each $X(v)$'s ancestor $X(w)$, we consider two supersets $\mathcal{P}_{vw} = \{(q, q', p^{qq'}_{vw}) | q, q' \in Q, S^{qq'}_{vw} \neq \emptyset\}$ (for v-w paths) and $\mathcal{P}_{wv}$ (for w-v paths) similarly, where $S^{qq'}_{vw} = \{p_{vw} | \mathbb{1}_{q,q'}(l(p_{vw})) = 1\}$ is the set of all v-w paths p_{vw} whose labels $l(p_{vw})$ can be accepted by the DFA from state q to state q', and $p^{qq'}_{vw}$ is the one in $S^{qq'}_{vw}$ with the shortest distance, i.e., $p^{qq'}_{vw} = \arg\min_{p \in S^{qq'}_{vw}} d(p)$. We can easily extract $\mathcal{P}^1_{vw}$ and $\mathcal{P}^2_{wv}$ from $\mathcal{P}_{vw}$ and $\mathcal{P}_{wv}$ by considering the initial state and final states, respectively. Note that one shortest path $p^{qq'}_{vw}$ in $\mathcal{P}_{vw}$ may not be in $\mathcal{P}_{wv}$ since it may not be a shortest w-v path when its label $l(p^{qq'}_{vw})$ is reversed.

The main idea of deriving $\mathcal{P}_{vw}$ and $\mathcal{P}_{wv}$ is that in the MDE algorithm, in the iteration of creating the node $X(v)$ that includes v and v's neighbors in G', we can regard all v's neighbors in G' (i.e., $X(v)\backslash\{v\}$) as a separator between v and w in G', as shown in Figure 4. Since the shortest paths in $\mathcal{P}_{vw}$ and $\mathcal{P}_{wv}$ must visit at least one v's neighbor $u \in X(v)\backslash\{v\}$, we can concatenate each of

Algorithm 1: PCSPJoin

```
   input : Two path sets $P_1$ and $P_2$ and a joined path set $P_{\text{res}}$
   output: Updated joined path set $P_{\text{res}}$
1  foreach $(q_1, q'_1, p_1) \in P_1$ do
2      foreach $(q_2, q'_2, p_2) \in P_2$ where $q'_1 = q_2$ do
3          if there exists no $(q_1, q'_2, p)$ in $P_{res}$ or
             $d(p_1) + d(p_2) < d(p)$ then
4              $p \leftarrow p_1 \oplus p_2$
5              Update $(q_1, q'_2, p)$ in $P_{\text{res}}$
6  $P_{\text{res}} \leftarrow P_2$ (or $P_1$) if $P_1$ (or $P_2$) is a set of $v$-$v$ paths for any $v$
```

v's incident edges (v, u) with the shortest u-w and w-u paths (i.e., $\mathcal{P}_{uw}$ and $\mathcal{P}_{wu}$) to update $\mathcal{P}_{vw}$ and $\mathcal{P}_{wv}$, respectively.

For the latter $\mathcal{P}_{uw}$ and $\mathcal{P}_{wu}$, we can follow a top-down manner recursively in the tree and obtain them from the previously computed hash table w.r.t. the node $X(u)$ and its ancestor $X(w)$ (if $X(w)$ is the ancestor of $X(u)$) or the node $X(w)$ and its ancestor $X(u)$ (otherwise). For the former v's incident edges, one problem is that some incident edges (v, u) are newly added before the iteration of creating $X(v)$. For example, in the iteration of creating $X(v')$, we may add (v, u) if both (v, v') and (v', u) exist. Moreover, (v, v') and (v', u) may also be newly added edges in the previous iterations. We associate two path sets $P_{(v,u)}$ and $P_{(u,v)}$ with each existing and new (undirected) edge (v, u) to preserve the information about the shortest paths when removing edges. Initially, for each $e = (v, u) \in E$, as a path with a single edge, $P_{(v,u)} = \{(q, q', \langle v, u\rangle) | q, q' \in Q, \mathbb{1}_{q,q'}(l(e)) = 1\}$ and $P_{(u,v)} = \{(q, q', \langle u, v\rangle) | q, q' \in Q, \mathbb{1}_{q,q'}(l(e)) = 1\}$. Each time two edges (v, v') and (v', u) are removed, we join $P_{(v,v')}$ and $P_{(v',u)}$ to update $P_{(v,u)}$ and $P_{(u,v')}$ and $P_{(v',v)}$ to update $P_{(u,v)}$. When concatenating each of v's incident edge (v, u) with paths in $\mathcal{P}_{uw}$ and $\mathcal{P}_{wu}$, we are joining $P_{(v,u)}$ and $\mathcal{P}_{uw}$ to update $\mathcal{P}_{vw}$ and $\mathcal{P}_{wu}$ and $P_{(u,v)}$ to update $\mathcal{P}_{wv}$.

To join two path sets, we concatenate two paths in the two respective sets such that they share a common intermediate state. For a pair (q, q') with $q, q' \in Q$, we only maintain the shortest path p with $\mathbb{1}_{q,q'}(l(p)) = 1$. For example, $\mathbb{1}_{q_0,q_2}(l(\langle v_7, v_6, v_8\rangle)) = \mathbb{1}_{q_0,q_2}(l(\langle v_7, v_9, v_8\rangle)) = 1$, and we only need to maintain $\langle v_7, v_6, v_8\rangle$ with a smaller distance w.r.t. (q_0, q_2). Let P_{res} denote the result of joining two path sets. For the concatenated path $p_1 \oplus p_2$ with $\mathbb{1}_{q,q'}(l(p_1 \oplus p_2)) = 1$, if there is no $(q, q', p) \in P_{\text{res}}$ w.r.t. the pair (q, q'), we directly put $(q, q', p_1 \oplus p_2)$ in P_{res}. If there exists $(q, q', p) \in P_{\text{res}}$ and $d(p) > d(p_1 \oplus p_2)$, we update p as $p_1 \oplus p_2$. Algorithm 1 describes the operation called PCSPJoin. In Line 1, we iterate the shortest path p_1 in P_1 with $\mathbb{1}_{q_1,q'_1}(l(p_1)) = 1$ for (q_1, q'_1) and only need to concatenate it with those p_2 in P_2 with $\mathbb{1}_{q_2,q'_2}(l(p_2)) = 1$ for (q_2, q'_2) where $q'_1 = q_2$ in Line 2. In Lines 3–5, if we have not stored a shortest path with $\mathbb{1}_{q_1,q'_2}(l(p_1) \cdot l(p_2)) = 1$ for (q_1, q'_2), we put it in the resulting path set P_{res}. Otherwise, we update the shortest path for (q_1, q'_2) if the sum of the two paths' distances is smaller than the current one. In Line 6, we handle a special case where $u = w$ that may happen in Lines 19–20 of Algorithm 1.

Algorithm 2 summarizes the whole procedure of preprocessing PCSPs. In Lines 1–3, we initialize $P_{(v,u)}$ and $P_{(u,v)}$ by considering

