

# Effective Indexing for Approximate Constrained Shortest Path Queries on Large Road Networks

Sibo Wang<sup>1</sup> Xiaokui Xiao<sup>1</sup> Yin Yang<sup>2</sup> Wenqing Lin<sup>3</sup>

<sup>1</sup>Nanyang Technological University  
{wang0759, xkxiao}@ntu.edu.sg

<sup>2</sup>Hamad Bin Khalifa University  
yyang@qf.org.qa

<sup>3</sup>Institute for Infocomm Research  
linw@i2r.a-star.edu.sg

## ABSTRACT

In a *constrained shortest path (CSP)* query, each edge in the road network is associated with both a length and a cost. Given an origin  $s$ , a destination  $t$ , and a cost constraint  $\theta$ , the goal is to find the shortest path from  $s$  to  $t$  whose total cost does not exceed  $\theta$ . Because exact CSP is NP-hard, previous work mostly focuses on approximate solutions. Even so, existing methods are still prohibitively expensive for large road networks. Two main reasons are (i) that they fail to utilize the special properties of road networks and (ii) that most of them process queries without indices; the few existing indices consume large amounts of memory and yet have limited effectiveness in reducing query costs.

Motivated by this, we propose *COLA*, the first practical solution for approximate CSP processing on large road networks. *COLA* exploits the facts that a road network can be effectively partitioned, and that there exists a relatively small set of landmark vertices that commonly appear in CSP results. Accordingly, *COLA* indexes the vertices lying on partition boundaries, and applies an on-the-fly algorithm called  $\alpha$ -*Dijk* for path computation within a partition, which effectively prunes paths based on landmarks. Extensive experiments demonstrate that on continent-sized road networks, *COLA* answers an approximate CSP query in sub-second time, whereas existing methods take hours. Interestingly, even without an index, the  $\alpha$ -*Dijk* algorithm in *COLA* still outperforms previous solutions by more than an order of magnitude.

## 1. INTRODUCTION

Nowadays, route planning via online mapping/navigation services has become an essential part of driving in many places. Most popular online maps today, such as Google Maps [3], compute routes based on a single criterion, which is usually either the total route length or the total travel time. In practice, however, the user often needs to consider multiple criteria when planning a route. Besides travel distance and time, a common criterion is toll payment. For example, many cities charge the road user a fee to use highways (e.g., in Tokyo), bridges and undersea tunnels (e.g., in New York City and Hong Kong); additionally, some densely populated metropolitan areas impose congestion charges (e.g., in London and

Singapore). Another common consideration is safety. For instance, in 2015, members of the New York City Council requested that Google Maps reduce the number of left turns in its suggested routes, since left turns lead to a higher rate of pedestrian crashes<sup>1</sup>. Finally, the shortest path is not necessarily the fastest or the most pleasant, e.g., the user may rather prefer driving through a university campus than on the highway. Currently, most online navigation systems address the problem by returning multiple paths, allowing the user to manually modify a path, and providing multiple options on how the best route is determined. None of these solutions is ideal since they do not take into consideration multiple criteria simultaneously.

The *constrained shortest path (CSP)* [19,24] addresses this problem by finding the best path based on one criterion with a constraint on another criterion. For instance, the user may want to compute a CSP that minimizes total travel time within a budget for toll payment. In an online navigation system, the constraint can be presented to the user in the form of a slider bar, which drastically simplifies user-system interactions. We focus on single-constraint CSP, because (i) tuning multiple parameters burdens the user, e.g., many parameter combinations may lead to no feasible solution and (ii) processing single-constraint CSP efficiently is already very challenging; for existing solutions, a single query may take hours on a continent-sized road network. Hence, we focus on single-constraint CSP and leave multiple-constraint CSP as future work.

Specifically, in CSP, each edge is assigned two attributes, which are used in the optimization objective and constraint respectively. Without loss of generality, we assume that these two attributes are edge length and cost, respectively. Given an origin  $s$ , a destination  $t$ , and a cost constraint  $\theta$ , CSP finds the path from  $s$  to  $t$  that minimizes its total length, while satisfying that its total cost does not exceed  $\theta$ . Besides online navigation systems, CSP also finds applications in railroad management, military aircraft management systems, telecommunications, etc. [29]. The CSP problem has been proven to be NP-hard [13, 19]. Hence, the majority of existing work (e.g., [19, 24, 31]) focuses on approximate solutions, which guarantee that the resulting path length is no longer than  $\alpha$  times of the optimal path length (where  $\alpha$  is a user specified approximation ratio), subject to the cost constraint  $\theta$ . Although there exist polynomial-time algorithms for approximate CSP (e.g. [19,24,31]), as we show in the experiments, the current state-of-the-art solutions are still prohibitively expensive for large road networks. There are two main reasons for their inefficiency. First, they aim at answering approximate CSPs on general graphs, rather than specifically on road networks; consequently, they fail to utilize the latter's special properties. Second and more importantly, most of them process

<sup>1</sup><http://www.digitaltrends.com/cars/new-york-city-to-google-reduce-the-number-of-left-turns-in-maps-navigation-directions/>

queries without an index. The few known indices are all designed for exact CSPs, and they consume large amounts of memory; furthermore, none of them succeeds at reducing query cost to a practical level.

We thus propose a novel and practical solution *COLA* for index-based approximate CSP processing on large road networks. *COLA* mainly exploits two important properties of the road network. First, real road networks are often (roughly) planar, and, thus, can be effectively split into partitions, each of which contains only a relatively small number of boundary vertices. Accordingly, *COLA* partitions the network, builds an overlay graph on the partitions, and indexes a set of selected paths between pairs of boundary vertices. Second, in practice there often exist a relatively small number of landmark vertices [16] in the road network that commonly appear in CSP results. Based on this property, we design an index-free algorithm  $\alpha$ -Dijk as a component of *COLA* for path computation within a partition, which achieves effective pruning using a landmark set. Extensive experiments using real continent-sized road networks containing tens of millions of vertices show that *COLA* answers an approximate CSP query within a second, whereas previous solutions need several hours. Further, even when an index is not available (e.g., when the edge lengths or costs change frequently), the  $\alpha$ -Dijk module still outperforms existing methods by over an order of magnitude.

## 2. BACKGROUND

In this section, we first provide formal definitions of the problem and the terminology used in this paper in Section 2.1. Then, in Section 2.2, we present the state-of-the-art solutions for exact and approximate CSP queries, and point out their drawbacks that render them inefficient for answering CSP queries. Other existing methods are reviewed in Section 2.3.

### 2.1 Formal Definitions

Let  $G = (V, E)$  be a directed road network with a vertex set  $V$  and an edge set  $E$ . Each edge  $e \in E$  is associated with a *length*  $\ell(e) \geq 0$  and a *cost*  $c(e) \geq 0$ . For a path  $P = \langle e_1, e_2, \dots, e_k \rangle$  in  $G$ , the *length* and *cost* of  $P$  are defined as  $\ell(P) = \sum_{i=1}^k \ell(e_i)$  and  $c(P) = \sum_{i=1}^k c(e_i)$ , respectively. Following previous work [19, 24, 31], we assume that the length of each edge is an integer. In practice this can be done by measuring the edge length in a sufficiently small unit, e.g., foot or meter, if the edge length represents travel distance. Similarly, we assume that the cost of each edge is also an integer. We use  $\ell_{max}$  (resp.  $c_{max}$ ) to denote the maximum length (resp. cost) of an edge in  $G$ . Meanwhile, we assume that  $\ell_{max}$  (resp.  $c_{max}$ ) is non-zero; otherwise, the problem can be trivially regarded as a conventional shortest path problem.

Given an origin vertex  $s \in V$ , a destination vertex  $t \in V$ , and a cost constraint  $\theta$ , a *constrained shortest path (CSP)* query asks for the shortest path  $P$  among all paths from  $s$  to  $t$  with costs no more than  $\theta$ . If there exist multiple CSPs with the same length, we break ties by the cost of the paths. The CSP problem has been proven to be NP-hard, if both  $\ell_{max}$  and  $c_{max}$  can be arbitrarily large [13, 19]. On the other hand, if either  $\ell_{max}$  or  $c_{max}$  is polynomial to the number of vertices, there exist polynomial-time solutions for CSP, e.g., [19, 21]. Nevertheless, as we review in Sections 2.2 and 2.3, these algorithms incur tremendous costs for large graphs, and, thus, are far from practical. As such, recently much effort has been devoted to solving approximate versions of the CSP problem. This paper follows a popular definition called  $\alpha$ -CSP, defined as follows.

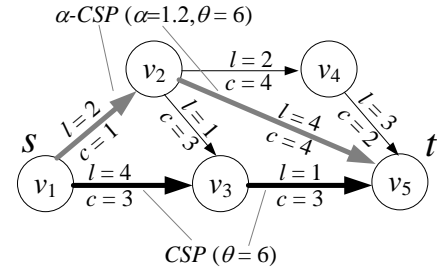
**DEFINITION 1 ( $\alpha$ -CSP QUERY).** *Given an origin  $s$ , a destination  $t$ , a cost constraint  $\theta$ , and an approximation ratio  $\alpha$ ,*

**Table 1: List of notations.**

Symbol	Meaning
$G = (V, E)$	Input graph
$n, m$	Numbers of vertices and edges in $G$
$\ell(e), c(e)$	Length and cost of an edge $e$
$\ell_{max}, c_{max}$	Maximum length and cost for any edge in $G$
$\alpha$	Approximation ratio in $\alpha$ -CSP
$s, t$	Query origin and destination vertices
$\mathcal{T}$	A partitioning of graph $G$
$G_s, G_t$	Subgraph in $\mathcal{T}$ containing $s$ and $t$ , respectively
$G^\circ = (V^\circ, E^\circ)$	Overlay graph of $G$ (refer to Section 3.1)

*an  $\alpha$ -CSP query returns a path  $P$ , such that  $c(P) \leq \theta$ , and  $\ell(P) \leq \alpha \cdot \ell(P_{opt})$ , where  $P_{opt}$  is the optimal answer to the exact CSP query with origin  $s$ , destination  $t$ , and cost constraint  $\theta$ .*  $\square$

**EXAMPLE 1.** Figure 1 illustrates an example of exact- and  $\alpha$ -CSP on a graph with 5 vertices  $v_1, v_2, \dots, v_5$ . The length and cost for each edge are also shown in the figure. For example, the edge from  $v_1$  to  $v_2$  has cost  $c = 1$  and length  $\ell = 2$ . Given origin  $s = v_1$ , destination  $t = v_5$ , and cost constraint  $\theta = 6$ , the CSP query returns the path  $P_{opt} = \langle (v_1, v_3), (v_3, v_5) \rangle$ , since (i)  $c(P_{opt}) = 6 \leq \theta$  and (ii) the length  $\ell(P_{opt}) = 5$  is the smallest among all paths from  $v_1$  to  $v_5$  with a cost no more than  $\theta$ . Meanwhile, for  $\alpha = 1.2$ , a valid result for the  $\alpha$ -CSP query with the same parameters  $s = v_1, t = v_5$  and  $\theta = 6$  is  $P_\alpha = \langle (v_1, v_2), (v_2, v_5) \rangle$ , since  $c(P_\alpha) = 5 \leq \theta$ , and  $\ell(P_\alpha) = 6 \leq \alpha \cdot \ell(P_{opt}) = 6$ .  $\square$



**Figure 1: Example of exact CSP and  $\alpha$ -CSP**

Two important concepts in solving  $\alpha$ -CSP are *dominance relationship* and *skyline*. We define dominance for  $\alpha$ -CSP as follows.

**DEFINITION 2 ( $\alpha$ -DOMINANCE).** Let  $P_1$  and  $P_2$  be two paths connecting the same origin and destination vertices.  $P_1$   $\alpha$ -dominates  $P_2$  iff  $c(P_1) \leq c(P_2)$  and  $\ell(P_1) \leq \alpha \cdot \ell(P_2)$ .  $\square$

For instance, consider paths  $P_1 = \langle (v_1, v_2), (v_2, v_5) \rangle$  and  $P_2 = \langle (v_1, v_3), (v_3, v_5) \rangle$  in the above example. When  $\alpha = 1.2$ ,  $P_1$   $\alpha$ -dominates  $P_2$ , since (i) the cost of  $P_1$  is  $c(P_1) = 5 \leq c(P_2) = 6$ , and (ii) the length of  $P_1$  is  $\ell(P_1) = 6 \leq \alpha \cdot \ell(P_2) = 1.2 \times 5 = 6$ .

Based on the above definition, a set of paths  $S$  is called a *skyline set*, iff no path in  $S$  is  $\alpha$ -dominated by another in the same set  $S$ . We say that a path  $P$  is a *skyline path* if  $P$  is in a skyline set. Note that if two paths  $P_1$  and  $P_2$  have the same cost, it is possible that they  $\alpha$ -dominate each other, in which case we put the path with smaller length in a skyline set. For exact CSP, we define dominance relationship and skyline in the same way, by simply fixing  $\alpha = 1$ . Table 1 summarizes common symbols throughout the paper.

### 2.2 State of the Art

We present the current state of the art for CSP processing. Additional literature review appears in Section 2.3.

**Exact CSP without index.** The state of the art index-free solution for exact CSP problem is the one proposed in [18], which we

call *Sky-Dijk* because it follows the general idea of Dijkstra’s algorithm [10]. The main difference between *Sky-Dijk* and Dijkstra’s algorithm is that the former incrementally maintains a set of paths at each vertex, rather than a single shortest path. Specifically, *Sky-Dijk* maintains a label set  $L(v)$  for each vertex  $v$ , which contains the current set of skyline paths from the origin  $s$  to  $v$ , i.e., those not dominated by another path in  $L(v)$ . Similar to Dijkstra’s algorithm, each  $L(v)$  is initialized to empty and updated iteratively.

Meanwhile, akin to Dijkstra’s algorithm, *Sky-Dijk* maintains a heap  $H$  of paths originating from  $s$ , in ascending order of their costs<sup>2</sup>. Initially,  $H$  contains only one trivial path with no edge, which both starts and ends at the origin vertex  $s$ . Then, in each iteration, *Sky-Dijk* pops the top path  $P$  from  $H$ . Let  $v$  be the last vertex in  $P$ . If  $v \neq t$ , i.e.,  $P$  has not reached the query destination, the algorithm enumerates each path  $P'$  that can be obtained by appending an edge  $(v, v')$  at the end of  $P$ , and checks whether  $P'$  exceeds the cost limit  $\theta$  or is dominated by any path in  $L(v')$ . If so,  $P'$  is simply discarded; otherwise, *Sky-Dijk* adds  $P'$  to both  $H$  and  $L(v')$ , and updates  $L(v')$  to eliminate paths dominated by  $P'$ . The algorithm terminates when  $H$  is empty, and returns the path in  $L(t)$  with the minimum length, where  $t$  is the destination vertex.

**EXAMPLE 2.** Consider again the example in Figure 1 with the same exact CSP query with origin  $s = v_1$ , destination  $t = v_5$ , and cost constraint  $\theta = 6$ . *Sky-Dijk* initializes the heap  $H$  with a trivial path  $P_0$  from  $v_1$  to  $v_1$  with no edge, and zero cost / length. Then, it pops  $P_0$  from  $H$ , and extends it to obtain paths  $P_1 = \langle (v_1, v_2) \rangle$  with cost  $c_1 = 1$  and length  $\ell_1 = 2$ , and  $P_2 = \langle (v_1, v_3) \rangle$  with  $c_2 = 3$  and  $\ell_2 = 4$ . The algorithm adds  $P_1$  and  $P_2$  to the label set  $L(v_2)$  and  $L(v_3)$  respectively, and both of them to  $H$ .

Next,  $P_1$  is popped from  $H$ , and *Sky-Dijk* extends it to obtain paths  $P_4 = \langle (v_1, v_2), (v_2, v_3) \rangle$ ,  $P_5 = \langle (v_1, v_2), (v_2, v_4) \rangle$  and  $P_6 = \langle (v_1, v_2), (v_2, v_5) \rangle$ , which are added to  $L(v_3)$ ,  $L(v_4)$  and  $L(v_5)$  respectively. Note that now  $L(v_3)$  contains two paths  $P_2$  ( $c_2 = 3, \ell_2 = 4$ ) and  $P_4$  ( $c_4 = 4, \ell_4 = 3$ ); neither dominates the other. Meanwhile, note that  $P_6$  does not need to be inserted to  $H$ , as it has reached the destination  $t = v_5$ , and, thus, cannot be extended further. After that,  $P_2$  is popped from  $H$ ; extending  $P_2$  generates  $P_7 = \langle (v_1, v_3), (v_3, v_5) \rangle$ , which is eventually returned as the CSP result. The algorithm terminates until  $H$  becomes empty, at which time it inspects  $L(v_5)$ , and returns  $P_7$ .  $\square$

The time complexity of *Sky-Dijk* is  $\mathcal{O}(\ell_{max}mn \cdot \log(\ell_{max}n))$ , where  $m$  (resp.  $n$ ) is the number of edges (resp. vertices) in the graph, and  $\ell_{max}$  is the maximum edge length [18]. Clearly, *Sky-Dijk* is a polynomial-time algorithm when  $\ell_{max}$  is polynomial to  $n$ . However, on real road networks, the performance of *Sky-Dijk* is very poor, since it does not optimize for such datasets at all. As shown in [30] and also in our experiments, *Sky-Dijk* incurs enormous costs for large graphs, and is clearly impractical.

