QHL: A Fast Algorithm for Exact Constrained Shortest Path Search on Road Networks

Paper ID: 580

ABSTRACT

Route planning is fundamental in our daily life. However, existing mapping applications focus on recommending routes by optimizing one single objective, which is inconsistent with some scenarios where users prefer the optimal route under a constraint. The constrained shortest path (CSP) query matches this requirement, but the query efficiencies of previous solutions are often low due to CSP's NP-hardness. In the era of big data, state-of-the-art indexes are getting larger to support faster query processing. Recent attempts to preprocess more intermediate results and reduce the number of table lookups have proved successful in solving the CSP. However, the best-known algorithm ignores some information in the CSP queries and tries to solve a more general problem before tackling the exact CSP. In this paper, we propose by far the fastest algorithm called QHL, which fully utilizes the pruning power of the CSP query information. Specifically, we preprocess our index by generating pruning conditions that can improve query efficiency. We also conducted extensive experiments on real-world datasets to demonstrate the superiority of our proposed algorithm. QHL could answer each CSP query in around 50 μs and run faster than the best-known algorithm by orders of magnitude.

KEYWORDS

Constrained Shortest Path, 2-hop Labeling

ACM Reference Format:

. 2023. QHL: A Fast Algorithm for Exact Constrained Shortest Path Search on Road Networks: Paper ID: 580. In *Proceedings of ACM SIGMOD conference (SIGMOD'23).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.475/nnn_mmm

1 INTRODUCTION

Route planning plays a critical role in many applications, such as tourists' trip planning, ride-hailing platforms, or food delivery services. Online mapping apps, such as Google Maps [1], offer navigation guidance to meet users' traveling demands. However, the recommended routes often optimize one single objective (e.g., the shortest travel time or distance) and ignore various users' preferences in different scenarios. For example, during a traffic jam, drivers may accept some slightly long detours to experience less congested road segments. Under travelers' limited budgets, the fastest route may be infeasible since it could utilize many highways and bridges with toll charges. Therefore, it is more flexible to consider the *constrained shortest path* (CSP) which minimizes an objective under a constraint.

Given a road network where road segments are represented by edges and their junctions are vertices, each edge can be characterized by two numbers w and c representing two metric values, and CSP asks for a path between a source vertex s and a destination vertex t which minimizes the sum of w values (of the path's edges) under the constraint that the sum of c values is no greater than an upper bound value C. For the first example above, we may regard the congestion degree (as green, yellow, and red lines in the live traffic layer of Google Maps [1]) and the distance of each road segment as w and c, respectively. Drivers may prefer the smoothest detour under a distance constraint. For the second one, we can consider the travel time and the charge of each road segment as w and c, respectively. The optimal route would have the shortest travel time under a given budget. In all, the two metrics have different meanings under different scenarios. Note that there are similar problems to CSP. Some studied how to find a set of *skyline paths* which are suboptimal in terms of the sum of either w or c values [14, 28], but presenting many skyline paths to users makes them confused about different choices. Others considered multiple objectives or constraints [11, 12, 18, 34, 36], but they could burden users' thinking and formulation process.

The CSP problem has been widely studied in the past decades [13, 17]. Early solutions considered the techniques of dynamic programming, linear programming, and approximation algorithms to solve the exact or approximate CSP [15, 17, 23, 24, 36]. However, since it is an NP-hard problem [15], these index-free solutions are unsalable to large road networks. Recent ones designed specific indexes to partition the large network [35] or prune the search space [9, 33]. They do run faster than previous index-free solutions for query processing but still have room for improvement.

As the data volume and the device memory are constantly growing larger, we may harness the power of larger indexes for faster solutions. One state-of-the-art fast solution for the shortest distance query, called *H2H*, considered the combination between the *tree embedding* of the road network and the *2-hop labeling* [26], where the former keeps the structural properties of the road network in a tree in order to enjoy the fast computation on the tree hierarchy, and the latter means that each shortest path can be divided into two subpaths (or two "hops", one from s to an intermediate vertex h and the other from h to t) with their distances (or "labels") stored in the index. In the query processing, the hops are concatenated by some intermediate vertices (also called "hoplinks") to form a set of candidate paths that contain the shortest one, and H2H compares them by necessary distance lookups in the tree. It runs faster than traditional baselines by several orders of magnitude, which is mainly due to its few distance (or label) lookups in the tree. [22] extends the idea of H2H to propose the *CSP-2Hop* to solve the exact CSP. It essentially wants to obtain the set of skyline paths between s and t since the CSP answer must be one of them. Its index, CSP-2Hop, then stores the skyline path sets of hops (instead of the distances) as the labels, and obtains the complete skyline path set for s and t also by some intermediate vertices (or hoplinks). For each hoplink, it considers joining the two skyline path sets of hops, which could be viewed as performing many *path concatenations*. It was also demonstrated that CSP-2Hop outperforms previous solutions by orders of magnitude in terms of query efficiency [22].

Though CSP-2Hop is currently the fastest solution to CSP, it has the following two weaknesses. First, it does not fully utilize the information that each CSP query provide, *i.e.*, the source vertex s, the destination vertex t, and the upper bound value C. Utilizing their pruning power can make the number of label lookups and path concatenations even smaller. Second, it focuses on generating the set of skyline paths between s and t and gets the CSP answer from them in the last minor step. It can be directly used to answer skyline path queries. Therefore, it overlooks some speedup possibilities since we only need the optimal path under the constraint instead of the complete set of skyline paths.

Motivated by the above limitations, in this paper, we propose by far the fastest algorithm called *Q*uery-aware *H*op *L*abeling (QHL) which attaches importance to the pruning power of the query information and the answer form. The basic framework remains the same. However, for the first weakness, we explore the chance of reducing label lookups under some *pruning conditions*. The additional small index that we maintain is about the pruning conditions. For the second one, we propose a new way of path concatenation with its query processing time complexity one fewer multiplier than that of CSP-2Hop. The experiments show that QHL could answer each CSP query in around 50 μs on New York's network and outperform CSP-2Hop by two orders of magnitude on a larger network, and its additional index space compared with CSP-2Hop is negligible.

We summarize our contributions as follows.

- We propose the QHL algorithm that answers the exact CSP queries efficiently. Its time complexity has one fewer multiplier than that of the best-known algorithm.
- We propose several speedup techniques, such as the pruning conditions and a new way of path concatenation. They fully utilize the pruning power of the query information.
- We empirically demonstrate the superiority of our QHL on real-world data. It could reduce the query processing time of the baseline by orders of magnitude while consuming a similar index space.

The remainder of the paper is organized as follows. Section 2 defines the problem. Section 3 presents our query processing algorithm. Section 4 gives the index construction details. Section 5 shows experimental results. Section 6 reviews the related work and Section 7 concludes our paper.

2 PRELIMINARIES

2.1 Problem Definitions

Definition 1 (Road Network). *A road network $G(V, E)$ is represented by a connected undirected graph, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. Each edge $e \in E$ is associated with a weight $w(e) \in \mathbb{R}^+$ and a cost $c(e) \in \mathbb{R}^+$.*

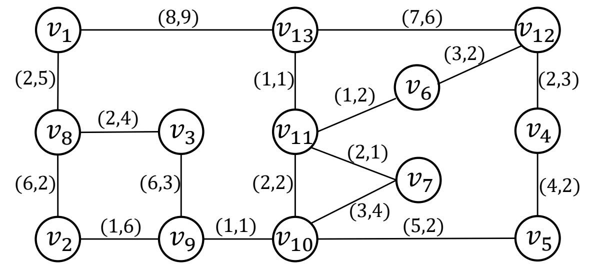
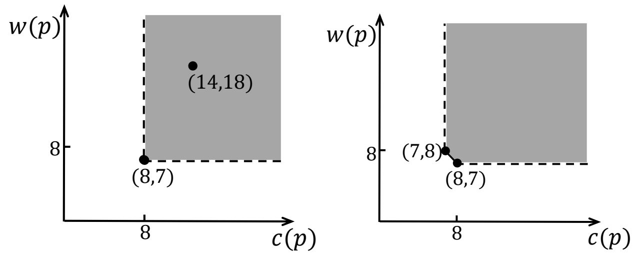


Figure 1: An example road network



(a) Path domination **(b) Skyline path set**

Figure 2: Skyline path example

Example 1. *Figure 1 shows an example road network with 13 nodes. The pair $(w(e), c(e))$ is shown beside each edge e. For example, $w((v_8, v_3)) = 2$ and $c((v_8, v_3)) = 4$.*

Definition 2 (Path). *A path p is a finite sequence of vertices $p = (v_0, v_1, \ldots, v_k)$ such that each $(v_{i-1}, v_i) \in E$ for $1 \le i \le k$. A path with its source vertex s and destination vertex t is called an s-t path. Abusing notations slightly, we define its weight and cost as $w(p) = \sum_{i=1}^{k} w((v_{i-1}, v_i))$ and $c(p) = \sum_{i=1}^{k} c((v_{i-1}, v_i))$, respectively. We denote the concatenation of two paths p_1 and p_2 by $p_1 \oplus p_2$ when p_1's destination is p_2's source. We define the weight-cost pair of a path p as $(w(p), c(p))$.*

It can be easily derived that the $w(p_1 \oplus p_2) = w(p_1) + w(p_2)$ and $c(p_1 \oplus p_2) = c(p_1) + c(p_2)$.

Definition 3 (CSP Query). *Given a road network G, a source vertex $s \in V$, a destination vertex $t \in V$, and a cost budget $C \in \mathbb{R}^+$, the CSP query returns the s-t path p^* that attains the minimum $w(p^*)$ and satisfies $c(p^*) \le C$.*

Example 2. *One CSP query could specify s as v_8, t as v_4, and $C = 13$. The answer to this query is the path $p^* = (v_8, v_2, v_9, v_{10}, v_5, v_4)$ with $(w(p^*), c(p^*)) = (17, 13)$. Any other path with its cost no larger than 13 has its weight larger than 17.*

2.2 Skyline Paths

Definition 4 (Path Domination). *For two paths p and p', p dominates p', denoted by $p \prec p'$, iff 1) $w(p) < w(p')$ and $c(p) \le c(p')$, or 2) $w(p) \le w(p')$ and $c(p) < c(p')$. We also define $p = p'$ iff $(w(p), c(p)) = (w(p'), c(p'))$, and $p \preceq p'$ iff $p \prec p'$ or $p = p'$.*

Example 3. *The weight-cost pairs of the path* (v_8, v_3, v_9) *and* $(v_8, v_1, v_{13}, v_{11}, v_{10}, v_9)$ *are* $(8, 7)$ *and* $(14, 18)$*, respectively. As shown in Figure 2a, if we use weight-cost pairs as coordinates of points and plot them in a coordinate system,* $(14, 18)$ *lies in the upper right part of* $(8, 7)$*, and hence* $(v_8, v_3, v_9) \prec (v_8, v_1, v_{13}, v_{11}, v_{10}, v_9)$.

Definition 5 (Path Set Domination). *For two sets of paths* P *and* P'*,* P *dominates* P'*, denoted by* $P \prec P'$*, if and only if 1) for any path* $p' \in P'$*, there exists a path* $p \in P$ *such that* $p \prec p'$*, and 2) for any path* $p \in P$*, there does not exist a path* $p' \in P'$ *such that* $p' \prec p$.