Algorithm 2: Preprocessing PCSPs

```
input  : A road network $G$
output : A tree decomposition and $\mathcal{P}_{vw}$ and $\mathcal{P}_{wv}$ for each
         node $X(v)$ and each $X(v)$'s ancestor $X(w)$
1  foreach $e = (v, u) \in E$ do
2      $P_{(v,u)} \leftarrow \{(q, q', \langle v, u\rangle) | q, q' \in Q, \mathbb{1}_{q,q'}(l(e)) = 1\}$
3      $P_{(u,v)} \leftarrow \{(q, q', \langle u, v\rangle) | q, q' \in Q, \mathbb{1}_{q,q'}(l(e)) = 1\}$
4  $G' \leftarrow G$
5  while $G'$ is not empty do
6      $v \leftarrow$ the vertex in $G'$ with the smallest degree
7      create a node $X(v)$ by $v$ and $v$'s current neighbors
8      foreach pair $(u, w)$ where they are $v$'s neighbors do
9          if the edge $(u, w)$ does not exist in $G'$ then
10             add an new edge $(u, w)$
11         $P_{(u,w)} \leftarrow$ PCSPJoin($P_{(u,v)}, P_{(v,w)}, P_{(u,w)}$)
12         $P_{(w,u)} \leftarrow$ PCSPJoin($P_{(w,v)}, P_{(v,u)}, P_{(w,u)}$)
13     Remove $v$ and its incident edges in $G'$
14 foreach node $X(v)$ do
15     set $X(v)$'s parent as $X(v')$ where $v'$ is the first
       eliminated vertex in $X(v)\backslash\{v\}$
16 foreach node $X(v)$ in a top-down manner do
17     foreach ancestor $X(w)$ of $X(v)$ do
18         foreach $u \in X(v)\backslash\{v\}$ do
19             $\mathcal{P}_{vw} \leftarrow$ PCSPJoin($P_{(v,u)}, \mathcal{P}_{uw}, \mathcal{P}_{vw}$)
20             $\mathcal{P}_{wv} \leftarrow$ PCSPJoin($\mathcal{P}_{wu}, P_{(u,v)}, \mathcal{P}_{wv}$)
```

all state pair (q, q'). Specifically, for each edge $e = (v, u)$, since we can only use one label $l(e)$, we only need to consider each $q \in Q$ and its next state q' that could be reached by using $l(e)$. In Lines 4–10 and 13–15, we follow the standard procedure of building the tree decomposition. In Lines 11-12, we update $P_{(u,w)}$ and $P_{(w,u)}$ by joining the path sets related to the two edges (u, v) and (v, w). In Lines 16-18, we consider a tree node $X(v)$, its ancestor $X(w)$, and the separator $X(v)\backslash\{v\}$. Finally, in Lines 19–20, we obtain $\mathcal{P}_{vw}$ and $\mathcal{P}_{wv}$ by joining the two path sets $P_{(v,u)}$ and $P_{(u,v)}$ related to the edge (v, u) and those related to u-w and w-u paths.

Example 4.7. We still use Example 2.9. For ease of presentation, we will identify each path p by its distance $d(p)$. The path retrieval procedure is given in Section 5. In Lines 1–3, we initialize $P_{(v,u)}$ and $P_{(u,v)}$ for each $e = (v, u) \in E$. For example, for (v_3, v_4), we set $P_{(v_3,v_4)} = \{(q_0, q_2, \langle v_3, v_4\rangle), (q_1, q_2, \langle v_3, v_4\rangle), (q_2, q_2, \langle v_3, v_4\rangle)\}$ and similarly $P_{(v_3,v_4)}$. For ease of presentation, we will identify each path p by its distance $d(p)$, and the path retrieval by using $d(p)$ is given in Section 5. In Lines 5–13, we build the tree decomposition. We add no new edge when creating $X(v_1)$ and $X(v_2)$. For $X(v_3)$, for the pair (v_4, v_{10}), we add a new edge (v_4, v_{10}) with $P_{(v_4,v_{10})}$ and $P_{(v_{10},v_4)}$. We join $P_{(v_4,v_3)} = \{(q_0, q_2, 3), (q_1, q_2, 3), (q_2, q_2, 3)\}$ and $P_{(v_3,v_{10})} = \{(q_0, q_2, 7), (q_1, q_2, 7), (q_2, q_2, 7)\}$ to obtain $P_{(v_4,v_{10})} = \{(q_0, q_2, 10), (q_1, q_2, 10), (q_2, q_2, 10)\}$. We similarly create the other nodes and path sets. In Lines 16–20, for $X(v_9)$ and its ancestor $X(v_{10})$, we create $\mathcal{P}_{v_9v_{10}} = P_{v_9v_{10}}$ and $\mathcal{P}_{v_{10}v_9} = P_{v_{10}v_9}$ since $\mathcal{P}_{v_{10}v_{10}} = \emptyset$. For $X(v_8)$, we similarly build $\mathcal{P}_{v_8v_{10}}$ and $\mathcal{P}_{v_{10}v_8}$ for its ancestor $X(v_{10})$ and $\mathcal{P}_{v_8v_9}$ and $\mathcal{P}_{v_9v_8}$ for its ancestor $X(v_9)$. We can then process the remaining nodes similarly.

Theorem 4.8. *Algorithm 2 correctly builds $\mathcal{P}^1_{vw}$ and $\mathcal{P}^2_{wv}$.*

Proof. It suffices to prove that we correctly build $\mathcal{P}_{vw}$ and $\mathcal{P}_{wv}$ since they are supersets of $\mathcal{P}^1_{vw}$ and $\mathcal{P}^2_{wv}$, respectively. We first prove by induction that in each iteration of creating $X(v)$, any LCSP between vertices in G' can be found by using $P_{(u,w)}$ and $P_{(w,u)}$ for each $e = (u, w) \in E(G')$. Next, we can prove the correctness of $\mathcal{P}_{vw}$ and $\mathcal{P}_{wv}$ by induction on the tree about using the separator $X(v)\backslash\{v\}$ to concatenate the incident edge with path sets, as stated previously. For the first statement, the base case for $G' = G$ holds since $P_{(u,w)}$ and $P_{(w,u)}$ for each $e = (u, w) \in E$ only store the corresponding edge information. Now assume that any LCSP can be found before v's incident edges are removed. After they are removed, any LCSP either traverses v or not. The latter case holds due to the hypothesis. For the former one, the LCSP consists of an s-u subpath, (u, v) and (v, w), and a w-t subpath. The first one and the last one can be restored by the hypothesis, and the middle one (u, v, w) can be restored by using $P_{(u,w)}$ and $P_{(w,u)}$ since we integrate the shortest path information in Lines 11–12. □

Theorem 4.9. *Algorithm 2's time complexity is $O(|V| \log |V| + |V|\omega^2|Q|^2 + |V|\eta\omega|Q|^2))$, where ω and η are the treewidth and tree-height, respectively, and $|Q|$ is the number of states. The space cost of PCSPs is $O(|V|\eta|Q|^2)$.*

Proof. The first term in the time complexity represents the time cost of selecting the vertex with the smallest degree (Line 6), which is implemented by a priority queue. The second term uses ω to bound the number of v's neighbors and $|Q|^2$ for the PCSPJoin operation (Algorithm 1). The third term is the time cost of Lines 16–20, where η and ω are upper bounds of the number of ancestors and $|X(v)\backslash\{v\}|$, respectively. The space cost is $O(|V|\eta|Q|^2)$ because we store $\mathcal{P}_{vw}$ and $\mathcal{P}_{wv}$ for each node $X(v)$ and its ancestor $X(w)$. □

It can be observed that both time and space complexities are related to the number of states $|Q|$. Thus, we perform DFA minimization to reduce $|Q|$, which can be done by Hopcroft's algorithm [26]. Moreover, our PCSP index only needs to store $\mathcal{P}^1_{vw}$ and $\mathcal{P}^2_{wv}$ (not the two supersets) for query processing, which would use $O(|V|\eta|Q|)$ and $O(|V|\eta|Q||F|)$ space, respectively.

4.3 Pruned Separator

Assume that we have known that the sources of some LCSP queries are all $v_s \in V$ (or the destinations are $v_t \in V$) in advance. Given a separator H from the node in the tree decomposition used to answer these queries, we can prune some hoplinks in H beforehand. The main idea of pruning a hoplink $h \in H$ is that each of the shortest v_s-h (or h-v_t) paths in $\mathcal{P}^1_{v_sh}$ is not superior to one v_s-h' (or h'-v_t) path via another hoplink $h' \in H$, which indicates that it is sufficient to check h' during query processing. We will then select some important separators and preprocess pruned separators for them. Note that we do not assume that we know both the source and the destination at the same time beforehand because it would require us to preprocess pruned separators for nearly $|V|^2$ queries, which is infeasible and prohibitive on large road networks.