**$\alpha$ -CSP without index.** The current state-of-the-art solution for  $\alpha$ -CSP is developed by Tsaggouris and Zaroliagis [31], dubbed as *CP-Dijk* in the following. Specifically, given an  $\alpha$ -CSP with origin  $s$ , destination  $t$ , cost constraint  $\theta$ , and approximation ratio  $\alpha$ , *CP-Dijk* applies the same data structures and follows the same steps as *Sky-Dijk*, with a single modification: that each label set  $L(v)$  maintains the set of paths that are not  $\sqrt[\alpha]{\alpha}$ -dominated (Definition 2) by another path in  $L(v)$ , where  $n$  is the total number of vertices in the input graph  $G$ . Because  $\sqrt[\alpha]{\alpha}$ -dominance is a relaxed condition of exact (i.e., 1-) dominance, this modification leads to faster query processing. However, for a large graph,  $\sqrt[\alpha]{\alpha}$  is very close to 1 even

<sup>2</sup>Note that, similar to the Dijkstra’s algorithm, it suffices to store in  $H$  only the length, cost and the last two vertices for each path. For ease of presentation we assume  $H$  contains full paths.

for a large  $\alpha$ . Consequently, the performance improvement of *CP-Dijk* over *Sky-Dijk* is often negligible, as shown in our experiments.

The time complexity of *CP-Dijk* is  $\mathcal{O}(\kappa mn \cdot \log(\kappa n))$ , where  $\kappa = \log(n \cdot \ell_{max}/\ell_{min})/(\alpha - 1)$  [31], and  $\ell_{max}, \ell_{min}$  are the maximum and minimum non-zero values of an edge length, respectively [31]. As explained before, in terms of practical performance, *CP-Dijk* obtains only marginal improvement over *Sky-Dijk*; nevertheless, *CP-Dijk* is at least no worse than *Sky-Dijk*. As discussed in Section 2.3, other polynomial-time solutions for  $\alpha$ -CSP can be far more costly. The fact that *CP-Dijk* is the state-of-the-art for  $\alpha$ -CSP processing reveals that previous research focuses mostly on asymptotic complexity, not practical performance.

**Exact CSP with index.** The state-of-the-art for indexed CSP processing is *CSP-CH* [30], which accelerates *Sky-Dijk* with contraction hierarchies [15], an indexing technique that has been shown to be effective for accelerating conventional shortest path processing on road networks [32]. Similar to [15], in each iteration *CSP-CH* removes a vertex from the graph, and substitutes it with new shortcut edges for the remaining vertices. Each shortcut  $(u, w)$  created during the removal of vertex  $v$  represents a path  $\langle (u, v), (v, w) \rangle$  that is not dominated by any other path from  $u$  to  $w$ . After that, *CSP-CH* answers query with a bidirectional *Sky-Dijk* search from both the origin  $s$  and the destination  $t$  simultaneously, utilizing the shortcuts to reduce the number of nodes to be traversed.

The problem of *CSP-CH* is that unlike conventional shortest path search, in CSP there can be numerous shortcuts (i.e., multiple skyline sets) for each removed vertex, leading to a prohibitively large index size. *CSP-CH* uses heuristics to alleviate this problem, e.g., by adding only a set of selected shortcuts, and by keeping the vertex in the graph instead of removing it. Such compromises, however, dramatically decrease the effectiveness of the index. Consequently, its query processing cost is still impractically high.

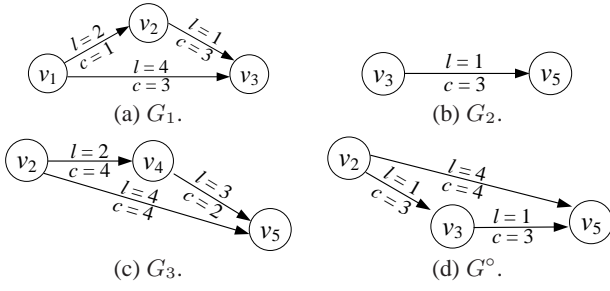
**$\alpha$ -CSP with index.** To our knowledge, we are not aware of any indexed solution for  $\alpha$ -CSP processing. In sum, none of the state-of-the-art methods optimizes for road networks, applies indexing effectively, or obtains acceptable query time for large networks.

## 2.3 Other Related Work

Joksch [21] first studies the CSP problem, and proposes a dynamic programming algorithm for exact CSP. Subsequently, Handler and Zang [17] propose two methods for exact CSP processing: one method formulates CSP as an integer linear programming (ILP) problem, and solves it with a standard ILP solver. This same methodology is used by Mehlhorn and Ziegelmann [25]. However, as shown in [25], these ILP-based solutions scale poorly, and incur tremendous processing costs on large road networks. The other solution in [17] reduces CSP to a *k-shortest path* problem, and repeatedly computes the next shortest path (in terms of total length) until reaching one that satisfies the cost constraint. Afterwards, Hansen [18] proposes an augmented Dijkstra’s algorithm [10] for exact CSP queries and is shown in [28] to outperform the *k-shortest path* solution. The state-of-the-art methods for exact CSP are described in Section 2.2, for index-free and indexed processing, respectively. Meanwhile, most recently, Sedeno et al. [27] propose several pruning strategies to improve the efficiency of *k-shortest path* search, and is shown to outperform the existing *k-shortest path* solutions. However, it does not compare with the *Sky-Dijk* solution. In our experiment, we include Sedeno et al.’s solution [27] as one of our competitors.

Regarding  $\alpha$ -CSP, Hansen [18] proposes the first solution, which runs in polynomial time but has a high complexity:  $\mathcal{O}(m^2 \frac{n^2}{\alpha-1} \log \frac{n}{\alpha-1})$ . Lorenz and Raz. [24] improve the complex-





**Figure 2: A partition  $\mathcal{T} = \{G_1, G_2, G_3\}$  of  $G$  in Figure 1 and the corresponding overlay graph  $G^\circ$ .**

ity to  $\mathcal{O}(nm \cdot (\log \log \frac{\ell_{max}}{\ell_{min}} + \frac{1}{\alpha-1}))$ . However, this solution is orders of magnitude slower than an exact CSP algorithm based on  $k$ -shortest path, as shown in [23]. Later on, Tsaggouris et al. [31] propose *CP-Dijk* based on the conservative pruning technique, i.e., the current state-of-the-art for  $\alpha$ -CSP as described in Section 2.2.

Delling et al. [9] study a related query, which returns the entire set of skyline paths between two given vertices. Their solution creates shortcuts similarly as *CSP-CH* [30], and can be adapted to answer CSP queries. However, this method is not scalable to large graphs, as shown in [12, 30]. In order to reach an acceptable processing time, [9] proposes to modify the problem setting by relaxing the definition of dominance. However, with this relaxation, the method can no longer be used to answer CSP or  $\alpha$ -CSP queries. Another related query type is to find the shortest path in terms of a weighted sum of edge costs [11, 14]. These methods, however, cannot be used to answer CSP or  $\alpha$ -CSP queries.

Finally, we briefly review classic shortest path and distance queries. One notable class of solutions [8, 20, 22] employ graph partitioning, as in the proposed method *COLA*. The representative is *MLD* [8], which combines partitioning and contraction hierarchy to improve query efficiency. Yu et al. [34] propose *CI-Rank*, which first identifies a number of star vertices, and then builds an overlay graph on these star vertices. The proposed method *COLA* differs from *CI-Rank* in two major aspects. First, in terms of data structure, the overlay graph in *COLA* correspond to skyline paths, rather than simple shortest paths as in *CI-Rank*. Second, in terms of algorithm, *COLA* builds an additional index structure on top of overlay graph, whereas *CI-Rank* processes the query directly using the overlay graph. Another important indexing technique is *2-hop labelling (2HL)* [6]. The state-of-the-art *2HL* algorithms pre-compute an order of vertices in the graph, and construct a *2HL* index based on this order, e.g., [4, 7, 33]. None of these methods applies to CSP or  $\alpha$ -CSP. Hence, we omit further discussions on classic shortest path processing for brevity, and we refer the reader to a recent survey [5].

### 3. COLA FRAMEWORK

This section presents the general framework of our proposed solution *constrained labeling (COLA)*. The implementation of several important components in *COLA* is described in Section 4. Basically, *COLA* partitions the road network and constructs an *overlay graph* on top of the partitions. The index structure in *COLA* is then built on the overlay graph, whose size is much smaller than the original graph, leading to much less query processing costs. In the following, Section 3.1 describes the overlay graph; Section 3.2 explains the index structure of *COLA*; Section 3.3 elaborates on query processing based on the *COLA* index; Section 3.4 presents several sophisticated optimizations that significantly reduce query costs.

#### 3.1 Overlay Graph

Given an input graph  $G$ , a partitioning of  $G$  consists of a set  $\mathcal{T} = \{G_1, G_2, \dots, G_{|\mathcal{T}|}\}$  of edge-disjoint subgraphs of  $G$ , such that the union of all  $G_i$  ( $1 \leq i \leq |\mathcal{T}|$ ) equals  $G$ . Given  $\mathcal{T}$ , we say that a vertex  $v$  is a *boundary vertex*, if  $v$  appears in more than one subgraphs in  $\mathcal{T}$ . Graph partitioning is a well-studied problem, and *COLA* could use any of the existing solutions, e.g., [8, 20, 22]. In our implementation, we use a state-of-the-art approach for road networks by Delling et al. [8].

We formally define an overlay graph as follows.

**DEFINITION 3 (OVERLAY GRAPH).** Given an input graph  $G$ , a partitioning  $\mathcal{T}$  of  $G$ , and the query approximation ratio  $\alpha^3$ , a graph  $G^\circ = (V^\circ, E^\circ)$  is an overlay graph of  $G$  with respect to  $\mathcal{T}$ , if it satisfies the following three conditions:

1.  $V^\circ$  consists of all boundary vertices with respect to  $\mathcal{T}$ ;
2. For each edge  $e^\circ \in E^\circ$  that starts at vertex  $v$  and ends at vertex  $v'$ , there exists a path  $P$  in  $G$  that goes from  $v$  to  $v'$ , such that  $c(P) = c(e^\circ)$  and  $\ell(P) = \ell(e^\circ)$ ;
3. For any pair of origin and destination vertices  $s, t \in G^\circ$ , and any path  $P$  in  $G$  from  $s$  to  $t$ , there exists a path  $P^\circ$  in  $G^\circ$  that goes from  $s$  to  $t$  that  $\alpha$ -dominates  $P$ , i.e.,  $c(P^\circ) \leq c(P)$  and  $\ell(P^\circ) \leq \alpha \cdot \ell(P)$ .  $\square$

Intuitively, an overlay graph compresses the input graph by (i) including only the boundary vertices of the partitions and removing all other vertices, (ii) using edges to represent paths in  $G$ , and (iii) reducing the number of edges by removing paths that are  $\alpha$ -dominated by others. Note that the above definition does not require the overlay graph to be minimal, i.e., for the same graph  $G$  and partitioning  $\mathcal{T}$ , there may be another possible overlay graph with fewer edges. Hence, there can be different ways to build an overlay graph, and we explain one such algorithm later in Section 4.2. Besides, for any two vertices  $v$  and  $v'$  in the overlay graph  $G^\circ$ , there can be multiple edges from  $v$  to  $v'$ , when there are multiple paths from  $v$  to  $v'$  in  $G$ .

**EXAMPLE 3.** Consider the input graph  $G$  in Figure 1. Figures 2(a), (b), and (c) show three subgraphs  $G_1$ ,  $G_2$  and  $G_3$  of  $G$  respectively. Clearly,  $\mathcal{T} = \{G_1, G_2, G_3\}$  is a partitioning of  $G$ , since (i) the set of edges of each subgraph is disjoint with the other subgraphs, and (ii) the union of edges in  $G_1$ ,  $G_2$  and  $G_3$  constitutes the set of edges of  $G$ . Meanwhile,  $v_2$ ,  $v_3$ , and  $v_5$  are the boundary vertices w.r.t.  $\mathcal{T}$ , since they appear in more than one subgraphs.

Assume that  $\alpha = 1.1$ , Figure 2(d) shows an overlay graph  $G^\circ$  w.r.t.  $\mathcal{T}$ . The set of vertices of  $G^\circ$  is  $V^\circ = \{v_2, v_3, v_5\}$ , which consists of the boundary vertices of  $\mathcal{T}$ . Observe that there is exactly one path from  $v_2$  to  $v_3$ , i.e.  $\langle (v_2, v_3) \rangle$ ; hence,  $G^\circ$  contains an edge  $e_1^\circ = (v_2, v_3)$  with length  $\ell(e_1^\circ) = 1$  and cost  $c(e_1^\circ) = 3$ . Similarly, there is exactly one path from  $v_3$  to  $v_5$ ; thus,  $G^\circ$  also includes an edge  $e_2^\circ = (v_3, v_5)$  with  $\ell(e_2^\circ) = 1$  and  $c(e_2^\circ) = 3$ . From  $v_2$  to  $v_5$  there are three paths:  $P_1 = \langle (v_2, v_3), (v_3, v_5) \rangle$ ,  $P_2 = \langle (v_2, v_5) \rangle$ , and  $P_3 = \langle (v_2, v_4), (v_4, v_5) \rangle$ . Note that  $P_2$   $\alpha$ -dominates  $P_3$ , and the two edges in path  $P_1$  already exist in  $G^\circ$ . Hence,  $G^\circ$  only includes one edge  $e_3^\circ$  from  $v_2$  to  $v_5$  with length  $\ell(e_3^\circ) = \ell(P_2) = 4$  and cost  $c(e_3^\circ) = c(P_2) = 4$ .  $\square$

Since the overlay graph can be pre-computed and has a smaller size than the original graph, it can be used as a low-cost index to accelerate  $\alpha$ -CSP processing, as follows. Given an  $\alpha$ -CSP query  $q$  on  $G$  with an origin  $s$ , a destination  $t$ , and a cost threshold  $\theta$ , we first identify the subgraphs  $G_s$  and  $G_t$  in  $\mathcal{T}$  that contain  $s$  and  $t$ , respectively. Then, we construct graph  $G^q$  (which we call an

<sup>3</sup>Note that the overlay graph and the *COLA* index both require the knowledge of  $\alpha$ , which we consider as a system parameter. The choice of  $\alpha$  is discussed further in Section 6.4.

extended graph) by merging  $G_s$ ,  $G_t$  and  $G^\circ$ , i.e.,  $G^q = (V_s \cup V_t \cup V^\circ, E_s \cup E_t \cup E^\circ)$ , where  $V_s$  (resp.  $E_s$ ) and  $V_t$  (resp.  $E_t$ ) are the vertex (resp. edge) sets of  $G_s$  and  $G_t$ , respectively. After that, we run an  $\alpha$ -CSP algorithm on  $G^q$ ; its result corresponds to a result for the original  $\alpha$ -CSP query, according to the following lemma<sup>4</sup>.

LEMMA 1. Let  $P_{opt}$  be a result of a CSP query on graph  $G$  with origin  $s$ , destination  $t$ , and cost constraint  $\theta$ . Meanwhile, let  $G_s = (V_s, E_s)$ ,  $G_t = (V_t, E_t)$  be the subgraphs in a partitioning  $\mathcal{T}$  that contains  $s$  and  $t$ , respectively. Then, any result  $P$  of an  $\alpha$ -CSP with parameters  $s$ ,  $t$ ,  $\alpha$  on the extended graph  $G^q = (V_s \cup V_t \cup V^\circ, E_s \cup E_t \cup E^\circ)$  satisfies  $c(P) \leq \theta$  and  $\ell(P) \leq \alpha \cdot \ell(P_{opt})$ .  $\square$

To translate an  $\alpha$ -CSP on  $G^q$  to an  $\alpha$ -CSP on  $G$ , we “unfold” each edge in  $E^\circ$  into a path in  $G$ , which is done according to Condition 2 in Definition 3. Because  $G^q$  can be viewed as a compressed version of  $G$  with significantly fewer vertices and edges, searching for an  $\alpha$ -CSP on  $G^q$  is expected to be faster than doing so on  $G$ . On the other hand, the speedup using an overlay graph is limited, since the query processing algorithm is the same, albeit on a smaller graph. Next we introduce a much more powerful index structure.

### 3.2 Constrained Labeling Index

The main *COLA* index is constructed on the overlay graph  $G^\circ = (V^\circ, E^\circ)$  described in the previous subsection. For each vertex  $v^\circ \in G^\circ$ , the index contains two label sets for  $v^\circ$ : an *in-label set*  $B_{in}(v^\circ)$  and an *out-label set*  $B_{out}(v^\circ)$ . Each entry in  $B_{out}(v^\circ)$  corresponds to a path<sup>5</sup> from  $v^\circ$  to another vertex in  $G^\circ$ . Symmetrically, each label in  $B_{in}(v)$  corresponds to a path from another vertex in  $G^\circ$  to  $v^\circ$ . The paths in the label sets are carefully chosen such that given any pair of origin and destination vertices  $s^\circ, t^\circ \in G^\circ$ , and a cost constraint  $\theta$ , we can construct the  $\alpha$ -CSP from  $s^\circ$  to  $t^\circ$  subject to  $\theta$  using only the paths in  $B_{out}(s^\circ)$  and  $B_{in}(t^\circ)$ . In other words, with the *COLA* index we do not need to search for the  $\alpha$ -CSP result; instead, we simply combine pre-computed paths in the label sets to form a result.