Definition 6 (Skyline Paths). *Given* s *and* t*, an* s*-*t *path* p *is called a skyline path for* s *and* t *if and only if there does not exist an* s*-*t *path* p' *such that* $p' \prec p$*. The set of all skyline paths for* s *and* t *(which dominates any other* s*-*t *path sets) is denoted by* $\overline{P_{st}}$.

Example 4. *Consider the skyline path set* $\overline{P_{v_8 v_9}}$ *between* v_8 *and* v_9*. It can be seen that the weight-cost pairs of the path* (v_8, v_3, v_9) *and the path* (v_8, v_2, v_9) *are* $(8, 7)$ *and* $(7, 8)$*, respectively. Since they cannot dominate each other and no other paths can dominate them,* $\overline{P_{v_8 v_9}} = \{(v_8, v_3, v_9), (v_8, v_2, v_9)\}$*. As shown in Figure 2b, if we link the skyline paths by lines, the points representing any other paths between* v_8 *and* v_9 *lie in the upper right grey area.*

It is shown that $\overline{P_{st}}$ suffices to answer the CSP query with s, t, and any value of C and CSP-2Hop derives $\overline{P_{st}}$ since $p^* \in \overline{P_{st}}$ [22]. The proof is as follows. We can sort all the skyline paths in $\overline{P_{st}}$ in the increasing order of their costs $c(p)$. The path with the largest $c(p^*) \le C$ is the answer p^*. This is because all the other skyline paths with $c(p) \le c(p^*)$ must have weights $w(p) > w(p^*)$. Otherwise, $p \prec p^*$ contradicts the skyline path set definition.

Example 5. *Back to Example 2,* $\overline{P_{st}}$ *has three paths with weight-cost pairs as follows:* $(18, 12), (17, 13), (16, 18)$*. The answer* p^* *lies in* $(17, 13)$ *since 13 is the largest possible value no larger than* C.

2.3 CSP-2Hop

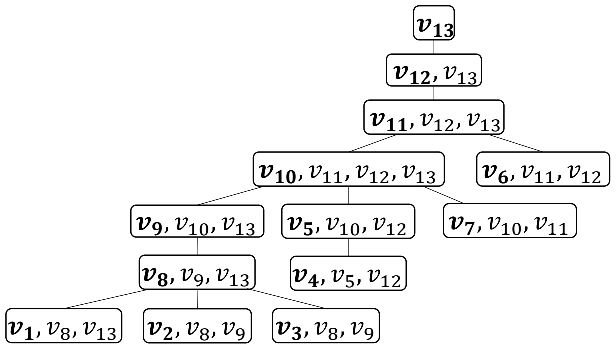
A *tree decomposition* is a rooted tree that is generated from the network G and satisfies certain conditions [26, 30]. See [4] for a survey. The conditions are omitted here for ease of illustration. Specifically, we use the tree decomposition generated by Algorithm 6 in [26] or Algorithm 1 in [22].

Definition 7 (Tree Decomposition [26]). *The tree decomposition of the graph* G *is a rooted tree, denoted by* T*. There is a bijection from* V *to the set of tree nodes, denoted by* X*. For each* $v \in V$*,* $X(v)$ *is its corresponding tree node. In each tree node* $X(v)$*, there is a subset of* V *of vertices which includes* v*. Abusing notations slightly, we also use* $X(v)$ *to denote this subset in* v*'s corresponding tree node. Hence,* $X(v) \subset V$ *and* $v \in X(v)$.

Example 6. *Figure 3 shows a tree decomposition for the graph in Figure 1. In each tree node* $X(v)$ *(where* v *is bolded), there is a set of vertices that includes* v*,* e.g., $X(v_8) = \{v_8, v_9, v_{13}\}$ *and* $v_8 \in X(v_8)$.

We will use "vertex" for the graph G and "node" for the tree to distinguish the two concepts. The valuable property of the tree decomposition is about the separator defined below.

Definition 8 (Separator). *The separator of two vertices* s *and* t *is a set of vertices such that after the removal of all the vertices in the separator,* s *and* t *are in two different connected components.*



$\boldsymbol{v_{13}}$
$\boldsymbol{v_{12}}, v_{13}$
$\boldsymbol{v_{11}}, v_{12}, v_{13}$
$\boldsymbol{v_{10}}, v_{11}, v_{12}, v_{13}$
$\boldsymbol{v_6}, v_{11}, v_{12}$
$\boldsymbol{v_9}, v_{10}, v_{13}$
$\boldsymbol{v_5}, v_{10}, v_{12}$
$\boldsymbol{v_7}, v_{10}, v_{11}$
$\boldsymbol{v_8}, v_9, v_{13}$
$\boldsymbol{v_4}, v_5, v_{12}$
$\boldsymbol{v_1}, v_8, v_{13}$
$\boldsymbol{v_2}, v_8, v_9$
$\boldsymbol{v_3}, v_8, v_9$

Figure 3: An example tree decomposition

Example 7. *In Figure 1, a separator of* v_8 *and* v_4 *could be* $\{v_{10}, v_{13}\}$ *since* v_8 *is disconnected from* v_4 *after* v_{10} *and* v_{13} *are removed.*

The separator also indicates that any s-t path has to visit at least one of the vertices in the separator of s and t.

Lemma 1 (Lemma 4 in [5]). *Given any two vertices* s *and* t*, suppose that there is no ancestor-descendant relationship between the two nodes* $X(s)$ *and* $X(t)$ *in the tree. Let* $X(l)$ *be the least common ancestor (LCA) of* $X(s)$ *and* $X(t)$*, and* $P = (X(s), \ldots, X(l), \ldots, X(t))$ *be the simple tree path connecting* $X(s)$ *and* $X(t)$*. The set* $X(l)$ *of vertices is a separator. For each* $X(v) \in P$ *where* $v \ne l$*, the set* $X(v) \backslash \{v\}$ *of vertices is also a separator.*

Example 8. *Given* v_8 *and* v_4*, since there is no ancestor-descendant relationship between the two nodes* $X(v_8)$ *and* $X(v_4)$*, and* $X(v_{10})$ *is the LCA,* $X(v_{10}) = \{v_{10}, v_{11}, v_{12}, v_{13}\}$ *is a separator for* v_8 *and* v_4.

Since the $X(l)$ of the LCA is a separator, CSP-2Hop considers dividing the problem into two parts; that is, each s-t path can be seen as the concatenation of two subpaths: one from s to a vertex $h \in X(l)$, also called a "hoplink", and the other from h to t. If we have the two skyline path sets $\overline{P_{sh}}$ and $\overline{P_{ht}}$ for each $h \in X(l)$, we can get $\overline{P_{st}}$ by iterating all the vertices in the separator $X(l)$ and concatenating any pair of two paths in $\overline{P_{sh}}$ and $\overline{P_{ht}}$. Formally, for the separator $X(l)$, $p^* \in \overline{P_{st}} \subseteq \bigcup_{h \in X(l)} \{p_1 \oplus p_2 : p_1 \in \overline{P_{sh}}, p_2 \in \overline{P_{ht}}\}$. We can find the optimal answer p^* in all the concatenated paths.

Specifically, CSP-2Hop preprocesses its index which stores a label for each $v \in V$, denoted by $L(v)$. Each label $L(v)$ is a set of pairs $(u, \overline{P_{vu}})$ for any $X(v)$'s ancestor $X(u)$ in the tree. For example, $L(v_{10}) = \{(v_{11}, \overline{P_{v_{10}v_{11}}}), (v_{12}, \overline{P_{v_{10}v_{12}}}), (v_{13}, \overline{P_{v_{10}v_{13}}})\}$. The labels can be generated by Algorithm 1 in [22]. Given a CSP query with s, t, and C, we answer it by using the labels to find $\overline{P_{st}}$ and further the optimal answer p^*. Algorithm 1 summarizes the query procedure. If $X(s)$ and $X(t)$ have the ancestor-descendant relationship, we directly find in the descendant's label the corresponding $\overline{P_{st}}$ in lines 2-5. Otherwise, since the $X(l)$ of the LCA of $X(s)$ and $X(t)$ is a separator, we can consider each vertex in $X(l)$ as a hoplink h and let *Hoplinks* be $X(l)$ in line 7. Furthermore, the tree decomposition has a useful property.

Property 1 (Property 2 in [26]). *For any vertex* $u \in X(v) \backslash \{v\}$*,* $X(u)$ *is an ancestor of* $X(v)$.

Algorithm 1: CSP-2Hop

```
input  : The label $L(v)$ for each $v \in V$ and a CSP query
output : The path $p$ with $c(p) \leq C$ and the minimum $w(p)$
1 $X(l) \leftarrow$ the LCA node of $X(s)$ and $X(t)$
2 if $l = s$ then
3     return the optimal path in $\overline{P_{st}}$ stored in $L(t)$
4 else if $l = t$ then
5     return the optimal path in $\overline{P_{st}}$ stored in $L(s)$
6 else
7     $Hoplinks \leftarrow X(l)$
8     return $p^*$ in $\bigcup_{h \in Hoplinks} \{p_1 \oplus p_2 : p_1 \in \overline{P_{sh}}, p_2 \in \overline{P_{ht}}\}$
```

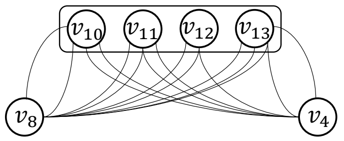


Figure 4: Path concatenation in CSP-2Hop

Example 9. *For $v_{11}, v_{12}, v_{13} \in X(v_{10})$, $X(v_{11}), X(v_{12}), X(v_{13})$ are ancestors of $X(v_{10})$.*

This property further indicates that for any $h \in X(l)$, we can find $\overline{P_{sh}}$ in the label $L(s)$ and $\overline{P_{ht}}$ in $L(t)$ since $X(h)$ is the ancestor of both $X(s)$ and $X(t)$ and we store the two sets $\overline{P_{sh}}$ and $\overline{P_{ht}}$ by the label definition. We can then obtain a super set of $\overline{P_{st}}$ that is the set of all concatenated paths $\bigcup_{h \in Hoplinks} \{p_1 \oplus p_2 : p_1 \in \overline{P_{sh}}, p_2 \in \overline{P_{ht}}\}$ and get p^* from it in line 8.

The query processing time complexity is $O(|X(l)||\overline{P_{sh}}||\overline{P_{ht}}|)$ which mainly lies in the last step of path concatenation. It is efficient in practice because $|X(l)|$ is bounded by $\max_v |X(v)|$ (also called the "treewidth") which is smaller than 1000 for nearly all road networks [22, 26]. The correctness of Algorithm 1 and the separator can be found in [22].

Note that the last line of Algorithm 1 can be done by updating the optimal path p^* while using C to filter out those concatenated paths with $c(p) > C$ without deriving the real $\overline{P_{st}}$. Also note that the extension to the directed graph and the path retrieval (since the labels store the weight-cost pairs for efficiency) can be found in [22], and ours are the same.