Algorithm 3: Separator Pruning for Sources

```
input : A separator $H$ and a source $v$
output: A pruned separator $H^{\text{sour}}(H, v)$
1 foreach $h \in H$ in the decreasing order of $\min_{p \in \mathcal{P}^1_{vh}} d(p)$ do
2     foreach $(q, p^{1q}_{v_s h}) \in \mathcal{P}^1_{v_s h}$ do
3         ConditionFlag$[q] \leftarrow false$
4         foreach $h'$ s.t. $\min_{p \in \mathcal{P}^1_{vh'}} d(p) < \min_{p \in \mathcal{P}^1_{vh}} d(p)$ do
5             if $d(p) = d(p^{1q}_{v_s h})$ where $(q, p) \in \mathcal{P}^1_{v_s h' h}$ then
6                 ConditionFlag$[q] \leftarrow true$
7     Prune $h$ if all ConditionFlag$[q]$ is true
```

4.3.1 Pruning Rules. Let $\mathcal{P}_{v_s h' h}$ denote the result of PCSPJoin (Algorithm 1) on $\mathcal{P}_{v_s h'}$ and $\mathcal{P}_{h'h}$. Let $\mathcal{P}^1_{v_s h' h} = \{(q', p) | (q, q', p) \in \mathcal{P}_{v_s h' h}, q = q_0\}$ be the set of paths in $\mathcal{P}_{v_s h' h}$ that starts with the initial state q_0. We similarly define $\mathcal{P}_{hh'v_t}$ and $\mathcal{P}^2_{hh'v_t}$. Then, we check if h can be pruned by comparing $\mathcal{P}^1_{v_s h}$ and $\mathcal{P}^1_{v_s h' h}$ (or $\mathcal{P}^2_{hv_t}$ and $\mathcal{P}^2_{hh'v_t}$). Specifically, we propose the following two pruning rules for sources and destinations:

(1) Given a source $v_s \in V$, a hoplink $h \in H$ can be pruned if for each $(q, p^{1q}_{v_s h}) \in \mathcal{P}^1_{v_s h}$, there exists one $h' \in H$ such that for $(q, p) \in \mathcal{P}^1_{v_s h' h}$, $d(p) = d(p^{1q}_{v_s h})$.

(2) Given a destination $v_t \in V$, a hoplink $h \in H$ can be pruned if for each $(q, p^{2q}_{hv_t}) \in \mathcal{P}^2_{hv_t}$, there exists one $h' \in H$ such that for $(q, p) \in \mathcal{P}^2_{hh'v_t}$, $d(p) = d(p^{2q}_{hv_t})$.

Note that in either rule, the hoplink h' that satisfies the condition can be different for each $(q, p^{1q}_{v_s h}) \in \mathcal{P}^1_{v_s h}$ or $(q, p^{2q}_{hv_t}) \in \mathcal{P}^2_{hv_t}$. The following lemma shows the correctness of the two rules.

Lemma 4.10. *Assume w.l.o.g. there is only one p^{opt}. Given a separator H, if one $h \in H$ can be pruned w.r.t. p^{opt}'s source s or destination t and p^{opt} traverses h, then p^{opt} traverses another $h' \in H$.*

Proof. Consider the case of p^{opt}'s source s. Let p be the s-h subpath of p^{opt}. There is only one q such that $\mathbb{1}_{q_0,q}(l(p)) = 1$ and $d(p)$ must be the minimum among all s-h paths p_s with $\mathbb{1}_{q_0,q}(l(p_{sh})) = 1$, which indicates that $(q, p) \in \mathcal{P}^1_{sh}$ and $p = p^{1q}_{sh}$ for q. According to the pruning rule, there is one h' and a path p' via h' such that $d(p') = d(p)$, which suggests that p^{opt} also traverses h'. For the case of p^{opt}'s destination t, we can similarly consider the h-t subpath of p^{opt}. Note that when there are several LCSP p^{opt} with the same distance, we can find one of them. □

By Lemma 4.10, we can prune one $h \in H$ since the LCSP p^{opt} also traverses another hoplink in H. After pruning h, we can find the next hoplink according to the pruning rule and repeat the procedure again. Note that we cannot alternatively use the two rules since no vertex can be both the source and the destination. For the first pruning rule, if there exists h' such that there are $p_{v_s h'}$ and $p_{h'h}$ with $d(p_{v_s h'} \oplus p_{h'h}) = d(p^{1q}_{v_s h})$, we can easily derive that $d(p_{v_s h'}) < d(p^{1q}_{v_s h})$. To find such h', a heuristic idea is to first sort all the hoplinks h in the increasing order of their unconstrained distances without the label constraint, i.e., $\min_{p \in \mathcal{P}^1_{v_s h}} d(p)$. We then find h' among those hoplinks with smaller unconstrained distances.

Algorithm 3 gives the procedure of pruning a separator H and a source v. Let $H^{\text{sour}}(H, v)$ denote the pruned separator. We omit a similar procedure for a destination v to get $H^{\text{des}}(H, v)$. In Line 1, we check the hoplinks in the decreasing order of their unconstrained shortest distances. In Lines 2–6, for each $(q, p^{1q}_{v_s h}) \in \mathcal{P}^1_{v_s h}$, we assign ConditionFlag$[q]$ as true if the pruning rule is discovered and false otherwise. We prune h finally in Line 7 if all flags are true.

Example 4.11. Given a separator $H = \{v_7, v_{10}\}$ and a source v_1, we show how to get $H^{\text{sour}}(H, v_1)$ by Algorithm 3. We can get $\mathcal{P}^1_{v_1 v_7} = \{(q_2, 6), (q_3, 16)\}$ and $\mathcal{P}^1_{v_1 v_{10}} = \{(q_2, 8)\}$. Since we have $\min_{p \in \mathcal{P}^1_{v_1 v_{10}}} d(p) = 8 \geq \min_{p \in \mathcal{P}^1_{v_1 v_7}} d(p) = 6$, we first process v_{10}. Let ConditionFlag$[q_2] = false$. For $(q_2, 8) \in \mathcal{P}^1_{v_1 v_{10}}$, we can find $h' = v_7$ such that $(q_2, 8) \in \mathcal{P}^1_{v_1 v_7 v_{10}}$ and set ConditionFlag$[q_2] = true$. We finally prune v_{10} and set $H^{\text{sour}}(H, v_1) = \{v_7\}$.

Theorem 4.12. *Algorithm 3 needs $O(|H| \log |H| + |H|^2 |Q|)$ time.*

Proof. The first term is because we sort all the hoplinks. The second term corresponds to the three for-loops. □

Algorithm 4: Preprocessing Pruned Separators

```
input : A set $R$ of random queries and a parameter $\alpha$
output: $H^{\text{sour}}(H, v)$ and $H^{\text{des}}(H, v)$ for top-$K$ separators
1 foreach $(s, t) \in R$ do
2     $X(c) \leftarrow$ the LCA of $X(s)$ and $X(t)$
3     if $X(c) = X(s)$ or $X(c) = X(t)$ then
4         continue
5     Let $c_s$ and $c_t$ be the two child nodes of $X(c)$ in the two
        branches containing $s$ and $t$, respectively
6     $H \leftarrow \arg\min_{H' \in \{X(c_s) \setminus \{c_s\}, X(c_t) \setminus \{c_t\}\}} |H'|$
7     $N(H) \leftarrow N(H) + \frac{1}{|R|}$
8 Sort all $H$ in the decreasing order of $N(H)$ to get $H_1, H_2, \ldots$
9 for $i = 1, 2, \ldots, \arg\min_K \sum_{i=1}^{K} N(H_i) \geq \alpha$ do
10    foreach $v$ s.t. $X(v)$ is the descendent of $H_i$ do
11        Get $H^{\text{sour}}(H_i, v)$ by Algorithm 3 and similarly
            $H^{\text{des}}(H_i, v)$
```

4.3.2 Preprocessing Pruned Separators. Algorithm 3 shows how to prune a separator. However, it is inefficient to preprocess $H^{\text{sour}}(H, v)$ and $H^{\text{des}}(H, v)$ for all the possible separators since each child of each branching node (with two or more children) can be a possible separator and the number of vertices $|V|$ can be large. It has been shown that there are some important branching nodes in the tree decomposition that would be selected as the LCA with high probabilities [13] due to the tree structure. Therefore, we would use a set R of random queries with sources and destinations that are randomly generated from V to find the set of separators with K largest frequencies.

