Efficient Indexing for Flexible Label-Constrained Shortest Path Queries in Road Networks

Paper ID: 1607

ABSTRACT

The point-to-point shortest path query is widely used in many spatial applications, e.g., navigation systems. However, the returned shortest path minimizing only one objective fails to satisfy users' various routing requirements in practice. For example, the user may specify the order of using several transportation modes in the planned route. The *Label-Constrained Shortest Path (LCSP)* query under regular languages is powerful enough to express diversified routing demands in a labeled road network where each edge is associated with a *label* to denote its road type. The complex routing demand can be formulated by a *regular language*, and the edge labels along each path should be a word under the given regular language. Previous LCSP solutions were either inefficient in query processing or inflexible in their use of the languages since they made some assumptions about the given language. In this paper, we propose an efficient index-based solution called *Border-based State Move (BSM)*, which can answer LCSP queries quickly with flexible use of the language constraint. Specifically, our BSM builds indexes to skip the exploration between a vertex and its *border* vertices during query processing. Our experiments conducted on real road networks demonstrated the superiority of our proposed BSM. It can greatly reduce the query time over state-of-the-art solutions by two orders of magnitude.

KEYWORDS

Label-Constrained Shortest Path, Road Network

ACM Reference Format:

. 2023. Efficient Indexing for Flexible Label-Constrained Shortest Path Queries in Road Networks: Paper ID: 1607. In *Proceedings of ACM SIGMOD conference (SIGMOD'23).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.475/nnn_mmm

1 INTRODUCTION

The point-to-point shortest path query has been an integral part of many spatial applications, such as navigation systems, online taxi-calling platforms, and food delivery services. Its query processing mainly regards the road network as a weighted graph (where vertices and edges represent road intersections and segments, respectively) and returns the path with the minimum sum of weights of its traversed edges. However, users' routing demands have become more diversified; they now have more detailed requirements for their personalized routing plans before reaching their destinations as soon as possible. The basic shortest path query fails to express these various routing demands since it can only minimize one metric value of the path (e.g., the distance and the travel time). In reality, the routing plan of a user going home from work may specify that s/he wants to first take a taxi to one special road (where s/he may want to collect some mail packages at a courier station), traverse the special road on foot, and finally take a bus or ride a bike to home. This routing plan involves the use of several transportation modes and the traversal of a special road in a certain order. We thus study the Label-Constrained Shortest Path (LCSP) query under the regular language that is powerful enough to describe these personalized routing plans [3].

LCSP queries are issued in a labeled road network where each edge is associated with an additional *label* (in addition to its weight) to denote its road type. In the previous example, we may use symbols α, β, θ, and γ to represent an expressway, a normal road, a bike lane, and the special road, respectively. We can then use regular languages to describe personalized routing plans, such as $(\alpha|\beta)^*\gamma(\beta^*|\theta^*)$ (where "*" means that zero or more occurrences of the preceding element are accepted and "|" means that the element either before or after "|" is accepted). A path satisfies a regular language when the edge labels along it (also known as the *path label*) can be seen as a word under the language. We are interested in the shortest path under the language constraint, also known as the LCSP. Regular languages are powerful enough to express many practical routing plans since we can easily specify the order and frequencies of edge labels in the resulting path.

There has been substantial effort devoted to designing efficient LCSP solutions [1–4, 7–13]. Early algorithms extended some techniques of the (unconstrained) shortest path algorithms, such as Dijkstra's search [3, 7, 12], bidirectional and A* search [8], access nodes [9], and landmarks [4]. However, they were inefficient in finding LCSPs and surpassed by later solutions. Recently, a number of LCSP solutions achieved higher query efficiency by utilizing state-of-the-art index-based techniques [1, 2, 10, 11, 13]. They mainly preprocess useful information in the index for efficient query processing. However, they all made rigid assumptions about using regular languages to express constraints. Some focused on the Kleene language [2, 10, 11], which is a special case of regular languages and only allows and prohibits the use of certain labels without caring about their *orders* and *frequencies*. For example, a user may want to first use local roads consistently and then highways, but Kleene languages are insufficient to describe this demand, and the returned LCSP may use highways and local roads alternately. For another example, a user may want to go to a restaurant first and then a shopping mall, which cannot be expressed by Kleene languages and handled by these solutions. Since a regular language can be equivalently transformed to a *deterministic finite automaton*

SIGMOD'23, June 2023, Houston, Texas, USA

ACM ISBN 123-4567-24-567/mm/dd.
https://doi.org/10.475/nnn_mmm

(DFA) [14] (which is often used to judge whether a path label satisfies the regular language), one study [13] considered a specific type of DFA that assumes using one fixed edge label among different DFA states. Its applications are limited because a user may prefer several transportation modes (which correspond to different labels in one state). Another study [1] assumed a known fixed regular language for all queries before query processing, which implies that we have to rebuild the index whenever the regular language changes. For example, for a workload of 1,000 LCSP queries with 50 different languages in New York, it needs 5,000 seconds to rebuild the index 50 times to answer all queries [1], whereas a flexible solution can build the index once to answer all queries.

Motivated by the above challenges, we propose a more efficient index-based solution, called *Border-based State Move* (BSM), which supports flexible use of regular languages without any assumption. It is non-trivial to design such an index-based LCSP solution. The major reason is that the regular language of each query can be quite different, which makes it hard to preprocess useful information. Specifically, the DFA graphs of various regular languages can have completely different graph structures. Previous solutions made some assumptions about the DFA so that they can preprocess information for their expected graph structures to avoid redundant search on the DFA graph. To support regular languages of any forms, we propose BSM by noticing the necessity of traversing the whole DFA graph during query processing and seeking efficiency optimization between steps of the DFA graph traversal. We design several acceleration techniques, including the initial pruning, Connected Component (CC) graph, and pruning bounds, with thorough theoretical analyses. We conducted experiments in several large networks and verified that our BSM is consistently faster than the best-known solution. This work can be further extended to applications beyond the context of road networks. For example, in biological networks, researchers who study a disease may need to find the shortest pathway that traverses a kinase, then several proteins, and finally a transcription factor, denoted by labels k, p, t, respectively. The requirement can be expressed by kp^*t. In a movie knowledge graph, where vertices include people and movies and edges are labeled with a for "acted" and d for "directed" relationships. One may want to find a list of movies acted by Leonardo and directed by James (expressed by ad). We summarize our contributions below.

- We propose an index-based solution called BSM. It can answer flexible LCSP queries under regular languages and run faster than state-of-the-art solutions.
- We propose several speedup techniques, including (1) the initial pruning and the CC graph (for efficient retrieval of border vertices), (2) the optimization of the weight calculation, and (3) the pruning strategy (for faster query processing).
- We theoretically analyze the correctness and complexities of all the proposed algorithms.
- We conducted extensive experiments to demonstrate the efficiency and effectiveness of the proposed BSM. The results show that BSM can greatly reduce the query times of the state-of-the-art baselines by two orders of magnitude while incurring small index costs.

The remainder of the paper is organized as follows. Section 2 states the problem. Section 3 shows an overview of BSM's indexing and query processing. Their details are given in Section 4 and Section 5. Section 6 presents our experiments. Section 7 reviews the related work. Section 8 concludes our paper.

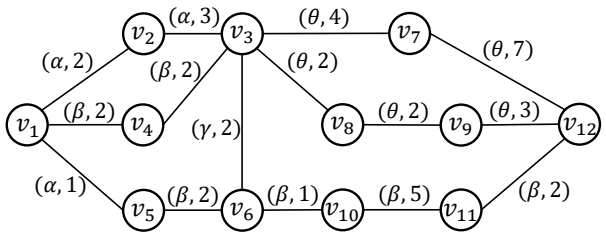


Figure 1: The labeled road network G

2 PRELIMINARIES

The labeled road network $G(V, E, \Sigma, l, w)$ is an undirected weighted graph, where V is a set of vertices, E is a set of edges, Σ is a set of labels, $l : E \to \Sigma$ is a function that assigns a label to each edge, and $w : E \to \mathbb{R}^+$ assigns a positive weight to each edge. An s-t path p is a finite sequence of vertices $\langle v_0 = s, v_1, v_2, \ldots, v_k = t \rangle$ such that each $e_i = (v_{i-1}, v_i) \in E$ for $1 \le i \le k$. For a path p, its weight is defined by $w(p) = \sum_{i=1}^{k} w(e_i)$, and its path label is defined by $l(p) = l(e_1) \cdot l(e_2) \ldots \cdot l(e_k)$ (or $l(e_1)l(e_2) \ldots l(e_k)$ for simplicity), where "$\cdot$" denotes the concatenation of two labels. The path label $l(p)$ is the concatenation of all edge labels along p. The terms "weight" and "distance" can be used interchangeably.

Example 1. *Figure 1 shows a labeled road network with 12 vertices and 15 edges. The label set $\Sigma = \{\alpha, \beta, \theta, \gamma\}$. For each edge e, its label $l(e)$ and weight $w(e)$ form a pair $(l(e), w(e))$ marked next to each edge. For example, the label and weight of (v_3, v_6) are γ and 2, respectively. Given a v_1-v_{12} path $p = \langle v_1, v_5, v_6, v_{10}, v_{11}, v_{12} \rangle$, its weight $w(p) = 11$ and its label $l(p) = \alpha\beta\beta\beta\beta$.*

To formalize the language constraint on the path, we use some notions of regular languages. A *word* is the concatenation of labels $\sigma_1\sigma_2 \ldots \sigma_k$ where $\sigma_i \in \Sigma$ for $i = 1, 2, \ldots, k$ (i.e., a path label). A language $\mathcal{L}$ is a set of words. Each regular language $\mathcal{L}$ can be defined by a *deterministic finite automaton* (DFA) that describes the words in $\mathcal{L}$. Specifically, a DFA $(Q, \Sigma, \delta, q_0, F)$ for $\mathcal{L}$ includes a state set Q, the alphabet Σ, a transition function $\delta : Q \times \Sigma \to Q$ (which defines the next state $\delta(q, \sigma)$ of a state q and a label σ), an initial state $q_0 \in Q$, and a set of accepting or final states $F \subseteq Q$. A word $\sigma_1\sigma_2 \ldots \sigma_k$ belongs to a language $\mathcal{L}$ (i.e., $\sigma_1\sigma_2 \ldots \sigma_k \in \mathcal{L}$) if and only if we can start with the initial state q_0, use the word's label σ_i sequentially for transitions (i.e., $\delta(q_0, \sigma_1)$, $\delta(\delta(q_0, \sigma_1), \sigma_2)$, ...), and stop at a final state $q \in F$ after using all labels of the word.

Each regular language can be easily expressed by a regular expression for practical use, which uses "$*$" (zero or more occurrences), "$\cdot$" (concatenation), and "$|$" (Boolean "or") for basic operations.