Formally, we define the *COLA* index as follows.

DEFINITION 4 (COLA INDEX). Given an overlay graph  $G^\circ$ , a *COLA* index contains label sets  $B_{in}(v^\circ)$  and  $B_{out}(v^\circ)$  for each vertex  $v^\circ \in G^\circ$  satisfying the following conditions:

1. Each entry in  $B_{in}(v^\circ)$  corresponds to a path from another vertex in  $G^\circ$  to  $v^\circ$ ;
2. Each entry in  $B_{out}(v^\circ)$  corresponds to a path from  $v^\circ$  to another vertex in  $G^\circ$ ;
3. For any path  $P$  between any two vertices  $s^\circ, t^\circ \in V^\circ$  ( $P$  may contain vertices in  $V \setminus V^\circ$ ), the *COLA* index contains both an out-label in  $B_{out}(s^\circ)$  with cost  $c_o$  and length  $\ell_o$  and an in-label in  $B_{in}(t^\circ)$  with cost  $c_i$  and length  $\ell_i$  such that  $c_o + c_i \leq c(P)$  and  $\ell_o + \ell_i \leq \alpha \cdot \ell(P)$ .  $\square$

Condition 3 in the above definition indicates that for any path  $P$  connecting two vertices  $s^\circ$  and  $t^\circ$  in the overlay graph, we can derive another path  $P'$  by concatenating two paths  $P_o$  and  $P_i$  from the out-label set of  $s^\circ$  and in-label set of  $t^\circ$  respectively, such that  $P'$   $\alpha$ -dominates  $P$ . Therefore, according to the definition of  $\alpha$ -CSP (Definition 1) and  $\alpha$ -dominance (Definition 2), we can obtain an  $\alpha$ -CSP result between  $s^\circ$  and  $t^\circ$  by joining the paths from their label sets, without searching for the result from scratch.

<sup>4</sup>We include all proofs in Appendix.

<sup>5</sup>Note that, similar to *Sky-Dijk* and *CP-Dijk*, it suffices to store the important path parameters such as its length, cost and last two vertices, instead of the entire path. These details are clarified in Section 5.2; for now, we assume that each entry in a label set is a path for simplicity.

EXAMPLE 4. Consider the overlay graph  $G^\circ$  in Example 3 with  $\alpha = 1.1$ . Let  $P_1^\circ = \langle (v_2, v_3) \rangle$ ,  $P_2^\circ = \langle (v_3, v_5) \rangle$ , and  $P_3^\circ = \langle (v_2, v_5) \rangle$ . Let  $P'_1$ ,  $P'_2$ , and  $P'_3$  be three trivial paths that go from  $v_2$  to  $v_2$ ,  $v_3$  to  $v_3$ , and  $v_5$  to  $v_5$ , with zero cost / length, respectively.

Then  $B_{out}(v_2) = \{P_1^\circ, P'_1\}$ ,  $B_{in}(v_2) = \{P'_1\}$ ,  $B_{out}(v_3) = \{P_2^\circ\}$ ,  $B_{in}(v_3) = \{P'_2\}$ ,  $B_{out}(v_5) = \{P_3^\circ\}$ , and  $B_{in}(v_5) = \{P'_3\}$  constitute an instance  $\mathcal{L}$  of *COLA* index. To explain, clearly,  $\mathcal{L}$  satisfies Conditions 1 and 2 in Definition 4. It remains to verify that  $\mathcal{L}$  satisfies Condition 3. Note that in the input graph  $G$ , there are five paths concerning nodes in  $G^\circ$ , i.e.,  $P_1 = \langle (v_2, v_4), (v_4, v_5) \rangle$ ,  $P_2 = \langle (v_2, v_3) \rangle$ ,  $P_3 = \langle (v_3, v_5) \rangle$ ,  $P_4 = \langle (v_2, v_5) \rangle$ , and  $P_5 = \langle (v_2, v_3), (v_3, v_5) \rangle$ . Consider the first path  $P_1 = \langle (v_2, v_4), (v_4, v_5) \rangle$  with cost  $c_1 = 6$  and  $\ell_1 = 5$ .  $P_1^\circ$  in  $B_{out}(v_2)$  and  $P_2^\circ$  in  $B_{in}(v_5)$  satisfy that  $c(P_1^\circ) + c(P_2^\circ) \leq c(P_1)$  and  $\ell(P_1^\circ) + \ell(P_2^\circ) \leq \alpha \cdot \ell(P_1)$ . Similarly, we can verify that  $\mathcal{L}$  also fulfills Condition 3 for the other four paths.  $\square$

One may wonder why we need both an in-label set and an out-label set, instead of just one of them. For example, given a pair of origin and destination vertices  $s^\circ$  and  $t^\circ$ , if the out-label set  $B_{out}(s^\circ)$  of  $s^\circ$  contains a path that ends at  $t^\circ$  and satisfies the cost constraint, we could simply return this path as the  $\alpha$ -CSP result, without checking the in-labels of  $t^\circ$ . The problem with having only out- (or in-) labels is that we must store for each vertex the complete set of labels containing skyline paths to (or from) every other vertex in  $G^\circ$ , leading to a prohibitively large index size. In contrast, by using both in-labels and out-labels, each label set only contains path to (or from) a selected subset of vertices, leading to a significantly reduced index size. This is akin to database normalization, where storing two separate base tables consumes less space than their join results.

### 3.3 Query Processing

This subsection clarifies the processing of an  $\alpha$ -CSP query with a pair of origin and destination vertices  $s, t \in G$  and a cost constraint  $\theta$ , using the overlay graph and the *COLA* index described in previous subsections. Note that if both  $s$  and  $t$  belong to the overlay graph  $G^\circ$ , we can simply join  $B_{out}(s)$  and  $B_{in}(t)$ , and select the  $\alpha$ -CSP result by concatenating a path from  $B_{out}(s)$  and another from  $B_{in}(t)$ , according to Condition 3 in Definition 4. However, either  $s$  or  $t$  may not appear in the overlay graph, where the  $\alpha$ -CSP queries with  $s$  and  $t$  cannot be answered purely by the *COLA* index which is built on the overlay graph. Note that we could build the *COLA* index on the original graph  $G$  instead of the overlay graph  $G^\circ$ . Nevertheless, doing so may lead to a prohibitively large index size, since  $G$  is far larger than  $G^\circ$ .

The main idea of *COLA* query processing is to build  $B_{out}(s)$  and  $B_{in}(t)$  during query time, using the *COLA* index as well as subgraphs  $G_s, G_t \in \mathcal{T}$  containing  $s$  and  $t$ , respectively. In particular,  $B_{out}(s)$  and  $B_{in}(t)$  must satisfy that the  $\alpha$ -CSP result can be obtained by concatenating a path from  $B_{out}(s)$  and another from  $B_{in}(t)$ . Formally, for any path  $P$  between  $s$  and  $t$ , there must exist paths  $P_o$  and  $P_i$  in  $B_{out}(s)$  and  $B_{in}(t)$  respectively, such that the concatenation of  $P_o$  and  $P_i$   $\alpha$ -dominates  $P$ . Given this property, the  $\alpha$ -CSP result can be obtained by joining  $B_{out}(s)$  and  $B_{in}(t)$  similarly as the case when  $s$  and  $t$  are boundary vertices.

The main challenge thus lies in the computation of  $B_{out}(s)$  and  $B_{in}(t)$ . We first focus on the former, initialized to empty. Let  $G_s \in \mathcal{T}$  be the sub-graph containing  $s$ . We perform a Dijkstra-like search from vertex  $s$  to every boundary vertex of  $G_s$ . This can be done, for example, using a slightly modified version (i.e., with multiple destinations) of the *CP-Dijk* algorithm described in Section 2.2. In our implementation, we use a novel algorithm  $\alpha$ -*Dijk*, detailed in Section 4, which is significantly more efficient than *CP-Dijk*. After

---

**Algorithm 1: COLA**

---

**input** :  $s, t, \theta, \alpha, G, G^\circ, B_{out}(s)$ , and  $B_{in}(t)$   
**output**: A path for the  $\alpha$ -CSP query with the origin  $s$ , the destination  $t$ , and cost threshold  $\theta$  on  $G$

- 1 Initialize both  $B_{out}(s)$  and  $B_{in}(t)$  to empty;
- 2 Perform a Dijkstra search from  $s$  within its partition  $G_s$  to obtain the set of skyline paths  $L(v^\circ)$  from  $s$  to each of the boundary vertex  $v^\circ$  in  $G_s$ ;
- 3 **for each** boundary vertex  $v^\circ \in G_s \cap G^\circ$  **do**
- 4     Join  $L(v^\circ)$  and  $B_{out}(v^\circ)$ ;     // optimized in Section 3.4
- 5     **for each** joined path  $P$  from  $s$  to  $w^\circ \in G^\circ$  **do**
- 6         **if**  $c(P) > \theta$  or  $P$  is  $\alpha$ -dominated by a path in  $B_{out}(s)$  to  $w^\circ$  **then**
- 7             Discard  $P$ ;
- 8         **else**
- 9             Add  $P$  to  $B_{out}(s)$ ;
- 10            Delete all paths in  $B_{out}(s)$  from  $s$  to  $w^\circ$  that are  $\alpha$ -dominated by  $P$ ;
- 11 Compute  $B_{in}(t)$  similarly as Lines 2-9 (see Section 3.3);
- 12 Join  $B_{out}(s)$  and  $B_{in}(t)$ ;     // optimized in Section 3.4
- 13 **return** the  $\alpha$ -SCPs from the above join result;

---

we finish the Dijkstra search, we extract the set of skyline paths (c.f. Section 2.1)  $L(v^\circ)$  for each boundary vertex  $v^\circ \in G_s \cap G^\circ$ . Then, we join  $L(v^\circ)$  with  $B_{out}(v^\circ)$  and add the results to  $B_{out}(s)$ . Specifically, for each path  $P_1$  in  $L(v^\circ)$  from  $s$  to  $v^\circ$ , and each path  $P_2$  in  $B_{out}(v^\circ)$  from  $v^\circ$  to another boundary vertex (say,  $w^\circ \in G^\circ$ ), we concatenate  $P_1$  and  $P_2$  into a path  $P$  from  $s$  to  $w^\circ$ , and insert  $P$  to  $B_{out}(s)$  if the latter does not contain a path that  $\alpha$ -dominates  $P$ . After that, we purge from  $B_{out}(s)$  all paths from  $s$  to  $w^\circ$  that are  $\alpha$ -dominated by  $P$ . The computation for  $B_{in}(t)$  is symmetric and omitted for brevity. Algorithm 1 summarizes the COLA query processing algorithm.

**EXAMPLE 5.** Consider the overlay graph in Example 3, and the COLA index  $\mathcal{L}$  in Example 4 with  $\alpha = 1.1$ . Given an  $\alpha$ -CSP query from  $v_1$  to  $v_5$  with cost threshold  $\theta = 7$ , COLA first checks whether  $v_1$  and  $v_5$  are boundary vertices. Since  $v_5$  is a boundary vertex, the method directly obtains  $B_{in}(v_5) = \{P_2^\circ, P_3^\circ, P_4^\circ\}$ . On the other hand, since  $v_1$  is not a boundary vertex, its out-label set  $B_{out}(v_1)$  needs to be computed on the fly. To do this, COLA initiates a Dijkstra-like search from  $v_1$ , and computes the set of skyline paths from  $v_1$  to the boundary nodes of  $v_1$ 's subgraph. In particular, it retrieves the skyline set from  $v_1$  to  $v_2$ , which contains only  $P_1 = \langle (v_1, v_2) \rangle$ . Then, it joins the skyline set with  $B_{out}(v_2)$ , and adds the joined paths into  $B_{out}(v_1)$  if they are not  $\alpha$ -dominated by any path in  $B_{out}(v_1)$ . After that, we have  $B_{out}(v_1) = \{P_1, P_1 \cdot P_1^\circ\}$ , where  $P_1 \cdot P_1^\circ$  denotes the concatenation of  $P_1$  and  $P_1^\circ$ .

Similarly, COLA retrieves the skyline set from  $v_1$  to  $v_3$ , and joins paths in the skyline set with  $B_{out}(v_3)$ . These paths are  $P_2 = \langle (v_1, v_3) \rangle$  and  $P_3 = \langle (v_1, v_2), (v_2, v_3) \rangle$ . Note that  $P_3$  is identical to  $P_1 \cdot P_1^\circ$ . Hence,  $P_3$  is not added to  $B_{out}(v_1)$ , which ends up with  $B_{out}(v_1) = \{P_1, P_1 \cdot P_1^\circ, P_2\}$ . By joining  $B_{out}(v_1)$  and  $B_{in}(v_5)$ , COLA retrieves a skyline set for all paths from  $v_1$  to  $v_5$ . Finally, it inspects the results, unfolds edges in  $G^\circ$  whenever necessary, and returns path  $P = \langle (v_1, v_2), (v_2, v_3), (v_3, v_5) \rangle$ .  $\square$

The query processing algorithm described so far contains several nested-loop join operations, which can be rather expensive for large label / skyline sets. Next we present effective optimizations that reduce the cost of such joins.

### 3.4 Optimizations

---

**Algorithm 2: LabelJoin**

---

**input** :  $\theta, \alpha, B_{out}(s)$ , and  $B_{in}(t)$   
**output**: An  $\alpha$ -CSP  $P^*$  with a cost not larger than  $\theta$  from  $s$  to  $t$

- 1 Sort paths in  $B_{out}(s)$  firstly in ascending order of end vertex ID, and secondly in ascending order of cost (which is also descending order of length);
- 2 Sort paths in  $B_{in}(t)$  firstly in ascending order of end vertex ID, and secondly in ascending order of length (which is also descending order of cost);
- 3 Initialize  $\alpha$ -CSP result  $P^*$  to empty;
- 4 **repeat**
- 5     Scan both  $B_{out}(s)$  and  $B_{in}(t)$  simultaneously, until reaching a matching pair  $P_o \in B_{out}(s)$  and  $P_i \in B_{in}(t)$ ;
- 6     **while**  $P_o$  matches  $P_i$ , i.e.,  $P_o$  ends at the origin vertex of  $P_i$  **do**
- 7         **if**  $c(P_o) + c(P_i) > \theta$  **then**
- 8             Set  $P_i$  to the next entry in  $B_{out}(s)$ ;
- 9         **else**
- 10            Update  $P^*$  to the concatenation of  $P_o$  and  $P_i$  if the combination of  $P_o$  and  $P_i$  has length smaller than  $P^*$ ;
- 11            Set  $P_o$  to next entry in  $B_{in}(t)$ ;
- 12 **until** reaching the end of either  $B_{out}(s)$  or  $B_{in}(t)$ ;
- 13 **return**  $P^*$ ;

---

First we optimize the join between  $B_{out}(s)$  and  $B_{in}(t)$ , which produces the  $\alpha$ -CSP result based on the following observation.

**OBSERVATION 1.** Let  $P_o$  and  $P_i$  be two arbitrary paths in  $B_{out}(s)$  and  $B_{in}(t)$  respectively that can be joined, i.e.,  $P_o$  ends at the starting vertex of  $P_i$ . We have the following:

1. If  $c(P_o) + c(P_i) > \theta$ , then joining  $P_o$  (resp.  $P_i$ ) with any path in  $B_{in}(t)$  (resp.  $B_{out}(s)$ ) with cost higher than  $P_i$  (resp.  $P_o$ ) cannot lead to an  $\alpha$ -CSP result;
2. If  $c(P_o) + c(P_i) \leq \theta$ , then joining  $P_o$  (resp.  $P_i$ ) with any path in  $B_{in}(t)$  (resp.  $B_{out}(s)$ ) with length longer than  $P_i$  (resp.  $P_o$ ) can be discarded.  $\square$

Based on the above observation, we accelerate the join between  $B_{out}(s)$  and  $B_{in}(t)$  through a careful ordering of the labels. Specifically, COLA sorts paths in  $B_{out}(s)$  by the IDs of their end vertices, breaking ties by total cost (in ascending order). Note that  $B_{out}(s)$  is a skyline set, meaning that paths with the same end vertex are also automatically sorted in descending order of their lengths (otherwise one path would dominate another). Similarly, COLA sorts paths in  $B_{in}(t)$  firstly by the IDs of their origin vertices, and secondly in ascending of their lengths / descending order of their costs. With such ordering, we propose a novel algorithm *LabelJoin* (as shown in Algorithm 2), which joins  $B_{out}(s)$  and  $B_{in}(t)$  with a *linear scan of each set*.