Algorithm 4 describes the procedure of preprocessing pruned separators. In Lines 1–6, we find the corresponding separator used for each random query. We use $N(H)$ to denote the frequency of

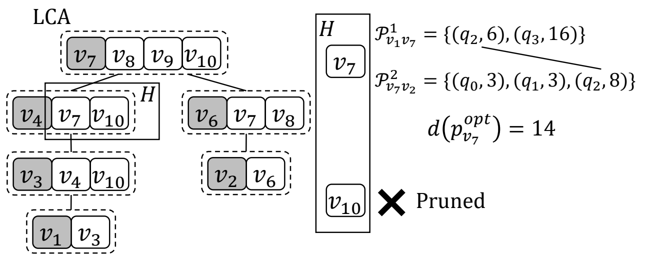


Figure 5: PCSP's query processing

H over the total number $|R|$ and update it in Line 7. In Line 8, we sort all the separators H in the decreasing order of $N(H)$ to get $H_1, H_2, \ldots$. In Lines 9–11, we only consider the top-K frequent separators with K largest $N(H)$ such that the sum of their $N(H)$ is at least α, where $\alpha \in (0,1)$ is a user parameter to control the ratio of queries that we would like to cover. We then apply the pruning rules (Algorithm 3) to obtain pruned separators $H^{\text{sour}}(H,v)$ and $H^{\text{des}}(H,v)$ for all vertices.

Example 4.13. Consider $\alpha = 0.6$ and $R = \{(v_1,v_2),(v_3,v_6),(v_3,v_5)\}$. For the first two queries, they all use $X(v_4)\backslash\{v_4\}$ and $N(X(v_4)\backslash\{v_4\}) = \frac{2}{3}$. For (v_3,v_5), they use $X(v_5)\backslash\{v_5\}$ and $N(X(v_5)\backslash\{v_5\}) = \frac{1}{3}$. We use the top-1 separator $X(v_4)\backslash\{v_4\}$ since $N(X(v_4)\backslash\{v_4\}) \geq 0.6$. We prune it by Algorithm 3 as stated before.

Theorem 4.14. *Algorithm 4 takes $O(|R|+B\log B+K|V|(|H|\log|H|+|H|^2|Q|))$ time and $O(K|V||H|)$ space, where H is at most the treewidth ω and B is the number of nodes that has two or more child nodes.*

Proof. Lines 1–7 uses $O(|R|)$ time since in each iteration, we can find the LCA in $O(1)$ time [9], set H and update $N(H)$ in $O(1)$ time. Lines 8 needs $O(B\log B)$ time to sort at most B separators since only branching nodes can be the LCA. The two for-loops in Lines 9–10 consider K separators and at most $|V|$ descendants of H. Line 11 follows Theorem 4.12. The space cost is because for the top-K separators, we store a flag for each hoplink w.r.t. a vertex. □

5 QUERY PROCESSING

Algorithm 5 summarizes how to process a query (s,t) by our PCSP index, which consists of PCSPs stored in $\mathcal{P}^1_{vw}$ and $\mathcal{P}^2_{wv}$ for each node $X(v)$ and its ancestor $X(w)$ and pruned separators $H^{\text{sour}}(H,s)$ and $H^{\text{des}}(H,t)$ for K separators. For ease of presentation, let $H^{\text{sour}}(H,s)$ and $H^{\text{des}}(H,s)$ be H if we do not preprocess them. In Lines 1–5, if $X(s)$ is the descendant or ancestor of $X(t)$, we directly return the LCSP from $\mathcal{P}^1_{st}$ or $\mathcal{P}^2_{st}$ as stated in Section 3. Otherwise, we find the separator H with a smaller size from $X(c_s)\backslash\{c_s\}$ and $X(c_s)\backslash\{c_s\}$ in Lines 6–7. In Line 8, we select the pruned separator with a smaller size as the set of hoplinks. We find p^{opt}_h by concatenating paths in Lines 9–10 and return the LCSP among all p^{opt}_h in Line 11.

Example 5.1. Back to Example 2.9, for the query (v_1,v_2), the LCA of $X(v_1)$ and $X(v_2)$ is $X(v_7)$. Then, $X(c_s) = X(v_4)$ and $X(c_t) = X(v_6)$ with equal sizes. The procedure is also shown in Figure 5. we set $Hoplinks = H^{\text{sour}}(X(v_4)\backslash\{v_4\},v_1) = \{v_7\}$. For $h = v_7$, we fetch $\mathcal{P}^1_{v_1v_7} = \{(q_2,6),(q_3,16)\}$ and $\mathcal{P}^2_{v_7v_2} = \{(q_0,3),(q_1,3),(q_2,8)\}$ from

Algorithm 5: Query Processing

```
input  : A query $(s,t)$, PCSPs, and pruned separators
output : LCSP $p^{opt}$
1 $X(c) \leftarrow$ the LCA of $X(s)$ and $X(t)$
2 if $X(c) = X(t)$ or $X(c)=X(s)$ then
3     return $p^{1q}_{st}$ in $\mathcal{P}^1_{st}$ with $q \in F$ or $p^{2q}_{st}$ in $\mathcal{P}^2_{st}$ with $q = q_0$
4 Let $c_s$ and $c_t$ be the two child nodes of $X(c)$ in the two
    branches containing $s$ and $t$, respectively
5 $H \leftarrow \arg\min_{H' \in \{X(c_s)\backslash\{c_s\}, X(c_t)\backslash\{c_t\}\}} |H'|$
6 $Hoplinks \leftarrow \arg\min_{H' \in \{H^{\text{sour}}(H,s), H^{\text{des}}(H,t)\}} |H'|$
7 foreach $h \in Hoplinks$ do
8     $p^{opt}_h \leftarrow$ the shortest path in
        $\{p^{1q}_{sh} \oplus p^{2q}_{ht} | q \in Q, (q,p^{1q}_{sh}) \in \mathcal{P}^1_{sh}, (q,p^{2q}_{ht}) \in \mathcal{P}^2_{ht}\}$
9 return $p^{opt} \leftarrow \arg\min_{h \in Hoplinks} d(p^{opt}_h)$
```

the index. We obtain $d(p^{opt}_{v_7}) = 14$ by concatenating $(q_2,6)$ and $(q_2,8)$ and finally return $p^{opt} = p^{opt}_{v_7}$ with $d(p^{opt}) = 14$.

Theorem 5.2. *Algorithm 5's time complexity is $O(|Q|\omega)$.*

Proof. Note that the LCSPs in Lines 3 and 5 can be preprocessed in the index and given in $O(1)$ time in query processing. Therefore, the query time mainly depends on Lines 9–10 with $O(|Q||H|)$ time (by the hashing technique), which is bounded by $O(|Q|\omega)$. □

Theorem 5.3. *Algorithm 5 correctly returns the LCSP p^{opt}.*

Proof. If $X(t)$ (or $X(s)$) is an ancestor of $X(s)$ (or $X(t)$), we correctly find it since $\mathbb{1}_{q_0,q_f}(l(p^{opt})) = 1$ for one $q_f \in F$ and it must be in $\mathcal{P}^1_{st}$ or $\mathcal{P}^2_{st}$ by definitions. Otherwise, we can find a separator H , and p^{opt} must traverse one $h \in H$. However, since we use the pruned separator as $Hoplinks$, h may not exist in $Hoplinks$. By Lemma 4.10, we know that p^{opt} also traverses one h'. If h' is not in $Hoplinks$, we could apply Lemma 4.10 repeatedly until one h'' that is in $Hoplinks$ and considered by Lines 10–11. Consider the s-h'' subpath p_1 and h''-t subpath p_2 of p^{opt}, we can find the only state q such that $\mathbb{1}_{q_0,q}(l(p_1)) = 1$ and $\mathbb{1}_{q,q_f}(l(p_2)) = 1$ for $q_f \in F$. Since p_1 and p_2 must be the shortest paths w.r.t. the state q (otherwise we can get a contradiction), p_1 and p_2 are in $\mathcal{P}^1_{sh}$ and $\mathcal{P}^2_{ht}$, respectively. We must consider p^{opt} in Line 11. □