Example 2. *Figure 2 presents a DFA for the regular language $\mathcal{L}$ represented by the expression $(\alpha|\beta)^*\gamma(\beta^*|\theta^*)$. Suppose that $\alpha, \beta, \theta, \gamma$ represent expressways, normal roads, bike lanes, and a special road where a courier station is located, respectively. The language $\mathcal{L}$ means that the path should first traverse some normal roads and expressways, then the special road, and finally bike lanes or normal roads*

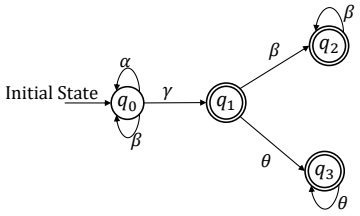


Figure 2: A DFA for $(\alpha|\beta)^*\gamma(\beta^*|\theta^*)$

Algorithm 1: Extended Dijkstra's Algorithm [3]

input : Two vertices s and t, a language $\mathcal{L}$
output: $w(p^{opt})$ of LCSP

1 $d[v][q] \leftarrow +\infty$ for $v \in V$ and $q \in Q$, $d[s][q_0] \leftarrow 0$
2 $Queue.push((s, q_0, 0))$
3 **while** $|Queue| > 0$ **do**
4 fetch (v, q, w) with the minimum weight w from $Queue$
5 **if** $v = t$ *and* $q \in F$ **then**
6 return w
7 **foreach** $(v, v') \in E$ *s.t.* $\delta(q, l(v, v'))$ *exists* **do**
8 $q' \leftarrow \delta(q, l(v, v'))$
9 **if** $d[v][q] + w(v, v') < d[v'][q']$ **then**
10 $d[v'][q'] \leftarrow d[v][q] + w(v, v')$
11 $Queue.push((v', q', d[v'][q']))$

consistently, corresponding to the user's routing plan in Section 1. The state set $Q = \{q_0, q_1, q_2, q_3\}$, where q_0 is the initial state and the final states are $q_1, q_2, q_3 \in F$. The arrows show the state transitions. For example, the word $\alpha\beta\gamma\theta\theta\theta$ satisfies $\mathcal{L}$ since the DFA can make state transitions from the initial state q_0 to a final state $q_3 \in F$.

Definition 1 (Label-Constrained Path). *Given a language $\mathcal{L}$, a path p is a* label-constrained path *if its label $l(p)$ belongs to $\mathcal{L}$, i.e., $l(p) \in \mathcal{L}$.*

Definition 2 (Label-Constrained Shortest Path (LCSP)). *Given a language $\mathcal{L}$ and two vertices s and t, a label-constrained shortest path (LCSP) is the s-t path p that has the minimum weight $w(p)$ among all label-constrained s-t paths.*

Example 3. *We still use the labeled road network G and the language $\mathcal{L}$ represented by the DFA in the previous examples. Let $s = v_1$ and $t = v_{12}$. Without the constraint, the shortest path is $\langle v_1, v_5, v_6, v_{10}, v_{11}, v_{12}\rangle$ with its weight of 11, but its path label $\alpha\beta\beta\beta\beta$ does not satisfy the language $\mathcal{L}$. Under the constraint, the label-constrained shortest path $p = \langle v_1, v_5, v_6, v_3, v_8, v_9, v_{12}\rangle$ with its minimum weight $w(p) = 12$ and its label $l(p) = \alpha\beta\gamma\theta\theta\theta$ accepted by the DFA from state q_0 to $q_3 \in F$.*

Problem Statement. Given a labeled network $G(V, E, \Sigma, l, w)$, an LCSP query is defined by $(s, t, \mathcal{L})$ with $s, t \in V$ and $\mathcal{L}$ which is a regular language as stated before. It asks for the LCSP p^{opt} between s and t under the constraint of $\mathcal{L}$. Our goal is to design an index based on G to answer each LCSP query efficiently.

Extended Dijkstra's Algorithm [3]. It mainly runs Dijkstra's algorithm on the *product* graph made up of the labeled network G and the DFA, which is also a graph. Specifically, in this product graph, the "vertex" set is $V \times Q$, and there is an "edge" between two "vertices" (v, q) and (v', q') if and only if $(v, v') \in E$ and there is a transition $\delta(q, l(v, v')) = q'$. In this way, the resulting path should satisfy the language constraint and have the minimum weight.

Table 1: Summary of main notations

Symbol	Meaning
$G = (V, E, \Sigma, l, w)$	the labeled road network
$\alpha, \beta, \theta, \gamma, \sigma$	edge labels
$l(e)$, $w(e)$	the edge label and weight of e
Q, F	the sets of DFA states and DFA final states
$\delta(q, \sigma)$	the transition function
$L_{\text{self}}(q)$, $L_{\text{move}}(q)$	the self-loop and state-move label sets of q
C_σ^i	the i-th group of $\{\sigma\}$-inner vertices
N_σ^i	the i-th set of $\{\sigma\}$-connected vertices
$B_L(v)$	the set of v's L-border vertices
$R_\sigma(N)$	the refined set of the node N
$d[v][q]$	the distance array for vertex-state pairs

The product graph does *not* have to be built *explicitly* by considering the vertex-state pairs (v, q) [12], which means that we do not need to identify each edge between any two vertex-state pairs. Algorithm 1 shows the whole procedure. As in Dijkstra's algorithm, in Line 1, it first initializes an array d to maintain the minimum weight for each vertex-state pair (v, q) (corresponding to an s-v path with its label accepted by the DFA from state q_0 to q) and sets $d[s][q_0] = 0$. It pushes (s, q_0) with its path weight of 0 to a priority queue in Line 2. Each pair (v, q) in the priority queue represents an s-v path with its path label accepted by the DFA from the initial state q_0 to q. In each iteration, the algorithm fetches the pair (v, q) with the minimum weight in the queue in Line 4. In Lines 7–11 of expanding (v, q), if there exists an adjacent edge label $l(v, v')$ for $(v, v') \in E$ that can be used to make a transition (i.e., $\delta(q, l(v, v')) = q'$), the algorithm pushes (v', q') with its new weight (i.e., $d[v][q] + w(v, v')$) to the queue when the new weight is smaller than the maintained minimum weight for (v', q'). The algorithm stops when it fetches the pair (v, q) such that v is t and $q \in F$ is a final state in Lines 5–6. However, it is inefficient because it does not utilize any index and can explore many unnecessary vertex-state pairs. Algorithm 1's time complexity is $O(|E||Q| \log(|V||Q|))$ following Dijkstra's complexity. Our solution can be much more efficient based on the power of a preprocessed index. Algorithm 1's space cost mainly depends on the array d. A simple implementation uses $O(|V||Q|)$ space. We can also use hashmap so that only visited vertex-state pairs with updated weights (in Line 10) incur the space cost. The number of visited vertex-state pairs is smaller than 200,000 in all experiments, which needs at most 1.6 MB.

We summarize all the main notations in Table 1.

3 OVERVIEW OF BSM

Starting from the pair (s, q_0), the extended Dijkstra's algorithm expands the current pair (v, q) by using an edge $e = (v, v')$ to find the next pair (v', q'). The DFA also makes a state transition by using the edge label at the same time. During the process, we can observe

that the state can still be the same (i.e., $q' = q$) after we use one or more edges. It actually depends on the "self-loop" state transitions in the DFA (i.e., $\delta(q, \sigma) = q$). We aim to improve query efficiency by avoiding the network search on these self-loops. Given a vertex-state pair (v, q), suppose that there are some self-loops on state q in the DFA. The main idea of BSM is to directly find all the last pairs (v', q) such that v''s adjacent edges can be used to make the state q change (which means no more self-loops after (v', q)). These vertices v' are also called v's "border" vertices. The search between v and v' can be preprocessed by an index (since it is a recursive process) and hence accelerated. We can imagine that there are "super edges" (or shortcuts) between v and v'. Overall, the query processing starts from the source, then goes to the border vertices repeatedly, and finally reaches the destination. In the meanwhile, the DFA state will always move one step forward until a final state.

According to the above process, BSM's index maintains the set of v's border vertices for each (v, q). Suppose that there is only one self-loop on state q with label σ (i.e., $\delta(q, \sigma) = q$). The key observation is that v and v's border vertices should be in a connected component that only contain edges with label σ. Therefore, BSM's index maintains connected components w.r.t. one edge label. However, there can be several self-loops on state q with different labels. We extend the idea for the case of only one self-loop to create a graph where nodes represent connected components, called the "Connected Component (CC) graph". It supports efficient retrieval of v's border vertices by traversing the graph nodes and fetching the corresponding border vertices in each connected component. To avoid the search between v and v's border vertex v', we use the concept of the super edge between them. Apart from identifying v' by the CC graph, we also need to find the weights of these super edges, which can be formulated by the Kleene-language constrained shortest path problem [2, 10, 11]. We adapt an existing solution [2] to achieve orders of magnitude speedups.

4 BSM INDEX CONSTRUCTION

For simplicity, we call the state transition $\delta(q, \sigma) = q' \neq q$ a *state move* (since $q' \neq q$), and the search can quickly stop if we can make a state move in each iteration of expanding (v, q) because the number of state moves between the initial state and a final state is often limited in the small DFA graph. To support efficient state moves, BSM essentially builds indexes for the set of v's border vertices. First, we formally define the border vertices. Then, we perform an initial pruning to identify some vertices that can never be border vertices. Next, we build a small CC graph to index the rest vertices to support efficient retrieval of their border vertices. Finally, we adapt an existing index to efficiently compute new weights between a vertex and its border vertex. The whole BSM index consists of the CC graph and the adapted index for weight calculation.

4.1 Inner and Border Vertices

For only one self-loop $\delta(q, \sigma) = q$, expanding (v, q) through several edges with the same label σ can make the state unmoved (i.e., still q). However, if there exists a label set L such that $\delta(q, \sigma) = q$ for each $\sigma \in L$, the state is unmoved after any sequence of edge labels in L. We want to skip the exploration where the state is unmoved. Formally, we define the following two sets for each state q.

Definition 3 (Self-loop and state-move label sets). *For each state q, its* self-loop label set $L_{\text{self}}(q) = \{\sigma \in \Sigma | \delta(q, \sigma) = q\}$, *and its* state-move label set $L_{\text{move}}(q) = \{\sigma \in \Sigma | \delta(q, \sigma) \neq q\}$. *Obviously,* $L_{\text{self}}(q) \cap L_{\text{move}}(q) = \emptyset$.

We next define the *inner* and *border* vertices w.r.t. a label set L and show that any self-loop ends at border vertices. Our BSM essentially finds necessary border vertices that have adjacent labels in $L_{\text{move}}(q)$ for state moves. It avoids the exploration from the current vertex to its border vertices and jumps to them directly.

Definition 4 (L-inner vertex). *Given a label set L, a vertex v is an L-inner vertex iff each of its adjacent edge labels belongs to L, i.e., $l(v, v') \in L$ for $(v, v') \in E$.*

In particular, each adjacent edge label of a $\{\sigma\}$-inner vertex is σ.

Definition 5 (L-connectivity). *Given a label set L, vertices u and v are L-connected iff there is a u-v path that only uses labels in L.*

Definition 6 (Set $B_L(v)$ of v's L-border vertices). *Given a label set L and two different vertices, namely u and v, we say that u is v's* L-border vertex *iff 1) there exists at least one of u's adjacent edge labels not in L (i.e., $l(u, u') \notin L$) and 2) u and v are L-connected. Let $B_L(v)$ denote the set of v's L-border vertices.*

Note that there are also some vertices other than the inner and border vertices. We ignore them because we make state moves on border vertices in each iteration.

Given a pair (v, q), if $L_{\text{self}}(q) \neq \emptyset$, it can be observed that the state moves only when the search visits one of v's $L_{\text{self}}(q)$-border vertices $u \in B_{L_{\text{self}}(q)}(v)$ since u has adjacent edge labels not in $L_{\text{self}}(q)$ for a state move. We can imagine that there is a "super edge" between v and u, representing the minimum-weight v-u path under the constraint that it only uses labels in $L_{\text{self}}(q)$. The main idea of BSM is to quickly retrieve the necessary border vertices from the index. A naive idea is to precompute $B_L(v)$ for each vertex v and each subset $L \subseteq \Sigma$. However, the indexing cost can be extremely large since each $|B_L(v)|$ can be $O(|V|)$ and the total cost is $O(|V|^2 2^{\Sigma})$. We focus on how to improve the efficiency of retrieving $B_L(v)$.

4.2 Initial Pruning

The initial pruning is based on two insights. The first one is that each $\{\sigma\}$-inner vertices for each $\sigma \in \Sigma$ can never be border vertices of others. The second one is that for each $\sigma \in \Sigma$, some connected $\{\sigma\}$-inner vertices have common sets of border vertices. Thus, we perform an initial pruning of these $\{\sigma\}$-inner vertices to avoid redundant computations and focus on the rest of the vertices.