Example 10. *We still use Example 2. Since $X(v_8)$ and $X(v_4)$ have no ancestor-descendant relationship, we find its LCA node $X(v_{10})$. We will use Hoplinks $= X(v_{10}) = \{v_{10}, v_{11}, v_{12}, v_{13}\}$. Figure 4 shows all the paths to be concatenated with their details omitted for simplicity. Take v_{10} as the first hoplink for example. Since $|\overline{P_{v_8 v_{10}}}| = 2$ and $|\overline{P_{v_{10} v_4}}| = 2$, we have to do 4 concatenations. For v_{11}, v_{12}, and v_{13}, we need to perform 4, 2, and 6 concatenations, respectively. After performing all the 16 concatenations, we get the final answer $p^* = (v_8, v_2, v_9, v_{10}, v_5, v_4)$ with the minimum weight of 17. Note that through our later examples, we will show that our proposed QHL only needs to do 3 concatenations.*

Algorithm 2: Query Processing Overview

```
input  : The QHL index and a CSP query with $s$, $t$, and $C$
output : The path $p$ with $c(p) \leq C$ and the minimum $w(p)$
1  $X(l) \leftarrow$ the LCA node of $X(s)$ and $X(t)$
2  if $l = s$ then
3      return the optimal path in $\overline{P_{st}}$ stored in $L(t)$
4  else if $l = t$ then
5      return the optimal path in $\overline{P_{st}}$ stored in $L(s)$
6  else
7      initialize two separators $H(s)$ and $H(t)$ (Section 3.2)
8      $\mathcal{H} \leftarrow$ the set of pruned separators if $s$, $t$, $C$, and $H(s)$ or
         $H(t)$ satisfy certain pruning conditions (Section 3.3)
9      $Hoplinks \leftarrow \arg\min_{H \in \mathcal{H}} T(H)$
10     foreach $h \in Hoplinks$ do
11         use $C$ to find the suboptimal path
           $p_h^* \in \{p_1 \oplus p_2 : p_1 \in \overline{P_{sh}}, p_2 \in \overline{P_{ht}}\}$ (Section 3.4)
12     return $p^* \leftarrow \arg\min_{p \in \{p_h^* : h \in Hoplinks\}} w(p_h^*)$
```

3 QUERY PROCESSING

We will start with the overview of the query processing of QHL and give details of each step in later sections.

3.1 Overview

CSP-2Hop's query processing time complexity is $O(|X(l)||\overline{P_{sh}}||\overline{P_{ht}}|)$. It consists of two parts: one is the size of the hoplinks and the other is the product of two sizes of the skyline path sets. To speed it up, we utilize the power of the query information (which includes s, t, and C) to reduce the size of the hoplinks and the time cost of path concatenation in the query processing. Note that in the following, we will use *Hoplinks* to represent the set of intermediate vertices in the final path concatenation as in Algorithm 1. Any separator is sufficient but probably redundant to be used as *Hoplinks*. Our Algorithm 2 runs in $O(|Hoplinks|(|\overline{P_{sh}}| + |\overline{P_{ht}}|))$ time, where *Hoplinks* is always no greater than $|X(l)|$.

Algorithm 2 gives an overview of the query processing. We start with the same labels $L(v)$ for any v as in CSP-2Hop. In lines 1-5, we use the same way to handle the case where $X(s)$ and $X(t)$ have an ancestor-descendant relationship.

When $X(s)$ and $X(t)$ have no ancestor-descendant relationship, which is also the most time-consuming part of the query processing, we propose our speed-up techniques. Note that the steps from lines 7-11 are also shown in Figure 5 for reference. Given s and t, CSP-2Hop directly uses the set $X(l)$ in the LCA node of $X(s)$ and $X(t)$ as *Hoplinks*. However, our algorithm in line 7 initializes two different separators $H(s)$ and $H(t)$, corresponding to the two child nodes of $X(l)$ in the same branches of $X(s)$ and $X(t)$ in the first step of Figure 5. Since using either of them as the final *Hoplinks* can invoke a different execution plan of the final path concatenation with a different time cost, they are just candidate separators. Next, utilizing the information of s, t, and C in line 8, we may prune $H(s)$ (or $H(t)$) based on some preprocessed *pruning conditions* to get one or two pruned separators which are better candidates. If $H(s)$ (or $H(t)$) cannot be pruned, we stick to the original $H(s)$ (or $H(t)$). Let

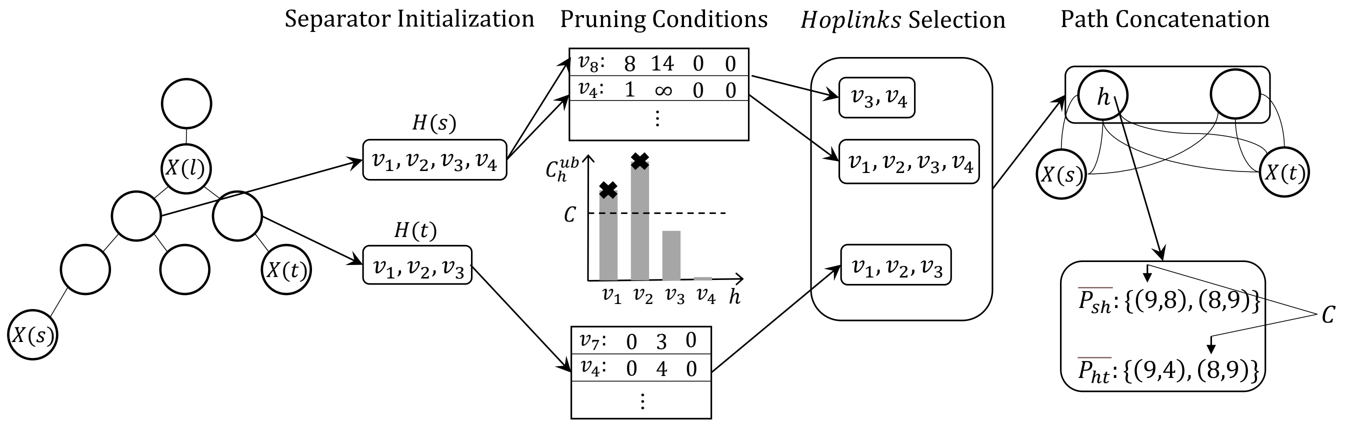


Figure 5: Querying processing overview

$\mathcal{H}$ denote the set of candidate separators and $|\mathcal{H}|$ could be 2,3,4. Since using any separator $H \in \mathcal{H}$ as the final *Hoplinks* may incur a different time cost, we will decide which one is better based on the cost estimation in line 9. The estimated time cost of a separator H is defined as $T(H) = \sum_{h \in H}(|\overline{P_{sh}}|+|\overline{P_{ht}}|)$, which is because we adopt a fast method for path concatenation within $O(|\overline{P_{sh}}| + |\overline{P_{ht}}|)$ time for each hoplink h in line 11. Since we are answering a CSP query with the cost budget C, we do not have to obtain the complete skyline path set $\overline{P_{st}}$ as in CSP-2Hop. Instead, we can use C to efficiently find the suboptimal path for each hoplink and then compare them to obtain the optimal one p^*. Specifically, let p_h^* denote the suboptimal path that has the minimum weight $w(p_h^*)$ and $c(p_h^*) \leq C$ among all the paths in $\{p_1 \oplus p_2 : p_1 \in \overline{P_{sh}}, p_2 \in \overline{P_{ht}}\}$ by using the hoplink h. It can be found in $O(|\overline{P_{sh}}| + |\overline{P_{ht}}|)$ time since we adopt a new way of path concatenation. The final answer p^* will be one of the paths p_h^* for $h \in Hoplinks$ with the minimum weight.

The time complexity of Algorithm 2 is $O(|Hoplinks|(|\overline{P_{sh}}| + |\overline{P_{ht}}|))$, where the main cost is still from path concatenation in lines 10-11. Each section later contains its corresponding detailed complexity analysis. Note that our additional index contains only the pruning conditions in line 8 (discussed in Section 4). We preprocess the same tree decomposition and labels as in CSP-2Hop.

3.2 Separator Initialization

In the first step in line 7 of Algorithm 2, we need to initialize the two separators $H(s)$ and $H(t)$, which are two temporary candidates for the final *Hoplinks*.

When $X(s)$ and $X(t)$ have no ancestor-descendant relationship, Lemma 1 tells us that each set $X(v)\backslash\{v\}$ for each node along the simple tree path connecting $X(s)$ and $X(t)$, except for the LCA node $X(l)$, could be a separator. We may simply want to choose the one with the minimum size along the tree path. However, our selected separators are required to be "feasible". By saying a separator is feasible, we mean that when we use it as *Hoplinks*, we could find for each hoplink h the sets $\overline{P_{sh}}$ and $\overline{P_{ht}}$ in the labels $L(s)$ and $L(t)$, respectively, for the final path concatenation. Recall that $L(v)$ only stores the pairs $(u, \overline{P_{vu}})$ for any $X(v)$'s ancestor $X(u)$ in the tree. In other words, for any u to be a hoplink, $X(u)$ should be the ancestor of both $X(s)$ and $X(t)$. Let $X(c_s)$ and $X(c_t)$ be the two child nodes of the LCA node $X(l)$ in the two branches containing $X(s)$ and $X(t)$, respectively. Only the two separators $X(c_s)\backslash\{c_s\}$ and $X(c_t)\backslash\{c_t\}$ are feasible due to the following property.

Property 2 (Lemma 2 in [6]). *For any child node $X(c)$ of $X(v)$, $X(c)\backslash\{c\} \subset X(v)$.*

By the above property, we know that for any $u \in X(c_s)\backslash\{c_s\}$, $u \in X(l)$ since $X(c_s)$ is the child node of the LCA $X(l)$. Using Property 1 for the same u, we know that $X(u)$ is either an ancestor of $X(l)$ when $u \neq l$ or $X(l)$ when $u = l$. In both cases, $X(u)$ is the ancestor of both $X(s)$ and $X(t)$, which makes $X(c_s)\backslash\{c_s\}$ a feasible separator. The same is true for $X(c_t)\backslash\{c_t\}$. We then define the two separators $H(s) = X(c_s)\backslash\{c_s\}$ and $H(t) = X(c_t)\backslash\{c_t\}$. They will be our two initial candidates for the final *Hoplinks*.

For the time complexity, since we can find the LCA node of $X(s)$ and $X(t)$ quickly [3], we just need $O(|H(s)| + |H(t)|)$ time to retrieve the two sets of vertices.

Example 11. *Given $s = v_8$ and $t = v_4$, the LCA node is $X(v_{10})$. $H(s) = X(v_9)\backslash\{v_9\} = \{v_{10}, v_{13}\}$ and $H(t) = X(v_5)\backslash\{v_5\} = \{v_{10}, v_{12}\}$. Both $H(s)$ and $H(t)$ have sizes smaller than $|X(v_{10})|$.*

3.3 Separator Pruning

Given a separator H, which could be either $H(s)$ or $H(t)$ in line 8 of Algorithm 2, we will generate one or two pruned separators by the *pruning conditions*. We will first give the definition and the usage of the pruning condition and then introduce a theorem to justify its correctness.