Path Retrieval. To save space cost, our PCSP index only stores five values for each shortest path p: its distance $d(p)$, source, destination, and a pair of states (q,q') such that $\mathbb{1}_{q,q'}(l(p)) = 1$. They are sufficient for all the algorithms. Finally, we also get these five values of the LCSP p^{opt}. To retrieve p^{opt} as a sequence of vertices, we store two more values $v^m(p)$ and $q^m(p)$ for each shortest path $p \in \mathcal{P}_{vw}, \mathcal{P}_{wv}$ to be the middle vertex w in Lines 19–20 of Algorithm 2 and the middle state q'_1 in Lines 11–12 of Algorithm 1 used to link p, respectively. For each edge $e = (u,v)$, as a path of a single edge, we set the two values for $\langle u,v\rangle$ and $\langle v,u\rangle$ as null. Each time we call PCSPJoin, in Line 5 of updating a path p concatenated by two paths p_1 and p_2, we update $v^m(p)$ and $q^m(p)$ accordingly. Finally, when retrieving the LCSP p^{opt}_{st}, we can unfold this s-t path p^{opt}

Table 2: Datasets

Dataset	Region	$\lvert V \rvert$	$\lvert E \rvert$	Storage
NY	New York City	264,346	733,846	17.9 MB
BAY	San Francisco Bay	321,270	800,172	19.6 MB
COL	Colorado	435,666	1,057,066	26.4 MB
FLA	Florida	1,070,376	2,712,798	68.5 MB

by adding $v^m(p^{opt})$ in the middle (i.e., $(s, v^m(p^{opt}), t)$) and recursively unfold the s-$v^m(p^{opt})$ path and $v^m(p^{opt})$-t path by using the middle vertex and state until the middle vertex is null.

Directed Graphs. We still build the tree decomposition by regarding each directed edge as undirected, which can only increase the connectivity and make all results about separators hold [13]. However, in Algorithm 2, we only maintain $P_{(v,u)}$ for each directed edge $e = (v, u)$ (instead of building $P_{(v,u)}$ and $P_{(u,v)}$ in Lines 11–12 of Algorithm 2) and concatenate two paths by considering their directions in PCSPJoin. We use the same Lines 16–20 of Algorithm 2 to generate $\mathcal{P}_{vw}$ and $\mathcal{P}_{wv}$ for v-w and w-v PCSPs, respectively, and then extract the two set $\mathcal{P}^1_{vw}$ and $\mathcal{P}^2_{wv}$ from them. The separator pruning and query processing algorithms remain the same because all these sets $\mathcal{P}_{vw}, \mathcal{P}_{wv}, \mathcal{P}^1_{vw}, \mathcal{P}^2_{wv}$ use directed edges.

6 EXPERIMENTS

6.1 Experimental Setup

We conducted all experiments on a machine with two Intel Xeon E5-2650 v4 2.2 GHz processors and 512 GB RAM running CentOS Linux distribution. All algorithms were implemented in C++ and compiled with GNU C++ compiler.

Datasets. Following existing work [43], we mainly used four publicly available real road networks from DIMACS [1] with their statistics shown in Table 2. For each edge (also a road segment), its spatial distance (in meters) and road category were used as the edge weight and label in our experiments, respectively. The road category code has two digits, where the first one denotes four main road types: 1) $\mathcal{A}$, Primary Highway With Limited Access; 2) $\mathcal{B}$, Primary Road Without Limited Access; 3) C, Secondary and Connecting Road; and 4) $\mathcal{D}$, Local, Neighborhood, and Rural Road, and the second digit ranging from 1 to 5 represents a finer classification of roads. For example, the code "$\mathcal{D}1$" denotes a local road, which is the most frequent edge label in any network.

Algorithms. We compared the following three LCSP solutions:

- Dijkstra [7]: a Dijkstra-based algorithm that searches the network by storing distances for vertex-state pairs.
- LSD [43]: the state-of-the-art index that searches the vertices in the tree decomposition and maintains the shortest distances w.r.t. different label sets.
- PCSP: our proposed index based on efficient table lookups and separator pruning.

For PCSP, we set the parameter $|R|$ = 1,000,000 and $\alpha = 0.9$ in Algorithm 4 which makes the pruned separators cover around 90% of the random queries in R. Note that PCSP is built based on all possible queries that satisfy the default language L. We studied the effect of α and justified this setting in Section 6.2. For the efficiency of LSD, we implemented set operations by bit operations.

6.2 Query Efficiency

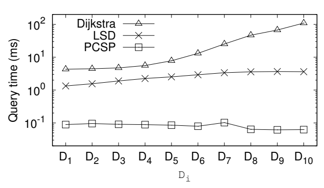
Exp-1: Query efficiency when varying the query distance. This experiment aims to evaluate algorithms' query efficiency by varying the shortest distance between sources and destinations of queries. Following [43], for each network, we generated 10 query sets with increasing query distances. Specifically, we first found the maximum distance l_{max} between any two vertices in a network and set a variable $x = (l_{max}/l_{min})^{1/10}$ where $l_{min} = 1,000$, e.g., $l_{max} = 154,745$ and $x = 1.6556$ in NY. We then generated 10 query sets $D_1, D_2, \ldots, D_{10}$, where the query distances in D_i lie in $(l_{min} \times x^{i-1}, l_{min} \times x^i]$ for $1 \leq i \leq 10$. Following [43], to compare LSD that only supports Kleene languages, we set the default language $L = (\cup_{a \in \Sigma_{10}} a)^*$ where Σ_{10} is the set of the top-10 frequent labels. Here, the frequency refers to number of edges in all the road networks having this label. We report the average query time in Figure 6.

For each network, it can be observed that Dijkstra and LSD tend to have a larger query time as the query distance increases. This is because they are all search-based solutions that search vertices from the source and destination to update distances recursively. As the query distance becomes larger, they all need to search more vertices. However, PCSP's query time is independent of the query distance since it preprocesses partial path answers in the hash tables and performs table lookups when checking very few related hoplinks. The query time is basically stable since the number of related hoplinks does not increase. Moreover, PCSP is the fastest one among the three algorithms. It can answer each LCSP query in around 100 microseconds and outperform LSD by almost two orders of magnitude on D_{10} of COL.

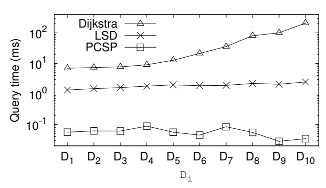
Exp-2: Query efficiency when varying the number of allowed labels. We used D_{10} as the default query set because the query distances of D_{10} are longer than those of the others, which implies that D_{10} is the hardest set and covers the whole range of the network. We tested the effect of the number of allowed labels in the Kleene languages by setting $L = (\cup_{a \in \Sigma_k} a)^*$, where Σ_k is the set of the top-k frequent labels for $k = 1, 2, \ldots, 10$. Figure 7 presents the average query time. It can be seen that all algorithms are insensitive to the number of allowed labels. The main reason is that we use the same query set with the same query distance. The label-constrained shortest distances for most queries do not vary much with more allowed labels. The search space of Dijkstra and LSD remains unchanged. For PCSP, it only uses a small number of hoplinks, which is independent of the number of allowed labels. We can draw a similar conclusion about the superiority of PCSP.

Exp-3: Query efficiency when varying the number of states. Since the number of states $|Q|$ is an important factor in query processing, we also studied its effect by considering the *linear regular language* in the form $a_1^* a_2^* \ldots$, which is also widely used [6, 37]. It has a useful property that the number of states is equal to the number of used labels (or the half length of the regular expression), which we vary from 1 to 10 by using the top-10 frequent labels incrementally. We still use D_{10} as the query set and show the average processing time in Figure 8. We omit LSD here because LSD cannot handle linear regular languages.