**LEMMA 2.** *Algorithm 2 correctly computes the  $\alpha$ -CSP result from  $B_{out}(s)$  and  $B_{in}(t)$  in linear time.*  $\square$

**EXAMPLE 6.** Consider an  $\alpha$ -CSP query from  $s$  to  $t$  with  $\alpha = 1.1$  and cost threshold  $\theta = 13$ . Assume that  $B_{out}(s) = \{P_1, P_2\}$  where both  $P_1$  and  $P_2$  end at  $w$ , with  $c(P_1) = 4$ ,  $\ell(P_1) = 7$ ,  $c(P_2) = 7$ , and  $\ell(P_2) = 4$ ;  $B_{in}(t) = \{P_3, P_4, P_5\}$  where all three paths start at  $w$ , with  $c(P_3) = 7$ ,  $\ell(P_3) = 5$ ,  $c(P_4) = 6$ ,  $\ell(P_4) = 6$ ,  $c(P_5) = 5$ , and  $\ell(P_5) = 7$ . Clearly,  $B_{out}(s)$  (resp.  $B_{in}(t)$ ) is sorted in ascending (resp. descending) order of cost.

By Algorithm 2, we first check  $P_1$  and  $P_3$ , to see if the concatenated path  $P' = P_1 \cdot P_3$  satisfies the cost constraint. As  $c(P') = 11 \leq \theta = 13$  and  $P^*$  is empty, the found shortest path under cost constraint is hence updated to  $P^* = P'$ . Afterwards, the *LabelJoin* algorithm proceeds to the next path  $P_2$  in  $B_{out}(s)$ , and concatenate it with  $P_3$ . Here,  $P_1$  is not further concatenated



with  $P_4$  and  $P_5$  due to Observation 1.2, i.e., concatenating  $P_1$  with  $P_4$  or  $P_5$  produces a path with a larger length than  $P'$ .

Consider the concatenated path  $P'_1 = P_2 \cdot P_3$ . Note that the cost of  $P'_1$  exceeds the cost threshold, and Algorithm 2 proceeds to the next path, i.e.  $P_4$ , in  $B_{in}(t)$ . Consider  $P'_2 = P_2 \cdot P_4$ . The cost of  $P'_2$  is 13, which satisfies the cost constraint, and the length of  $P'_2$  is 10, smaller than  $P^*$ . Hence  $P^*$  is updated to  $P'_2$ . Since there is no more path in  $B_{out}(s)$ , the *LabelJoin* algorithm stops checking the labels and returns  $P^*$  as the result.  $\square$

Next, we focus on the join between a skyline set  $L(v^\circ)$  and a label set  $B_{out}(v^\circ)$  in the *COLA* index (Line 4 of Algorithm 1). The case for joining  $L(v^\circ)$  with  $B_{in}(v^\circ)$  is symmetric and omitted for brevity. The basic idea for the optimization is not to compute the complete join results, but only those results that can possibly lead to an  $\alpha$ -CSP result. Specifically, we avoid generating certain join results based on the following observation.

**OBSERVATION 2.** Let  $P$  be a path from  $s$  to  $w^\circ$  from the join result of  $L(v^\circ)$  and  $B_{out}(v^\circ)$ .  $P$  cannot possibly lead to an  $\alpha$ -CSP result, if any of the following is true.

1.  $w^\circ$  cannot possibly reach  $t$  on the input graph  $G$ ;
2. The minimum cost for any path from  $w^\circ$  to  $t$  exceeds  $\theta - c(P)$ ;
3. There exists a path  $P'$  from  $s$  to  $t$  satisfying the cost constraint, such that the minimum length of any path from  $w^\circ$  to  $t$  exceeds  $\ell(P')/\alpha - \ell(P)$ .  $\square$

According to the above observation, before performing any join operation, we first select a set of end vertices  $W$  for the join results that can possibly lead to an  $\alpha$ -CSP result, which is incrementally pruned using Observation 2. Then, we filter the *COLA* index, and use only the labels that reach a vertex in  $W$ . Algorithm 3 shows the algorithm for computing  $W$ . Filtering the *COLA* index before joining it with the skyline sets can be understood as using a semi-join to improve join performance in database systems. Note that the *COLA* index is constructed before we know the query parameters, hence, it usually contains a large number of labels not needed for answering the query at hand.

Algorithm 3 shows the pseudo-code for computing  $W$ .  $W$  is initialized with all boundary vertices reachable from both  $s$  and  $t$ . Then, we prune those vertices that cannot lead to a path from  $s$  to  $t$  within cost threshold  $\theta$ . After that, the algorithm computes an upper bound for the length of a path from  $s$  to  $t$ , and uses it to prune more vertices in  $W$  based on the third condition in Observation 2.

#### 4. $\alpha$ -DIJK

Recall that our query processing involves computing the skyline paths to (resp. from) the boundary vertex from the origin vertex  $s$  (resp. to the destination  $t$ ). In this section, we present an efficient algorithm  $\alpha$ -*Dijk* to accelerate this process. Apart from this,  $\alpha$ -*Dijk* has three other main uses: (i) for intra-partition search during query processing in *COLA*, (ii) for building the *COLA* index, and (iii) as a standalone index-free solution for  $\alpha$ -CSP.

Similar to *CP-Dijk* (refer to Section 2.2),  $\alpha$ -*Dijk* is based on *Sky-Dijk* with enhanced pruning based on the relaxed  $\alpha$ -dominance definition. On the other hand, the pruning strategy of  $\alpha$ -*Dijk* is radically different from that in *CP-Dijk*. The intuition is as follows. Imagine that we have a total “budget” for pruning along a path; the higher the budget allocated to a vertex, the stronger pruning power it is allowed to apply to reduce the size of its set of associated paths. *CP-Dijk* simply distributes this budget equally to each vertex on the path. Since there can be a large number of vertices on a path, each of them only receives little pruning power, leading to ineffective pruning everywhere. In contrast,  $\alpha$ -*Dijk* concentrates the pruning

---

#### Algorithm 3: PruneLabel

---

**input** :  $s, t, \theta, \alpha, L, B_{out}$ , and  $B_{in}$   
**output**:  $B_{out}$  and  $B_{in}$  with labels filtered

- 1 Initialize vertex set  $W$  with all boundary vertices reachable from both  $s$  and  $t$ , according to  $B_{out}(s)$  and  $B_{in}(t)$ ; // Pruning condition 1
- 2 **for each** vertex  $w$  in  $W$  **do**
- 3     **for each** vertex  $v^\circ$  in  $G_s$  **do**
- 4         Compute the minimum cost  $c_1$  from  $s$  to  $v^\circ$  in  $L(v^\circ)$ ;
- 5         Compute the minimum cost  $c_2$  from  $v^\circ$  to  $w$  in  $B_{out}(v^\circ)$ ;
- 6         Set  $c_s(w) = c_1 + c_2$ ;
- 7     Compute  $c_t(w)$  similarly as Lines 3-6;
- 8     **if**  $c_s(w) + c_t(w) > \theta$  **then**
- 9         Remove  $w$  from  $W$ ; // Pruning condition 2
- 10 Initialize  $\ell_{max}$  to  $-\infty$ ;
- 11 **for each** vertex  $w$  in  $W$  **do**
- 12     **for each** boundary vertex  $v^\circ$  in  $G_s$  **do**
- 13         Compute the max length  $\ell_{max}^1$  from  $s$  to  $v^\circ$  in  $L(v^\circ)$ , and the max length  $\ell_{max}^2$  from  $v^\circ$  to  $w$  in  $B_{out}(v^\circ)$ ;
- 14         Set  $\ell_{max}^s(w) = \ell_{max}^1 + \ell_{max}^2$ ;
- 15     Compute  $\ell_{max}^t(w)$  similarly as Lines 12-14;
- 16     Update  $\ell_{max}$  if  $\ell_{max}^s(w) + \ell_{max}^t(w) > \ell_{max}$ ;
- 17 **for each** vertex  $w$  in  $W$  **do**
- 18     Compute minimum lengths  $\ell_{min}^s(w)$  and  $\ell_{min}^t(w)$  similarly as Lines 12-14;
- 19     **if**  $\ell_{min}^s(w) + \ell_{min}^t(w) > \ell_{max}$  **then**
- 20         Remove  $w$  from  $W$ ; // Pruning condition 3
- 21 Filter  $B_{out}$  (resp.  $B_{in}$ ) by removing paths that do not end (resp. originate) at a vertex in  $W$ ;
- 22 **return**  $B_{out}$  and  $B_{in}$ ;

---

power to vertices that are associated with a large number of paths, and does not allocate pruning power at all to vertices with relatively few paths. In a real road network, there are usually a small number of landmark vertices that appear frequently in CSPs, which tend to accumulate a large number of paths. As a result, concentrating the pruning power to such vertices leads to effective reduction of the total number of paths to be examined, and thus, accelerates query processing.

Formally,  $\alpha$ -*Dijk* applies  $\alpha_i$ -dominance to prune entries in  $L(v_i)$ , where the values of  $\alpha_i \in [1, \alpha]$  depends on  $v_i$ . In other words, the amount of pruning applied in each  $L(v_i)$  is *adaptive*, rather than fixed as in *CP-Dijk*. Let  $P_{opt}$  be the CSP from an origin vertex  $s$  to a destination vertex  $t$  under a cost threshold  $\theta$ . Let  $k$  be the number of edges in  $P_{opt}$ ,  $\ell_i$  ( $i \in [1, k]$ ) be the length of the  $i$ -th edge in  $P_{opt}$ , and  $v_j$  be the vertex that the  $i$ -th edge points to. By adopting  $\alpha_i$ -dominance into the construction of  $L(v_i)$ ,  $\alpha$ -*Dijk* always returns a path  $P$  from  $s$  to  $t$ , such that  $c(P) \leq \theta$  and

$$\begin{aligned} \ell(P) &\leq \ell_1 \prod_{i=1}^k \alpha_i + \ell_2 \prod_{i=2}^k \alpha_i + \ell_3 \prod_{i=3}^k \alpha_i + \dots + \ell_k \prod_{i=k}^k \alpha_i \\ &= \sum_{j=1}^k \left( \ell_j \cdot \prod_{i=j}^k \alpha_i \right). \end{aligned}$$

Therefore,  $P$  can be an  $\alpha$ -CSP if each  $\alpha_j$  is carefully selected to ensure that

$$\sum_{j=1}^k \left( \ell_j \cdot \prod_{i=j}^k \alpha_i \right) \leq \sum_{j=1}^k (\ell_j \cdot \alpha). \quad (1)$$

A straightforward approach to enforce Inequality (1) is to set  $\alpha_i = \alpha^{\frac{1}{n}}$  (as in the case of *CP-Dijk*), but it leads to ineffective pruning in each  $L(v_i)$ , as we discuss in Section 2.2. One may wonder whether we can apply pruning everywhere, i.e., maximizing the

punning power. The following example shows that pruning everywhere may cause problems.

**EXAMPLE 7.** Given the graph  $G$  in Figure 1, consider an  $\alpha$ -CSP query with origin  $v_1$ , destination  $v_5$ , cost constraint  $\theta = 7$ , and  $\alpha = 1.35$ . Assume that  $\alpha$ -dominance is used to do the pruning during the traversal.

Similar to *Sky-Dijk*, it first adds path  $P_1 = \{(v_1, v_2)\}$  (with  $c_1 = 1$ ,  $\ell_1 = 2$ ) to  $L(v_2)$ ,  $P_2 = \{(v_1, v_3)\}$  (with  $c_2 = 4$ ,  $\ell_2 = 3$ ) to  $L(v_3)$ , and then pushes these two paths into heap  $H$ . Next  $P_1$  is popped from  $H$ , and  $P_3 = \{(v_1, v_2), (v_2, v_3)\}$  (with  $c_3 = 4$ ,  $\ell_3 = 3$ ),  $P_4 = \{(v_1, v_2), (v_2, v_4)\}$  (with  $c_4 = 5$ ,  $\ell_4 = 4$ ),  $P_5 = \{(v_1, v_2), (v_2, v_5)\}$  (with  $c_5 = 5$ ,  $\ell_5 = 6$ ) are extended from  $P_1$ . As  $c_2 = 3 \leq c_3 = 4$  and  $\ell_2 = 4 \leq \alpha \cdot \ell_3 = 1.35 \times 3$ ,  $P_3$  is  $\alpha$ -dominated by  $P_2$  that is stored in  $L(v_3)$ . Then by applying the greedy pruning strategy,  $P_4$  is omitted and is not added into  $H$ .  $P_4$  and  $P_5$  are added into  $L(v_4)$  and  $L(v_5)$ , respectively, and are pushed into  $H$ .

Next,  $P_2$  has the minimum cost in  $H$ , and is popped from  $H$ .  $P_6 = \{(v_1, v_3), (v_3, v_5)\}$  (with  $c_6 = 6$  and  $\ell_6 = 5$ ) is then extended from  $P_2$ . Meanwhile, as  $c_5 = 5 \leq c_6$  and  $\ell_5 = 6 \leq \alpha \cdot \ell_6 = 1.35 \cdot 5$ ,  $P_6$  can be  $\alpha$ -dominated by  $P_5$  stored in  $L(v_5)$ , and  $P_6$  is discarded.

When the traversal finishes,  $L(v_5)$  ends up with only  $P_5$  and  $P_5$  satisfies the cost constraint, so it returns  $P_5$  as the query result. However, the answer for the exact CSP query is path  $P_7 = \{(v_1, v_2), (v_2, v_3), (v_3, v_5)\}$ , with  $c_7 = 7$  and  $\ell_7 = 4$ . Note that  $\ell_5 = 6 > \alpha \cdot \ell_7 = 1.35 \times 4$ , which means  $P_7$  is not  $\alpha$ -dominated by  $P_5$ , and  $P_5$  is not a valid  $\alpha$ -CSP query answer.  $\square$

In this paper, we propose to set  $\alpha_i = 1$  for some vertices  $v_i$  but  $\alpha_j > \alpha_i$  for the others  $v_j$ . That is, we give up pruning in some entry lists, in exchange for more effective optimization in the other entry lists. Intuitively, this allows us to focus our pruning effort on the entry lists that are important for the given  $\alpha$ -CSP query to achieve higher efficiency. To implement this idea, however, we have to address two crucial issues. First, on which entry lists  $L(v_i)$  should we have  $\alpha_i > \alpha_i$ ? Second, given that  $\ell_1, \ell_2, \dots, \ell_k$  are unknown in advance, how can we appropriately select  $\alpha_i$  without invalidating Equation (1)? In the following, we clarify these issues by elaborating the details of our  $\alpha$ -Dijk algorithm. Algorithm 4 shows the pseudo-code of  $\alpha$ -Dijk. Given an  $\alpha$ -CSP query with an origin vertex  $s$ , a destination vertex  $t$ , and a cost threshold  $\theta$ ,  $\alpha$ -Dijk first invokes the vanilla Dijkstra's algorithm to compute, for each vertex  $v$ , the minimum-length path  $P_\ell(v)$  and the minimum-cost path, i.e., the path with the minimum cost (ties broken by smaller length),  $P_c(v)$  from  $s$  to  $v$  (Line 1). Then it records the length of  $P_\ell(v)$  and  $P_c(v)$  as  $\ell^\perp(v)$  and  $\ell^\top(v)$  (Line 2), respectively. Note that  $\ell^\perp(v)$  and  $\ell^\top(v)$  are lower- and upper- bounds on the total path length, respectively. To explain why  $\ell^\top(v)$  is an upper-bound, consider a path  $P'$  that has length larger than  $\ell^\top(v)$ . Clearly,  $P'$  can be  $\alpha$ -dominated by  $P_c(v)$ . Therefore, we can keep only  $P_c(v)$  and discard all paths with length larger than  $P_c(v)$ , meaning that its length  $\ell^\top(v)$  is the upper bound for total path length.

Next,  $\alpha$ -Dijk creates a min-heap  $H$  of skyline paths sorted on path costs similarly as in *Sky-Dijk* and *CP-Dijk*, except that in  $\alpha$ -Dijk each heap entry  $\rho = \langle P, \tau \rangle$  contains both a path  $P$  and an additional *pruning surrogate*  $\tau \in [\ell/\alpha, \ell]$  that facilitates adaptive allocation of the "pruning budget" as mentioned earlier<sup>6</sup>. The heap

<sup>6</sup>To conserve memory, in our implementation instead of a full path  $P$  each entry only stores its length  $\ell$ , cost  $c$ , last vertex  $v$  and the vertex  $v_{pred}$  before  $v$ . The full path can be reconstructed by using  $v$  and  $v_{pred}$ . Details can be found in Section 5.2.