Definition 9 (Pruning Conditions of a Separator). *For one separator H, there is a set of pruning conditions, each in the form of (v_{end}, C^{ub}) where $v_{end} \in V$ is called an end vertex and $C^{ub} = \{C_h^{ub} \in \mathbb{R}_0^+ : h \in H\}$ is a set of values as upper bounds. A pruning condition is satisfied if either s or t is v_{end}. It is then used to prune any vertex $h \in H$ such that $C_h^{ub} > C$.*

Example 12. *For a separator $H = \{v_{10}, v_{13}\}$, suppose that one of its pruning conditions specifies $v_{end} = v_8$, $C_{v_{10}}^{ub} = 0$, and $C_{v_{13}}^{ub} = 14$.*

Algorithm 3: Pruned Separators

```
input  : The CSP query, the set of pruned separator $\mathcal{H}$, a
         separator $H$ and its set of pruning conditions
         $\{(v_{end}, C^{ub})\}$
output : The set of pruned separator $\mathcal{H}$
1 $\mathcal{H} \leftarrow \mathcal{H} \cup \{H\}$
2 if $s$ matches some $v_{end}$ of $(v_{end}, C^{ub})$ then
3     $\mathcal{H} \leftarrow \mathcal{H} \backslash \{H\} \cup \{\{h \in H : C \geq C^{ub}_h\}\}$
4 if $t$ matches some $v_{end}$ of $(v_{end}, C^{ub})$ then
5     $\mathcal{H} \leftarrow \mathcal{H} \backslash \{H\} \cup \{\{h \in H : C \geq C^{ub}_h\}\}$
6 return $\mathcal{H}$
```

Using the CSP query in Example 2, since $s = v_8 = v_{end}$ satisfies this pruning condition, we then prune v_{13} because $C^{ub}_{v_{13}} = 14 > C = 13$. We cannot prune v_{10} because $C^{ub}_{v_{10}} = 0 \leq C$. After using this pruning condition, we have one fewer hoplink than the original $|X(l)|$. Note that $t = v_4$ does not satisfy this pruning condition because $v_{end} \neq v_4$.

The pruning conditions are obtained during preprocessing. In the query processing, the set of pruned separators $\mathcal{H} = \emptyset$ initially. We then apply Algorithm 3 for both $H(s)$ and $H(t)$ to get pruned separators to expand $\mathcal{H}$. If the separator has a pruning condition whose v_{end} is s in lines 2-3, we then prune any $h \in H$ with $C^{ub}_h > C$ by using the pruning condition. Similarly, we can do this if some v_{end} is t in lines 4-5. The final $\mathcal{H}$ will contain 2, 3, or 4 separators after we call Algorithm 3 twice for $H(s)$ and $H(t)$. In all, the time complexity of line 8 is $O(|H(s)|+|H(t)|)$. Checking the satisfiability of v_{end} can be fast in $O(1)$ by hashing.

Example 13. *Suppose that we only have one pruning condition for $H = \{v_{10}, v_{13}\}$ as in the previous example. Initially, $\mathcal{H} = \{H\}$. For $H = \{v_{10}, v_{13}\}$, we update $\mathcal{H} = \{\{v_{10}\}\}$ since s matches the v_{end} of the pruning condition. There is no pruning condition whose v_{end} matches $t = v_4$. For the separator $H = \{v_{10}, v_{12}\}$, we do nothing because there is no pruning condition for it. The final $\mathcal{H} = \{\{v_{10}\}, \{v_{10}, v_{12}\}\}$.*

Let $\theta \in \mathbb{R}^+$ and P be a set of paths. Let $P^\theta = \{p \in P : c(p) < \theta\}$. We will next justify the pruning conditions by the following theorem for any CSP queries with fixed s and C, and any t. Its proof is deferred.

Theorem 1. *Given a CSP query with s, C, and any t, for a fixed $h \in H$, it can be safely pruned if there exists a value $\theta \in \mathbb{R}^+$ and a vertex $u \in H$ other than h such that $C < \theta$ and $\overline{P_{sh}}^\theta \subseteq \{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}, p_2 \in \overline{P_{uh}}\}^\theta$. If the latter two conditions hold, we simply say that h is pruned by u under θ for s.*

By the symmetry of s and t, the theorem is also true for a CSP query with fixed t and C, and any s.

Example 14. *Consider the previous CSP query with $s = v_8$, $t = v_4$, and $C = 13$. Supposet that $H = \{v_{10}, v_{13}\}$ and $h = v_{13}$. For ease of notations, we will use the weight-cost pair to represent different paths. Since u could only be v_{10}, $\overline{P_{sh}} = \{(12, 11), (11, 12), (10, 14)\}$, $\overline{P_{su}} = \{(9, 8), (8, 9)\}$, and $\overline{P_{uh}} = \{(3, 3)\}$. Thus, $\{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}, p_2 \in \overline{P_{uh}}\} = \{(12, 11), (11, 12)\}$. It can be seen that any value $\theta \in (13, 14]$ can all make the latter two condition hold and v_{13} is pruned by v_{10} under $\theta \in (13, 14]$ for $s = v_8$.*

For a fixed h, suppose that we have found some u and θ such that h is pruned by u under θ for s. We can directly build a pruning condition by setting $v_{end} = s$ and $C^{ub}_h = \theta$ because safely pruning h for s requires that $v_{end} = s$ and $C < \theta = C^{ub}_h$ by Theorem 1, which is consistent with the usage of the pruning condition. When many values of θ satisfy Theorem 1, we can set C^{ub}_h to be the largest one to make more CSP queries with different C fit the condition. Some intuition of Theorem 1 is that if $C < \theta$, we can derive the condition of interest for the CSP query with C, *i.e.*, $\overline{P_{sh}}^C \subseteq \{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}, p_2 \in \overline{P_{uh}}\}^C$ since $C < \theta$ should be more strict. When $\overline{P_{sh}}^C \subseteq \{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}, p_2 \in \overline{P_{uh}}\}^C$ holds, it means that all the skyline paths between s and h could be replaced by the paths from s to h via u, and it is sufficient to use u as the hoplink to find the answer. An efficient way of finding such relationships is described in Section 4.1.

Before proving Theorem 1, we first show some lemmas.

Lemma 2. *If $P_1 \subseteq P_2$, $P_1^\theta \subseteq P_2^\theta$.*

Proof. If $p \in P_1^\theta$, $p \in P_1 \subseteq P_2$ and $c(p) \leq \theta$. Thus, $p \in P_2^\theta$. □

Lemma 3. $\{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}, p_2 \in \overline{P_{uh}}\}^\theta = \{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}^\theta, p_2 \in \overline{P_{uh}}\}^\theta$.

Proof. RHS ⊆ LHS because for any path $p_1 \oplus p_2 \in$ RHS, we know that $p_1 \in \overline{P_{su}}^\theta \subseteq \overline{P_{su}}$, which implies $p_1 \oplus p_2 \in$ LHS. LHS ⊆ RHS because for any path $p_1 \oplus p_2 \in$ LHS, we have $c(p_1) < \theta$. Otherwise, $\theta \leq c(p_1) < c(p_1 \oplus p_2)$, contradicting the fact $p_1 \oplus p_2 \in P^\theta$ where $P = \{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}, p_2 \in \overline{P_{uh}}\}$. Since $c(p_1) < \theta$, it implies $p_1 \in \overline{P_{su}}^\theta$ and $p_1 \oplus p_2 \in$ RHS. □

Lemma 4. *For the same s and θ, if h is pruned by u, $\overline{P_{su}}^\theta \prec \overline{P_{sh}}^\theta$.*

Proof. We prove it by following Definition 5. By Lemma 3, we have $\overline{P_{sh}}^\theta \subseteq \{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}^\theta, p_2 \in \overline{P_{uh}}\}^\theta$. For any path $p \in \overline{P_{sh}}^\theta$, there exist $p_1 \in \overline{P_{su}}^\theta$ and $p_2 \in \overline{P_{uh}}$ such that $p = p_1 \oplus p_2$. Since $w(p_2) > 0$ and $c(p_2) > 0$, we have $p_1 \prec p$. For any path $p_1 \in \overline{P_{su}}^\theta$, if there exists a path $p \in \overline{P_{sh}}^\theta$ such that $p \prec p_1$, the path p can be made up by $p_3 \oplus p_4$ where $p_3 \in \overline{P_{su}}^\theta$ and $p_4 \in \overline{P_{uh}}$. However, since $w(p_4) > 0$ and $c(p_4) > 0$, this indicates that $p_3 \prec p \prec p_1$ where both p_1 and p_3 are in $\overline{P_{su}}^\theta$, which contradicts the definition of a skyline path set. □

Lemma 5. *For the same s and θ, suppose that there is a finite sequence of vertices $(v_1, v_2, \ldots, v_k)$ such that each v_i is pruned by v_{i+1} for $i = 1, 2, \ldots, k-1$. These vertices are distinct from each other.*

Proof. By Lemma 4, we have $\overline{P_{sv_{i+1}}}^\theta \prec \overline{P_{sv_i}}^\theta$ for $i = 1, 2, \ldots, k-1$ since v_i is pruned by v_{i+1}. Suppose that $v_i = v_j$ holds for $i < j$. We could take the subsequence between v_i and v_j. By applying Lemma 4 repeatedly, we can derive that $\overline{P_{sv_i}}^\theta \prec \overline{P_{sv_j}}^\theta = \overline{P_{sv_i}}^\theta$, which contradicts Definition 5. □

For a separator H, let H' be the separator after we prune it.

Algorithm 4: Path Concatenation

```
input  : The CSP query, the hoplink $h$, sorted $\overline{P_{sh}}$ and $\overline{P_{ht}}$
output : The path $p_h^*$
1 $i \leftarrow 1, j \leftarrow |\overline{P_{ht}}|$
2 while $i \neq |\overline{P_{sh}}| + 1$ and $j \neq 0$ do
3     if $c(p_{sh}^{(i)} \oplus p_{ht}^{(j)}) \leq C$ then
4         if $w(p_{sh}^{(i)} \oplus p_{ht}^{(j)}) < w(p_h^*)$ then
5             $p_h^* \leftarrow p_{sh}^{(i)} \oplus p_{ht}^{(j)}$
6         $i \leftarrow i + 1$
7     else
8         $j \leftarrow j - 1$
9 return $p_h^*$
```

Corollary 1. *For the sequence above, there must exists the last vertex v_k that cannot be pruned by any one, and hence $v_k \in H'$.*

Proof. By Lemma 5, the length k of the sequence is at most $|H|$. There exists no vertex to prune the last vertex v_k. □

Since p^* must pass through some vertex $h \in H$. Let p^*_{sh} and p^*_{ht} be the two sub-path with the vertex $h \in H$ as the destination and the source, respectively, and $p^*_{sh} \oplus p^*_{ht} = p^*$.