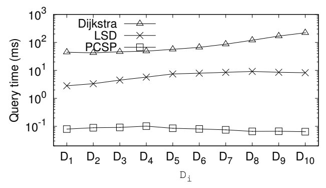
It can be discovered that the query times of Dijkstra and PCSP are irrelevant to the number of states. For Dijkstra, the query distance is the dominant factor in query efficiency, but it is the same for all



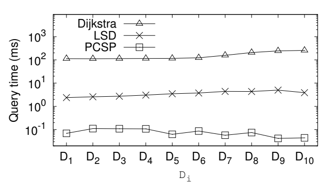
(a) NY



(b) BAY

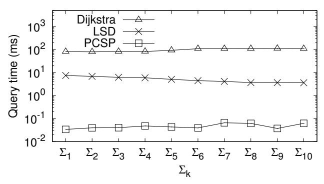


(c) COL

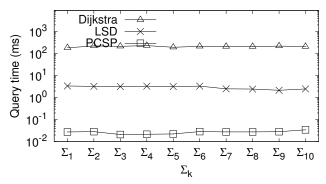


(d) FLA

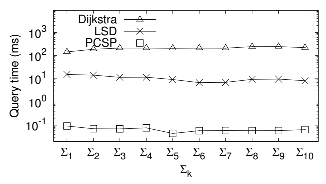
Figure 6: Query processing time (ms) when varying the query distance D



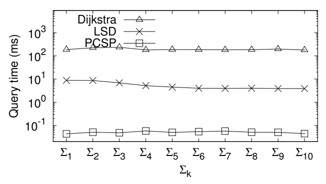
(a) NY



(b) BAY

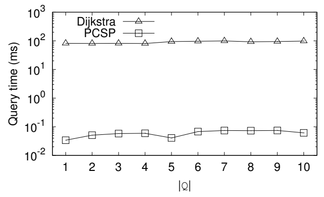


(c) COL

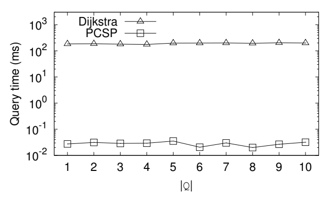


(d) FLA

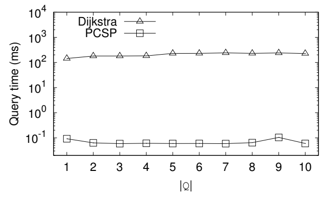
Figure 7: Query processing time (ms) when varying the number of allowed labels $|\Sigma_A|$



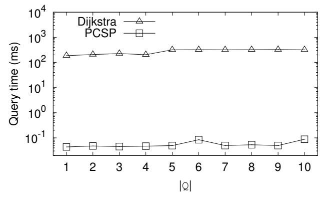
(a) NY



(b) BAY

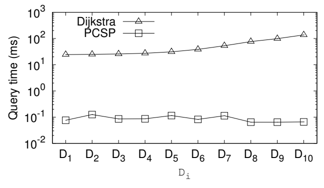


(c) COL

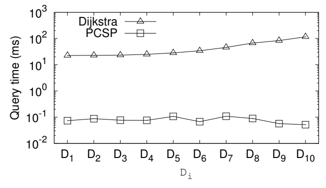


(d) FLA

Figure 8: Query processing time (ms) when varying the number of states $|Q|$

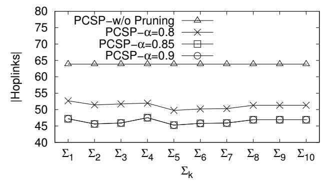


(a) Highway usage

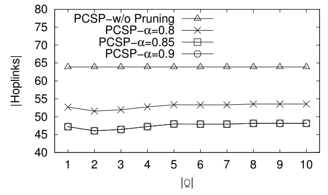


(b) Regional transfer

Figure 9: Complex languages on NY



(a) Varying $|\Sigma_k|$



(b) Varying $|Q|$

Figure 10: Ablation study on NY

settings since we all use D_{10}. For PCSP, the number of hoplinks mainly affects its query time. Similarly, it is the same since we all use D_{10}. PCSP still runs fast within 100 microseconds.

Exp-4: Query efficiency on complex languages. Following existing work [5], we also evaluated the performance for some complex languages. Let the label set $\Sigma_{\mathcal{A}} = \{\mathcal{A}1, \mathcal{A}2, \mathcal{A}3, \mathcal{A}4, \mathcal{A}5\}$ and define $\Sigma_{\mathcal{B}}$, Σ_{C}, and $\Sigma_{\mathcal{D}}$ similarly. We consider two languages [5]: 1) highway usage, $(\cup_{a\in\Sigma_C\cup\Sigma_{\mathcal{D}}} a)^*(\cup_{a\in\Sigma_{\mathcal{A}}\cup\Sigma_{\mathcal{B}}} a)^+(\cup_{a\in\Sigma_C\cup\Sigma_{\mathcal{D}}} a)^*$ (since it only allows us to use highways and primary roads in the middle) and 2) regional transfer, $(\cup_{a\in\Sigma_{\mathcal{B}}\cup\Sigma_C\cup\Sigma_{\mathcal{D}}} a)^*$ (since highways of type $\mathcal{A}$ are forbidden). The results on query sets of increasing query distances are reported in Figure 9. We have similar findings to previous ones. Dijkstra's query time increases since the distance increases due to larger search space, whereas PCSP is still efficient with stable and small query processing times.

Exp-5: Ablation study of separator pruning. We studied the effect of our proposed separator pruning technique by considering four variants, called "PCSP-w/o Pruning", "PCSP-α=0.8", "PCSP-α=0.85", "PCSP-α=0.9", where the first one does not utilize the pruning technique, and the later three preprocess Top-K separators by varying α in $\{0.8, 0.85, 0.9\}$ in Algorithm 4. The average number of used hoplinks per query for D_{10} is shown in Figure 10. Note

Table 3: Index construction cost

Data	Alg.	ω, η	Index Time	Pruning Time	Index Size
NY	LSD	148, 330	37s	-	34.2 MB
	PCSP		87s	13s	2.3 GB
BAY	LSD	100, 238	14s	-	27.8 MB
	PCSP		70s	8s	2.0 GB
COL	LSD	221, 400	24s	-	40.2 MB
	PCSP		173s	27s	4.8 GB
FLA	LSD	149, 399	34s	-	109.0 MB
	PCSP		410s	30s	10.3 GB
EST	LSD	222, 1113	259s	-	327.2 MB
	PCSP		1,917s	438s	76.8 GB
WST	LSD	257, 1094	344s	-	547.0 MB
	PCSP		3,552s	571s	130.5 GB

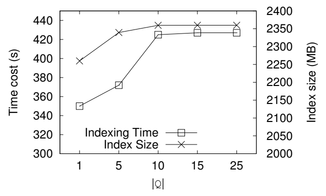
that as α increases, K increases since we need to preprocess more separators with K largest $N(H)$ to cover a ratio α of the random queries in Algorithm 4.

We can find that the number of hoplinks for *PCSP-w/o Pruning* is always around 65 because we use the same default query set D_{10}. In Figure 10a, for Σ_5, the number of hoplinks is reduced by 15 and 20 when we consider $\alpha = 0.8$ and $\alpha = 0.85$, respectively. The reason is that more preprocessed separators would make more queries available for pruning, hence a reduction in the number of used hoplinks. However, using $\alpha = 0.9$ further does not reduce the number of hoplinks compared with using $\alpha = 0.85$. Their lines overlap with each other. This is because the 1000 queries in D_{10} are basically all covered by the 1,000,000 random queries. In Figure 10b, where we vary $|Q|$, we can obtain similar findings. In practice, using $\alpha = 0.9$ to preprocess top-K separators would cover most queries, and using a larger α would increase preprocessing costs but gain little improvement over query efficiency.

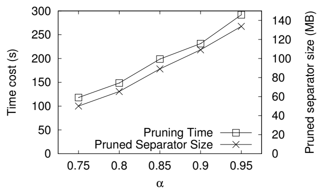
6.3 Index Cost

Index construction cost. We summarized the index construction costs of PCSP and LSD for the default language in Table 3. Note that Dijkstra builds no index. We also evaluated index construction costs on larger networks of Eastern USA ("EST" for short with 3,598,623 vertices and 8,778,114 edges) and Western USA ("WST" for short with 6,262,104 vertices and 15,248,146 edges) from DIMACS. The third and fourth columns show the treewidth ω and treeheight η of the tree decomposition, respectively. They are the same for LSD and PCSP for all data since we use the same tree decomposition. For PCSP, we separately report the indexing (Algorithm 2) and pruning (Algorithm 4) time costs in the fifth and sixth columns, respectively. The index sizes are given in the last column.