---

#### Algorithm 4: $\alpha$ -Dijk

---

**input** : An  $\alpha$ -CSP query on  $G$  with an origin  $s$ , a destination  $t$ , and a cost threshold  $\theta$   
**output**: An answer  $P$  to the  $\alpha$ -CSP query

- 1 Calculate the minimum-length path  $P_\ell(v)$  and minimum-cost path  $P_c(v)$  from  $s$  to each vertex  $v$ ;
- 2 Let  $\ell^\perp(v)$  and  $\ell^\top(v)$  be the length of  $P_\ell(v)$   $P_c(v)$ , respectively;
- 3 Create a min-heap  $H$  with entries in the form  $\langle P, \tau \rangle$ , sorted in ascending order of path costs, breaking ties with path lengths;
- 4 Create an entry list  $L(v)$  for each vertex  $v$  in  $G$ ;
- 5 Insert an entry  $\langle P = \langle s \rangle, \tau = 0 \rangle$  into  $H$ ;
- 6 **while**  $H$  is not empty **do**
- 7 Pop the top entry  $\rho = \langle P, \tau \rangle$  in  $H$ ;
- 8 Let  $\langle P', \tau' \rangle$  be the entry in  $L(v)$  with the largest cost;
- 9 **if**  $\tau' \leq \tau$  **then**
- 10 **continue**; //  $\rho$  is pruned
- 11 Insert  $\rho$  into  $L(v)$ ;
- 12 **if**  $v \neq s$  and  $|L(v)| > \log_\alpha \frac{\ell^\top(v)}{\ell^\perp(v)}$  **then**
- 13 **Modify**  $\rho$  to set  $\tau = \max\{\ell/\alpha, \ell^\perp(v)\}$ ;
- 14 **for each outgoing edge**  $e = (v, v')$  **of**  $v$  **do**
- 15 **Construct** path  $P_{new}$  by extending  $P$  with  $e$ ;
- 16 **if**  $c + c(e) \leq \theta$  **then**
- 17 **Push**  $\langle P_{new}, \tau + \tau(e) \rangle$  into  $H$ ;

18 **return** the length-shortest path in  $L(t)$ ;

---

$H$  is initialized with a single entry that contains a trivial path that contains only one vertex  $s$ , and a pruning surrogate with value 0 (Line 5). In addition,  $\alpha$ -Dijk initializes a list  $L(v)$  of heap entries for each vertex  $v$  (Line 4).

After that,  $\alpha$ -Dijk iteratively pops the top entry  $\rho = \langle P, \tau \rangle$  from  $H$  and processes it as follows. Let  $v$  be the last vertex in  $P$ .  $\alpha$ -Dijk first examines  $L(v)$ , and retrieves the entry  $\rho' = \langle P', \tau' \rangle$  in  $L(v)$  with the largest path cost (Line 8). The algorithm guarantees that the cost of  $P'$  is smaller than that of  $P$ , since  $H$  is sorted in ascending order of path costs. Then,  $\alpha$ -Dijk compares the pruning surrogates  $\tau$  and  $\tau'$  of the two entries, and prunes  $\rho$  iff.  $\tau' \leq \tau$  (Lines 9-10). Let  $l(P)$  be the total length of  $P$  and  $l(P')$  be the length of  $P'$ , we have:

$$\ell(P') \leq \alpha \cdot \tau' \leq \alpha \cdot \tau \leq \alpha \cdot \ell(P),$$

which means that  $P'$   $\alpha$ -dominates  $P$  (note that the former also has a lower cost explained above). One may wonder why we prune based on surrogates rather than the path lengths. The reason is that the surrogates  $\tau$  and  $\tau'$  control the amount of pruning at vertex  $v$ . To see this, if we set  $\tau' = \ell'/\alpha$  and  $\tau = \ell$ , then  $P'$  can prune  $P$  whenever the former  $\alpha$ -dominates the latter, which indicates that we use the maximum pruning power at  $v$ . Conversely, when  $\tau' > \ell'/\alpha$  or  $\tau < \ell$ , pruning at  $v$  is not performed with full power, i.e., it is possible that  $P$  is not pruned even if  $P'$   $\alpha$ -dominates  $P$ . Note that *pruning with full power everywhere leads to incorrect results*, as shown in Example 7. Meanwhile, *pruning with the same power everywhere is inefficient*, as explained at the beginning of this subsection. The use of surrogates enables *adaptive pruning*, the key idea of  $\alpha$ -Dijk.

It remains to clarify how  $\alpha$ -Dijk computes the surrogate values. This is done in Lines 12-13. In particular, if  $\rho$  passes pruning, it is inserted into  $L(v)$ . At this point, the algorithm adjusts its surrogate  $\tau$  based on the following heuristic: if the number of entries  $|L(v)| > \log_\alpha \frac{\ell^\top(v)}{\ell^\perp(v)}$ , then  $\alpha$ -Dijk sets  $\tau = \max\{\ell/\alpha, \ell^\perp(v)\}$ , which grants  $\rho$  the maximum pruning power (Line 13). Otherwise,  $\alpha$ -Dijk sets  $\tau = \ell$ , minimizing the pruning capability of  $\rho$ . To explain, observe that  $\log_\alpha \frac{\ell^\top(v)}{\ell^\perp(v)}$  is an upper bound of  $|L(v)|$  when we



apply  $\alpha$ -dominance in the construction of  $L(v)$ . Intuitively, if the size of the current  $L(v)$  exceeds this upper bound, then applying aggressive pruning in  $L(v)$  is likely to reduce  $|L(v)|$  and help improve query efficiency. On the other hand, if  $|L(v)| \leq \log_{\alpha} \frac{\ell^{\top}(v)}{\ell^{\perp}(v)}$ , then pruning entries in  $L(v)$  tends to be ineffective, in which case it is more preferable to omit pruning in  $L(v)$  in order to enable aggressive pruning at other vertices.

After that, for an entry that is not pruned, the algorithm continues to extend the corresponding path  $P$ , by adding one more edge  $e = (v, v')$  (Lines 14-15). If the resulting path  $P_{new}$  satisfies the cost constraint,  $\alpha$ -Dijk creates a new entry and inserts it into  $H$ . The surrogate value for  $P_{new}$  is computed by adding  $\tau$  and the edge surrogate value  $\tau(e)$  (Line 17), obtained as follows. If edge  $e$  is an original edge in the input graph, then  $\tau(e)$  is simply the length of  $e$ . Otherwise, i.e.,  $e$  is added during the construction of the overlay graph which corresponds to a path in the original graph,  $\tau(e)$  is the surrogate value corresponding to that path, as we clarify in the next subsection. Finally, after  $H$  depletes,  $\alpha$ -Dijk retrieves the entry in  $L(t)$  with the smallest length, and returns the corresponding path as the the answer to the  $\alpha$ -CSP query.

We demonstrate how  $\alpha$ -Dijk works with the following example.

**EXAMPLE 8.** Consider an  $\alpha$ -CSP query on graph  $G$  in Figure 1, with origin  $v_1$ , destination  $v_5$ , cost constraint  $\theta = 7$ , and  $\alpha = 1.35$ . First,  $\alpha$ -Dijk computes  $\ell^{\perp}(v)$  and  $\ell^{\top}(v)$  for all vertices in  $G$ , and obtains  $\ell^{\perp}(v_2) = \ell^{\top}(v_2) = 2$ ,  $\ell^{\perp}(v_3) = 3$ ,  $\ell^{\top}(v_3) = 4$ ,  $\ell^{\perp}(v_4) = \ell^{\top}(v_4) = 4$ , and  $\ell^{\perp}(v_5) = 4$ ,  $\ell^{\top}(v_5) = 7$ .

$\alpha$ -Dijk initializes heap  $H$  with a single entry  $\rho_0 = \langle P_0, 0 \rangle$ , where  $P_0 = \langle v_1 \rangle$ . Then, two paths  $P_1 = \langle (v_1, v_2) \rangle$  ( $c_1 = 1$ ,  $\ell_1 = 2$ ) and  $P_2 = \langle (v_1, v_3) \rangle$  ( $c_2 = 3$ ,  $\ell_2 = 4$ ) are extended from  $P_0$ . Hence, the algorithm inserts into  $H$  entries  $\langle P_1, \tau_1 = 2 \rangle$  and  $\langle P_2, \tau_2 = 3 \rangle$ . Next,  $\rho_1 = \langle P_1, \tau_1 = 2 \rangle$  is popped from  $H$ . As  $L(v_2)$  is empty,  $\rho_1$  is inserted into  $L(v_2)$ . Meanwhile, as  $|L(v_2)| > \log_{\alpha} \frac{\ell^{\top}(v_2)}{\ell^{\perp}(v_2)} = \log_{1.35} \frac{2}{2}$ ,  $\alpha$ -Dijk sets  $\tau_1 = 2$ .

Afterwards,  $P_3 = \langle (v_1, v_2), (v_2, v_3) \rangle$  ( $c_3 = 4$ ,  $\ell_3 = 3$ ),  $P_4 = \langle (v_1, v_2), (v_2, v_4) \rangle$  ( $c_4 = 5$ ,  $\ell_4 = 4$ ), and  $P_5 = \langle (v_1, v_2), (v_2, v_5) \rangle$  ( $c_5 = 5$ ,  $\ell_5 = 6$ ) are extended from  $P_1$ . As a result, three entries  $\langle P_3, \tau_3 = 3 \rangle$ ,  $\langle P_4, \tau_4 = 4 \rangle$ , and  $\langle P_5, \tau_5 = 6 \rangle$  are pushed into  $H$ .

After that,  $\rho_2 = \langle P_2, \tau_2 \rangle$  is popped from  $H$ , and is inserted into  $L(v_3)$ . As  $|L(v_3)| > \log_{\alpha} \frac{\ell^{\top}(v_3)}{\ell^{\perp}(v_3)}$ ,  $\tau_2$  is set to  $\max\{\ell_2/\alpha, \ell^{\perp}(v_3)\}$ , which is 3. Then, path  $P_6 = \langle (v_1, v_3), (v_3, v_5) \rangle$  ( $c_6 = 6$ ,  $\ell_6 = 5$ ) is extended from  $P_2$  and an entry  $\langle P_6, \tau_6 = 4 \rangle$  is pushed into  $H$ .

Subsequently,  $\rho_3 = \langle P_3, \tau_3 \rangle$  is popped from  $H$ . Notice that  $L(v_2)$  contains an entry pertinent to path  $P_2$ , and  $\tau_2 \leq \tau_3$ . Therefore,  $P_3$  is pruned. Afterwards,  $\rho_4 = \langle P_4, \tau_4 \rangle$  and  $\rho_5 = \langle P_5, \tau_5 \rangle$  are popped from  $H$ , and  $\rho_4$  and  $\rho_5$  are inserted into  $L(v_4)$  and  $L(v_5)$ , respectively. Since  $|L(v_4)| > \log_{\alpha} \frac{\ell^{\top}(v_4)}{\ell^{\perp}(v_4)}$ ,  $\tau_4$  is set to 4. Similarly,  $\tau_5$  is set to  $\max\{\ell_5/\alpha, 4\} = 4.44$ . For path  $P_4$ , a path  $P_7 = \langle (v_1, v_2), (v_2, v_4), (v_4, v_5) \rangle$  ( $c_7 = 7$ ,  $\ell_7 = 7$ ) is extended from it, and an entry  $\langle P_7, \tau_7 = 7 \rangle$  is pushed into  $H$ .

Next,  $\rho_6 = \langle P_6, \tau_6 \rangle$  is popped from  $H$ , and is compared with the only entry in  $L(v_5)$ . Notice that  $\tau_6 = 4 < \tau_5 = 4.4$ , which means that the entry cannot be pruned. Hence,  $\langle P_6, \tau_6 \rangle$  is added into  $L(v_5)$ . Then, it updates  $\tau_6$  to  $\max\{\ell_6/\alpha, \ell^{\perp}(v_5)\} = 4$ . Finally,  $P_7$  is pruned as  $\tau_7 > \tau_5$ . Thus,  $\alpha$ -Dijk returns  $P_6$  with cost  $c_6 = 6$ , length  $\ell_6 = 5$  as the  $\alpha$ -CSP query answer.  $\square$

**Theoretical Analysis.** The following theorem establishes the correctness of  $\alpha$ -Dijk.

---

### Algorithm 5: Overlay Graph Construction

---

**input** : A partition  $\mathcal{T} = \{G_1, G_2, \dots, G_k\}$  of  $G$   
**output**: An overlay graph  $G^{\circ} = (V^{\circ}, E^{\circ})$  of  $G$

- 1 Let  $E^{\circ} = \emptyset$ , and  $V^{\circ}$  be the set of boundary vertices defined by  $\mathcal{T}$ ;
- 2 **for each** subgraph  $G_i \in \mathcal{T}$  **do**
- 3     **for each** boundary vertex  $v^{\circ}$  in  $G_i$  **do**
- 4         Feed  $v^{\circ}$  and  $G_i$  as input to the single-source  $\alpha$ -Dijk, which outputs an entry list  $L(v)$  for each vertex  $v$  in  $G_i$ ;
- 5         **for each** boundary vertex  $v$  in  $G_i$  **do**
- 6             Let  $S = \emptyset$ ;
- 7             **for each** entry  $\rho$  in  $L(v)$  in ascending order of  $c(\rho)$  **do**
- 8                 Let  $\rho'$  be the last entry inserted into  $S$ ;
- 9                 **if**  $S$  is empty or  $\ell(\rho') > \alpha \cdot \tau(\rho)$  **then**
- 10                     Insert  $\rho$  into  $S$ ;
- 11                 **else**
- 12                      $\tau(\rho') = \min\{\tau(\rho'), \tau(\rho)\}$ ; //  $\rho$  is pruned
- 13             **for each** entry  $\rho$  in  $S$  **do**
- 14                 Insert an edge  $e^{\circ} = (v^{\circ}, v)$  into  $E^{\circ}$ , with  $c(e^{\circ}) = c(\rho)$ ,  $\ell(e^{\circ}) = \ell(\rho)$ , and  $\tau(e^{\circ}) = \tau(\rho)$ ;
- 15 **return**  $G^{\circ} = (V^{\circ}, E^{\circ})$ ;

---

**THEOREM 1.** For each vertex  $v$ , let  $L(v)$  be the entry list of  $v$  when  $\alpha$ -Dijk terminates. Then, for any path  $P$  from  $s$  to  $v$  with a cost smaller than  $\theta$ , there exists an entry  $\rho$  in  $L(v)$  with a cost at most  $c(P')$  and a length at most  $\alpha \cdot \ell(P')$ .  $\square$

Next, we discuss the time complexity of our  $\alpha$ -Dijk algorithm. For each vertex  $v$ , the number of stored entries in  $L(v)$  is bounded by  $\ell_{max} \cdot n$ . When adding labels for outgoing edges of each vertex  $v$ , it incurs at most  $d_v \cdot \ell_{max} \cdot n$  labels, where  $d_v$  is the out-degree of  $v$ . To sum up with, there are at most  $\ell_{max} \cdot n \cdot m$  labels added in the whole procedure. To insert / pop  $\ell_{max} \cdot n \cdot m$  entries into the heap, it requires  $\mathcal{O}(\log(\ell_{max}n))$  time for each operation, ending up with  $\mathcal{O}(\ell_{max} \cdot m \cdot n \cdot \log(\ell_{max} \cdot n))$  time complexity.

## 5. COLA IMPLEMENTATION

This section details the implementation of COLA. Section 5.1 clarifies the algorithm for building the overlay graph. Section 5.2 describes the construction of the COLA index.

### 5.1 Overlay Graph Construction

Given a partitioning  $\mathcal{T} = \{G_1, G_2, \dots, G_k\}$  of  $G$ , we construct an overlay graph  $G^{\circ} = (V^{\circ}, E^{\circ})$  defined in Definition 3 using  $\alpha$ -Dijk with two minor modifications: (i) there is no cost constraint, i.e.,  $\theta = +\infty$  and (ii) instead of a path, the algorithm returns the entry list  $L(v)$  for every vertex  $v$ . We refer to this modified algorithm as *single-source  $\alpha$ -Dijk*. To simplify our notations, in the following we use  $c(\rho)$ ,  $\ell(\rho)$  and  $\tau(\rho)$  to denote the path cost, path length and pruning surrogate of an entry  $\rho \in L(v)$ , respectively.

Algorithm 5 shows the pseudo-code of our overlay graph construction algorithm. Initially, the algorithm sets  $V^{\circ}$  to the set of all boundary vertices defined by  $\mathcal{T}$ , and  $E^{\circ}$  to empty (Line 1). After that, it processes each subgraph  $G_i$  (Lines 2-14). In particular, for each boundary vertex  $v^{\circ}$  in  $G_i$ , it invokes the single-source  $\alpha$ -Dijk to compute an entry list  $L(v)$  for each vertex  $v$  in  $G_i$  (Line 4). If  $v$  is also a boundary vertex in  $G_i$ , then some of the entries in  $L(v)$  may be converted into edges between  $v^{\circ}$  and  $v$  in  $E^{\circ}$  (Lines 5-14), as follows. First, the algorithm creates a set  $S = \emptyset$  for storing entries (Line 6). Then, it inspects the entries in  $L(v)$  in ascending order of their costs, and compares each entry  $\rho$  with the last entry  $\rho'$  inserted into  $S$ . If  $\rho'$  does not exist (i.e.,  $S$  is empty) or  $c(\rho') \leq c(\rho)$  or  $\ell(\rho') \leq \alpha \cdot \tau(\rho)$ , then the algorithm inserts  $\rho$

into  $S$  as an entry to be converted into an edge in  $E^\circ$  (Lines 9-10). Otherwise, the path represented by  $\rho$  must be  $\alpha$ -dominated by  $\rho'$ , and hence, the algorithm omits  $\rho$ , and modifies  $\rho'$  to set  $\tau(\rho') = \min\{\tau(\rho'), \tau(\rho)\}$  (Lines 11-12). The change of  $\tau(\rho')$  is important to ensure that the resulting graph  $G^\circ$  satisfies Definition 3. After all entries in  $L(v)$  are examined, the algorithm retrieves each entry  $\rho$  that has been inserted in  $S$ , and converts it into an edge  $e^\circ \in E^\circ$  with  $c(e^\circ) = c(\rho)$  and  $\ell(e^\circ) = \ell(\rho)$ . In addition, we define the surrogate value of  $e^\circ$  as  $\tau(e^\circ) = \tau(\rho)$ , which is used in our COLA index construction, clarified in the next subsection. Once all subgraphs in  $\mathcal{T}$  are processed, the algorithm terminates and returns  $G^\circ = (V^\circ, E^\circ)$ . We have the following theorem for Algorithm 5.