We first identify the $\{\sigma\}$-inner vertices for each $\sigma \in \Sigma$ and then maintain their border vertices in the index. The following lemma shows why we focus on the $\{\sigma\}$-inner vertices.

Lemma 1. *For each $\sigma \in \Sigma$, a $\{\sigma\}$-inner vertex u cannot be v's L-border vertex for any $v \in V$ and $L \subseteq \Sigma$.*

Proof. If u is v's L-border vertex, u and v are L-connected. Since u is a $\{\sigma\}$-inner vertex, each of u's adjacent edge labels must be σ, which indicates that $\sigma \in L$ because u and v are L-connected. But, there does not exist another u's adjacent edge label not in L (as required in Definition 6), which is a contradiction. □

Algorithm 2: Initial Pruning

input : The network $G(V, E, \Sigma, l, w)$
output: C_σ^i for each $\sigma \in \Sigma$

1 $k_\sigma \leftarrow 0$ for each $\sigma \in \Sigma$, $V_{rest} \leftarrow V$
2 **foreach** $v \in V$ **do**
3 **if** v *is a* $\{\sigma\}$*-inner vertex* **then**
4 perform a BFS from v by only using edges with σ, for each visited vertex, add $\{\sigma\}$-inner ones into C_σ^i, and stop expansion at non-$\{\sigma\}$-inner ones
5 $V_{rest} \leftarrow V_{rest} \setminus C_\sigma^i$
6 $k_\sigma \leftarrow k_\sigma + 1$
7 return C_σ^i for each $\sigma \in \Sigma$

For all the non-$\{\sigma\}$-inner vertices, denoted by V_{rest}, we handle $B_L(v)$ for $v \in V_{rest}$ in the next section. For each $\{\sigma\}$-inner vertex v, we find its $B_L(v)$ by the following lemma and corollary, which suggest that its $B_L(v)$ can be found by $B_L(u)$ for $u \in V_{rest}$.

Lemma 2. *For two $\{\sigma\}$-inner vertices v and u, if $(v,u) \in E$, $B_L(v) = B_L(u)$ for any $L \subseteq \Sigma$.*

Proof. We prove that $B_L(v) \subseteq B_L(u)$ (which also implies the next Corollary 1), and the reverse is true by symmetry. For one of v's L-border vertex $v' \in B_L(v)$, v' has at least one adjacent edge labels not in L, and v' and v are L-connected. Since each v-v' path must traverse at least one of v's adjacent edges with its label σ, we know $\sigma \in L$, which means that v' and u are L-connected via (v,u) and has at least one adjacent edge label not in L. □

Corollary 1. *For a $\{\sigma\}$-inner vertex v, if $(u,v) \in E$, $B_L(v) \subseteq B_L(u)$ for any $L \subseteq \Sigma$. If $\sigma \in L$, $B_L(v) = B_L(u)$. If $\sigma \notin L$, $B_L(v) = \emptyset$.*

Proof. The first part follows Lemma 2. If $\sigma \in L$, $B_L(u) \subseteq B_L(v)$, for each $v' \in B_L(u)$, v' and u are L-connected. Since v is a σ-inner vertex and $l(u,v) = \sigma \in L$, v' and v are also L-connected. □

Lemma 2 further indicates that $B_L(v)$ is the same among connected $\{\sigma\}$-inner vertices. We can then put connected $\{\sigma\}$-inner vertices in the same group, which is easily done by breadth-first search (BFS). For each $\sigma \in \Sigma$, suppose that there are k_σ groups. Let C_σ^i denote the i-th group of connected $\{\sigma\}$-inner vertices. The set of non-$\{\sigma\}$-inner vertices is $V_{rest} = V \setminus \bigcup_{\sigma \in \Sigma} \bigcup_{i=1}^{k_\sigma} C_\sigma^i$. For each $v \in C_\sigma^i$ and any label set L, we find $B_L(v)$ by using $B_L(u)$ where $u \notin C_\sigma^i$ and $(u,v) \in E$ by Corollary 1. Section 4.3 will discuss how to retrieve these $B_L(u)$ for $u \in V_{rest}$.

Algorithm 2 summarizes the procedure of maintaining the groups C_σ^i for $\{\sigma\}$-inner vertices. Specifically, Line 1 initializes k_σ and V_{rest}. In Line 3, if v's all adjacent edge labels are σ, v is a $\{\sigma\}$-inner vertex by Definition 4. In Line 4, we start a BFS from a $\{\sigma\}$-inner vertex by using a queue. For each visited vertex, if it is a $\{\sigma\}$-inner vertex, we put it into the current group C_σ^i and the queue for further expansion (i.e., visiting its neighbor vertices). Otherwise, we stop expansion at this vertex by not putting it into the queue.

Example 4. *Figure 3 shows all inner vertices (with colors) in the network G. Algorithm 2 first visits v_2 since v_2 is an $\{\alpha\}$-inner vertex. It sets $C_\alpha^1 = \{v_2\}$ because v_1 and v_3 are non-α-inner vertices. Next, it visits the $\{\beta\}$-inner vertex v_4 and similarly sets $C_\beta^1 = \{v_4\}$. For the $\{\theta\}$-inner vertex v_7, $C_\theta^1 = \{v_7\}$. Similarly, for v_8, $C_\theta^2 = \{v_8, v_9\}$. Finally, $C_\beta^2 = \{v_{10}, v_{11}\}$. The rest of the vertices without colors form the set V_{rest}, and border vertices are among them.*

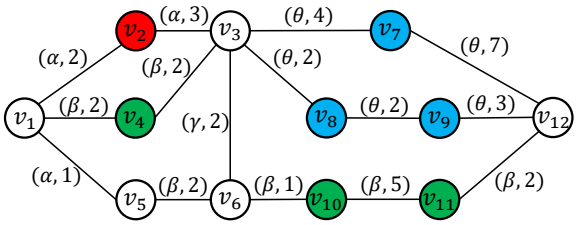


Figure 3: All $\{\sigma\}$-inner vertices for $\sigma \in \Sigma$ (i.e., colorful ones)

Theorem 1. *Algorithm 2's time complexity is $O(|V| + |E|)$. Its space complexity is $O(|V|)$.*

Proof. The time is $O(|V| + |E|)$ because we can perform a BFS to traverse all vertices and edges. The space is $O(|V|)$ because we put all vertices into different sets. □

4.3 Connected Component Graph

The goal of building the Connected Component (CC) graph is to efficiently retrieve $B_L(v)$ for any label set $L \subseteq \Sigma$ and $v \in V_{rest}$, where $V_{rest} = V \setminus \bigcup_{\sigma \in \Sigma} \bigcup_{i=1}^{k_\sigma} C_\sigma^i$ is the rest of vertices after the initial pruning. Given a vertex $v \in V_{rest}$, we know that $B_L(v) \subseteq V_{rest}$ by Lemma 1. However, it is still costly to check each vertex in V_{rest}. We separate the task of retrieving $B_L(v)$ into two steps following Definition 6: first retrieving the set of v's L-connected vertices in V_{rest} and then retaining those that have adjacent edge labels not in L.

For the first step, we propose the CC graph that considers the $\{\sigma\}$-connected vertices for each $\sigma \in \Sigma$ and combines them to form L-connected vertices in V_{rest}. It mainly includes a set of nodes N_σ^i for each $\sigma \in \Sigma$, where each N_σ^i is the i-th set of $\{\sigma\}$-connected vertices in V_{rest}. These nodes are helpful because any two vertices inside N_σ^i are σ-connected, which reduces the search space inside N_σ^i. We perform BFS by only using edges with label σ to find N_σ^i. If we visit a vertex $v \in V_{rest}$, we put it into N_σ^i.

We also maintain the mapping from each vertex to its CC node. For each $v \in V_{rest}$ and each $\sigma \in \Sigma$, let $N_\sigma(v)$ be the unique node (which is one N_σ^i) that v belongs to (i.e., $v \in N_\sigma^i$) if it exists and null otherwise. For each $v \notin V_{rest}$, which is a $\{\sigma\}$-inner vertex, we also assign its $N_\sigma(v)$ to better find its L-connected vertices in V_{rest}. It can be done when we process Algorithm 2. Specifically, in Line 4 of Algorithm 2, for a $\{\sigma\}$-inner vertex v, when we stop expansion at some non-$\{\sigma\}$-inner ones, they and v must belong to the same connected component and thus, the same CC node. Thus, we can set $N_\sigma(v)$ as the CC node of those non-$\{\sigma\}$-inner ones. For each $v \in V_{rest}$, since any pair of v's two adjacent edges with different labels are connected via v, we add a link between $N_\sigma(v)$ and $N_{\sigma'}(v)$ where σ and σ' are two different adjacent edge labels if the link does not exist.

For each $v \in V$, let $\mathcal{N}_L(v)$ be the set of nodes that v belongs to w.r.t. all $\sigma \in L$, i.e, $\mathcal{N}_L(v) = \{N_\sigma(v) | \sigma \in L, N_\sigma(v) \neq \text{null}\}$.

Algorithm 3: CC Graph Construction

input : The network $G(V, E, \Sigma, l, w)$
output : CC graph including the refined sets $R_\sigma(N)$ for each node N and $\sigma \in \Sigma$

1 **foreach** $\sigma \in \Sigma$ **do**
2 $i \leftarrow 0, V' \leftarrow V$
3 **foreach** $v \in V'$ **do**
4 **if** *v has an adjacent label equal to σ* **then**
5 perform a BFS from v by only using edges with label σ, and for each visited vertex u, add u into N^i_σ if $u \in V_{rest}$
6 **foreach** $\sigma' \in \Sigma \backslash \{\sigma\}$ **do**
7 $R_{\sigma'}(N^i_\sigma) \leftarrow \{v \in N | \exists l(v, v') = \sigma'\}$
8 $V' \leftarrow V' \setminus N^i_\sigma$
9 $i \leftarrow i + 1$
10 **foreach** $v \in V_{rest}$ **do**
11 **for** $(v, v'), (v, v'') \in E$ *where* $l(v, v') \neq l(v, v'')$ **do**
12 add a link between $N_{l(v,v')}(v)$ and $N_{l(v,v'')}(v)$
13 return the CC graph of G

Let $\mathcal{N}'_L(v)$ be the set of nodes that are some N_σ for $\sigma \in L$ and connected to those in $\mathcal{N}_L(v)$ in the CC graph, i.e., $\mathcal{N}'_L(v) = \{N_\sigma | \sigma \in L, N \in \mathcal{N}_L(v), \exists$ a path between N_σ and N in the CC graph$\}$. We show how to use the CC graph by the following lemma.