For high query efficiency, PCSP consumes much space, but it achieves orders of magnitude improvement over the query efficiency, and it can handle more complex languages. Moreover, it is common that the index cost is large for shortest path problems with complex constraints [32, 33, 41]. The indexing cost can be readily acceptable for modern machines with powerful computational capability and large memory space. Furthermore, the source and destination of each query are usually located within the same city and not far from each other in most scenarios.

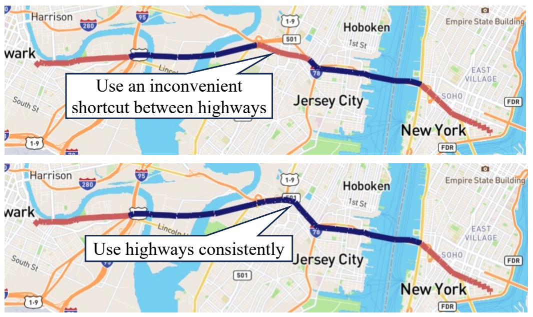


(a) Indexing cost (Alg. 2)



(b) Pruning cost (Alg. 4)

Figure 11: PCSP's index cost on NY



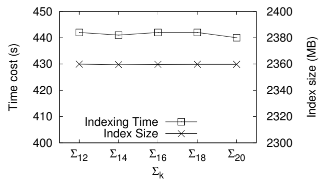
Use an inconvenient shortcut between highways
Use highways consistently
Harrison
Hoboken
Jersey City
New York
Empire State Building

Figure 12: Case study on NY

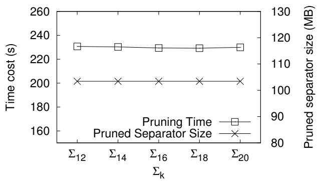
Exp-6: Indexing cost when varying the number of states. Since the number of states $|Q|$ is an important factor in both the time and space complexities, we explored its effect on the indexing cost (Algorithm 2) by using the linear regular languages introduced before in Figure 11a with $K = 10$. Note that the indexing time and index size follow the left and right y-axes, respectively. It can be seen that both the indexing time and size become larger as $|Q|$ increases. Their results are consistent with complexities. When $|Q|$ reaches 10, the index time and size do not increase because the top-10 frequent labels cover nearly 99% of labels, as stated before. Since all labels are used sequentially in the linear regular languages, the remaining 1% would make little difference in the index cost.

Exp-7: Pruning cost when varying the number of pruned separators. We reported the pruning cost (Algorithm 4) when varying the ratio $\alpha \in (0.75, 1)$ in Figure 11b with $|Q| = 10$. Note that the pruning time and pruned separator size follow the left and right y-axes, respectively. As α increase, we need to process more separators (the top-K separators with K largest frequencies) to ensure that the sum of their frequencies should be larger than the ratio α. It can be deducted from the complexities that the pruning time and pruned separator size increase as K increases. Besides, we find that the pruned separator size is much smaller than the index size in Figure 11a since we only need to store a Boolean flag for each hoplink in each pruned separator.

Exp-8: Case study. Figure 12 demonstrates a case study of LCSP queries in New York. We use blue and red lines to represent edges with labels in $\Sigma_{\mathcal{A}} \cup \Sigma_{\mathcal{B}}$ (for primary highways) and $\Sigma_{\mathcal{C}} \cup \Sigma_{\mathcal{D}}$ (for local roads), respectively. The top figure shows the shortest path returned by LSD when all labels are allowed, whereas the bottom one presents the LCSP (with the same s and t) returned by PCSP under the regular language constraint of highway usage, i.e.,



(a) Indexing cost (Alg. 2)



(b) Pruning cost (Alg. 4)

Figure 13: PCSP's index cost for more labels on NY

$(\cup_{a\in\Sigma_C\cup\Sigma_{\mathcal{D}}} a)^*(\cup_{a\in\Sigma_{\mathcal{A}}\cup\Sigma_{\mathcal{B}}} a)^+(\cup_{a\in\Sigma_C\cup\Sigma_{\mathcal{D}}} a)^*$, as introduced before. We can observe that the path returned by LSD uses an inconvenient shortcut between two parts of highways, which is unrealistic since highways can charge drivers a toll twice. The LCSP returned by PCSP can overcome this weakness by specifying a regular language to express the routing requirement of using highways consistently. Specifically, we do not allow local roads among highways since we remove the labels of $\Sigma_C \cup \Sigma_{\mathcal{D}}$ in the middle part $(\cup_{a\in\Sigma_{\mathcal{A}}\cup\Sigma_{\mathcal{B}}} a)^+$.

Exp-9: Index cost when varying the number of labels. Since we evaluated the index cost for the top-10 frequent labels Σ_{10} previously by default, we also studied the effect of more labels by varying k in $\{12, 14, 16, 18, 20\}$ in $L = (\cup_{a\in\Sigma_k} a)^*$ in Figure 13. Note that $\Sigma_{20} = \Sigma$ uses all labels. It can be observed that both the indexing cost (Algorithm 2) and pruning cost (Algorithm 4) are not affected by the number of labels. This is mainly because all the factors in the time and space complexities of Algorithms 2 and 4 (Theorems 4.9 and 4.14, respectively) all remain the same.

7 RELATED WORK

7.1 Shortest Path

Without additional constraints, the classical shortest path problem has been extensively studied since Dijkstra's algorithm [18]. Early index-free solutions included bidirectional search [14] and A* search [23], which mainly reduce the search space in Dijkstra's framework (by starting two simultaneous searches from the source and destination and utilizing goal-directed priority weights, respectively). However, they run slowly in large road networks because they need to update many distances from scratch. Subsequent research focused on designing indexes for higher efficiency, such as ALT [21], Arc Flags [25], Transit Nodes [8], and Reach [22]. State-of-the-art indexes could be divided into two categories: 1) pruning Dijkstra's search space by abstracting detailed paths in different hierarchies [12, 17, 20, 36, 39, 42] and 2) looking up a small number of relevant paths preprocessed in hash tables [4, 13, 28–30, 35, 44–47]. Those of the former category incur less index cost (in terms of time and space) but have longer query processing times than those of the latter. Among those of the latter one based on table lookups, [35] proposed H2H to first use the tree decomposition in road networks due to its small treewidth, which allows us to focus on a small separator set of hoplinks, as introduced in Section 4.1. Although we also adopted the tree decomposition to find this small separator to improve query efficiency, H2H is significantly different from our PCSP. In the unconstrained shortest path problem, it is clear that the index should store only one single shortest path between each vertex $v \in V$ and each hoplink in the separator. For LCSP, especially for the more complicated regular languages, it is unclear what kinds of paths could be preprocessed efficiently in the index and further used to answer queries quickly by table lookups. Our proposed PCSP clearly defines such paths and gives a query procedure that utilizes them properly.

7.2 Label-Constrained Shortest Path

Given a regular language, the problem of finding label-constrained paths in graph databases were first proposed by [34]. The hardness of different problem variants were analyzed in [7]. It also proposed the first polynomial LCSP algorithm for regular languages, which mainly runs Dijkstra's search on a composite graph where each vertex is now a vertex-state pair, and each edge exists only when there are connections on both the network and NFA. An implementation of this algorithm was later evaluated for linear regular languages [6]. It was also shown that the composite graph does not have to be built explicitly [40]. Note that they adopted NFA for its simple implementation, we used a minimized DFA that usually has fewer states [26]. Later, many successful techniques for shortest path queries were shown to be useful for LCSP. Their extensions for LCSP were based on bidirectional and A* search [5], Transit Nodes [15], Contraction Hierarchies [16, 37], ALT [27]. There were also some approximate LCSP solutions [11, 31] and work for dynamic index maintenance [19, 24, 37]. They were beaten by later indexes in terms of query efficiency. [24] proposed the Edge-Disjoint Partitioning (EDP) index for Kleene languages, which links paths from some partitions of the network. The state-of-the-art index LSD [43] outperforms EDP by orders of magnitude. It mainly uses the tree decomposition to reduce the search space. It recursively updates the distances from s (or t) to vertices in the tree nodes along the simple tree path between s (or t) to the LCA node. Note that though LSD is also based on the tree decomposition, our PCSP is entirely different since we directly looks up the hash tables for the shortest distances to provide fast query processing.. LSD's query time complexity is $O(\eta\omega\rho)$, where ρ is the number of shortest distances maintained in each node, whereas ours is $O(|Q|\omega)$ with $|Q| \leq \rho$. Our proposed PCSP could run faster than LSD by orders of magnitude and also support the more general regular languages.