Proof of Theorem 1. Given a CSP query with s, C, and any t, we will show that we can still find the final answer p^* by using H' as *Hoplinks*. We consider two cases of p^*. If p^* passes through any vertex $h \in H'$, the correctness is guaranteed since we will consider h as a hoplink for the path concatenation. Otherwise, p^* passes through some vertex $h \in H \backslash H'$ that is pruned by some vertex u. We will first show that if h is pruned by u and p^* passes through h, p^* must pass through u. Since $\overline{P_{sh}}^{\theta} \subseteq \{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}, p_2 \in \overline{P_{uh}}\}^{\theta}$ and $C < \theta$, we have $\overline{P_{sh}}^{C} \subseteq \{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}, p_2 \in \overline{P_{uh}}\}^{C}$ by Lemma 2. Since $p^*_{sh} \in \overline{P_{sh}}^{C}$, there exists $p_1 \oplus p_2 = p^*_{sh}$ such that $p_1 \in \overline{P_{su}}, p_2 \in \overline{P_{uh}}$. It further implies that p^* passes through u. Since u could also be pruned by another vertex, there is a sequence $(v_1, v_2, \ldots, v_k)$ such that $v_1 = h$, $v_2 = u$, and v_i is pruned by v_{i+1} for $i = 1, 2, \ldots, k-1$ and $k \geq 2$. If p^* passes through some v_i and v_i is pruned by v_{i+1}, p^* must pass through v_{i+1}. Using the above statement repeatedly, p^* must pass through $v_k \in H'$ which will be a hoplink. In all, we can find p^* by using H'. □

3.4 Path Concatenation

Given a hoplink $h \in Hoplinks$, in line 11 of Algorithm 2, we propose to first find the path p^*_h such that it has the minimum weight $w(p^*_h)$ and $c(p^*_h) \leq C$ among all the paths in $\{p_1 \oplus p_2 : p_1 \in \overline{P_{sh}}, p_2 \in \overline{P_{ht}}\}$. The main idea is that if $\overline{P_{sh}}$ and $\overline{P_{ht}}$ are sorted in the increasing order of the cost in advance, we can use C to skip some unnecessary path concatenations on the two sorted path sequences.

Let $p^{(i)}_{sh}$ and $p^{(j)}_{ht}$ denote the path with the i-th smallest cost in $\overline{P_{sh}}$ and the path with the j-th smallest cost in $\overline{P_{ht}}$, respectively. Algorithm 4 summarizes the whole procedure. We set two pointers i and j which represent the two paths $p^{(i)}_{sh}$ and $p^{(j)}_{ht}$ that we are about to concatenate. Initially, in line 1, we set $i = 1$ and $j = |\overline{P_{ht}}|$, which indicates that we will first concatenate the path with the smallest cost in $\overline{P_{sh}}$ and the path with the largest cost in $\overline{P_{ht}}$. In each iteration, if the concatenated path $p^{(i)}_{sh} \oplus p^{(j)}_{ht}$ has its cost no greater than C (in line 3), we will update the answer p^*_h if the concatenated path has a smaller weight than p^*_h's current one in lines 4-5. We next increment i by 1 in line 6, which indicates that any path concatenation $p^{(i)}_{sh} \oplus p^{(j')}_{ht}$ for $1 \leq j' < j$ could be ignored since they could only have smaller costs (by the increasing order of the cost in $\overline{P_{ht}}$) and hence a larger weight (by the skyline definition). If the concatenated path $p^{(i)}_{sh} \oplus p^{(j)}_{ht}$ has its cost greater than C (in line 3), we decrement j by 1 in line 8, which indicates that any path concatenation $p^{(i')}_{sh} \oplus p^{(j)}_{ht}$ for $i < i' \leq |\overline{P_{sh}}|$ could be ignored since they could only have greater costs (by the increasing order of the cost in $\overline{P_{sh}}$), also violating C. The loop stops when $i = |\overline{P_{sh}}| + 1$ or $j = 0$, which indicates that there is no more path concatenation that we need to check.

Note that $w(p^*_h)$ can be initialized to $+\infty$ at first. Algorithm 4's time complexity is $O(|\overline{P_{sh}}| + |\overline{P_{ht}}|)$ because i is increased by $|\overline{P_{sh}}|$ and j is decreased by $|\overline{P_{ht}}|$ in the worst case.

Example 15. *Consider $s = v_8$, $t = v_4$, $C = 13$, and $h = v_{10}$. $\overline{P_{sh}} = \{(9,8),(8,9)\}$ and $\overline{P_{ht}} = \{(9,4),(8,9)\}$ sorted in the increasing order of costs. In the beginning, $i = 1$ and $j = |\overline{P_{ht}}| = 2$, indicating that we first concatenate the path with $(9,8) \in \overline{P_{sh}}$ and $(8,9) \in \overline{P_{ht}}$. Since its cost $8+9 > C$, we will set $j = 1$ and next consider $(9,4) \in \overline{P_{ht}}$ and get $(9+9, 8+4) = (18,12)$ with its cost of $12 \leq C$. We will update the answer p^*_h to this path with its weight of 18 and set $i = 2$. Next, we consider $(8,9) \in \overline{P_{sh}}$ and get $(8+9, 9+4) = (17,13)$ with its cost of $13 \leq C$. We will further set $i = 3$ and update p^*_h since its weight of 17 is smaller than the current weight of 18. We finally stop the algorithm since $i = 3$ and return p^*_h.*

4 INDEX CONSTRUCTION

Our index consists of two parts. One is the tree and label indexes as in CSP-2Hop [22], and the other is the pruning conditions. We will first explain how to find a pruning condition for a separator H and a given end vertex v_{end}. However, it is costly to find all pruning conditions for all separators and all end vertices. We will then propose an efficient strategy for selecting appropriate combinations of separators and end vertices. Finally, we will discuss the feasibility of other forms of pruning conditions by combining s, t, and C.

4.1 Fixed Separator and End Vertex

Given a separator H and an end vertex v_{end}, we need to find the corresponding C^{ub}_h for each $h \in H$ by Theorem 1. Initially, we can set $C^{ub}_h = 0$ for any $h \in H$ to represent that no vertex h could be pruned currently since $C^{ub}_h = 0 < C$ for any C. We will first describe how to find C^{ub}_h for fixed h and u and then show how to find C^{ub}_h for all $h \in H$.

4.1.1 Fixed h and u. By Theorem 1, for fixed h, v_{end}, and u, we need to find some $\theta \in \mathbb{R}^+$ such that $\overline{P_{sh}}^{\theta} \subseteq \{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}, p_2 \in \overline{P_{uh}}\}^{\theta}$. We will then set C^{ub}_h as the largest θ which makes

Algorithm 5: Setting C_h^{ub}

```
input  : The three sorted sets $\overline{P_{v_{end}h}}$, $\overline{P_{v_{end}u}}$, and $\overline{P_{uh}}$
output : The upper bound $C_h^{ub}$
1  $C_h^{ub} \leftarrow 0, j \leftarrow 1,$
   $P' \leftarrow \overline{P_{v_{end}h}}, P'' \leftarrow \{p_1 \oplus p_2 : p_1 \in \overline{P_{v_{end}u}}, p_2 \in \overline{P_{uh}}\}$
2  sort $P''$ in the increasing order of the cost
3  for $i = 1, \ldots, |P'|$ do
4      while $j \leq |P''|$ do
5          if $p'^{(i)} = p''^{(j)}$ then
6              break
7          else
8              $j \leftarrow j + 1$
9      if $j = |P''| + 1$ then
10         return $C_h^{ub} \leftarrow c(p'^{(i)})$
11 return $C_h^{ub} \leftarrow +\infty$
```

the condition holds since this pruning condition could be applied to more CSP queries with different C.

We directly compute the two sets $P' = \overline{P_{v_{end}h}}$ and $P'' = \{p_1 \oplus p_2 : p_1 \in \overline{P_{v_{end}u}}, p_2 \in \overline{P_{uh}}\}$ first. Suppose that the two sets have been sorted in the increasing order of the cost, and let $p'^{(i)}$ and $p''^{(j)}$ denote the path with the i-th smallest cost in P' and the path with the j-th smallest cost in P'', respectively. We then have to check if each $p'^{(i)}$ is in P'' until some i and determine C_h^{ub}.

Algorithm 5 illustrates the main procedure. We initialize $C_h^{ub} = 0$, $j = 1$ as a pointer for the path in P'', and the two sets P' and P''. We have to sort P'' in line 2 because it was just obtained by the path concatenation in line 1. Then, for each $p'^{(i)}$, we have to check if there exists $p''^{(j)}$ equal to it in lines 3-10. We will check $p''^{(j)}$ by increasing j in lines 4-8. If $p'^{(i)} = p''^{(j)}$, we know that $p'^{(i)}$ is in P'' and break the loop in lines 5-6. Otherwise, we just increment j by 1 in lines 7-8. If there is no such j, in line 9, we set $C_h^{ub} = c(p'^{(i)})$ in line 10 since we have checked that previous paths are all in P''. When $i > |P'|$ and $P' \subseteq P''$, we can set $C_h^{ub} = +\infty$, indicating that any C can make the condition hold by Lemma 2.

Its time complexity is $O(|P''| \log |P''| + \max(|P'|, |P''|))$, where the first term is from line 2 and the second one is because either i is increased to P' or j is increased to P''.

Example 16. *Consider $v_{end} = v_8$, $h = v_{13}$, and $u = v_{10}$. $P' = \overline{P_{sh}} = \{(12, 11), (11, 12), (10, 14)\}$, $\overline{P_{su}} = \{(9, 8), (8, 9)\}$, and $\overline{P_{uh}} = \{(3, 3)\}$. $C_{v_{13}}^{ub} = 0$ at first. $P'' = \{(12, 11), (11, 12)\}$. For $i = 1$ and $i = 2$, we can easily find $p'^{(1)} = p''^{(1)}$ and $p'^{(2)} = p''^{(2)}$, respectively, and j is also increased to 2. When $i = 3$ and $p'^{(3)} \neq p''^{(2)}$, j is increased to $3 = |P''| + 1$. We can then return $C_{v_{13}}^{ub} = c(p'^{(3)}) = 14$.*

4.1.2 *Finding C_h^{ub} for $h \in H$.* A straightforward idea is to apply Algorithm 5 on all the combinations of $u, h \in H$ where $u \neq h$. If h can be pruned by two or more vertices of u under different values of θ, we just use the one which gives us a larger C_h^{ub} since it could make more CSP queries fit the pruning condition. However, it is time-consuming to call Algorithm 5 $O(|H|^2)$ times and using different u to prune the same h only makes C_h^{ub} slightly larger. Therefore, for each h, we will set its C_h^{ub} by trying to use only one u. Besides, the following lemma can help us to find those u that can be used to pruned each h.

Algorithm 6: Finding C_h^{ub} for all $h \in H$

```
input  : The separator $H$ and the end vertex $v_{end}$
output : The upper bounds $C_h^{ub}$ for all $h \in H$
1  for $i \leftarrow 2, 3, \ldots |H|$ do
2      $j \leftarrow$ randomly chosen from $\{1, \ldots, i-1\}$
3      Applying Algorithm 5 on $\overline{P_{v_{end}h^{(i)}}}$, $\overline{P_{v_{end}h^{(j)}}}$, and
       $\overline{P_{h^{(i)}h^{(j)}}}$ to update $C_{h^{(i)}}^{ub}$
4  return $C_h^{ub}$ for all $h \in H$
```

Lemma 6. *If h is pruned by u for v_{end}, $c(p_{v_{end}h}^{(1)}) > c(p_{v_{end}u}^{(1)})$.*

Proof. There exist $p_1 \in \overline{P_{v_{end}u}}$ and $p_2 \in \overline{P_{uh}}$ such that $p_{v_{end}h}^{(1)} = p_1 \oplus p_2$. Since $c(p_1) \geq c(p_{v_{end}u}^{(1)})$, $c(p_{v_{end}h}^{(1)}) > c(p_1) \geq c(p_{v_{end}u}^{(1)})$. □

We can then sort $\overline{P_{v_{end}h}}$ for all h in the increasing order of $c(p_{v_{end}h}^{(1)})$. Let $h^{(i)}$ be the hoplink whose $c(p_{v_{end}h^{(i)}}^{(1)})$ is the i-th smallest one for all $h^{(i)}$. For $h^{(i)}$, we only need to consider one j such that $1 \leq j < i$ as u by the above lemma.