Lemma 3. *Given a label set L, the set of v's L-connected vertices is $\bigcup_{N \in \mathcal{N}_L(v) \cup \mathcal{N}'_L(v)} N$.*

Proof. For one of v's L-connected vertices u, consider any v-u path $\langle v_0 = v, v_1, v_2, \ldots, v_k = u \rangle$ where each $l(v_{i-1}, v_i) \in L$ for $1 \leq i \leq k$. We prove by induction that v_i is in one of the nodes N in $\mathcal{N}_L(v) \cup \mathcal{N}'_L(v)$. First, $v_0 = v \in N_{l(v_0,v_1)}(v) \in \mathcal{N}_L(v)$ because $l(v_0, v_1) \in L$. If $v_i \in N_\sigma(v_i) \in \mathcal{N}_L(v) \cup \mathcal{N}'_L(v)$ for one σ, we can get that $N_{l(v_i,v_{i+1})}(v_i) \in \mathcal{N}_L(v) \cup \mathcal{N}'_L(v)$ since $l(v_i, v_{i+1}) \in L$ and $N_{l(v_i,v_{i+1})}(v_i)$ should be linked to $N_\sigma(v_i)$. We also know that $v_{i+1} \in N_{l(v_i,v_{i+1})}(v_i)$ since v_i and v_{i+1} are $l(v_i, v_{i+1})$-connected and should belong to the same node, which shows that $v_{i+1} \in N_{l(v_i,v_{i+1})}(v_i) \in \mathcal{N}_L(v) \cup \mathcal{N}'_L(v)$. For each $u \in \bigcup_{N \in \mathcal{N}_L(v) \cup \mathcal{N}'_L(v)} N$, we first find the two nodes that contain u and v and then a path in the CC graph between the two nodes. Since each link (N, N') in this path represents that all vertices in N and N' are L-connected, we show that u and v are L-connected. □

For the second step, to retain only v's L-connected vertices that have adjacent labels not in L, we can classify the vertices in each node by their adjacent labels. Specifically, for each node N, let $R_\sigma(N)$ be the refined set of vertices in N that have at least one adjacent label equal to σ, i.e., $R_\sigma(N) = \{v \in N | \exists l(v, v') = \sigma\}$ for each $\sigma \in \Sigma$. Then, for $v \in V_{rest}$, $B_L(v) = \bigcup_{\sigma \in \Sigma \backslash L} \bigcup_{N \in \mathcal{N}_L(v) \cup \mathcal{N}'_L(v)} R_\sigma(N)$.

Algorithm 3 describes the construction of the CC graph. For each label σ (Line 1), we separately find the σ-connected components to form the nodes N^i_σ by using BFS in Line 5. The BFS only uses edges with label σ for expansion. We then refine the node set N^i_σ by assigning vertices into different $R_{\sigma'}(N^i_\sigma)$ for $\sigma' \in L \backslash \{\sigma\}$ in Lines 6-7. In Lines 10-12, for each vertex, we add a link between any pair of $N_\sigma(v)$ and $N_{\sigma'}(v)$ where $\sigma = l(v, v')$ and $\sigma' = l(v, v'')$ are v's two adjacent labels.

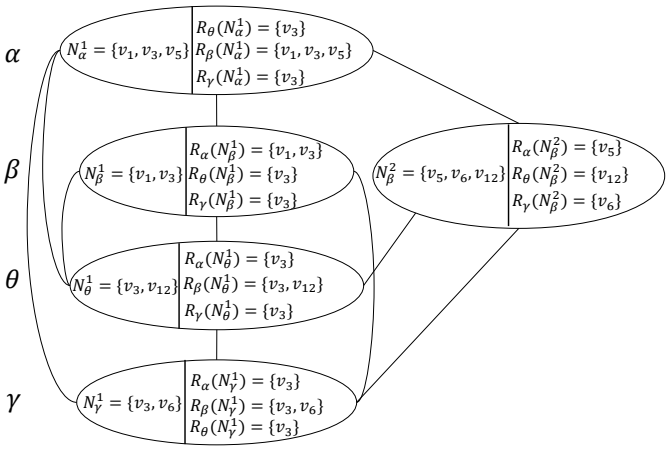


Figure 4: The CC graph of G

Example 5. *For label α, Algorithm 3 uses BFS to find $\{v_1, v_2, v_3, v_5\}$. We set $N^1_\alpha = \{v_1, v_3, v_5\}$ because $v_2 \notin V_{rest}$. For each label other than α, we set $R_\theta(N^1_\alpha) = \{v_3\}$ since only v_3 has adjacent labels $l(v_3, v_7) = l(v_3, v_8) = \theta$. Similarly, we can set $R_\beta(N^1_\alpha)$ and $R_\gamma(N^1_\alpha)$. For the label β, we can find two connected components $\{v_1, v_3, v_4\}$ and $\{v_5, v_6, v_{10}, v_{11}, v_{12}\}$ by the BFS. After removing those not in V_{rest}, we get $N^1_\beta = \{v_1, v_3\}$ and $N^2_\beta = \{v_5, v_6, v_{12}\}$ and their refined sets $R_\sigma(N)$. We similarly process N^1_θ and N^1_γ. Next, we add links between CC nodes. For v_1, $l(v_1, v_2) = \alpha \neq l(v_1, v_4) = \beta$, we add a link between N^1_α and N^1_β. Similarly, we can add links between other nodes by considering other vertices in V_{rest}.*

To make a state move, given the current vertex-state pair (v, q), we consider the set $B_{L_{\text{self}}(q)}(v)$ where we use $L_{\text{self}}(q)$ to replace L. Furthermore, not all labels in $\Sigma \backslash L$ can be used for state moves, we only need to focus on those in $L_{\text{move}}(q)$. We focus on the set of vertices in $B_{L_{\text{self}}}(v)$ such that they have adjacent labels in $L_{\text{move}}(q)$ for state moves. Thus, we define $B_{L_{\text{self}}(q)}(v, L_{\text{move}}(q))$ where we use $L_{\text{move}}(q)$ to replace $\Sigma \backslash L$.

Definition 7 (The set of v's L border vertices w.r.t. the label set $L_{move}(q)$). *At state q, we define $B_{L_{self}(q)}(v, L_{move}(q)) = \bigcup_{\sigma \in L_{move}(q)} \bigcup_{N \in \mathcal{N}_{L_{self}(q)}(v) \cup \mathcal{N}'_{L_{self}(q)}(v)} R_\sigma(N)$.*

Algorithm 4 summarizes the procedure of fetching the set of border vertices $B_{L_{\text{self}}(q)}(v, L_{\text{move}}(q))$ used in query processing. Specifically, in Lines 2–4, we first find the nodes in the CC graph that are connected to $N_\sigma(v)$ for $\sigma \in L_{\text{self}}(q)$ and put them in $\mathcal{N}$. These nodes are connected by labels in $L_{\text{self}}(q)$ according to the construction of the CC graph. To make a state move, we consider each label $\sigma \in L_{\text{move}}(q)$ (Line 5) and put $R_\sigma(N)$ for each $N \in \mathcal{N}$ (Line 6) in the final answer since the vertices in $R_\sigma(N)$ have at least one adjacent label σ for state moves.

Algorithm 4: Fetching $B_{L_{\text{self}}(q)}(v, L_{\text{move}}(q))$

input : CC graph, a vertex v, and a state q
output: $B_{L_{\text{self}}(q)}(v, L_{\text{move}}(q))$

1 $\mathcal{N} \leftarrow \emptyset$, $B \leftarrow \emptyset$
2 **foreach** $\sigma \in L_{\text{self}}(q)$ **do**
3 $\mathcal{N} \leftarrow \mathcal{N} \cup \{N_\sigma(v)\}$
4 Perform a BFS from $N_\sigma(v)$ by using CC graph links and add each $N_{\sigma'}(v)$ in $\mathcal{N}$ s.t. $\sigma' \in L_{\text{self}}(q)$
5 **foreach** $\sigma \in L_{\text{move}}(q)$ **do**
6 **foreach** $N \in \mathcal{N}$ **do**
7 $B \leftarrow B \cup R_\sigma(N)$
8 return B

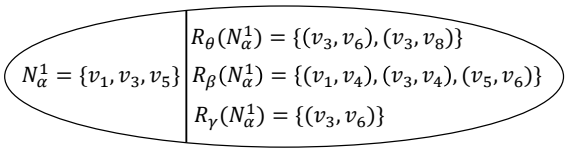


Figure 5: Edge representation of the CC node N^1_α

Example 6. *Suppose that we want to find $B_{L_{self}(q_0)}(v_1, L_{move}(q_0))$ for (v_1, q_0). First, since $L_{self}(q_0) = \{\alpha, \beta\}$, we find $\mathcal{N}_{L_{self}(q_0)}(v_1) = \{N^1_\alpha, N^1_\beta\}$ by looking at the first row of α where N^1_α contains v_1 and the second row of β where N^1_β contains v_1 in Figure 4. We also get $\mathcal{N}'_{L_{self}(q_0)}(v_1) = \{N^2_\beta\}$ because N^2_β is connected to N^1_α. Then, $\mathcal{N} = \{N^1_\alpha, N^1_\beta, N^2_\beta\}$. In Line 7, we can then use the union of $R_\gamma(N)$ for $N \in \mathcal{N}$ to form $B = \{v_3, v_6\}$.*

Algorithm 4 is correct because we fetch all of v's $L_{\text{self}}(q)$-connected vertices by Lemma 3 and retain only those with adjacent labels in $L_{\text{move}}(q)$ by $R_\sigma(N)$.

To avoid scanning the neighbors of each vertex v in $R_\sigma(N)$, we can directly store v's incident edges with label σ in each CC node and form B as a set of edges. For ease of illustration, we just list vertices in each $R_\sigma(N)$ of each CC node.

Example 7. *Figure 5 shows how we store the edges in the CC node N^1_α. For $v_3 \in R_\theta(N^1_\alpha)$, we can directly store the two incident edges (v_3, v_6) and (v_3, v_8) with label θ. Similarly, we can process $R_\beta(N^1_\alpha)$ and $R_\gamma(N^1_\alpha)$ to store the edges with corresponding labels.*

Theorem 2. *Algorithm 3's time complexity is $O(|\Sigma||E| + |V|D^2)$, where D is the average degree of each vertex. Its space complexity is $O(|\Sigma||V| + |V|D^2)$.*

Proof. In each iteration, the BFS takes $O(|E|)$ time and there are $|\Sigma|$ iterations. The last two for-loops uses $O(|V|D^2)$ time. Since we need to maintain the mapping from each vertex to its unique node for each $\sigma \in \Sigma$, we use $O(|\Sigma||V|)$ space. There could be at most $O(|V|D^2)$ links. □

Theorem 3. *Algorithm 4's time complexity is $O(|E|)$.*

Proof. In the worst case, the number of links in the CC graph is $|E|$ and we perform a BFS using $O(|E|)$ time. □

4.4 Weight Calculation

Given a vertex-state pair (v, q), if $L_{\text{self}}(q) \neq \emptyset$, we can find one of its border vertices $v' \in B_{L_{\text{self}}(q)}(v)$ for a state move. We also need to calculate the minimum weight of the v-v' path that only uses labels in $L_{\text{self}}(q)$. It corresponds to the Kleene-language constrained shortest path problem, which is widely studied [2, 10, 11, 15]. However, we note that they cannot handle general regular languages and that the query efficiency of the best-known index, called LSD [2], can be still improved since it sacrifices much query efficiency to achieve a small index cost. We will adapt the LSD index to compute the minimum weight of the v-v' path that only uses labels in a given non-empty label set L (which is $L_{\text{self}}(q)$ in our problem), denoted by $w^{\min}_L(p_{vv'})$. It is ∞ if v and v' are not L-connected. Our modified LSD can be orders of magnitude faster than the original version of LSD, as shown in our experiments.

We first introduce the original LSD. Let $s = v$ and $t = v'$. To compute $w^{\min}_L(p_{st})$, LSD mainly utilizes two concepts: vertex cuts and non-dominated paths. First, a vertex cut H for s and t is any set of vertices such that s and t are disconnected after the vertices in H are removed from the graph. It further means that any s-t path must traverse one vertex in H and that $w^{\min}_L(p_{st}) = \min_{h \in H}\{w^{\min}_L(p_{sh}) + w^{\min}_L(p_{ht})\}$ [2]. LSD mainly builds a tree decomposition [16, 17], which is a tree structure that maps each vertex v to a tree node $X(v) \subset V$ representing a vertex set. Given any s and t, we can quickly find a small vertex cut H that is the least common ancestor (LCA) node X_{LCA} of $X(s)$ and $X(t)$. Second, to define the non-dominated path, we use a pair $(L_p, w(p))$ to represent a path p with its path weight $w(p)$ and label set L_p, made up by all edge labels along p. A path (L_1, w_1) dominates another path (L_2, w_2) if 1) $L_1 \subset L_2$ and $w_1 \leq w_2$ or 2) $L_1 = L_2$ and $w_1 < w_2$. The set $\mathcal{S}_{sh}$ of non-dominated paths between s and h includes all the s-h paths that are not dominated by any other s-h paths. Given $\mathcal{S}_{sh}$, we can quickly find $w^{\min}_L(p_{sh})$ by returning the minimum distance of a path $p \in \mathcal{S}_{sh}$ such that $L_p \subseteq L$.