**THEOREM 2.** *Algorithm 5 correctly constructs an overlay graph that satisfies Definition 3.*  $\square$

## 5.2 Labeling Index Construction

This subsection details the construction of the COLA index. Note that the index structure is not unique, and there are various ways to build it. In our implementation, we apply a standard technique in the literature of *2-hop labeling* (e.g., [4, 7, 33]) for conventional shortest paths, which introduces a ranking function  $r$  of all vertices in  $G^\circ$ , whose values reflect the relative importance of the vertices. Then, for each  $v^\circ \in G^\circ$ , we require that (i) each entry in  $B_{in}(v^\circ)$  is a path  $P_i$  originating at a vertex  $w$  that has the highest rank among all vertices in  $P_i$ , and symmetrically, (ii) each entry in  $B_{out}(v^\circ)$  is a path  $P_o$  ending at a vertex  $w$  that has the highest rank among all vertices in  $P_o$ . To reduce memory consumption, in each entry we can substitute a full path with a tuple  $\langle v, c, l, v_{pred} \rangle$ , where  $c$  and  $l$  are the cost and length of the path and  $v$  and  $v_{pred}$  are the last and second-to-last vertices, respectively.

Similar to conventional 2-hop labeling, the choice of the ordering plays an important role for the effectiveness of the index. We follow a similar approach as in previous work [7], and will discuss the details in Section 5.3. We further define the rank of a path as follows.

**DEFINITION 5 (RANK OF A PATH).** *Let  $P$  be an arbitrary path in  $G^\circ$ . The rank  $r(P)$  of  $P$  is the highest rank among all vertices in  $P$ .*  $\square$

Our index construction algorithm runs in iterations. After finishing  $i$  iterations, the labels constructed in  $B_{out}$  and  $B_{in}$  guarantee that for any path  $P$  from  $u$  to  $v$  ( $u, v \in G^\circ$ ) whose rank is no more than  $i$ , there exists a label entry  $\rho_u \in B_{out}(u)$  corresponding to a path  $P_1$ , and a label entry  $\rho_v \in B_{in}(v)$  concerning a path  $P_2$  in  $G^\circ$  such that  $c(\rho_u) + c(\rho_v) \leq c(P)$  and  $\ell(\rho_u) + \ell(\rho_v) \leq \alpha \cdot \ell(P)$ . In other words, by concatenating  $P_1$  and  $P_2$ , we find a path that  $\alpha$ -dominates  $P$ .

Algorithm 6 shows the procedure for the index construction. It produces indices in  $n$  iterations. We explain how labels in  $B_{in}$  are computed, and  $B_{out}$  can be explained similarly. In the  $i$ -th iteration, the vertex  $u_i$  with  $r(u_i) = i$  is selected. A modified version of  $\alpha$ -Dijk is invoked to produce a set of label lists  $L(v)$  for  $v \in G^\circ$ . The main modification is that: when deriving the lower bound of the length  $\ell_\perp(v)$  from  $u_i$  to  $v$ , it requires to use the  $\tau$  value of a path instead of the length of the path. The main reason is that  $G^\circ$  is a simplified version of  $G$  to capture the paths from  $u_i$  to  $v$ , and the minimum length path  $P$  in  $G$  might not be preserved in  $G^\circ$ ; however, the  $\tau$  value of a path denotes the lower bound of the minimum length path that it has pruned, indicating the  $\tau$  value of the minimum- $\tau$  path is a lower bound of the length from  $u_i$  to  $v$ .

Meanwhile, when a label entry  $\rho = \langle v, c, \ell, \tau, v_{pred} \rangle$  is popped from the heap  $H$ . A dominance checking procedure is proceeded, which checks if a path with cost  $c$  and length  $\tau$  will be dominated by existing labels. In particular, it uses the labels constructed in the previous  $i - 1$  iterations to do a dominance checking. Specifically, it checks if there exists a label  $\rho^+ \in B_{out}(u_i)$  and a label  $\rho^- \in B_{in}(v)$  such that  $c(\rho^+) + c(\rho^-) \leq c$  and  $\ell(\rho^+) + \ell(\rho^-) \leq \tau$ . If the answer is yes, this label entry can be skipped in this iteration without affecting the query correctness as will be proved in Lemma 3.

When the  $\alpha$ -Dijk's algorithm finishes, it outputs the label list into  $L$ . For each vertex  $v$ , the most simple approach is to subsequently select the label from  $L(v)$  with the smallest cost and that cannot be  $\alpha$ -dominated by previous selected labels. This approach, however, cannot efficiently help path reconstruction. In particular, given an in-label entry  $\rho^- = \langle w, c, \ell, v_{pred} \rangle$ , and denote the path pertinent to this label as  $P$ , the path reconstruction requires the knowledge of the label pertinent to the sub-path of  $P$  that starts from  $u$  to  $u_i$ . We denote this label as the *child* of  $\rho^-$ . This requirement, however, cannot be guaranteed by the simple selection strategy. Instead, we adopt a more advanced approach to guarantee that, for each out-label in  $B_{out}$  (resp. in-label in  $B_{in}$ ), its child is always in  $B_{out}$  (resp.  $B_{in}$ ).

Observe that, in the  $i$ -th iteration, if a label  $\langle w, c, \ell, \tau, v_{pred} \rangle$  is stored in  $L$ , there must exist a label  $\langle v_{pred}, c', \ell', \tau', v'_{pred} \rangle$  stored in  $L$ . This is due to the traversing order of a path in  $\alpha$ -Dijk. If a path  $P$  from  $u$  to  $w$  has been traversed, its sub-path from  $u$  to  $w$ 's predecessor must also be traversed. Note that  $c' \leq c$  and  $\ell' \leq \ell$ . By storing the labels in descending order of the cost, with ties broken by the length, we can guarantee that each label is ordered before its child. Denote the sorted label as  $L_i$ . This procedure is proceeded as shown in Lines 8-9 of Algorithm 6. After that, we maintain a hash-table  $T$  to record the child of the added labels. Besides, we record the last pruned label concerning  $v$ , and denote it as  $\rho_{prune}(v)$ . Note that  $\rho_{prune}(v)$  is initially set to the first label concerning  $v$  in  $L_i$ . Then in order of the sorted list  $L_i$ , we inspect whether (i) the current label  $\rho = \langle w, c, \ell, \tau \rangle$  satisfies that it is a child of a previous stored label, and (ii) whether the next label  $\rho_p$  concerning  $w$  satisfies that  $\ell(\rho_p) \leq \tau(\rho_{prune}) \cdot \alpha$ . If both conditions are not satisfied (Line 13-15), this label can be pruned safely. Otherwise, a label  $\langle u_i, c, \ell, v_{pred} \rangle$  is inserted into  $B_{in}(w)$ ; its child is inserted into the hash-table  $T$ ; and  $\rho_{prune}$  is updated to  $\rho_p$  as shown in Lines 17-20.

With the guarantee that a label's child will always be in COLA index. Then we can reconstruct the path as follows. Given a label  $\rho = \langle w, c, \ell, v_{pred} \rangle \in B_{in}(v)$ , we can find its child label  $\rho' = \langle w, c', \ell', v'_{pred} \rangle \in B_{in}(v_{pred})$  with  $\mathcal{O}(1)$  time, by storing a pointer to  $\rho'$  in  $\rho$ . Afterwards, we can then recursively unfold the edges in  $G^\circ$  by checking  $\mathcal{O}(k)$  labels, where  $k$  is the number of edges in  $P^\circ$ . Afterwards, we apply a bidirectional traversal to find the path in  $G$  corresponding to each edge in  $P^\circ$ .

When the label entries are updated, a backward version of  $\alpha$ -Dijk is proceeded, and entries are added in to  $B_{out}$ . After that, the input graph of the next iteration is updated by removing vertex  $u_i$  and its incident edges. After  $n$  iteration, the algorithm finishes and ends up with two label sets  $B_{out}$  and  $B_{in}$  that guarantees the query correctness as shown in Lemma 3.

**LEMMA 3.** *Algorithm 6 correctly produces a COLA index that satisfies Definition 4.*  $\square$

## 5.3 Vertex Ordering

Recall that every in-label (resp. out-label) of each vertex  $v$  in the COLA index is a skyline path  $P$  originating (reps. ending) at a

---

**Algorithm 6: Index Construction**

---

**input** : An overlay graph  $G^\circ$ , and a rank for all vertices in  $G^\circ$   
**output**: a COLA index  $\mathcal{L}$

- 1 Create a min-heap  $H$  that (i) stores entries in the form of  $\langle v, c, \ell, \tau, v_{pred} \rangle$ , and (ii) sorts the entries in ascending order of their  $c$  values, with ties broken by their  $\ell$  values;
- 2 Let  $G_r^\circ = G^\circ$ ;
- 3 **for**  $i = 1, 2, \dots, n$  **do**
- 4   Let  $u_i$  be the vertex whose rank is  $i$ ;
- 5   Invoke Algorithm 4 on  $G_r^\circ$  except that (i) to derive  $\ell^\perp(v)$ , it requires to compute the path  $P$  with minimum  $\tau$  value instead of minimum-length; (ii) when a label entry  $\rho = \langle w, c, \ell, \tau, v_{pred} \rangle$  is popped from the heap, it invoke a dominance checking procedure to see if a path with cost  $c$  and length  $\tau$ , will be dominated by existing labels with a linear label scanning similar to Algorithm 2; (iii) output the label list  $L$ ;
- 6   Let  $L_i$  be an empty list;
- 7   **for**  $v \in V^\circ$  whose  $L(v)$  is not empty **do**
- 8     Insert all entries in  $L(v)$  into  $L_i$ ;
- 9   Sort all entries in  $L_i$  in descending order of cost, with ties broken by length;
- 10   Initiate a hash-table  $T$  to record preserved labels, and the key is of the form  $\langle w, c, \ell \rangle$ ;
- 11   Record the last label  $\rho_{prune}(v)$  pruned with  $\alpha$ -dominance, initially it is set to the first label in  $L_i$  concerning  $v$ ;
- 12   **for each** label entry  $\rho = \langle w, c, \ell, \tau, v_{pred} \rangle$  in  $L_i$  **do**
- 13     Let  $\rho_p \in L_i$  be the next label concerning  $w$  after  $\rho$ ;
- 14     **if**  $\langle w, c, \ell \rangle \notin T$  **and**  $\ell(\rho_p) \leq \tau(\rho_{prune}(w)) \cdot \alpha$  **then**
- 15       **continue**;
- 16     **else**
- 17       Add  $\rho' = \langle u_i, c, \ell, v_{pred} \rangle$  into  $B_{in}(w)$ ;
- 18       Let  $P$  be the path pertinent to  $\rho'$ , and let  $e$  be the edge that points from  $v_{pred}$  to  $w$ ;
- 19       Insert an entry  $\langle v_{pred}, c - c(e), \ell - \ell(e) \rangle$  into  $T$ ;
- 20       Update  $\rho_{prune}$  to be  $\rho_p$ ;
- 21     Repeat the above procedure with a backward modified  $\alpha$ -Dijk and add entries into  $B_{out}$ ;
- 22     Update  $G_{i+1}^\circ$  by removing the incident edges to  $u_i$ ;
- 23 **return**  $B_{in}, B_{out}$ ;

---

vertex  $v'$  that has the highest rank among all vertices in  $P$ , i.e., the rank of  $v'$  is higher than the rank of  $v$ . In other words, the ordering of vertices would affect the total number of labels in COLA, which is similar to the situation in conventional 2-hop labeling, e.g., [4, 7, 33]. Note that, computing a 2-hop labeling with minimal size is NP-hard. However, the case in COLA is even worse, since COLA index is built on the overlay graph, which allows multiple edges from a vertex to another. Besides, given an ordering of vertices, it is highly costly to the exact index size of COLA, whose cost is equivalent to building the index.

To produce a COLA index with small size and also overcome the computational difficulty, we present in this section a sampling based approach that works in  $n$  iterations, each of which orders one vertex of of the overlay graph  $G^\circ$ . Specifically, in each iteration, we first estimate the number of labels pertinent to each vertex, and then select the vertex  $v$  to order higher than the rest if  $v$  results in the smallest estimated index size. Once a vertex  $v$  is ordered, we remove  $v$  from  $G^\circ$ , and continue to order the rest vertices in residual graph  $G_r^\circ$  until  $G_r^\circ$  is empty.

In order to estimate the number of labels pertinent to each vertex  $v$ , we first define the *coverage* of  $v$  as the set  $\mathcal{P}_v$  of skyline paths  $P$  in  $G_r^\circ$  that goes through  $v$ , denoted by  $C(v) = |\mathcal{P}_v|$ . Intuitively, the larger coverage of the vertex  $v$  in  $G_r^\circ$ , the more index space would be saved if  $v$  is ranked higher than the rest vertices in  $G_r^\circ$ , as we can avoid the computation of the paths in  $\mathcal{P}_v$  in the

subsequent iterations. However, it is highly expensive to compute all skyline paths in  $G_r^\circ$ , especially when  $G_r^\circ$  is sufficiently large. To avoid the deficiency, we devise a sampling based approach to estimate the coverage of  $v$  in  $G_r^\circ$ . To explain, let  $u$  be a randomly sampled node, and  $\beta$  be a randomly sampled number in  $[0, 1]$ . Define  $c(e) \cdot \beta + \ell(e) \cdot (1 - \beta)$  as the *weight* of an edge  $e$ . Then we sample from  $u$  a spanning tree with the minimum total weight. The rationale is that, if a path is a skyline path, then there exists a  $\beta$  in  $[0, 1]$  such that the path has the minimal weight. Hence, we sample  $k$  trees  $T_1, T_2, \dots, T_k$  using the above strategy, where  $k$  is sufficiently large. Let  $C_j(v)$  be the size of subtree with root  $v$  in  $T_j$  ( $1 \leq j \leq k$ ). Based on that, we can estimate the coverage of  $v$  as  $\sum_{j=1}^k C_j(v)/k$ . Then, we order the vertex  $v$  of the largest estimated coverage among vertices in  $G_r^\circ$ . After that, we remove all subtrees rooted at  $v$  from  $T_1, T_2, \dots, T_k$  and update the estimated coverage for the rest vertices respectively.

A larger sample size  $k$  tends to provide a more accurate estimated coverage for each vertex. However, it comes at a higher computational cost. To explain, let  $n_o$  be the number vertices in  $G^\circ$ , then the number of nodes in sampled tree is at most  $n_o$ . Therefore, the time complexity in computing the sampled trees by the above method is  $O(kn_o^2)$ . Let  $m_o$  be the number of edges in  $G^\circ$ . To reduce the cost, we constrain the total number of edges sampled from all trees to be at most  $h \cdot m_o$ , where  $h = 16$  following the setting in previous work [7].

## 6. EXPERIMENTS

This section experimentally evaluates COLA against the current state-of-the-art methods. Section 6.1 explains the experimental settings. Sections 6.2 and 6.3 present evaluation results in terms of query efficiency and indexing overhead, respectively. Section 6.4 provides insights for choosing an appropriate value for  $\alpha$ .

### 6.1 Experimental Settings

All methods are implemented in C++, compiled with full optimizations, and tested on a Linux machine with an Intel Xeon 2.6GHz CPU and 64GB RAM. We repeat each experiment 5 times and report the average results.

**Datasets.** We use 8 real road networks from the 9th DIMACS Implementation Challenge [2] as shown in Table 2. In the datasets, each vertex represents a road junction, and each edge represents a road segment. Among the eight road networks, EU is directed and the others are undirected. Each edge in the dataset contains two attributes: the travel time and the travel distance. Following previous work [27], we use the travel time as the cost, and the travel distance as the length. The dataset sizes vary from small cities to the full USA road networks. Table 2 summarizes the properties of the datasets, where  $|V|$ ,  $|E|$ ,  $|V^\circ|$  and  $|E^\circ|$  are the cardinalities of vertices in the road network, edges in the road network, vertices in the overlay graph (Ref. Section 3 for definition of overlay graph) and edges in the overlay graph, respectively.

