Accelerating Skyline Path Enumeration with a Core Attribute Index on Multi-attribute Graphs

YUANYUAN ZENG, Chinese University of Hong Kong, Shenzhen, China
YIXIANG FANG*, Chinese University of Hong Kong, Shenzhen, China
WENSHENG LUO, Chinese University of Hong Kong, Shenzhen, China
CHENHAO MA*, Chinese University of Hong Kong, Shenzhen, China

As a building block of many graph-based areas, the *s-t* path enumeration problem aims to find all paths between *s* and *t* by satisfying a given constraint, e.g., hop numbers. In many real-world scenarios, graphs are multi-attribute, where vertices and edges are associated with numerical attributes, such as expense or distance in road networks. However, existing methods have not fully leveraged all attributes in *s-t* path analysis. Hence, in this paper, we study the problem of skyline path enumeration, which aims to identify paths that balance multiple attributes, ensuring that no skyline result is dominated by another, thus meeting diverse user needs. To efficiently tackle this problem, we design a task-oriented core attribute index, called CAI, to rule out all redundant vertices and edges not located in any skyline path. Additionally, we introduce a hop-dependency label propagation strategy to construct the CAI index in parallel, improving the indexing process. Based on this index, we further design a CAI-based querying strategy that reduces fruitless explorations between candidate vertices not in the same skyline path, significantly optimizing query processing time. Experimental evaluations on fifteen real-world graphs show that CAI outperforms existing methods by up to four orders of magnitude in speed while demonstrating enhanced scalability and well-bound memory costs.

CCS Concepts: • **Computing methodologies → Shared memory algorithms**; • **Mathematics of computing → Paths and connectivity problems**.

Additional Key Words and Phrases: Multi-attribute graph, Path enumeration, Parallel computing, Skyline path

ACM Reference Format:
Yuanyuan Zeng, Yixiang Fang, Wensheng Luo, and Chenhao Ma. 2025. Accelerating Skyline Path Enumeration with a Core Attribute Index on Multi-attribute Graphs. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 124 (June 2025), 26 pages. https://doi.org/10.1145/3725261

1 Introduction

Graphs are fundamental data structures composed of vertices and edges, used to represent complex relationships among entities [6, 7, 11, 23, 24, 37, 39]. Among the core problems in graph analytics, *s-t* path enumeration involves returning all specific paths between a source node *s* and a target node *t* that satisfy predefined constraints. This problem serves as a building block in various graph-based domains [5, 8, 17, 29, 33, 40]. For example, in E-commerce, transaction loops can indicate potential fraudulent activities among users, and path enumeration helps identify new loops resulting from user transactions [29, 33]; In biological networks, path enumeration aids in analyzing

*Yixiang Fang and Chenhao Ma are the corresponding authors.

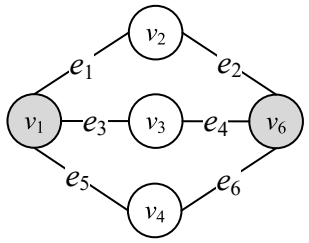
Authors' Contact Information: Yuanyuan Zeng, zengyuanyuan@cuhk.edu.cn, Chinese University of Hong Kong, Shenzhen, Shenzhen, China; Yixiang Fang, fangyixiang@cuhk.edu.cn, Chinese University of Hong Kong, Shenzhen, Shenzhen, China; Wensheng Luo, luowengsheng@cuhk.edu.cn, Chinese University of Hong Kong, Shenzhen, Shenzhen, China; Chenhao Ma, machenhao@cuhk.edu.cn, Chinese University of Hong Kong, Shenzhen, Shenzhen, China.

ACM 2836-6573/2025/6-ART124
https://doi.org/10.1145/3725261

chains of interactions between substances [5]; In knowledge graphs, this technique can enhance the connections between entities for improving the overall quality of the knowledge graph [40].

Motivation. In many real-world applications, the nodes/edges are often associated with numerical attributes which can be obtained from their profiles or the statistical information computed by different network analysis methods (e.g., the degree, PageRank, influence, etc.). For example, the edges in road networks are associated with many attributes, including distance, driving time, the number of traffic lights, and maximum load bearing, etc [14, 18, 26]; In addition, each author in the Aminer scientific collaboration network (aminer.org) has several numerical attributes, including the number of published papers, h-index, activity, diversity, sociability, etc [19]; Such network data is typically modeled as a multi-attribute graph where each node/edge is associated with d ($d \geq 1$) numerical attributes.

In multi-attribute graphs, users often need to consider multiple criteria simultaneously when enumerating s-t paths, e.g., travel time and expenses in road networks. Existing path enumeration approaches [5, 17, 29, 33, 40] often prioritize one numerical attribute for query efficiency, such as enumerating all paths whose length is within a certain threshold. However, this single-attribute focus limits their ability to provide query results that fully represent all relevant attribute features, potentially leading to suboptimal or unsuitable paths in practical applications. For example, the shortest path may be the most expensive, or it may traverse edges with insufficient load-bearing capacity for heavy vehicles. To address these limitations, the skyline operator [18, 26] is introduced to identify all non-dominated paths, where a path is non-dominated if it excels in at least one numerical attribute compared to others, making them highly suitable for varied graph-based applications and enhancing decision-making accuracy [14, 18, 22, 26]. By enumerating skyline paths, users are provided with a comprehensive set of optimal choices, accommodating diverse preferences and requirements.