Given s, t, and a label set L, LSD first obtains a vertex cut H from the tree decomposition. Next, it computes $w^{\min}_L(p_{sh})$ for each $h \in H$, then $w^{\min}_L(p_{ht})$ for each $h \in H$ similarly, and $w^{\min}_L(p_{st}) = \min_{h \in H}\{w^{\min}_L(p_{sh}) + w^{\min}_L(p_{ht})\}$. To compute $w^{\min}_L(p_{sh})$ for each $h \in H$, LSD finds several sets of vertices $V_1, V_2, \ldots, V_n = H$ from the tree decomposition in a bottom-up manner. $V_1, V_2, \ldots V_n$ contain some middle vertices with distances to v from near to far. In LSD's index, it maintains $\mathcal{S}_{sv}$ for $v \in V_1$ so that $w^{\min}_L(p_{sv})$ for $v \in V_1$ can be computed easily, as stated above. It also maintains some $\mathcal{S}_{vv'}$ for $v \in V_i$ and $v' \in V_{i+1}$ (for $i = 1, \ldots n-1$) to compute $w^{\min}_L(p_{vv'})$. In this way, it can recursively use $w^{\min}_L(p_{sv})$ for $v \in V_i$ to compute $w^{\min}_L(p_{sv})$ for $v \in V_{i+1}$ until $V_{i+1} = H$.

We can find that the original LSD needs to recursively compute the distances $w^{\min}_L(p_{sv})$ for $v \in V_1, V_2, \ldots, H$ from near to far. To improve its efficiency, we adapt LSD by directly preprocessing non-dominated path sets $\mathcal{S}_{sh}$ and $\mathcal{S}_{ht}$ for each $h \in H$ in the index. By directly looking up the sets in the index, we can compute $w^{\min}_L(p_{sh})$

and $w_L^{\min}(p_{ht})$ more efficiently. It actually suggests that we preprocess the recursive computations during index construction. Though the indexing time and space costs of the adapted version can be much larger than the original one, its query efficiency can be orders of magnitude higher. Since s and t can be different in query processing, we need to consider all possible vertex cuts. Specifically, the vertex cut can only be the LCA node X_{lca} of $X(s)$ and $X(t)$, which means that we only need to consider the branching nodes that have at least two child nodes. There is also a useful property that for each $v \in X_{\text{lca}}$, $X(v)$ is an ancestor of both $X(s)$ and $X(t)$. Therefore, we maintain $\mathcal{S}_{vu}$ in the index whenever $X(v)$ is an ancestor of $X(u)$. The adapted index can be built recursively in a top-down manner in the tree decomposition, similarly to [17].

Note that any faster Kleene-language constrained shortest path solutions can replace this weight calculation module. A trivial speedup technique is to precompute some Kleene-language constrained distances between vertices and their border ones. We directly use LSD since the weight calculation is not our focus.

5 BSM QUERY PROCESSING

The query processing algorithm efficiently searches the graph by utilizing goal-directed priority values, state-move expansion, and pruning bounds. First, we use the CC graph to obtain the border vertices to make state moves in each iteration of expansion, which helps us to quickly reach the final states in the DFA graph. Second, we follow the idea of A* search [4, 18], which uses a priority queue that tends to first fetch the vertex-state pair closer to the destination for expansion. We utilize a lower bound for the priority values of vertex-state pairs [4]. Third, we prune unnecessary searches by using the lower and upper bounds of the path weight.

BSM query processing mainly utilizes the CC graph to obtain border vertices for state moves. Given the current vertex-state pair (v, q), if $L_{\text{self}}(q)$ exists, we can directly find its border vertices $B_{L_{\text{self}}(q)}(v, L_{\text{move}}(q))$ by Algorithm 4. This can be imagined as using some "super edges" (which represents paths) between v and each $u \in B_{L_{\text{self}}(q)}(v, L_{\text{move}}(q))$ for expansion. We avoid searching the vertices along the corresponding v-u path. Similar to Algorithm 1, we use the array d to maintain the minimum weight for each vertex-state pair (v, q), corresponding to an s-v path with its label accepted by the DFA from state q_0 to q. We use one of the super edges for expansion when $d[u][q]$ is larger than $d[v][q] + w_{L_{\text{self}}}(q)^{\min}(p_{vu})$, which means that the s-v path concatenated with the super edge between v and u is better than the previously seen s-u paths. Furthermore, $u \in B_{L_{\text{self}}(q)}(v, L_{\text{move}}(q))$ has at least one incident edge with its label used for state moves.

The extended Dijkstra's algorithm uses a priority queue with the current weight $d[v][q]$ as the priority value of each vertex-state pair (v, q), whereas A* search uses $f(v, q) = d(v, q) + lb(v, q)$, where $lb(v, q)$ denotes a lower bound of the minimum weight of the path from (v, q) to (t, q_f) for one $q_f \in F$ to ensure the correctness. A simple way to set $lb(v, q)$ is to use v and t's coordinates to compute the Euclidean distance between v and t. Noticing that any unconstrained shortest distance between v and t is also a lower bound, [4] chooses a set S_{lm} of vertices, called landmarks, and precomputes the unconstrained distances between each vertex v and $u \in S_{lm}$. Note that the original idea comes from ALT for unconstrained shortest paths [6]. Then, the absolute value between the v-u and t-u distances for each $u \in S_{lm}$ can be a lower bound $lb(v, q)$ due to the triangle inequality. We can use the maximal lower bound among all landmarks since it can prune more search space.

We can also use the upper bound of the optimal LCSP for pruning. For the upper bound of $w(p^{opt})$ of LCSP, denoted by w^{ub}, the weight of any label-constrained s-t path is an upper bound. Each time we visit the pair (t, q) for each $q \in F$, we maintain the minimal upper bound if $d[t][q]$ is smaller than the current w^{ub}. Given the current pair (v, q), there are two places where we can use the upper bound for pruning. First, after we fetch one of v's border vertices $u \in B_{L_{\text{self}}(q)}(v, L_{\text{move}}(q))$, the lower bound of the path weight via u is $d[v][q] + w^{\min}_{L_{\text{self}}(q)}(p_{vu}) + lb(u, q)$, where the three terms represent the weights of s-v, v-u, and u-t subpaths, respectively. We then prune a border vertex u if the lower bound is no smaller than the current upper bound w^{ub}. Second, we can prune a pair (v', q') for expansion when the lower bound $d[v'][q'] + lb(v', q')$ is no smaller than w^{ub} because the real weight of the path via v' should be at least the lower bound and thus w^{ub}, which means that we cannot find the LCSP by expanding (v', q'). Note that w^{ub} is designed to maintain the upper bound of the optimal weight $w(p^{opt})$ w.r.t. any $q \in F$, whereas $d[t][q]$ maintain the minimum weight for q.

Algorithm 5 summarizes the whole procedure of query processing. In Line 1, we initialize the array d, the upper bound w^{ub}, and the priority queue *Queue*. Each quadruple in *Queue* includes its priority value $f(v, q)$, vertex v, state q, and the current weight w. In Lines 4–11, we handle the case where q is a final state. Specifically, in Lines 7–11, we check if there is a "super edge" between v and t by using $L_{\text{self}}(q)$ and this supper edge gives a better s-t path from state q_0 to q. Let X be the set of vertices to be expanded. We initially put v into X in Line 12. In Lines 13–17, we check the "super edge" between v and each of the border vertices from the CC graph. Specifically, we prune a border vertex u if the lower bound $d[v][q] + w^{\min}_{L_{\text{self}}(q)}(p_{vu}) + lb(u, q) > w^{ub}$. We also check if we have visited a better path to (u, q). In Lines 18–24, we perform the expansion for each vertex $u \in X$. We only consider each of u's adjacent labels $l(u, v')$ that makes a state move to $\delta(q, l(u, v')) = q' \neq q$. In Lines 23–24, we prune the pair (v, q) if its lower bound $d[v'][q'] + lb(v', q')$ is no smaller than w^{ub}.

Example 8. *We still use the same Example 3 and simply assume that all the lower bounds $lb(v, q) = 0$. Initially, we fetch $(0, v_1, q_0, 0)$ from Queue. Since $q_0 \notin F$ and $L_{\text{self}}(q_0) = \{\alpha, \beta\} \neq \emptyset$, in Line 14, we consider the set $B_{L_{\text{self}}(q_0)}(v_1, L_{\text{move}}(q_0)) = \{v_3, v_6\}$ as detailed in Example 6. For v_3, we update $d[v_3][q_0] = d[v_1][q_0] + w^{\min}_{L_{\text{self}}(q_0)}(p_{v_1v_3}) = 0 + 4 = 4$ and put v_3 in X. For v_6, we update $d[v_6][q_0] = d[v_1][q_0] + w^{\min}_{L_{\text{self}}(q_0)}(p_{v_1v_6}) = 0 + 3 = 3$ and put v_6 in X. In Line 19, since $q' = \delta(q_0, \gamma) = q_1 \neq q$, we can only use the label γ for expansion. For v_3, we update $d[v_6][q_1] = 6$ and put $(6, v_6, q_1, 6)$ in Queue. Similarly, we update $d[v_3][q_1] = 5$ and put $(5, v_3, q_1, 5)$ in Queue. We next fetch $(5, v_3, q_1, 5)$ from Queue. Since $q_1 \in F$, $v_3 \neq t$, and $L_{\text{self}}(q_1) = \emptyset$, we directly put expand (v_3, q_1) in Line 18–24. We update $d[v_4][q_2] = 7$ and put $(7, v_4, q_2, 7)$ in Queue, update $d[v_7][q_3] = 9$ and put $(9, v_7, q_3, 9)$ in Queue, and update $d[v_8][q_3] = 7$ and put $(7, v_8, q_3, 7)$ in Queue. Suppose that we next fetch $(7, v_8, q_3, 7)$. Since $q_3 \in F$ and $L_{\text{self}}(q_3) = \{\theta\}$, in Line 8, we update $d[v_{12}][q_3] = 7 + 5 = 12$ and*

Algorithm 5: BSM Query Processing

input : Two vertices s and t, $\mathcal{L}$, and BSM index
output: $w(p^{opt})$ of LCSP

1 $d[v][q] \leftarrow +\infty$ for $v \in V$ and $q \in Q$, $d[s][q_0] \leftarrow 0$, $w^{ub} \leftarrow +\infty$, $Queue.push((0, s, q_0, 0))$
2 **while** $|Queue| > 0$ **do**
3 fetch (f, v, q, w) with the minimum f from $Queue$
4 **if** $q \in F$ **then**
5 **if** $v = t$ **then**
6 return w
7 **else if** $L_{\text{self}}(q) \neq \emptyset$ *and* $d[t][q] > d[v][q] + w^{\min}_{L_{\text{self}}(q)}(p_{vt})$ **then**
8 $d[t][q] \leftarrow d[v][q] + w^{\min}_{L_{\text{self}}(q)}(p_{vt})$
9 **if** $d[t][q] < w^{ub}$ **then**
10 $w^{ub} \leftarrow d[t][q]$
11 $Queue.push((d[t][q], t, q, d[t][q]))$
12 $X \leftarrow \{v\}$
13 **if** $L_{\text{self}}(q) \neq \emptyset$ **then**
14 **foreach** $u \in B_{L_{\text{self}}(q)}(v, L_{\text{move}}(q))$ *(Algorithm 4)* **do**
15 **if** $w^{ub} \geq d[v][q] + w^{\min}_{L_{\text{self}}(q)}(p_{vu}) + lb(u, q)$ *and* $d[u][q] \geq d[v][q] + w^{\min}_{L_{\text{self}}(q)}(p_{vu})$ **then**
16 $d[u][q] \leftarrow d[v][q] + w^{\min}_{L_{\text{self}}(q)}(p_{vu})$
17 $X \leftarrow X \cup \{u\}$
18 **foreach** $u \in X$ **do**
19 **foreach** $(u, v') \in E$ *s.t.* $\delta(q, l(u, v')) \in L_{\text{move}}(q)$ **do**
20 $q' \leftarrow \delta(q, l(u, v'))$
21 **if** $d[v'][q'] > d[u][q] + w(u, v')$ **then**
22 $d[v'][q'] \leftarrow d[u][q] + w(u, v')$
23 **if** $d[v'][q'] + lb(v', q') < w^{ub}$ **then**
24 $Queue.push((d[v'][q'] + lb(v', q'), v', q, d[v'][q']))$

$w^{ub} = d[v_{12}][q_3] = 12$. *We put* $(12, v_{12}, q_3, 12)$ *in Queue and fetch it from Queue in the later iteration. After some similar steps, when we visit* v_{12}*, we can return the answer* $w(p^{opt}) = 12$.