**Query sets.** For each dataset, we generate 5 query sets Q1-Q5, each containing 100 queries. Q1-Q5 are generated as follows. First, we randomly choose the query origin  $s$  and destination  $t$  among the vertices of the road networks. Then, we classify the query into one of the 5 query sets, based on the length from  $s$  to  $t$ . Specifically, we first compute a lower bound for the graph diameter (the longest length of all shortest paths) using an existing approximate algorithm [26], which is at most 2 times the value of diameter. Let  $d_{min}$  be this lower bound. We then generate queries as follows: if the length of these two nodes is in range  $[d_{min}/2, +\infty)$ , we add it into Q5; if the distance is in range  $[d_{min}/4, d_{min}/2)$ , we add it into Q4; if the distance is in the range  $[d_{min}/8, d_{min}/4)$ , we add it



dataset	$ V $	$ E $	$ V^\circ $	$ E^\circ $
New York City (NY)	0.3M	0.7M	4.8K	0.2M
Florida (FLA)	1.1M	2.7M	4.2K	0.2M
Northwest USA (NW)	1.2M	2.8M	4.2K	0.2M
Northeast USA (NE)	1.5M	3.9M	7.0K	0.7M
Great Lakes (LKS)	2.8M	6.9M	10.9K	1.8M
Western USA (W)	6.3M	15.2M	7.8K	1.2M
Europe (EU)	18.0M	42.2M	16.9K	9.3M
Full USA (USA)	23.9M	58.3M	17.7K	9.5M

**Table 2: Road networks ( $K=10^3$ ,  $M=10^6$ ).**

into Q3, etc. For each query set, we report the average time for the 100 queries. The differences among results for different query sets reflect the impact of the travel distance.

Next, we clarify the generation of the cost constraint  $\theta$ . For each query, we first compute  $c_{min}$ , the minimum cost of any path, and  $c_{max}$ , the cost of the minimum-length path from  $s$  to  $t$ ; then, we select  $\theta$  uniformly at random from  $[c_{min}, c_{max}]$ . Note that if  $\theta < c_{min}$ , the query cannot return any result; if  $\theta > c_{max}$ , the minimum-length path is always a valid result to the CSP query.

Another important parameter is  $\alpha$ , which determines the approximation guarantee of  $\alpha$ -CSP queries. We view  $\alpha$  as a system parameter rather than part of the query, since  $\alpha$  controls the trade-off between query accuracy and system efficiency, e.g., space consumption, which is more relevant to the system environment than individual queries. In order to find a suitable value for  $\alpha$ , we performed a set of experiments with varying  $\alpha$ . The results shown in in Section 6.4 demonstrate that the value  $\alpha = 1.1$  leads to a good balance among the space consumption, preprocessing time, query efficiency and query accuracy. Hence, in the rest of our experiments, we fix  $\alpha = 1.1$ .

**Methods.** We compare *COLA* against three state-of-the-art methods: *Sky-Dijk* [18], *CP-Dijk* [31] and *CSP-CH* [30], described in Section 2.2. Meanwhile, we also include a most recent *k-shortest path* based solution [27] as our competitor, dubbed as *KSP*. Besides, we also include  $\alpha$ -*Dijk* (see Section 4) in our comparisons, which answers  $\alpha$ -CSP queries without an index.

## 6.2 Query Efficiency

Figure 3 plots the average query execution time (in seconds) for all methods. For brevity, we only show the results on 4 representative datasets: NY, LKS, EU, and USA. On these 4 datasets, we inspect the performance on all 5 query sets Q1-Q5. Note that the y-axis is in logarithmic scale, and we use error bars to present the variations of the query performance for *COLA* and  $\alpha$ -*Dijk*.

The most apparent observation is that regardless of the dataset or query set, our main proposal *COLA* consistently outperforms all other methods by several orders of magnitude. Further, the query time of *COLA* is always within one second, even on the largest network covering the entire USA. Besides, on large directed road networks, e.g., EU, *COLA* still outperforms the existing methods by two orders of magnitude, which demonstrates the effectiveness of our *COLA* index on both directed and undirected road networks.

For state-sized networks, *COLA* always finishes within milliseconds. In contrast, the rest of the methods require at least 1 second to finish, except for a few settings that involve both a small dataset and a query set with very close query origin and destination. For queries with relatively far apart origin / destination pairs, these methods (except for  $\alpha$ -*Dijk*) can take several hours to process one query.

From these observations, we conclude that *COLA* not only outperforms its competitors, but brings down the query processing cost from prohibitively high to a practically low. In other words, *COLA* makes  $\alpha$ -CSP feasible on current hardware.

Besides *COLA*, the next fastest method is our index-free algorithm  $\alpha$ -*Dijk*, again in all settings. The performance gap between  $\alpha$ -*Dijk* and the competitors is also notable, often more than an order of magnitude. Moreover, both *COLA* and  $\alpha$ -*Dijk* demonstrate small variation for the query processing time, which is generally close to the average. In contrast, as shown in [27, 30], existing CSP algorithms vary significantly in terms of query time, and may incur much longer query time than the average. This demonstrates the robustness of our methods to the input query. Interestingly, our index-free  $\alpha$ -*Dijk* outperforms the indexed method *CSP-CH*, which is given the advantage of large amounts of pre-computations. This suggests that exact CSP as a query type may not be practical for large networks; in such situations, it is necessary to relax the query definition.

Comparing the four existing methods, *CSP-CH* outperforms *CP-Dijk* and *Sky-Dijk* in all settings, and outperforms *KSP* for queries with far apart origin / destination pairs. Meanwhile, *KSP* turns out to be very efficient when the origin and destination are close. However, as the distance between the origin and destination increases, the query efficiency degrades rapidly, due to the exponential growth in the number of paths from the origin to the destination. In addition, we observe that the performance improvement of *CP-Dijk* (for  $\alpha$ -CSP) over *Sky-Dijk* (for exact CSP) is negligible, in all settings. This confirms that the *CP-Dijk* (and also other previous work on  $\alpha$ -CSP, as *CP-Dijk* is the current state of the art) focuses on theoretical improvement in terms of asymptotic complexity rather than practical performance.

The impact of the distance between the origin  $s$  and destination  $t$  becomes apparent, once we compare results for different query sets. When they are close to each other (e.g., Q1 and Q2), the query cost for all methods are small, especially on small networks. As  $s$  and  $t$  become farther apart, query costs increase rapidly. This is expected, as the longer the trip is, the more vertices are involved during CSP processing. An important observation is that *COLA*'s performance is robust against varying query sets. This is mainly due to its use of the overlay network. Specifically, *COLA* utilizes the pre-computed distances between boundary vertices of different partitions, and the number of boundary vertices involved slowly increases with the distance between  $s$  and  $t$ , compared to the number of vertices in the network. Besides, the query cost of our  $\alpha$ -*Dijk* algorithm increases slowly, which demonstrates the effectiveness of its adaptive pruning. On the contrary, the other four existing methods scale poorly; for large datasets, they usually require hours to process a single query. Furthermore, on the USA dataset, none of these four methods can answer queries in Q3-Q5 due to prohibitive memory consumption (>64GB).

Finally, we compare results on different datasets. As expected, the larger the network is, the more expensive it is to process a CSP query. Once again, unlike its competitors, the query cost of *COLA* grows slowly with the network size, thanks to its use of the overlay network and the constrained labeling index, whose effects become more pronounced as the dataset becomes larger.

## 6.3 Index Size and Construction Time

Figure 4 illustrates the memory consumption for all methods, before running any query. The results for the non-indexed approaches *Sky-Dijk*, *CP-Dijk*, *KSP* and  $\alpha$ -*Dijk* are simply the size of the road network. On the other hand, the results for *CSP-CH* and *COLA* indicate their respective index sizes. In addition, we also show the size of the overlay graph in *COLA* in the same figure.

The most important observation is that the index size of *COLA* is no more than 5GB on the largest dataset USA and such memory requirement can be easily satisfied on a modern server. Comparing *CSP-CH* and *COLA*, the former uses a smaller amount of memory

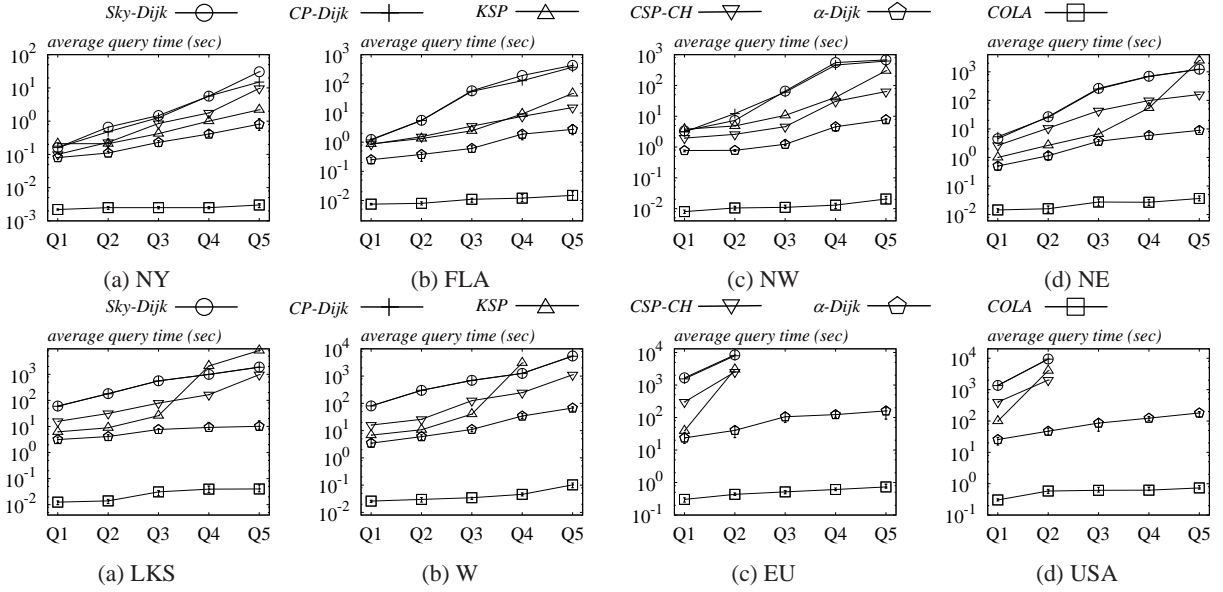


Figure 3: Query efficiency vs. Query sets.

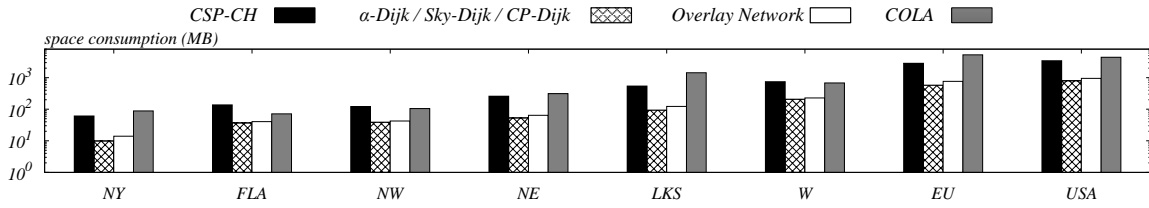


Figure 4: Space consumption.

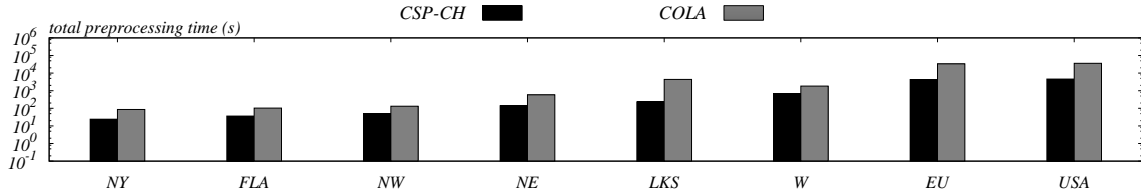


Figure 5: Total preprocessing time.

than the latter, and yet the index size of *CSP-CH* is considerable compared to that of non-indexed methods, i.e., the size of the input graph. Although *COLA* requires a larger index, its memory overhead is affordable, which is more than compensated by its high query performance as shown in Figure 3. The results also show that the size of the overlay graphs is negligible, indicating that the space consumption of *COLA* is mainly attributed to its constrained labeling index.

Figure 5 presents the total pre-processing time of *CSP-CH* and *COLA*. Note that the non-indexed methods are not shown as they do not need pre-processing. Compared to *CSP-CH*, *COLA* takes on average 3x to 4x processing time. Nevertheless, the cost of pre-processing in *COLA* is still modest, i.e., within 12 hours on the largest dataset USA, using a single server. Considering that existing methods require hours to process even one query, the pre-processing cost of *COLA* is worth paying for.

Summarizing the experiments, *COLA* effectively reduces the processing time of  $\alpha$ -CSP queries from hours to sub-second, with moderate index size (no more than 5GB). Hence, it is clearly the method of choice for  $\alpha$ -CSP processing. When an index is not available, we recommend the  $\alpha$ -Dijk algorithm, which might be suitable for applications that do not require fast response. Since all previous methods are prohibitively expensive and  $\alpha$ -CSP has

promising use cases, the proposed methods might become key enablers for new online navigation services based on  $\alpha$ -CSP.

## 6.4 Tuning $\alpha$

In this set of experiments, we evaluate the impact of  $\alpha$  on *COLA*. In particular, we measure the query accuracy and space consumption of *COLA* by varying  $\alpha$  from 1.005 to 1.4. Due to space limitations, we show the results for 4 representative datasets.

Figures 6(a)-(b) show the memory consumption, i.e., the index size, and preprocessing costs of our *COLA*, on FLA, NE, W, and USA datasets. As we can observe, when  $\alpha$  changes from 1.005 to 1.4, both the space consumption and pre-processing time decrease, since a larger  $\alpha$  tends to help the *COLA* labels prune more paths, resulting in a smaller index size.

Note that the impact of  $\alpha$  is more pronounced on pre-processing time than on memory consumption. To explain, the index construction algorithm requires examining a large number of paths (even if the paths are added into label sets), and a larger  $\alpha$  can help prune more paths, and hence can help save more preprocessing time. However, only a subset of the paths traversed are stored in the index, and hence the pruning effect is less significant in terms of space consumption than pre-processing time.

Besides, the query time of *COLA* is relatively insensitive to  $\alpha$ . In particular, when  $\alpha$  decreases from 1.4 to 1.005, the query time of *COLA* only increases by around 1.5x. To explain, the size of

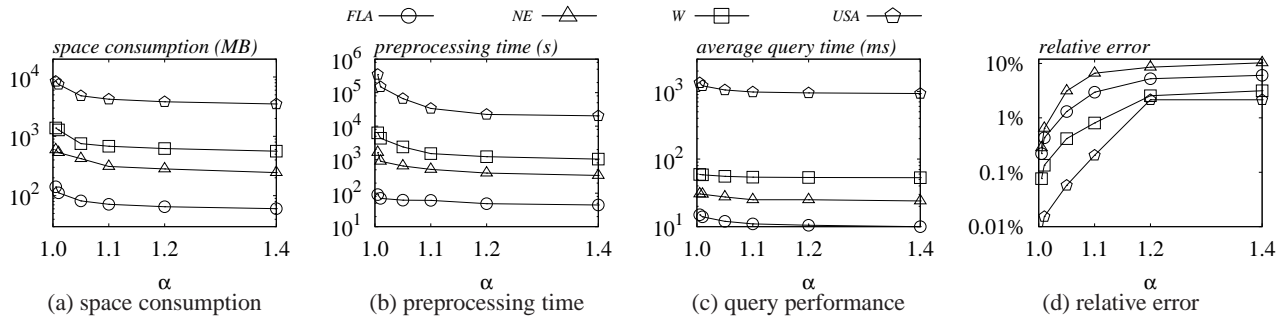


Figure 6: Tuning  $\alpha$  for COLA.

the COLA index is not significantly increased with decreasing  $\alpha$  as shown in the results; and hence the query time is not significantly affected.

In terms of the query accuracy, we use *relative error* of 1000 random queries as the evaluation. More specifically, given a query  $q$ , let  $P^*$  be the solution of the exact CSP query and  $P'$  be an  $\alpha$ -CSP, the relative error of the latter is computed as  $\frac{\ell(P') - \ell(P^*)}{\ell(P^*)}$ . Figure 6(b) shows the relative error of COLA on the same datasets. A higher value of  $\alpha$  leads to a larger relative error. Nevertheless, the relative error is generally smaller than the worst case bound. For example, the relative error of  $\alpha$ -Dijk for  $\alpha = 1.1$ , is around 3% on FLA dataset, which is less than a third of the worst case bound, i.e., 10%. We set  $\alpha$  to 1.1 for COLA since it strikes a good balance among query accuracy, query efficiency, space consumption and preprocessing time.

## 7. CONCLUSIONS

We present COLA, a novel and practical solution for approximate constrained shortest path processing, which can be applied to navigation services where a single criterion on the path is not sufficient to capture the user's requirement. COLA utilizes important properties for real road networks, and applies effective indexing which leads to orders of magnitude reduction in query execution time. Meanwhile, COLA also includes an algorithm,  $\alpha$ -Dijk, which significantly outperforms existing techniques for  $\alpha$ -CSP processing without an index. As future work, we plan to investigate (i) how to avoid reconstruction of indices for different values of  $\alpha$  and (ii)  $\alpha$ -CSP processing in denser graphs compared to road networks.