8 CONCLUSION

In this work, we present PCSP, a practical index that answers LCSP queries efficiently in road networks with label constraints expressed by regular languages. PCSP mainly preprocesses two types of shortest paths whose labels can be accepted by the DFA from an initial state to an intermediate state and from an intermediate state to a final state. In this way, we can process each query by concatenating two PCSPs. We also propose a pruning technique that can prune the set of related hoplinks further. Extensive experiments on real networks show that PCSP answers each query within 100 microseconds. For future work, we may explore how to efficiently maintain the index to dynamically update edge labels and distances.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their constructive comments on this paper. This work is supported by WEB24EG01-H.

REFERENCES

[1] 2010. 9th DIMACS Implementation Challenge. http://www.diag.uniroma1.it/challenge9/download.shtml.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

[4] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. 2014. Fast Shortest-path Distance Queries on Road Networks by Pruned Highway Labeling. In *ALENEX*. 147–154.

[5] Christopher L. Barrett, Keith R. Bisset, Martin Holzer, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner. 2008. Engineering Label-Constrained Shortest-Path Algorithms. In *AAIM*. 27–37.

[6] Christopher L. Barrett, Keith R. Bisset, Riko Jacob, Goran Konjevod, and Madhav V. Marathe. 2002. Classical and Contemporary Shortest Path Problems in Road Networks: Implementation and Experimental Analysis of the TRANSIMS Router. In *ESA*. 126–138.

[7] Christopher L. Barrett, Riko Jacob, and Madhav V. Marathe. 2000. Formal-Language-Constrained Path Problems. *SIAM J. Comput.* 30, 3 (2000), 809–837.

[8] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. 2007. Fast routing in road networks with transit nodes. *Science* 316, 5824 (2007), 566–566.

[9] Michael A. Bender and Martin Farach-Colton. 2000. The LCA Problem Revisited. In *LATIN*. 88–94.

[10] Hans L. Bodlaender and Arie M. C. A. Koster. 2010. Treewidth computations I. Upper bounds. *Inf. Comput.* 208, 3 (2010), 259–275.

[11] Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Antti Ukkonen. 2014. Distance oracles in edge-labeled graphs. In *EDBT*. 547–558.

[12] Zi Chen, Bo Feng, Long Yuan, Xuemin Lin, and Liping Wang. 2023. Fully Dynamic Contraction Hierarchies with Label Restrictions on Road Networks. *Data Sci. Eng.* 8, 3 (2023), 263–278.

[13] 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.

[14] George Dantzig. 1963. *Linear programming and extensions.* Princeton university press.

[15] Daniel Delling, Thomas Pajor, and Dorothea Wagner. 2009. Accelerating Multi-modal Route Planning by Access-Nodes. In *ESA*. 587–598.

[16] Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. 2012. User-Constrained Multi-Modal Route Planning. In *ALENEX*. 118–129.

[17] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2014. Customizable Contraction Hierarchies. In *SEA*. 271–282.

[18] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269–271.

[19] Bo Feng, Zi Chen, Long Yuan, Xuemin Lin, and Liping Wang. 2023. Contraction Hierarchies with Label Restrictions Maintenance in Dynamic Road Networks. In *DASFAA*. 269–285.

[20] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transp. Sci.* 46, 3 (2012), 388–404.

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

[22] Ronald J. Gutman. 2004. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *ALENEX/ANALC*. 100–111.

[23] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.* 4, 2 (1968), 100–107.

[24] Mohamed S. Hassan, Walid G. Aref, and Ahmed M. Aly. 2016. Graph Indexing for Shortest-Path Finding over Dynamic Sub-Graphs. In *SIGMOD*. 1183–1197.

[25] Moritz Hilger, Ekkehard Köhler, Rolf H Möhring, and Heiko Schilling. 2009. Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge* 74 (2009), 41–72.

[26] John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations.* 189–196.

[27] Dominik Kirchler, Leo Liberti, Thomas Pajor, and Roberto Wolfler Calvo. 2011. UniALT for regular language contrained shortest paths on a multi-modal transportation network. In *ATMOS*. 64–75.

[28] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling Distance Labeling on Small-World Networks. In *SIGMOD*. 64–75.

[29] 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 *SIGMOD*. 1367–1381.

[30] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. 2017. An Experimental Study on Hub Labeling based Shortest Path Algorithms. *PVLDB* 11, 4 (2017), 445–457.

[31] Ankita Likhyani and Srikanta J. Bedathur. 2013. Label constrained shortest path estimation. In *CIKM*. 1177–1180.

[32] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, Pingfu Chao, and Xiaofang Zhou. 2021. Efficient Constrained Shortest Path Query Answering with Forest Hop Labeling. In *ICDE*. 1763–1774.

[33] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2022. FHL-Cube: Multi-Constraint Shortest Path Querying with Flexible Combination of Constraints. *PVLDB* 15, 11 (2022), 3112–3125.

[34] Alberto O. Mendelzon and Peter T. Wood. 1989. Finding Regular Simple Paths in Graph Databases. In *VLDB*. 185–193.

[35] 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.

[36] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. 2020. Efficient Shortest Path Index Maintenance on Dynamic Road Networks with Theoretical Guarantees. *PVLDB* 13, 5 (2020), 602–615.

[37] Michael N. Rice and Vassilis J. Tsotras. 2010. Graph Indexing of Road Networks for Shortest Path Queries with Label Restrictions. *PVLDB* 4, 2 (2010), 69–80.

[38] Neil Robertson and Paul D. Seymour. 1984. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B* 36, 1 (1984), 49–64.

[39] Peter Sanders and Dominik Schultes. 2012. Engineering highway hierarchies. *ACM J. Exp. Algorithmics* 17, 1 (2012).

[40] Hanif D. Sherali, Chawalit Jeenanunta, and Antoine G. Hobeika. 2006. The approach-dependent, time-dependent, label-constrained shortest path problem. *Networks* 48, 2 (2006), 57–67.

[41] Libin Wang and Raymond Chi-Wing Wong. 2023. QHL: A Fast Algorithm for Exact Constrained Shortest Path Search on Road Networks. *Proc. ACM Manag. Data* 1, 2 (2023), 155:1–155:25.

[42] Victor Junqiu Wei, Raymond Chi-Wing Wong, and Cheng Long. 2020. Architecture-Intact Oracle for Fastest Path and Time Queries on Dynamic Spatial Networks. In *SIGMOD*. 1841–1856.

[43] Junhua Zhang, Long Yuan, Wentao Li, Lu Qin, and Ying Zhang. 2021. Efficient Label-Constrained Shortest Path Queries on Road Networks: A Tree Decomposition Approach. *PVLDB* 15, 3 (2021), 686–698.

[44] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. 2021. Dynamic Hub Labeling for Road Networks. In *ICDE*. 336–347.

[45] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2021. Efficient 2-Hop Labeling Maintenance in Dynamic Small-World Networks. In *ICDE*. 133–144.

[46] Yikai Zhang and Jeffrey Xu Yu. 2022. Relative Subboundedness of Contraction Hierarchy and Hierarchical 2-Hop Index in Dynamic Road Networks. In *SIGMOD*. 1992–2005.

[47] Bolong Zheng, Yong Ma, Jingyi Wan, Yongyong Gao, Kai Huang, Xiaofang Zhou, and Christian S. Jensen. 2023. Reinforcement Learning based Tree Decomposition for Distance Querying in Road Networks. In *ICDE*. 1678–1690.