Theorem 4. *Algorithm 5 is guaranteed to find the LCSP.*

Proof. We mainly follow the framework of Algorithm 1. The difference is that we add some "super edges" between (v, q) and (u, q) such that u is a $L_{\text{self}}(q)$-border vertex. Consider the graph G with these "super edges". We additionally explore these super edges with weights $w^{\min}_{L_{\text{self}}(q)}(p_{vu})$. They do not affect the correctness since each LCSP in the original graph G still has the same path weight in the new graph. □

Theorem 5. *Algorithm 5 uses* $O(|E||Q|\log(|V||Q|))$ *time.*

Proof. In the worst case, we have a DFA where $L_{\text{self}}(q) = \emptyset$ for each $q \in Q$. There can be $|E||Q|$ "push" operations, and each "fetch" operation takes $O(\log(|V||Q|))$ time. □

Though the worst-case time complexity does not improve, the query efficiency can be significantly higher since we use border vertices to reduce search space. The space cost of Algorithm 5 is similar to that of Algorithm 1, as discussed in Section 2.

Path retrieval. The procedure uses the same idea of Dijkstra's algorithm. We maintain an array $prev$ that stores the previous pair of (v, q) and update it as the array d is updated. When we return the answer in Line 5, we start with the last pair (t, q) and use the array $prev$ to find previous vertex-state pair repeatedly until we find (s, q_0). Note that if the array d is updated in Lines 15–16 of Algorithm 5, we restore the path by querying the index of the weight calculation component.

Practical implementation. First, we can transform Algorithm 5 to a bidirectional algorithm that searches from both s and t to reduce the search space. We need to maintain two distance arrays $df[v][q]$ and $db[v][q]$ for the forward and backward searches.

Second, the time-consuming part of Algorithm 5 lies in querying the LSD index [2] to obtain the distances $w^{\min}_{L_{\text{self}}(q)}(p_{vu})$ between v and its border vertex u. A potential speedup technique is to preprocess some unconstrained distances for vertices and their border vertices. The unconstrained distance between v and u is denoted by $w^{\min}_{\Sigma}(p_{vu})$ and always a lower bound of $w^{\min}_{L_{\text{self}}(q)}(p_{vu})$ for any $L_{\text{self}}(q) \subseteq \Sigma$. With these lower bounds, we can prune a border vertex u by the condition $w^{ub} < d[v][q] + w^{\min}_{\Sigma}(p_{vu}) + lb(u, q)$ before querying LSD in Line 15.

Third, noticing that some Kleene-language constrained shortest distances $w^{\min}_{L_{\text{self}}(q)}(p_{vt})$ in Line 7 can be infinity, which means that v and t are not $L_{\text{self}}(q)$-connected, we can perform a label-constrained reachability check before Line 7. Specifically, we do not query LSD if v and t are not $L_{\text{self}}(q)$-connected. There have been many efficient solutions [19–22] since this problem is irrelevant to edge weights and thus much easier. The state-of-the-art solution P2H [22] can answer a label-constrained reachability query in several microseconds on a large graph. Note that we do not have to perform the label-constrained reachability check before Line 15 since v and u must be $L_{\text{self}}(q)$-connected by the correctness of the CC graph and Algorithm 3.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our proposed BSM on four common real road network datasets. We first introduce the experimental setup in Section 6.1, present the results by considering the query and index efficiencies in Section 6.2, and summarize our finding in Section 6.3.

6.1 Experimental Setup

We implemented all algorithms in C++ and compiled them with GNU C++ compiler. The implementations could be found in an anonymous link [1]. All experiments were performed on one machine with Intel Xeon Processor E5-2650 v4 (30M Cache, 2.20 GHz) and 512 GB DDR4 RAM. **Datasets and Workloads.** Following existing studies [1, 2], we considered four real road networks obtained from DIMACS [2]. Their statistics were summarized in Table 2. The weight

[1] https://anonymous.4open.science/r/BSM-7B71
[2] http://www.dis.uniroma1.it/challenge9/download.shtml

Table 2: Dataset Statistics

Dataset	Region	$\|V\|$	$\|E\|$	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

$w(e)$ and label $l(e)$ of each edge were set to its spatial distance and road category. Specifically, each road category was represented by a two-digit code. The description of DIMACS stated that the first digit is used to denote four main road types: 1) $\mathcal{A}$, Primary Highway With Limited Access (e.g. interstates); 2) $\mathcal{B}$, Primary Road Without Limited Access (e.g. US highways); 3) $\mathcal{C}$, Secondary and Connecting Road (e.g. state highways); and 4) $\mathcal{D}$, Local, Neighborhood, and Rural Road. The second digit ranging from 1 to 5 represents a finer level of roads. For example, the most frequent code is "$\mathcal{D}1$", which means that the road with this label is a local road with level 1.

We generated each workload by specifying the source s, destination t, and regular language $\mathcal{L}$ of each query $q = (s, t, \mathcal{L})$. Specifically, following existing studies [1, 2], for each road network, we first found the maximum unconstrained distance D_{max} between any two vertices (set to an approximate value for efficiency) and created 10 workloads, called $D_1, D_2, \ldots, D_{10}$ [1, 2]. Each workload is formed by 1,000 queries. For all queries in D_i, the unconstrained distances between sources and destinations lie in $(x^{i-1}D_{min}, x^iD_{min})$ for $1 \le i \le 10$, where $D_{min} = 1,000$ and $x = (D_{max}/D_{min})^{1/10}$. In this way, the maximum distance in D_{10} is $x^{10}D_{min} = D_{max}$. For regular languages, we mainly follow the settings of [1, 2]. The label set $\Sigma = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{D}_5\}$. We sort them in the descending order of their frequencies among all edges. Let Σ_i be the set of top-i frequent labels. To make LSD capable of handling the language constraint, we set the default language $\mathcal{L} = (\cup_{\sigma\in\Sigma_{10}}\sigma)^*$, which only allows the use of the top-10 frequent labels in the LCSP. It further means that $\mathcal{L}$ prohibits the use of labels in $\Sigma\backslash\Sigma_{10}$, corresponding to some routing preferences on certain types of roads. To test the effect of the DFA, we also conduct experiments about more complex types of languages, including the linear and the composite forms, which should cover most routing demands in real scenarios.

Compared Algorithms. We compared the following four solutions in experiments.

- ExtDijkstra [3]: the extended Dijkstra's algorithm as illustrated in Algorithm 1.
- SDALT [4]: a search-based algorithm that uses some landmark vertices to provide lower bounds of distances to direct the search and allows the flexible use of regular languages.
- LSD [2]: the state-of-the-art solution for Kleene languages that reduces the search space by the tree decomposition.
- AdaLSD: our adapted LSD introduced in Section 4.4.
- BSM: our proposed index-based solution that prunes the search between each vertex and its border vertices.

Note that we use 32 landmarks following the default setting in [4]. Also note that we would omit LSD if the experiments considered more general constraints that can only be expressed by regular languages but not Kleene ones.

6.2 Experiment Results

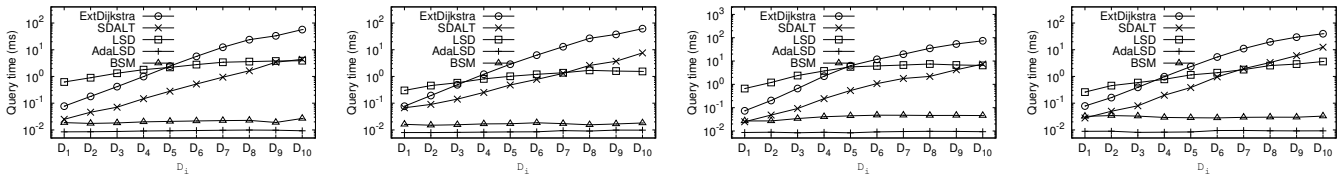
6.2.1 *Query Efficiency.*

Exp-1: Query efficiency for different query distances D_i. We test the effect of the query distance while fixing the language as the default one. The results for the four datasets are shown in Figure 6. It can be observed that the query times of all algorithms increase when the query distance is larger, which is mainly due to the expansion of the search range. For the performance, BSM is the faster than all algorithms except AdaLSD in all datasets. AdaLSD is a bit faster than BSM because it does not support general regular languages and avoids making some tedious processing related to DFA. When AdaLSD is handling languages other than Kleene languages, AdaLSD could not answer it with its index since the index is particularly designed for Kleene languages. However, our BSM could answer any form of regular language. The Extended Dijkstra's algorithm has the largest query time because it utilizes no speedup technique. SDALT is faster than ExtDijkstra since it uses landmarks to guide the search, but it cannot handle queries with long distances in larger road networks efficiently. LSD performs well for longer query distances because it uses the tree decomposition to reduce the search space.

Exp-2: Query efficiency for different query distances $|\Sigma_i|$. We also study the effect of the number of labels by varying Σ_i and using the workload D_{10} with the same query distance. Specifically, for Σ_i, we set the regular language as $L = (\cup_{\sigma\in\Sigma_i}\sigma)^*$. We summarize the results in Figure 7. Since Σ_i with a larger i indicates that the constraint allows more labels, ExtDijkstra has to check more labels during the search, resulting in more time cost. However, when the constraint allows more edges, it becomes looser since in the extreme case, there is no constraint when all labels are allowed (i.e., Σ_{20}). As i in Σ_i increases, the lower bounds used in SDALT are closer to the actual label-constrained distances, which are more precise and can guide the search quickly to the destination. We can observe that SDALT's time cost gradually decreases when the constraint is looser. LSD tends to have smaller query times because it checks fewer non-dominated paths as the constraint is looser. AdaLSD and BSM nearly have the same query times in all settings and outperform the other competitors by a large margin.

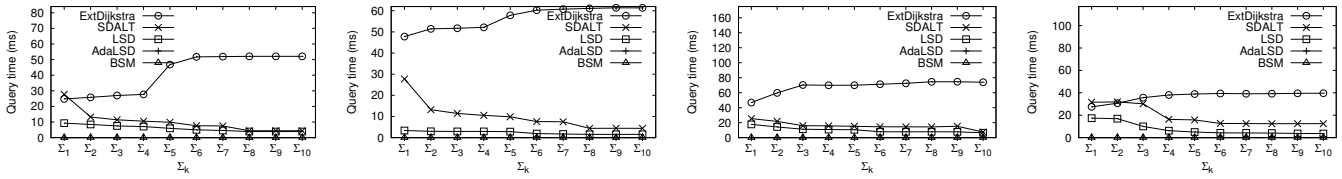
Exp-3: Query efficiency for different numbers of states $|Q|$. Since the number of states $|Q|$ is an important factor that determines the cost for state transitions, we vary it by considering the regular language $\mathcal{L}_i = \sigma_1^*\sigma_2^*\ldots\sigma_i^*$ where σ_i is the i-th top frequent label. Here, we use the top-10 frequent labels in Σ_{10} sequentially and still consider the workload D_{10}. Note that we omit LSD because it cannot handle this type of language. Figure 8 presents the query processing times of different $|Q|$. We can see that BSM outperforms the other two algorithms in all cases. Since the number of states increases, all algorithms take more time cost because they all need to check more vertices and states during the search. The increasing trend slows down mainly because the less frequent labels (e.g., σ_{10}) rarely appear in the LCSP and the search can quickly prune them.