(a)

Edge	c_1	c_2	c_3	
e_1	4	10	15	p_1
e_2	5	15	20	
e_3	6	10	20	p_2
e_4	9	10	15	
e_5	8	15	20	p_3
e_6	8	15	20	

(b)

Fig. 1. (a) a road network G and (b) the 3-dimensional cost vector of G where c_1, c_2, and c_3 represent distance (*mi*), expense ($), and the maximum load-bearing value (*ton*), respectively.

To effectively compute skyline paths in multi-attribute graphs, it is crucial to define suitable cost functions that accurately capture the attributes of interest [14, 18, 22, 26]. To our knowledge, most skyline query methods adopt the sum of node/edge numerical attributes as the cost function of paths [18, 26] and aim to find the paths equipped with the smallest cost value in at least one dimension [14, 22]. However, we find that this setting does not work for a part of attributes in many real-world applications. For example, the route planning service in the road network aims to provide suitable routes from the start node to the destination node. In the road network G (Figure 1(a)), each edge has a 3-dimensional cost vector: distance c_1 (miles), expense c_2 (dollars), and load-bearing capacity c_3 (tons). The goal is to find skyline paths from v_1 to v_6. (1) Path $p_1 = \langle v_1, v_2, v_6 \rangle$ minimizes distance: $C_1(p_1)$=$c_1(e_1) + c_1(e_2) = 9$ mi. (2) Path $p_2 = \langle v_1, v_3, v_6 \rangle$ minimizes expense: $C_2(p_2) = c_2(e_3) + c_2(e_4) = 20$ \$. However, for load-bearing capacity, the minimum value along a path determines suitability. Both p_1 and p_2 have a maximum load-bearing capacity of 15 tons, i.e., $C_3(p_1) = \min\{c_3(e_1), c_3(e_2)\} = 15$, making them unsuitable for trucks heavier than 15 tons.

Path $p_3 = \langle v_1, v_4, v_6 \rangle$ is better for heavier trucks, as it has a higher load-bearing capacity ($C_3(p_3) = \min\{c_3(e_5), c_3(e_6)\} = 20$ tons). Similarly, assessing the closeness between two individuals is a pivotal concern in social networks [24, 38, 39]. A standard measure for this is the distance separating them, with smaller distances typically indicating greater proximity. Additionally, the tie strength between individuals can be represented as an edge attribute to enhance the precision of such assessments. Compared with the sum operator, the minimal operator is more suitable for capturing the tie strength attribute of a path, as larger minimal values typically signify a closer relationship. Consequently, the skyline path query can be employed to offer users high-quality results by enumerating the paths that are not dominated by others in terms of both distance and tie strength attributes. Based on the above analysis, we introduce two different skyline cost functions to improve the diversity and accuracy of query results. The first cost function, called "sum", is the sum of node/edge numerical attributes in each dimension, while the second cost function, called "min", is the minimal value of node/edge numerical attributes in each dimension. Note that we pursue the minimal and maximal values of the "sum" and "min" cost functions, respectively.

Challenges. The significance of the skyline path enumeration problem in multi-attribute graphs necessitates an effective and efficient resolution. However, efficiently addressing the problem involves overcoming two primary hurdles:

(1) **Unpredictable and explosive search space.** The number of possible paths between two nodes can grow exponentially with the hop number value [5, 29, 33], especially in large-scale graphs. When multiple attributes are considered for skyline computation, the search space becomes even more extensive and unpredictable. Exhaustively exploring all possible paths is computationally infeasible, necessitating strategies to effectively prune the search space without missing any skyline paths.
(2) **Massive redundant and non-skyline path.** Many paths are dominated by others across all attribute dimensions and thus are not part of the skyline. Enumerating these non-skyline paths leads to redundant computations and significant performance degradation. Efficiently identifying and pruning these dominated paths early in the search process is challenging, especially without prior knowledge of the skyline cost values.

Although various approaches have been developed for s-t path enumeration with hop constraints [5, 29, 33], they face significant challenges when extended to skyline path enumeration. Specifically, hop-constrained methods may enumerate redundant paths that are dominated and thus not part of the skyline, leading to unnecessary computations. Moreover, by imposing strict hop limits, these methods can miss longer skyline paths that exceed the hop constraint but are optimal in certain attribute dimensions. Consequently, these methods are inadequate for efficiently identifying all skyline paths, as they both generate redundant results and potentially overlook optimal paths. In addition, existing skyline path query methods [18, 26] focus on finding limited optimal paths with minimal cost values, resulting in incomplete enumerated results. The detailed analysis of the above techniques is listed in Section 3.