Algorithm 6 illustrates the main procedure. In line 1, we do not consider $h^{(1)}$ because no vertex can prune it. In line 2, we consider a random $j \in \{1, \ldots, i-1\}$ as u. In line 3, we use Algorithm 5 to check if $h^{(i)}$ could be pruned by $h^{(j)}$ under some value for v_{end}, If so, we update the corresponding $C_{h^{(i)}}^{ub}$.

Algorithm 6's time complexity is $O(|H|(\max(|P''| \log |P''|, |P'|)))$, where the second part is from Algorithm 5. The space complexity is simply $O(|H|)$ for each separator H and v_{end}.

Example 17. *For the separator $H = \{v_{10}, v_{13}\}$ and $v_{end} = v_8$, we first sort them by using the two smallest costs in $\overline{P_{v_8v_{10}}}$ and $\overline{P_{v_8v_{13}}}$ and get $h^{(1)} = v_{10}$ and $h^{(2)} = v_{13}$. We will then use the procedure in the previous example to update $C_{v_{13}}^{ub}$ as 14.*

4.2 Finding Separators and End Vertices

It is time-consuming to build pruning conditions for all the combinations of separators and end vertices. First, the number of separators is large because there are many branching nodes (*i.e.*, the LCA nodes) in the tree with many child nodes as our initial separators. Second, for each of these child nodes as a separator, the number of possible end vertices (which are descendants of the child node) is also large. Therefore, we do not build pruning conditions for all the combinations. We will use a set of random CSP queries, denoted by Q_{index}, to help us find those combinations of separators and end vertices that are frequently visited in the tree and give us more chance of pruning separators and time cost reduction. It can be generated by uniformly sampling from past workloads. For each $q \in Q_{\text{index}}$, we simply try to build pruning conditions for the four combinations where the separator could be $H(s)$ and $H(t)$ and the

Table 1: Real dataset description

Name	Region	$\lvert V\rvert$	$\lvert E\rvert$	$d_{max} \approx$
NY	New York City	264,346	733,846	154 km
BAY	San Francisco Bay Area	321,270	800,172	320 km
COL	Colorado	435,666	1,057,066	832 km

end vertex could be s or t. Note that the space complexity of each combination of the separator and the end vertex is only $O(|H|)$, which is negligible compared to the size of the labels.

One speedup technique for building pruning conditions is that if we have established that for a separator H, $h \in H$ is pruned by $u \in H$ and $u \neq h$ under C_h^{ub} for v_{end}, this relationship between h and u also holds for any other separator as long as both h and u are all in it. This is because the second condition in Theorem 1 holds, and we only need to ensure that u is available in the separator when it prunes h to make it unavailable. Therefore, we can maintain such relationships which help us save the number of calls for Algorithm 5 in line 3 of Algorithm 6.

4.3 Other Forms of Pruning Conditions

Each CSP query specifies its s, t, and C which could be used for acceleration. Our proposed pruning conditions are based on the combinations of (s, C) and (t, C). Only considering s (or t) means that the pruning conditions should be applicable for a given s and any t and C (or vice versa). It is actually a special case of our form since it requires that $\overline{P_{sh}} \subseteq \{p_1 \oplus p_2 : p_1 \in \overline{P_{su}}, p_2 \in \overline{P_{uh}}\}$ without the constraint of θ (or $\theta = +\infty$) in Theorem 1. Only considering C means that we prune any hoplink h such that any path passing through h has its cost larger than C. This pruning idea is reflected in Algorithm 4 which uses the power of C. For the two more complicated combinations (s, t) and (s, t, C), the pruning condition is required to match both s and t, which is a small probability event with its probability around $1/|V|^2$. Though one may store some information for frequently pairs of (s, t), it is impractical when we handle any possible CSP queries.

5 EXPERIMENTS

5.1 Experimental Setup

In our experiments, all algorithms were implemented in C++ and compiled by the GNU C++ compiler with the O3 optimization. The programs were performed in a machine with two Intel Xeon Gold 5220R 2.2GHz processors and 512GB RAM installed with CentOS 7 Linux distribution. The implementations are available through an anonymous link[1].

Datasets. Following existing work [22], we used three publicly available road networks from DIMACS [2], including NY, BAY, and COL. Their details could be found in Table 1. The last column of Table 1 shows the diameter of the network, defined as the maximum shortest distance of any s-t paths and denoted by d_{max}. It was used to generate the query sets explained below. Since DIMACS provided both the travel time and the distance of each edge, we used the travel time as the weight w and the distance as the cost c. Following existing work on path queries [22, 26], we generated 10 query sets with each size of 1000 by varying two factors: the shortest distance, denoted by d, and the cost budget C. Specifically, for the former one, each query set Q_i, where $i = 1, 2, 3, 4, 5$, consists of random queries with their shortest distances lie in $[d_{max}/2^{6-i}, d_{max}/2^{5-i}]$. Each query in Q_i uses $C = 0.5C_{max} + 0.5C_{min}$, where $C_{max} = d_{max}/2^{5-i}$ and C_{min} is its shortest distance d (since there is no CSP answer for $C < d$). For the latter one, each query set R_i, where $i = 1, 2, 3, 4, 5$, consists of the same queries as in Q_3 with each $C = rC_{max} + (1 - r)C_{min}$, where $r = (2i - 1) \times 0.1$, $C_{max} = d_{max}/4$ and C_{min} is the shortest distance d. The ratio r is used to vary C.

Compared algorithms. Since we focused on query efficiency, we compared our QHL with the fastest known algorithm, CSP-2Hop. We did not compare the other CSP solutions because they are slower than CSP-2Hop by orders of magnitude as shown in [22]. For the default settings, we used a set Q_{index} of 50,000 random queries to build the pruning conditions, which is sufficient to produce desirable results. For the number of random queries, we will explore its effect in Section 5.2.2.

5.2 Experiment Results

5.2.1 *Query Performance.*

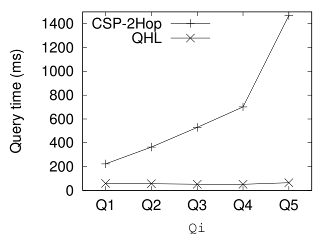
Figure 6 shows the query times of processing 1000 queries with different Q and r on three networks. The results of varying Q and r are shown in the first and second columns of Figure 6, respectively.

Query time of varying Q**.** For any of the three figures in the first column, it can be observed that when we vary the query set Q_i by increasing i, the query times of the algorithms on all datasets become larger, though the increases for QHL are hard to see. Recall that the average shortest distance of the queries in Q_i increases with i. Since a long distance between s and t indicates that there are many path choices between s and t, the size of the skyline path set between s and t also increases quickly. It further means that the algorithms have to deal with more skyline paths and hence incur more time cost.

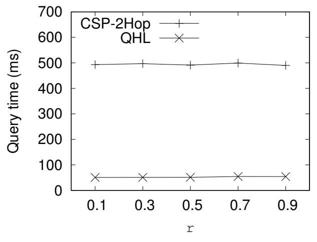
For the two algorithms, QHL always runs faster than CSP-2Hop in any datasets. In Figure 6e, QHL could reduce the query time of CSP-2Hop by two orders of magnitude in COL's network. The difference is significant on Q_5 since the long distance means more skyline paths as explained above, and CSP-2Hop utilizes less query information and handles the large skyline path sets directly. When $|V|$ and $|E|$ get much larger for COL, CSP-2Hop is not scalable to such large data and runs inefficiently with its nonlinear increase of time costs. However, our QHL still takes very short query time and runs fast on large data. This is because QHL uses the query information to prune many unnecessary hoplinks and paths.

For different road networks, it can be seen that the query times are similar on NY and BAY for the same Q_i and algorithm, though both $|V|$ and $|E|$ of BAY are slightly larger than those of NY. It is worth noting that the network of NY has its dense grid-like structure where there are many paths between any s and t, whereas the network of BAY circles around some bays, suggesting that the number of skyline paths between some s and t on Bay's network may not be as many as the one in NY's dense network. For COL, its network could be even denser around Denver with more edges.

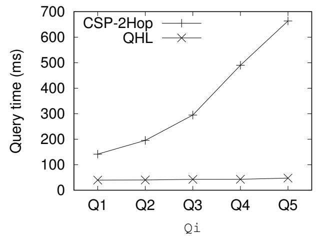
[1]https://anonymous.4open.science/r/QHL4CSP-51B8
[2]http://www.dis.uniroma1.it/challenge9/download.shtml



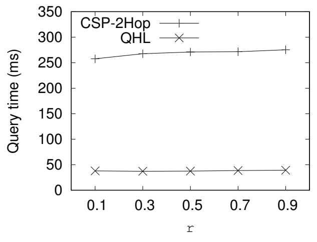
(a) Query time of varying Q (NY)



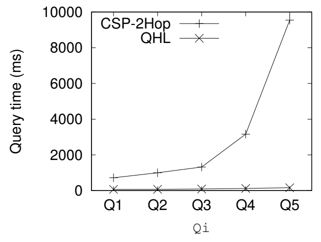
(b) Query time of varying r (NY)



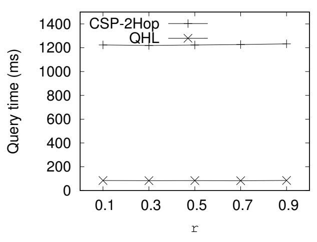
(c) Query time of varying Q (BAY)



(d) Query time of varying r (BAY)

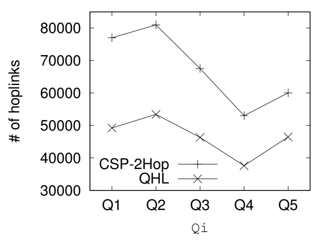


(e) Query time of varying Q (COL)

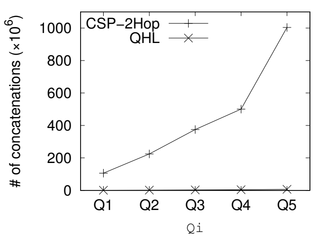


(f) Query time of varying r (COL)

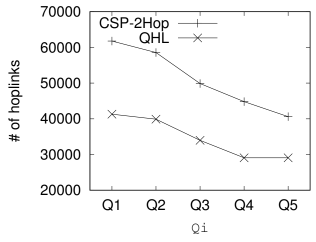
Figure 6: Query time of varying the query set Q and the ratio r for setting C on NY, BAY, and COL datasets



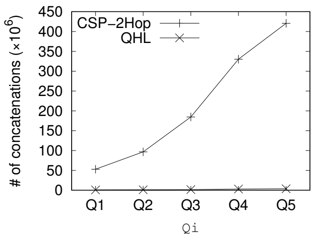
(a) # of hoplinks (NY)



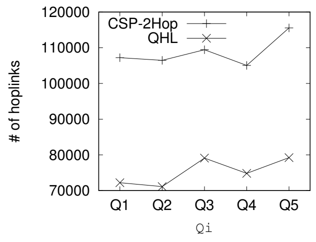
(b) # of path concatenations (NY)



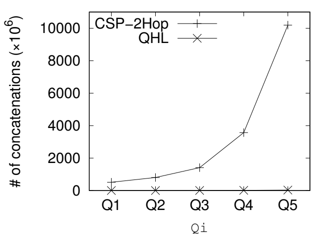
(c) # of hoplinks (BAY)



(d) # of path concatenations (BAY)



(e) # of hoplinks (COL)



(f) # of path concatenations (COL)

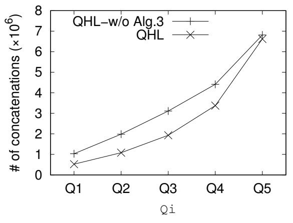
Figure 7: The numbers of hoplinks and path concatenations of varying the set Q on NY, BAY and COL datasets

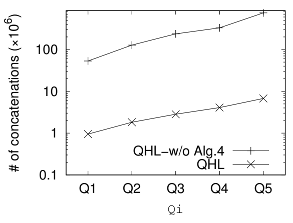
With larger skyline path sets, CSP-2Hop needs to perform a huge number of path concatenations to find all skyline paths.