Exp-4: Query efficiency for more labels. We use more labels than those in previous experiments and show the results on NY data in Figure 9. Specifically, in Figure 9a, we use D_{10} dataset and $L = (\cup_{\sigma\in\Sigma_i}\sigma)^*$ where $i = 11, \ldots, 20$. In Figure 9b, we use $L = (\cup_{\sigma\in\Sigma_{20}}\sigma)^*$ and vary D_i from D_1 to D_{10}. The flat lines are consistent



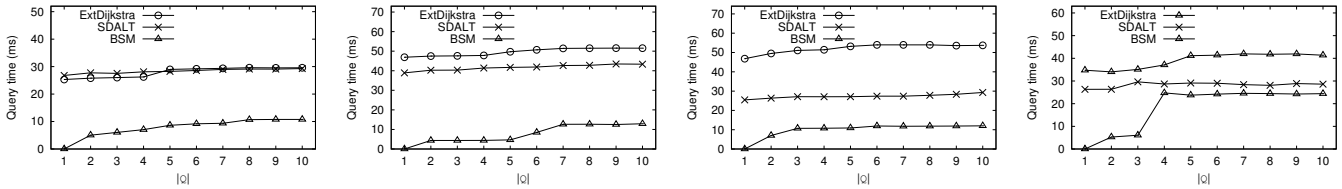
(a) NY **(b) BAY** **(c) COL** **(d) FLA**

Figure 6: Query processing times (on a log-10 scale) for different query distances D_i



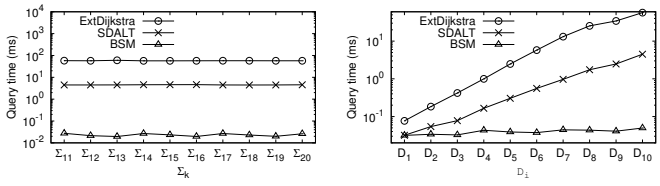
(a) NY **(b) BAY** **(c) COL** **(d) FLA**

Figure 7: Query processing times for different numbers of labels $|\Sigma_i|$



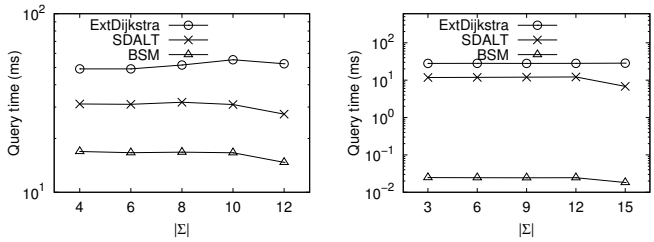
(a) NY **(b) BAY** **(c) COL** **(d) FLA**

Figure 8: Query processing times for different numbers of states $|Q|$



(a) Varying $|\Sigma_i|$ **(b) Varying D_i**

Figure 9: More labels on NY



(a) Highway usage **(b) Regional transfer**

Figure 10: More complex languages on NY

with those in Figure 7a after Σ_8. This is mainly because we use labels from frequent to less frequent ones and the frequencies follow the power law distribution. Suppose that all 20 labels are denoted as $\sigma_1, \sigma_2, \ldots, \sigma_{20}$ in the descending order of their frequencies. The difference between $\Sigma_{11} = \{\sigma_1, \sigma_2, \ldots, \sigma_{11}\}$ and Σ_{12} is σ_{12}, which is a rare label and has less influence on the query time. We can obtain similar findings that BSM outperforms the other baselines by at least two orders of magnitude.

Exp-5: Query efficiency for more complex languages. We also evaluate the performance for more complex languages. Following [1, 8], we consider two types of languages: (1) highway usage $L_1 = (\cup_{\sigma \in \Sigma_C \cup \Sigma_D} \sigma)^* (\cup_{\sigma \in \Sigma_A \cup \Sigma_B} \sigma)^* (\cup_{\sigma \in \Sigma_C \cup \Sigma_D} \sigma)^*$ and (2) regional transfer $L_2 = (\cup_{\sigma \in \Sigma_B \cup \Sigma_C \cup \Sigma_D} \sigma)^*$, where $\Sigma_A, \Sigma_B, \Sigma_C, \Sigma_D$ represent the label sets containing four types of labels, respectively. Figure 10 gives the results when we increase the number of distinct labels in the language. We can draw a similar conclusion that BSM is still the most efficient one among all algorithms.

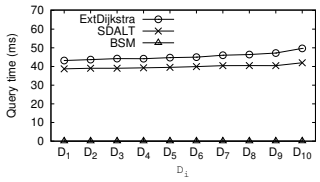
Exp-6: Query efficiency for stricter language constraints. Figure 11 shows the comparison for two stricter languages constraints. The first one uses the same form as the one in Section 1,

Table 3: Index construction time

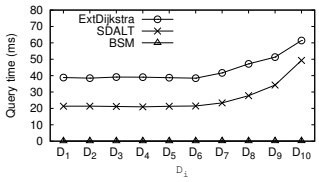
Dataset	SDALT	LSD	AdaLSD	Alg. 2	Alg. 3	Total BSM
NY	3,616s	52s	585s	0.06s	0.17s	585.23s
BAY	4,416s	14s	280s	0.06s	0.15s	280.21s
COL	6,016s	102s	1,244s	0.08s	0.25s	1,244.33s
FLA	14,720s	545s	7,837s	0.28s	0.79s	7,838.07s

Table 4: Index storage size

Data	SDALT	LSD	AdaLSD	CC Graph	Total BSM
NY	67.7 MB	34.2 MB	5.1 GB	4.7 MB	5.1 GB
BAY	82.3 MB	27.8 MB	3.9 GB	5.7 MB	3.9 GB
COL	111.6 MB	40.2 MB	13.4 GB	7.5 MB	13.4 GB
FLA	274.1 MB	108.9 MB	50.2 GB	22.5 MB	50.2 GB

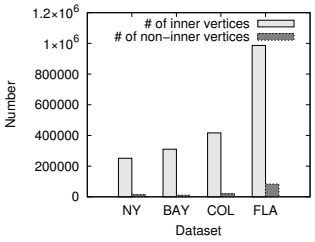


(a) $(\mathcal{D}_1|\mathcal{D}_2)^*\mathcal{D}_4(\mathcal{D}_2^*|\mathcal{D}_3^*)$

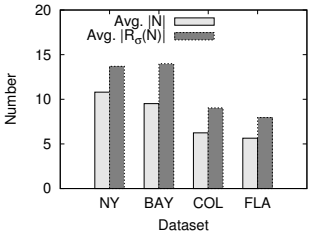


(b) $(\mathcal{D}_1^*|\mathcal{D}_2^*)\mathcal{D}_3\mathcal{D}_1^*$

Figure 11: Stricter language constraints on NY



(a) Statistics of Alg. 2



(b) Statistics of Alg. 3

Figure 12: Statistics of BSM index

and the second one indicates that the returned LCSP should first use highways, then a special road, and finally local roads. Note that we use the road type $\mathcal{D}$ (for local roads) because using other types (e.g., type $\mathcal{A}$ for highways) may not result in any label-constrained path between the source and destination. It can be found that ExtDijkstra and SDALT have worse performance than that in Figure 6 for Exp-1 because Exp-1 uses looser constraint with $\mathcal{L} = (\cup_{\sigma\in\Sigma_{10}}\sigma)^*$. Even if the unconstrained distance between the source and destination increases from D_1 to D_{10}, the label-constrained distances can be long and irrelevant to the unconstrained ones (e.g., the path should traverse a special road). However, we can observe that BSM outperforms others by orders of magnitude and can answer each query in less than one microseconds.

6.2.2 Index Efficiency.

Index construction time. We record the index construction time for each algorithm and dataset in Table 3. It can be noticed that all the algorithms tend to incur larger time costs when the road network is larger. We can also find that SDALT takes the longest time among all algorithms. This is because it has to precompute the distances between each vertex and each landmark, which can be time-consuming since the number of pre-computed distances is large. LSD takes acceptable time to build its index, but its query efficiency can be low as shown in previous experiments, and it cannot support regular languages. AdaLSD needs much indexing time but achieves a higher query efficiency. BSM's total precomputation time includes Algorithms 2 and 3 and the time of building the weight calculation index (i.e., AdaLSD). Algorithms 2 and 3 runs fast because both of them mainly scan the network once by using BFS. Though the time of building the AdaLSD index can be long, we only need to build it once to support any forms of regular languages, and its query time can be orders of magnitude faster than that of the original LSD. Any state-of-the-art solution for the Kleene-language constrained distance can be easily plugged in since we could regard this component as a black box.

Index storage size. The storage sizes of all indexes are listed in Table 4. Since the road network has more vertices and edges, all the indexes consume larger space. The total BSM index consists of the CC graph and the AdaLSD index for weight calculation. The space cost of the total BSM index is a bit larger than that of AdaLSD, but it is not reflected in the table due to rounding. It can be found that the CC Graph built by our BSM uses less than 30 MB in all datasets. Though the total index size of BSM is large, it allows flexible use of regular languages and can answer each LCSP query orders of magnitude faster than other competitors.

Statistics of BSM index Figure 12 shows some key variables in Algorithms 2 and 3 in detail. Specifically, Figure 12a gives the numbers of $\{\sigma\}$-inner vertices and non-$\{\sigma\}$-inner vertices for each dataset. The $\{\sigma\}$-inner vertices occupy a large part of the whole vertices, which further demonstrates the necessity of performing an initial pruning procedure on the original graph since there are many $\{\sigma\}$-inner vertices with common border vertices. Figure 12b presents the average size of the CC nodes $|N|$ and the average

refined set size $|R_\sigma(N)|$. The former is actually the number of non-$\{\sigma\}$-inner vertices in a connected component, while the latter refines the vertices by retaining those with at least one adjacent label σ. The two numbers are all smaller than 20, which indicates that the number of border vertices for the next state move is limited and efficient. Note that we directly count the number of edges with adjacent label σ as $|R_\sigma(N)|$, which makes its average value a bit larger than that of $|N|$. When the road network is larger, the two numbers become smaller, which is due to the increase in the number of CC nodes as shown in Table 4.

6.3 Summary

(1) Our proposed *BSM* can answer the flexible LCSP query without any assumption on the language type. It can outperform the state-of-the-art baselines by two orders of magnitude in terms of query processing time.

(2) The BSM index can be built efficiently with acceptable time and space consumption.

(3) BSM can easily handle problem instances with more labels and complex languages.

7 RELATED WORK

This section discusses related work from the *unconstrained shortest path* (Section 7.1) and the *label-constrained shortest path* (Section 7.2). The former one can be basically classified into *index-free* and *index-based* solutions. For the latter one, some build no indexes and thus, allow the flexible use of regular languages, and others make assumptions on the languages and preprocess some indexes.

7.1 Unconstrained Shortest Path

The shortest path problem has been widely studied since Dijkstra's algorithm [23]. To improve its efficiency, early solutions extended Dijkstra's idea by performing the bidirectional search (which runs two simultaneous Dijkstra's algorithms from both the source and destination) [24] and goal-directed A* search (which uses a priority values that prefer vertices close to the destination) [18]. However, they are all *index-free* methods and inefficient in large networks because they have to search the network from scratch.