## 8. REFERENCES

- [1] 9th DIMACS Implementation Challenge. <http://www.dis.uniroma1.it/challenge9/download.shtml>.
- [2] Google Maps. [www.google.com/maps](http://www.google.com/maps).
- [3] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pages 349–360, 2013.
- [4] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. *CoRR*, abs/1504.05140, 2015.
- [5] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.
- [6] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Robust distance queries on massive networks. In *ESA*, pages 321–333, 2014.
- [7] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. F. Werneck. Graph partitioning with natural cuts. In *IPDPS*, pages 1135–1146, 2011.
- [8] D. Delling and D. Wagner. Pareto paths with sharc. In *SEA*, pages 125–136, 2009.
- [9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
- [10] S. Funke, A. Nusser, and S. Storandt. On k-path covers and their applications. *PVLDB*, 7(10):893–902, 2014.
- [11] S. Funke and S. Storandt. Polynomial-time construction of contraction hierarchies for multi-criteria objectives. In *SOCS*, pages 41–54, 2013.
- [12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979.
- [13] R. Geisberger, M. Kobitzsch, and P. Sanders. Route planning with flexible objective functions. In *ALLENEX*, pages 124–137, 2010.
- [14] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.
- [15] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, pages 156–165, 2005.
- [16] G. Y. Handler and I. Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10(4):293–309, 1980.
- [17] P. Hansen. Bicriterion path problems. In *Multiple criteria decision making theory and application*, pages 109–127, 1980.
- [18] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations research*, 17(1):36–42, 1992.
- [19] N. Jing, Y. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Trans. Knowl. Data Eng.*, 10(3):409–432, 1998.
- [20] H. C. Jokschi. The shortest route problem with constraints. *Journal of Mathematical analysis and applications*, 14(2):191–197, 1966.
- [21] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Trans. Knowl. Data Eng.*, 14(5):1029–1046, 2002.
- [22] F. A. Kuipers, A. Orda, D. Raz, and P. V. Mieghem. A comparison of exact and  $\epsilon$ -approximation algorithms for constrained routing. In *NETWORKING*, pages 197–208, 2006.
- [23] D. H. Lorenz and D. Raz. A simple efficient approximation scheme for the restricted shortest path problem. *Oper. Res. Lett.*, 28(5):213–219, 2001.
- [24] K. Mehlhorn and M. Ziegelmann. Resource constrained shortest paths. In *ESA*, pages 326–337, 2000.
- [25] U. Meyer and P. Sanders.  $\Delta$ -stepping: a parallelizable shortest path algorithm. *J. Algorithms*.
- [26] A. Sedeño-Noda and S. Alonso-Rodríguez. An enhanced K-SP algorithm with pruning strategies to solve the constrained shortest path problem. *Applied Mathematics and Computation*, 265:602–618, 2015.
- [27] A. J. V. Skriver and K. A. Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computers & OR*, 27(6):507–524, 2000.
- [28] O. J. Smith, N. Boland, and H. Waterer. Solving shortest path problems with a weight constraint and replenishment arcs. *Computers & OR*, 39(5):964–984, 2012.
- [29] S. Storandt. Route planning for bicycles-exact constrained shortest paths made practical via contraction hierarchy. In *ICAPS*, volume 4, page 46, 2012.
- [30] G. Tsaggouris and C. D. Zaroliagis. Multiobjective optimization: Improved FPTAS for shortest paths and non-linear objectives with applications. *Theory Comput. Syst.*, 45(1):162–186, 2009.
- [31] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *PVLDB*, 5(5):406–417, 2012.



- [32] Y. Yoshida. Almost linear-time algorithms for adaptive betweenness centrality using hypergraph sketches. In *SIGKDD*, pages 1416–1425, 2014.
- [33] X. Yu and H. Shi. Ci-rank: Ranking keyword search results based on collective importance. In *ICDE*, pages 78–89, 2012.

## APPENDIX

### A. PROOFS

**Proof of Lemma 1.** Let  $v_1^\circ$  be the first boundary vertex that  $P_{opt}$  goes through, and  $v_2^\circ$  be the last boundary vertex that  $P_{opt}$  goes through. We consider three sub-paths  $P_1$ ,  $P_2$ , and  $P_3$ , where  $P_1$  is the path from  $s$  to  $v_1^\circ$ ,  $P_2$  is the path from  $v_1^\circ$  to  $v_2^\circ$ , and  $P_3$  is the path from  $v_2^\circ$  to  $t$ . Clearly,  $P_1$  is within  $G_s$ , otherwise, it must go to another subgraph, which indicates that there must exist a boundary vertex  $v^\circ$  such that  $P_1$  goes through. This contradicts to the assumption that  $v_1^\circ$  is the first boundary vertex in  $P_{opt}$ . Similarly,  $P_3$  is within  $G_t$ . Meanwhile, based on the property of overlay graph, there exists a path  $P^\circ$  in  $G^\circ$  such that  $P^\circ$  can  $\alpha$ -dominate  $P_2$ , i.e.,  $c(P^\circ) \leq c(P_2)$ ,  $\ell(P^\circ) \leq \alpha \cdot \ell(P_2)$ . Let  $P'$  be the path by concatenating  $P_1$ ,  $P^\circ$ , and  $P_3$ , then  $c(P') \leq c(P_{opt}) \leq \theta$ , and  $\ell(P') \leq \alpha \ell(P_{opt})$ . This finishes the proof.  $\square$

**Proof of Lemma 2.** From Algorithm 2, it is easy to verify that *LabelJoin* algorithm since it never goes back to an entry previously scanned. So we mainly focus on the correctness proof here. We say a path  $P$  is covered by *COLA* if it can be reconstructed by an out-label  $\rho_o$  and an in-label  $\rho_i$ .

Let  $P$  be the path with cost no more than  $\theta$  and has a minimum length concatenated from  $B_{out}(s)$  and  $B_{in}(t)$ . Then it is an  $\alpha$ -CSP query. The reason is that, if there exists an  $\alpha$ -CSP query, then *COLA* index contains an out-label  $\rho_o$  and an in-label  $\rho_i$  to cover it. Let the path  $P'$  be the  $\alpha$ -CSP path covered by *COLA*. Then as  $P$  is minimum among all paths covered by *COLA*,  $\ell(P') \leq \ell(P)$ . If  $P$  is an  $\alpha$ -CSP query, then  $P'$  will also be an  $\alpha$ -CSP query.

On the other hand, we prove *LabelJoin* can correctly derive  $P'$ . In particular, during the scan of the labelling, if the labels concerns different vertices, then, by concatenating these two labels, we will not be able to derive a path, so it will not miss any  $\alpha$ -CSP. On the other hand, as  $B_{out}(s)$  is sorted in increasing order of cost, and  $B_{in}(t)$  is sorted in decreasing order of cost, if an out-label and an in-label concerning a vertex  $w$ , has a total cost larger than  $\theta$ , then going to the next label in  $B_{in}(v)$  will not miss any path since a previous entry in  $B_{in}(v)$  either has a larger cost, which indicates that the total cost of the labels will also exceeds  $\theta$  and will not be a valid answer. Meanwhile, if a previous label includes another node, it will not be a valid path either.

On the other hand, if the total cost of the two labels does not exceed  $\theta$ , then it compares with the existing path  $P^*$ , if it is larger than  $P^*$ , then it will not be the minimum path, and can be discarded. Then  $B_{out}(s)$  goes to the next label entry. This operation, we prove that will also not miss  $P'$ . In particular, as label entries are stored in ascending of the cost, which indicates the paths are also stored in descending order of the lengths. Then, a previous label entry in  $B_{out}(s)$  will have a larger length, considering the total length with the current label entry in  $B_{in}(t)$ . So in either case, *LabelJoin* will not miss  $P'$ , indicates that it will find the minimum-path covered by  $B_{out}(s)$  and  $B_{in}(t)$ . This finishes the proof.  $\square$

**Proof of Theorem 1.** For convenience, we abuse notation and use  $\tau(\rho)$  to denote the surrogate value of an entry  $\rho$  in  $L(v)$ . Assume that  $P'$  contains  $k$  edges  $e_1, e_2, \dots, e_k$ , such that  $e_i$  ( $i \in [1, k]$ ) points to a node  $v_i$ . We recursively construct  $k$  paths  $P_1, P_2, \dots, P_k$ , and define a surrogate value  $\tau(P_i)$  for each path  $P_i$  as follows. First, we define  $P_1$  as a path that contains only  $e_1$ . Observe that  $L(v_1)$  must contain an entry  $\rho_1$  that represents  $P_1$  (i.e.,  $\rho_1 = \langle v_1, c(P_1), \ell(P_1), *, s \rangle$ ), since such an entry

would be inserted into  $L(v_i)$  right after  $\alpha$ -Dijk pops the first entry  $\langle s, 0, 0, 0, null \rangle$  from the min-heap  $H$ . We set  $\tau(P_1) = \tau(\rho_1)$ .

Assume that  $P_i$  ( $i \in [1, k - 1]$ ) is constructed, and  $L_i$  contains an entry  $\rho_i$  that represents  $P_i$ . Then, right after  $\rho_i$  is popped from  $H$  and inserted into  $L_i$ ,  $\alpha$ -Dijk would insert an entry into  $H$  for each outgoing edge  $e$  of  $v_i$ . Since  $e_{i+1}$  is an outgoing edge of  $v_i$ , one of the entries inserted should be  $\rho' = \langle v_{i+1}, c(P_i) + c(e_{i+1}), \ell(P_i) + \ell(e_{i+1}), \tau(P_i) + \ell(e_{i+1}), v_i \rangle$ . Then, when  $\rho'$  is popped from  $H$ , it is either pruned by an existing entry in  $L(v_{i+1})$  or inserted into  $L(v_{i+1})$ . In either case, there must exist an entry  $\rho_{i+1}$  with a cost at most  $c(P_i) + c(e_{i+1})$  and a surrogate value  $\tau(\rho_{i+1}) \leq \tau(P_i) + \ell(e_{i+1})$ . We set  $\tau(P_{i+1}) = \tau(\rho_{i+1})$ .

With the above constructions, the following equations hold for any  $i \in [1, k - 1]$ :

$$\begin{aligned} c(P_{i+1}) &\leq c(P_i) + c(e_{i+1}); \\ \tau(P_{i+1}) &\leq \tau(P_i) + \ell(e_{i+1}); \\ \ell(P_{i+1}) &\leq \alpha \cdot \tau(P_{i+1}) \leq \alpha \cdot \ell(P_{i+1}). \end{aligned}$$

Consequently, we have

$$\begin{aligned} c(P_k) &\leq c(P_{k-1}) + c(e_k) \leq c(P_{k-2}) + c(e_{k-1}) + c(e_k) \\ &\leq \dots \leq \sum_{i=1}^k c(e_i) = c(P'); \end{aligned} \quad (2)$$

$$\begin{aligned} \tau(P_k) &\leq \tau(P_{k-1}) + \ell(e_k) \leq \tau(P_{k-2}) + \ell(e_{k-1}) + \ell(e_k) \\ &\leq \dots \leq \sum_{i=1}^k \ell(e_i) = \ell(P'); \\ \ell(P_k) &\leq \alpha \cdot \tau(P_k) \leq \alpha \cdot \ell(P'). \end{aligned} \quad (3)$$

Equations (2) and (3) indicate that  $P'$  is  $\alpha$ -dominated by a path  $P_k$  represented by an entry in  $L(v_k)$ . This completes the proof.  $\square$

**Proof of Theorem 2.** Before proving Theorem 2, we first establish the following lemma.

**LEMMA 4.** Let  $s$  and  $v$  be two vertices in a graph  $G_i$ , and  $L(v)$  be the entry list of  $v$  constructed by the single-source  $\alpha$ -Dijk given  $G_i$  and  $s$  as input. Let  $\mathcal{P}$  be the set of all paths from  $s$  to  $v$  in  $G_i$ . Then, for any entry  $\rho \in L(v)$ , there exists a path  $P \in \mathcal{P}$  with  $c(P) = c(\rho)$ ,  $\ell(P) = \ell(\rho)$ , and  $\tau(\rho) \in [\frac{1}{\alpha} \ell(P), \ell(P)]$ . Furthermore, for any path  $P' \in \mathcal{P}$ , there exists an entry  $\rho \in L(v)$  with  $c(\rho) \leq c(P')$  and  $\tau(\rho) \leq \ell(P')$ .  $\square$

Let  $v$  be the destination node and  $G_i$  be the input graph for Algorithm 4, then base on Theorem 1 and how labels are derived, it is guaranteed that each label is constructed from a path. Let this path be  $P$ , and the  $\tau$  value is set to be no smaller than  $\ell(P)$  based on Algorithm 4. In addition, for any path  $P'$ , as the label stores paths that cannot  $\alpha$ -dominate each other from the smallest cost to the largest cost, if a path  $P'$  has cost  $c$ , let  $\rho$  be the label entry for path  $P$  that has cost smaller than  $P'$ , and the next label has cost larger than  $P'$ , then if  $P$  has length such that  $\tau(P) \leq \ell(P')$ , then the requirement is satisfied. Meanwhile, if  $\tau(P) > \ell(P')$ , since  $P'$  has a cost smaller than the label entry next to the one concerning  $P$ , then it should be added into  $L(v)$  based on the Algorithm 4 Lines 9-11. So in either cases,  $P'$  satisfy that there exists a label  $\rho$  such that  $c(\rho) \leq c(P')$  and  $\tau(\rho) \leq \ell(P')$ , which finishes the proof of Lemma 4.

Afterwards, we set the origin as an arbitrary vertex  $v_1 \in G^\circ$  and destination as an arbitrary vertex  $v_2 \in G^\circ$ . Then, it satisfies that for any path  $P$  from  $v_1$  to  $v_2$ , there exists a label in  $L(v_2)$  can  $\alpha$ -dominate  $P$ . Meanwhile, based on how the edges are constructed

(Algorithm 5 Line 9), it is guaranteed that for any path  $P$  from  $v_1$  to  $v_2$ , there exists a stored entry that can  $\alpha$ -dominate  $P$ . As such entries are mapped to an edge  $e^\circ$ , and these entries are pertinent to a path from  $v_1$  to  $v_2$ . As a consequence, the constructed graph satisfies all conditions in Definition 3, which finishes the proof.  $\square$

**Proof of Theorem 3** We prove that after the  $i$ -th iteration, for all paths  $P$  whose rank (ref. Section 5.2)  $r(P) \leq i$ , it will be  $\alpha$ -dominated by a path  $P$  that can be concatenated by an in-label and an out-label.

In the first iteration, let  $u_1$  be the node with  $r(u_1) = 1$ . Since initially  $B_{out}$  and  $B_{in}$  are empty, Algorithm 6 Line 5 condition 2 will not be triggered. The correctness is guaranteed by Theorem 1.

Consider the  $i$ -th iteration, an  $\alpha$ -Dijk will be started from vertex  $u_i$  with  $r(u_i) = i$ . During the traversal, if a path  $P$  from  $u_i$  to  $v$  is pruned by  $\alpha$ -Dijk, the subsequent correctness can be guaranteed by 1. However, if a path is pruned by label checking, i.e., Line 5 condition 2, then there exists an out-label in  $B_{out}(u_i)$  and an in-label in  $B_{in}(v)$  such that their concatenated path  $P'$  satisfy that  $c(P') \leq c(P)$  and  $\ell(P') \leq \tau(P)$ . In this case, consider any path  $P^+$  that contains  $P$  as a sub-path. We prove that for any such paths  $P^+$ , there will exist some label that can  $\alpha$ -dominate  $P^+$ .

Denote the path corresponding to the labels that prune  $P$  as  $P_\alpha$ ,

clearly  $r(P_\alpha) < i$ . For each path  $P^+$ , we divide them into two sub-paths with  $P_1 = P$ , and  $P_2$  is the remaining sub-path. Let  $P_\alpha^+$  be the path by concatenating  $P_\alpha$  and  $P_2$ . Then it is guaranteed that,  $c(P_\alpha^+) \leq c(P^+)$ , and  $\ell(P_\alpha^+) \leq \ell(P^+)$ . Based on the induction assumption, path  $P_\alpha^+$  satisfies that there exist a label in  $B_{out}$  and a label in  $B_{in}$ , such that their corresponding path  $P_x$  satisfy that  $c(P_x) \leq c(P_\alpha^+)$ , and  $\ell(P_x) \leq \alpha \cdot \ell(P_\alpha^+) \leq \alpha \cdot \ell(P^+)$ . In other words, for any path  $P^+$  derived by the above method, there will exist a path pertinent to an out-label and an in-label that can  $\alpha$ -dominate  $P^+$ . So pruning such paths will be safe, and will not affect the correctness.

Consider any path  $P$  whose rank is  $i$ , based on the above construction, for the sub-path  $P_1$  from  $u_i$  to the ending node of  $P$ , there exists a label  $\rho$  that will be stored such that  $c(\rho) \leq c(P_1)$  and  $\ell(\rho) \leq \alpha \cdot \ell(P_1)$ . As a reversed version of  $\alpha$ -Dijk is applied. Similarly, there exists a label  $\rho'$  such that, for the sub-path  $P_2$  from the source of  $P$  to  $u_i$ , such that  $c(\rho') \leq c(P_2)$  and  $\ell(\rho') \leq \alpha \cdot \ell(P_2)$ . Adding them together, we can conclude that for any path  $P$  whose rank is  $i$ , it will be  $\alpha$ -dominated by a path  $P$  concatenated from an in-label and an out-label.

This indicates that after  $n$  iterations, the label constructed will satisfy Definition 4, which finishes the proof.  $\square$