Query time of varying r. In the second column of Figure 6, we study the query time of varying r. Recall that r is used to vary the cost budget C, and a large r means a large C since $C = rC_{max} + (1 - r)C_{min}$. For all three figures, the two algorithms are all insensitive to the change of r. The main reason is that the two algorithms all use the idea of path concatenation in the last step, though the procedures are different. For CSP-2Hop, it essentially computes the skyline path set of s and t and uses C to prune this set. The procedure of computing the skyline path set is independent of C. For QHL, in Algorithm 4, the two pointers still need to go through all the elements in the two path sets in the worst case. Besides, the slight increase of C (determined by C_{max}) could only allow few pruning conditions satisfy the requirements, which slightly increase the number of the remaining hoplinks and affect the query processing time less.

For the two algorithms, similarly, we can still find that QHL outperforms CSP-2Hop with a significant improvement on the query efficiency, which is mainly due to its use of the pruning power of the query information. It can be also found that the query processing times on the largest network of COL are greater than those on the other two networks, and those on the networks of NY and BAY are similar.

Number of hoplinks. The first column of Figure 7 presents the numbers of hoplinks (or $|Hoplinks|$) of varying Q. It can be found that QHL always uses fewer hoplinks than CSP-2Hop because QHL uses the pruning conditions. From the three figures, we can derive that $|Hoplinks|$ are independent of the distance owing to the properties of the tree decomposition. Specifically, $|Hoplinks|$ are bounded by the treewidth, denoted by $tw = \max_v |X(v)|$. The treewidth is determined by the tree decomposition algorithm [26] which only uses V and E but not w and c. Therefore, $|Hoplinks|$ depends only on the s and t of queries in Q_i but not their shortest distances. Besides, nearly all treewidths of road networks are smaller than 1000, also demonstrated in Table 2. Then, we can know that for each Q_i, the total number of hoplinks should be smaller than $tw \cdot |Q_i|$. For example, it is smaller than 148000 for NY since the treewidth is 148 and each $|Q_i| = 1000$. However, we should notice that the skyline path set sizes (*e.g.*, $|\overline{P_{sh}}|$ and $|\overline{P_{ht}}|$) could be large for large networks and queries of long distances. Pruning the hoplinks is still helpful since we could avoid some large skyline path sets associated with the hoplinks. Besides, since we estimate the time cost by the function $T(H)$ and use it to choose the separator with





(a) # of concatenations w/o Alg. 3 (b) # of concatenations w/o Alg. 4

Figure 8: Ablation study on NY

the minimum $T(H)$, we can correctly identify those hoplinks with large skyline path sets and avoid using them.

Number of path concatenations. The second column of Figure 7 gives the numbers of path concatenations of varying Q. It correlates with the query time since the path concatenation is the dominant term in QHL's time complexity $O(|Hoplinks|(|\overline{P_{sh}}| + |\overline{P_{ht}}|))$. By the discussion above, we know that $|Hoplinks|$ is bounded and the two skyline path sets could be large. Hence, for the three figures, we can observe that all lines have similar trends as their counterparts in the first column of Figure 6 about query processing times. Moreover, we can obtain similar conclusions that QHL is superior to CSP-2Hop in terms of query efficiency, and for Figure 7f, CSP-2Hop could have a nonlinear increase in the number of path concatenations because the distances of queries become longer and the network is so dense that the skyline path sets become very large.

Ablation study. We tested the module effects of QHL by considering two variants. The first one, called "QHL-w/o Alg. 3", is the algorithm that uses no pruning conditions (or equivalently, all $C_h^{ub} = 0$). It then directly chooses the one between $H(s)$ and $H(t)$ with smaller estimated cost $T(H)$. The second one, called "QHL-w/o Alg. 4", uses the Cartesian product of the two skyline path sets as in CSP-2Hop, instead of Algorithm 4. We compare the number of path concatenations because the differences between the variants and QHL are more obvious. The results are shown in Figure 8.

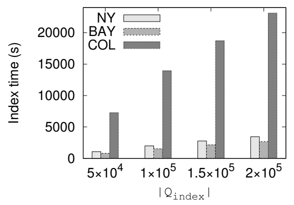
Figure 8a reports the number of path concatenations without Algorithm 3. It can be seen that the number of path concatenations is larger. Algorithm 3 could reduce the number of path concatenations by 50% for $Q1$ and Q_2. When i grows, the difference between the variant and QHL gets smaller because some C_h^{ub} are no longer larger than C (which increases with i since C_{min} is at least the shortest distance) and cannot prune the hoplinks. This coincides with the intuition that we have to check more paths for queries of long distances. Figure 8b gives the results without Algorithm 4. We can found that the variant has dramatic increases on the numbers. The differences are very large because the time complexity becomes $O(|Hoplinks||\overline{P_{sh}}||\overline{P_{ht}}|)$ with one more multiplier. In summary, the differences for Algorithm 3 and Algorithm 4 all demonstrate the effectiveness of the proposed techniques.

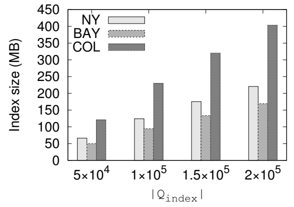
5.2.2 *Index Cost.*

We consider the time and space consumption during the index construction. Since our index consists of two parts, including the tree index with the labels as in CSP-2Hop and the pruning conditions, we will discuss them separately. Note that our additional space consumption is from only the pruning conditions.

Table 2: Tree and label index costs

Name	tw	th	Avg. th	Tree time	Label time	Label size
NY	148	330	269	120s	1533s	26.7GB
BAY	100	238	193	41s	706s	22.6GB
COL	143	423	276	756s	5419s	149GB





(a) Index time of varying $|Q_{\text{index}}|$ (b) Index size of varying $|Q_{\text{index}}|$

Figure 9: Index cost of varying Q for pruning conditions

Tree index cost. There are two steps. The first one is to generate a tree decomposition based on the network structure $G = (V, E)$, and the second one is to assign labels $L(v)$ for each v, which requires finding the skyline path sets $(u, \overline{P_{vu}})$ for all $X(v)$'s ancestors $X(u)$ [22]. We show the statistics of the tree and label index costs for three networks in Table 2. The first and second columns give the treewidths and treeheights, denoted by tw and th, respectively. Each label size $|L(v)|$ is bounded by th since each v can only have th ancestors. The third column shows the average treeheights over all tree nodes which reflects the number of stored large skyline path sets. The fourth and fifth columns give the time cost in seconds to build the tree and generate the labels, respectively. The last column lists the label sizes in terms of gigabytes. It is proportional to the sum of the sizes of all skyline path sets.

It can be seen that NY and BAY have similar statistics, and COL has its tree decomposition with its moderate treewidth and treeheight but needs large index time and space. There are two reasons. The first one is that the network is dense with so many edges concentrated in the center of Denver, as stated before. The second one is that there are many unimplemented tricks that can reduce the index size and time greatly since the building process is similar to building the contraction hierarchy [10]. This is out of the scope of this paper because we focus on improving query efficiency and there is a line of research studying how to reduce the index size [2, 25]. For a simple heuristic idea to reduce the index time and space, one may divide the large network into small networks and apply QHL for these small ones and combine the results.

Pruning conditions. Figure 9 depicts the time and space consumption of constructing the pruning conditions when we use the random query sets Q_{index} with different sizes. For the index time, in Figure 9a, it can be seen that the time increases linearly with $|Q_{\text{index}}|$ on all the three networks. For the fixed $|Q_{\text{index}}|$, the index times of the three networks are actually proportional to the label sizes of the three tree indexes. This is because the label construction covers similar work to Algorithm 6 that builds pruning conditions. Recall that the time complexity is $O(|H||\overline{P_{v_{end}h}}|)$. Each $X(h)$ for $h \in H$ is $X(v_{end})$'s ancestor which is also processed in the label construction to get $\overline{P_{v_{end}h}}$. Hence, building a small tree index will definitely decrease the time cost of building pruning conditions. However, reducing the index size is out of the scope of this paper. Besides, parallelization is also feasible since each pruning condition is independent of others.

For the index size, in Figure 9b, it can be observed that the space cost grows linearly with $|Q_{\text{index}}|$ on all the three networks because the space complexity for each Q_{index} is $O(|Q_{\text{index}}||H|)$. It can be smaller than 200 MB on NY and BAY, and 450 MB on COL, all within 1% of the label index size.

It is worth noting that more random queries and pruning conditions can surely improve the query performance. However, there is a trade-off between processing more random queries and gaining more acceleration. Besides, we also found that most random queries basically visit less than 10 frequent separators, which is determined by the tree structure. It further means that there is a bottleneck where more queries produce no improvement, just after nearly all $v \in V$ build pruning conditions on the frequent separators.

5.3 Summary

(i) Our proposed *QHL* that uses the pruning power of the query information runs faster than the best-known solution for the exact CSP by orders of magnitude with 1% more index space.

(ii) Our QHL is scalable to large networks in terms of time and space efficiencies. It runs within 50 μs for one single CSP query on standard networks of cities. Its additional index has negligible space consumption compared with the whole index.

(iii) The proposed pruning techniques are all useful in accelerating the query processing.

6 RELATED WORK

In this section, we review related work for two categories, including the 2-hop labeling and CSP queries.

6.1 2-Hop Labeling

2-Hop labeling refers to the approach of using a set of hoplinks to concatenate two paths (or "hops") with their related information stored in the labels. The seminal work directly finds a set of paths, also called a 2-hop cover, such that any path with some properties (such as shortest paths) can be covered by the concatenation of two paths (or hops) in the 2-hop cover [8]. However, the size of the 2-hop cover can be very large and hence inefficient. To control the size of the hoplinks and also the label size, [26] proposed H2H which uses the tree decomposition to efficiently answer shortest path distance queries. Later work considered its further optimization [6, 20, 21] or dynamic maintenance [39–41]. It was also studied that 2-hop labeling could be used to answer different types of queries, such as reachability [7, 16, 42], shortest path counting [29], keyword search [32], or constrained shortest path search [22]. However, except for the last one, they consider different problems. Our solution is specific to the CSP query and hence tackles different challenges.