Later solutions preprocessed useful network information in the *index* to achieve high query efficiency. These solutions included *ALT* (which uses pre-computed distances to provide better priority values in A* search) [6], *Arc Flags* (which marks unnecessary edges during query processing in advance) [25], *Transit Nodes* (which pre-computes pairwise distances among a set of vertices) [26], and *Reach* (which prunes vertices based on pre-computed distances) [27]. Recent indexes could be classified into two categories: some mainly prune *Dijkstra's search space* [28–33], and others precompute distances stored in *hash tables* [17, 34–41]. The methods in the second category could run faster than those in the first one but have larger space consumption. Specifically, the classical method in the first category is contraction hierarchy [30], which creates some additional "shortcut" edges to represent a long path and uses a vertex hierarchy to efficiently search the path made up by shortcut edges. The state-of-the-art H2H [17] in the second category first finds a set of vertices from the *tree decomposition*, also called the *hoplink*, such that any *s-t* path must traverse at least one hoplink, then concatenates the paths from sources and destinations to these hoplinks, and finally finds the shortest path among all concatenated paths. However, LCSP is completely different from the shortest path since LCSP involves the label constraint.

7.2 Label-Constrained Shortest Path

The seminal work [42] considered finding simple paths with labels accepted by regular languages in graph databases. The following work studied some variants of LCSP w.r.t. different types of formal languages and also proposed the first polynomial LCSP algorithm (i.e., the extended Dijkstra's algorithm) for regular langauges [3]. It was later analyzed and implemented in a system called *TRANSIMS* [7]. To further improve its efficiency, early solutions extended some ideas of unconstrained shortest path algorithms, such as *bidirectional and A* search* [8], *Transit Nodes* [9], and *ALT* [4]. Approximate solutions were also proposed to accelerate the query processing [15, 43]. In sum, they made no assumption on the regular languages, but their solutions, which were non-index approaches, were inefficient in answering LCSP queries in large road networks.

Recent index-based solutions assumed some special cases of regular languages so that they could preprocess some information and achieve much higher query efficiency [1, 2, 10, 11, 13]. [10, 13] utilized the idea of contraction hierarchy to prune the search space. Specifically, for Kleene languages, which are special cases of regular languages, [10] proposed CHLR that additionally creates some "shortcut" edges between vertex pairs to allow efficient retrieval of some sub-LCSPs. [13] assumed that the transformed DFA should have a certain pattern and built UCCH that preprocesses "shortcut" edges for paths with the same labels (different from CHLR since the shortcuts in CHLR could use different labels). Following [10] for Kleene languages, [11] presented EDP that partitions the graph according to edge labels and preprocessed some LCSPs in each partition. Its query processing then links the LCSPs among necessary partitions. EDP also uses the idea of connected components and bridge vertices. However, for the set of L-border vertices, it actually considers the trivial case where the label set L includes only one label and thus, cannot handle regular languages. In contrast, our BSM builds the CC graph to handle more labels in L and can answer more general queries under regular languages. The other study similarly utilized the partition and boundary vertices [5], but its problem does not involve labels and its partition is generated by unlabeled graphs. Based on the tree decomposition technique, LSD finds the LCSP by searching the tree decomposition in a bottom-up manner [2]. The currently fastest-known solution was PCSP proposed by [1]. It mainly returns the LCSP by concatenating two sub-LCSPs in the index, where one has a label accepted by the DFA from the initial state to a middle state and the other has a label accepted by the DFA from the middle state to a final state. It could outperform previous ones by orders of magnitude in terms of query efficiency. We admit that PCSP could run faster than our BSM, but it has to rebuild its index for different regular languages and does not support the flexible setting of the language constraint, which limits its application. In sum, they all made assumptions on the regular languages. [2, 10, 11] considered the Kleene language that specifies a label set and only allows the use of edge labels in it. It is

a special case of regular languages and it is hard to express various routing requirements in practice. [13] also focused on a special case of regular languages where the corresponding DFA is restricted such that each state has only one self-loop transition and there is only one link label between different states. [1] assumes the prior knowledge of the regular languages before query processing. Its solution has to rebuild its index several times when handling different regular languages in a workload. In contrast, we do not make any assumptions on the regular languages used in query processing. We only need to build the index once to answer queries with different languages, which is more flexible and efficient.

8 CONCLUSION

In this work, we study how to efficiently answer Label-Constrained Shortest Path (LCSP) queries with flexible use of regular languages as constraints. Specifically, we allow any different forms of regular languages for each query. Noticing that the time-consuming part of query processing lies in the self-loop of DFA states, which makes the DFA state unchanged, we propose an index-based solution called BSM that uses *border vertices* to skip the exploration in the network and make the state change in each iteration. Our experiments conducted on real road networks show that our BSM can significantly outperform other competitors and support more flexible use of languages. For future work, we may study more general languages (e.g., context-free and context-sensitive languages) and other types of networks (e.g., biological and social networks).

REFERENCES

[1] L. Wang and R. C. Wong, "PCSP: efficiently answering label-constrained shortest path queries on road networks," *PVLDB*, vol. 17, no. 10, 2024.
[2] J. Zhang, L. Yuan, W. Li, L. Qin, and Y. Zhang, "Efficient label-constrained shortest path queries on road networks: A tree decomposition approach," *PVLDB*, vol. 15, no. 3, pp. 686–698, 2021.
[3] C. L. Barrett, R. Jacob, and M. V. Marathe, "Formal-language-constrained path problems," *SIAM J. Comput.*, vol. 30, no. 3, pp. 809–837, 2000.
[4] D. Kirchler, L. Liberti, T. Pajor, and R. W. Calvo, "Unialt for regular language contrained shortest paths on a multi-modal transportation network," in *ATMOS*, 2011.
[5] S. Wang, X. Xiao, Y. Yang, and W. Lin, "Effective indexing for approximate constrained shortest path queries on large road networks," *PVLDB*, 2016.
[6] A. V. Goldberg and C. Harrelson, "Computing the shortest path: *A* search meets graph theory," in *SODA*, 2005.
[7] C. L. Barrett, K. R. Bisset, R. Jacob, G. Konjevod, and M. V. Marathe, "Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router," in *ESA*, 2002.
[8] C. L. Barrett, K. R. Bisset, M. Holzer, G. Konjevod, M. V. Marathe, and D. Wagner, "Engineering label-constrained shortest-path algorithms," in *AAIM*, 2008.
[9] D. Delling, T. Pajor, and D. Wagner, "Accelerating multi-modal route planning by access-nodes," in *ESA*, 2009.
[10] M. N. Rice and V. J. Tsotras, "Graph indexing of road networks for shortest path queries with label restrictions," *PVLDB*, vol. 4, no. 2, pp. 69–80, 2010.
[11] M. S. Hassan, W. G. Aref, and A. M. Aly, "Graph indexing for shortest-path finding over dynamic sub-graphs," in *SIGMOD*, 2016.
[12] H. D. Sherali, C. Jeenanunta, and A. G. Hobeika, "The approach-dependent, time-dependent, label-constrained shortest path problem," *Networks*, vol. 48, no. 2, pp. 57–67, 2006.
[13] J. Dibbelt, T. Pajor, and D. Wagner, "User-constrained multi-modal route planning," in *ALENEX*, 2012.
[14] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
[15] F. Bonchi, A. Gionis, F. Gullo, and A. Ukkonen, "Distance oracles in edge-labeled graphs," in *EDBT*, 2014.
[16] N. Robertson and P. D. Seymour, "Graph minors. III. planar tree-width," *J. Comb. Theory, Ser. B*, vol. 36, no. 1, pp. 49–64, 1984.
[17] D. Ouyang, L. Qin, L. Chang, X. Lin, Y. Zhang, and Q. Zhu, "When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks," in *SIGMOD*, 2018.
[18] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, 1968.
[19] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang, "Computing label-constraint reachability in graph databases," in *SIGMOD*, 2010.
[20] L. Zou, K. Xu, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao, "Efficient processing of label-constraint reachability queries in large graphs," *Inf. Syst.*, vol. 40, pp. 47–66, 2014.
[21] L. D. J. Valstar, G. H. L. Fletcher, and Y. Yoshida, "Landmark indexing for evaluation of label-constrained reachability queries," in *SIGMOD*, 2017.
[22] Y. Peng, Y. Zhang, X. Lin, L. Qin, and W. Zhang, "Answering billion-scale label-constrained reachability queries within microsecond," *PVLDB*, vol. 13, no. 6, pp. 812–825, 2020.
[23] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.
[24] G. Dantzig, *Linear programming and extensions*. Princeton university press, 1963.
[25] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling, "Fast point-to-point shortest path computations with arc-flags," *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, vol. 74, pp. 41–72, 2009.
[26] H. Bast, S. Funke, P. Sanders, and D. Schultes, "Fast routing in road networks with transit nodes," *Science*, vol. 316, no. 5824, pp. 566–566, 2007.
[27] R. J. Gutman, "Reach-based routing: A new approach to shortest path algorithms optimized for road networks," in *ALENEX/ANALC*, 2004.
[28] P. Sanders and D. Schultes, "Engineering highway hierarchies," *ACM J. Exp. Algorithmics*, vol. 17, no. 1, 2012.
[29] J. Dibbelt, B. Strasser, and D. Wagner, "Customizable contraction hierarchies," in *SEA*, 2014.
[30] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," *Transp. Sci.*, vol. 46, no. 3, pp. 388–404, 2012.
[31] V. J. Wei, R. C. Wong, and C. Long, "Architecture-intact oracle for fastest path and time queries on dynamic spatial networks," in *SIGMOD*, 2020.
[32] D. Ouyang, L. Yuan, L. Qin, L. Chang, Y. Zhang, and X. Lin, "Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees," *PVLDB*, vol. 13, no. 5, pp. 602–615, 2020.
[33] Z. Chen, B. Feng, L. Yuan, X. Lin, and L. Wang, "Fully dynamic contraction hierarchies with label restrictions on road networks," *Data Sci. Eng.*, vol. 8, no. 3, pp. 263–278, 2023.
[34] T. Akiba, Y. Iwata, K. Kawarabayashi, and Y. Kawata, "Fast shortest-path distance queries on road networks by pruned highway labeling," in *ALENEX*, 2014.
[35] Y. Li, L. H. U, M. L. Yiu, and N. M. Kou, "An experimental study on hub labeling based shortest path algorithms," *PVLDB*, vol. 11, no. 4, pp. 445–457, 2017.
[36] W. Li, M. Qiao, L. Qin, Y. Zhang, L. Chang, and X. Lin, "Scaling up distance labeling on graphs with core-periphery properties," in *SIGMOD*, 2020.
[37] ——, "Scaling distance labeling on small-world networks," in *SIGMOD*, 2019.
[38] Z. Chen, A. W. Fu, M. Jiang, E. Lo, and P. Zhang, "P2H: efficient distance querying on road networks by projected vertex separators," in *SIGMOD*, 2021.
[39] Y. Zhang and J. X. Yu, "Relative subboundedness of contraction hierarchy and hierarchical 2-hop index in dynamic road networks," in *SIGMOD*, 2022.
[40] M. Zhang, L. Li, W. Hua, R. Mao, P. Chao, and X. Zhou, "Dynamic hub labeling for road networks," in *ICDE*, 2021.
[41] M. Zhang, L. Li, W. Hua, and X. Zhou, "Efficient 2-hop labeling maintenance in dynamic small-world networks," in *ICDE*, 2021.
[42] A. O. Mendelzon and P. T. Wood, "Finding regular simple paths in graph databases," in *VLDB*, 1989.
[43] A. Likhyani and S. J. Bedathur, "Label constrained shortest path estimation," in *CIKM*, 2013.