6.2 Constrained Shortest Path

6.2.1 Approximate CSP. Since CSP is known as an NP-Hard problem, early work studied the optimization on the full polynomial approximation schemes (FPAS) which compute the path with its cost no greater than $(1+\epsilon)$ times the cost of the optimal path and with its running time polynomial in $O(1/\epsilon)$ and input size [15, 23, 36]. Their solutions were based on the technique of rounding costs in order to better check the feasibility. One later solution CP-CSP considered the extension of Dijkstra's idea which allows each vertex is visited multiple times and prunes some paths $\sqrt[|V|]{1+\epsilon}$-dominated by others (where α-dominance extends the definition of path domination) [34]. However, they could still run slower than some exact methods [19], and they are all index-free algorithms. The recent index-based solution, COLA [35], partitions the graph into subgraphs and indexes some selected paths between boundary vertices for combining on-the-fly results from subgraphs. However, it builds no index when searching the subgraphs and gives approximate answers. Besides, some approximate algorithms were also designed for a different problem about the CSP on the time-dependent graph where costs and weights follow time-dependent functions [38].

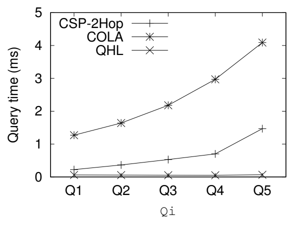
6.2.2 Exact CSP. The first work used dynamic programming to solve the exact CSP [17]. The problem is then formulated by integer linear programming and solved by Lagrangian relaxation to get rid of some constraints [13]. Linear relaxation is also used to get upper and lower bounds of the weights to prune the search space [24]. A more efficient solution that extends Dijkstra's idea was also proposed [14]. It searches the path incrementally while allowing vertices to be visited many times and pruning paths that are dominated by others. Another solution considered using the k shortest paths (in terms of costs) to incrementally find the final answer [31]. However, they are all incapable of handling large graphs because they do not build any index. Since computing skyline paths can be also used to solve CSP queries, some work adapted the indexes of answering shortest distances to find skyline paths [9, 22, 33]. Among them, it was shown that CSP-2Hop runs faster than the others by several orders of magnitude [22]. To reduce the index size and construction time, it further proposed the *forest labeling* which basically partitions the graph and also the tree decomposition, but it sacrifices the query efficiency. In all, our proposed QHL outperforms CSP-2Hop in terms of query efficiency while incurring similar space costs.

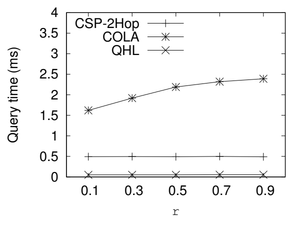
7 CONCLUSION

This paper considers the problem of the exact constrained shortest path search on road networks. The state-of-the-art algorithm, called CSP-2Hop, beats previous ones with a dramatic improvement on query efficiency. Its success is mainly due to the use of the tree hierarchy and few table lookups. However, it overlooks the pruning power of the CSP query information and hence the chance of further acceleration. We propose by far the fastest algorithm, called QHL, which builds pruning conditions based on the query information and concatenate paths in an economic way. Our QHL runs faster than CSP-2Hop by orders or magnitude with a similar space consumption. It uses much fewer table lookups than CSP-2Hop and has one fewer multiplier in its time complexity. The experimental results demonstrate the superiority of the proposed QHL in query efficiency. For future work, one may consider similar constrained queries, such as those with multiple objectives or constraints. The other interesting topic is about reducing the index cost in terms of both time and space.

REFERENCES

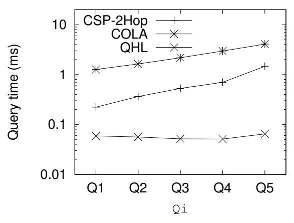
[1] 2022. Google Maps. https://www.google.com/maps.
[2] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. 2010. Preprocessing Speed-Up Techniques Is Hard. In *CIAC*.
[3] Michael A. Bender and Martin Farach-Colton. 2000. The LCA Problem Revisited. In *LATIN*, Gaston H. Gonnet, Daniel Panario, and Alfredo Viola (Eds.).
[4] Hans L. Bodlaender. 1993. A Tourist Guide through Treewidth. *Acta Cybern.* 11, 1-2 (1993), 1–21.
[5] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Hong Cheng, and Miao Qiao. 2012. The exact distance to destination in undirected world. *VLDB J.* 21, 6 (2012), 869–888.
[6] 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*.
[7] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. 2006. Fast Computation of Reachability Labeling for Large Graphs. In *EDBT*.
[8] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2002. Reachability and distance queries via 2-hop labels. In *SODA*.
[9] Daniel Delling and Dorothea Wagner. 2009. Pareto Paths with SHARC. In *SEA*.
[10] 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.
[11] Qixu Gong and Huiping Cao. 2022. Backbone Index to Support Skyline Path Queries over Multi-cost Road Networks. In *EDBT*.
[12] Qixu Gong, Huiping Cao, and Parth Nagarkar. 2019. Skyline Queries Constrained by Multi-cost Transportation Networks. In *ICDE*.
[13] Gabriel Y. Handler and Israel Zang. 1980. A dual algorithm for the constrained shortest path problem. *Networks* 10, 4 (1980), 293–309.
[14] Pierre Hansen. 1980. Bicriterion path problems. In *Multiple criteria decision making theory and application*. Springer, 109–127.
[15] Refael Hassin. 1992. Approximation Schemes for the Restricted Shortest Path Problem. *Math. Oper. Res.* 17, 1 (1992), 36–42.
[16] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-HOP: a high-compression indexing scheme for reachability query. In *SIGMOD*.
[17] H.C Joksch. 1966. The shortest route problem with constraints. *J. Math. Anal. Appl.* 14, 2 (1966), 191–197.
[18] Hans-Peter Kriegel, Matthias Renz, and Matthias Schubert. 2010. Route skyline queries: A multi-preference path planning approach. In *ICDE*.
[19] Fernando A. Kuipers, Ariel Orda, Danny Raz, and Piet Van Mieghem. 2006. A Comparison of Exact and *epsilon*-Approximation Algorithms for Constrained Routing. In *NETWORKING*.
[20] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling Distance Labeling on Small-World Networks. In *SIGMOD*.
[21] 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*.
[22] 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*.
[23] Dean H. Lorenz and Danny Raz. 2001. A simple efficient approximation scheme for the restricted shortest path problem. *Oper. Res. Lett.* 28, 5 (2001), 213–219.
[24] Kurt Mehlhorn and Mark Ziegelmann. 2000. Resource Constrained Shortest Paths. In *ESA*.
[25] Nikola Milosavljevic. 2012. On optimal preprocessing for contraction hierarchies. In *CTS-SIGSPATIAL*, Stephan Winter and Matthias Müller-Hannemann (Eds.).
[26] 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*.
[27] 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* (2020).
[28] Dian Ouyang, Long Yuan, Fan Zhang, Lu Qin, and Xuemin Lin. 2018. Towards Efficient Path Skyline Computation in Bicriteria Networks. In *DASFAA*.
[29] Yu-Xuan Qiu, Dong Wen, Lu Qin, Wentao Li, Ronghua Li, and Ying Zhang. 2022. Efficient Shortest Path Counting on Large Road Networks. *PVLDB* (2022).
[30] Neil Robertson and Paul D. Seymour. 1984. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B* 36, 1 (1984), 49–64.
[31] Antonio Sedeño-Noda and Sergio Alonso-Rodríguez. 2015. An enhanced K-SP algorithm with pruning strategies to solve the constrained shortest path problem. *Appl. Math. Comput.* 265 (2015), 602–618.
[32] Yuxuan Shi, Gong Cheng, and Evgeny Kharlamov. 2020. Keyword Search over Knowledge Graphs via Static and Dynamic Hub Labelings. In *WWW*.
[33] Sabine Storandt. 2012. Route Planning for Bicycles - Exact Constrained Shortest Paths Made Practical via Contraction Hierarchy. In *ICAPS*.
[34] George Tsaggouris and Christos D. Zaroliagis. 2009. Multiobjective Optimization: Improved FPTAS for Shortest Paths and Non-Linear Objectives with Applications. *Theory Comput. Syst.* 45, 1 (2009), 162–186.
[35] Sibo Wang, Xiaokui Xiao, Yin Yang, and Wenqing Lin. 2016. Effective Indexing for Approximate Constrained Shortest Path Queries on Large Road Networks. *PVLDB* (2016).
[36] Arthur Warburton. 1987. Approximation of Pareto Optima in Multiple-Objective, Shortest-Path Problems. *Oper. Res.* 35, 1 (1987), 70–79.
[37] 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*.
[38] Ye Yuan, Xiang Lian, Guoren Wang, Yuliang Ma, and Yishu Wang. 2019. Constrained Shortest Path Query in a Large Time-Dependent Graph. *PVLDB* (2019).
[39] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. 2021. Dynamic Hub Labeling for Road Networks. In *ICDE*.
[40] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2021. Efficient 2-Hop Labeling Maintenance in Dynamic Small-World Networks. In *ICDE*.
[41] Yikai Zhang and Jeffrey Xu Yu. 2022. Relative Subboundedness of Contraction Hierarchy and Hierarchical 2-Hop Index in Dynamic Road Networks. In *SIGMOD*.
[42] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *SIGMOD*.

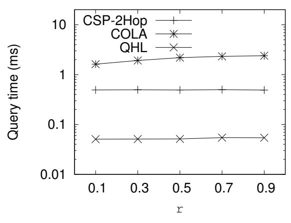




(a) Query time of varying Q (NY) (b) Query time of varying r (NY)

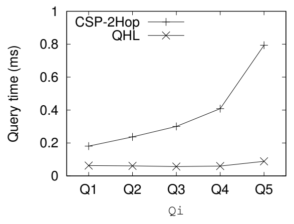
Figure 10: Query time on NY datasets

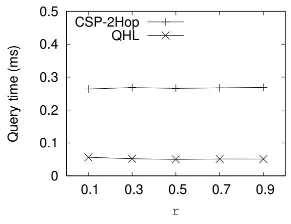




(a) Query time of varying Q (NY) (b) Query time of varying r (NY)

Figure 11: Query time in the log scale





(a) Query time of varying Q (NY) (b) Query time of varying r (NY)

Figure 12: Effects of independent weights and costs

A APPENDIX

A.1 Comparison with COLA

We compared COLA [35] by using its implementation [3] and setting its approximation ratio as 1 to generate exact CSP answers. The results of query processing time are shown in Figure 10. The same results in the log scale are given in Figure 11. It can be seen that QHL is always the fastest one and can outperform COLA by two orders of magnitude on NY datasets. Other observations can be found in Section 5.2.1.

A.2 Correlation between Weight and Cost

We further explored the case where weights and costs have no correlation. We simulated the scenario where the number of traversed traffic signals and the travel distance are the weight and the cost, respectively. To assign edge weights, we first selected some vertices of high degrees as the positions of traffic signals and then set the edge weights as 1 if the edges are incident to traffic signal vertices (since using those edges will traverse the traffic signals) and 0 otherwise. There are 58986 traffic signals for NY data. The query times are all shown in Figure 12. We could obtain similar results to previous ones.

[3]https://sourceforge.net/projects/cola2016/