Our Approach. Based on the above analysis, we design a task-oriented Core Attribute Index, called CAI, to efficiently enumerate all skyline paths of any query task. By executing a hop-dependency label propagation strategy, this index not only exactly computes the cost values of all types of skyline paths between the source and target nodes, but also identifies the position of each candidate node on the corresponding skyline paths. Specifically, given a query task $q(s, t)$, it is required that each node v maintains a label set $L(v)$ where each label entry in $L(v)$ records the position of v in the corresponding skyline path between s and t. Specifically, the CAI index is equipped with the following excellent properties.

(1) **Redundant vertices/edges elimination.** The CAI index can accurately rule out all redundant vertices and edges that are not located in any skyline path, thus effectively avoiding fruitless explorations of these elements.

(2) **Exact skyline paths computation.** The positions of each node on the corresponding skyline paths can be determined exactly based on the CAI index, thereby avoiding fruitless explorations of any two vertices that are not located in the same skyline path
(3) **Parallel index construction.** The label propagation strategy not only avoids the significant time cost caused by path traversal recursion but can also accelerate the construction of CAI via parallel optimization, largely reducing the time cost.

Contributions. In this paper, we make the following principal contributions:

- We design a hop-dependency label propagation strategy to build the CAI index in parallel. This index can directly rule out all redundant vertices and edges that are not located in any skyline path
- We design a CAI-based searching method to further avoid the fruitless explorations of any two candidate vertices that are not located in the same skyline path, thus reducing the processing time.
- Extensive experiments demonstrate the superior performance of CAI, which achieves up to four orders of magnitude speedup than the state-of-the-art path enumeration methods.

Roadmap. The rest of the paper is organized as follows. Section 2 presents the SkyPE problem and analyzes existing methods. Section 3 describes the extension of existing methods on SkyPE. Section 4 describes the index structure and querying strategy of CAI and the index construction method is introduced in Section 5. Section 6 evaluates the performance of our method. Finally, Section 7 reviews important related work and Section 8 concludes the paper.

2 Preliminary

In this section, we begin by presenting the skyline path enumeration problem (SkyPE). Subsequently, we thoroughly summarize the most related approaches to the skyline path query problem. Table 1 summarizes frequently used notations in this paper.

Table 1. **Notations and meanings.**

Notations	Meanings
$G(V, E, X)$	a κ-attribute undirected graph
(s, t)	a vertex pair
$N(v)$	the neighbor set of v in G
$deg(v)$	the degree of v
$X(e)$	a κ-dimensional vector of the edge e
k	the dimension number of the "min" operator
$p(s, t)$	a path between s to t
$\|p(s, t)\|$	the hop number of $p(s, t)$
$C(p)$	a κ-dimensional cost vector of the path p
$C_l(p)$	the l-th dimension cost value of p
$SP(s, t)$	a set of all skyline paths between s to t
$P^d(s, t)$	a set of paths where $\forall p^* \in P^d$ satisfies $\|p^*\| = d$

2.1 Problem Description.

Let $G(V, E, X)$ be a multi-attribute undirected graph where V and $E \subseteq V \times V$ are sets of n vertices and m edges respectively, and X is a set of κ-dimensional vectors. Here, each edge $e(u, v) \in E$ is associated with a κ-dimensional real-valued vector denoted by $X(e(u, v)) = [x_1(e), \cdots, x_\kappa(e)]$. For simplicity, $X(e(u, v))$ is abbreviated as $X(u, v)$. $N(v) = \{u | e(u, v) \in E\}$ is the neighbor set of v in G. $deg(v) = |N(v)|$ denotes the degree of v. We represent a path between s and t as $p(s, t) = \langle v_0 = s, \cdots, v_j, \cdots, v_k = t \rangle$, where $e(v_i, v_{i+1}) \in E$ for $i \in [0, k-1]$ and $p(s, t) = p(s, v_j) \parallel p(v_j, t)$. Here,

$p(s, v_j)$ is a subpath in $p(s, t)$, denoted as $p(s, v) \subset p(s, t)$. $|p(s, t)|$ denotes the hop number of $p(s, t)$ and $p(s, t)$=$\langle s, t\rangle$ is a 1-hop path if $e(s, t) \in E$. For convenience, $e_j = e(v_j, v_{j+1})$ denotes the j-th edge in the path $p(s, t)$. The l-th dimension vector of $p(s, t)$ is denoted as $X_l(p) = [x_l(e_1), \cdots, x_l(e_{|p|})]$. Then, we formally define the "min" and "sum" cost functions of the l-th dimension about $p(s, t)$ as

$$C_l(p) = \begin{cases} min(X_l(p)) = \min_{j\in[1,|p|]} x_l(e_j) & \text{if } l \le k, \\ sum(X_l(p)) = \sum_{j=1}^{|p|} x_l(e_j) & \text{otherwise.} \end{cases} \tag{1}$$

In Equation 1, the "min" operator computes the minimal values of the first k dimensional features with $k < \kappa$, while the "sum" operator computes the cumulative value of the remaining $\kappa-k$ dimensional features. Accordingly, $C(p)$=$[C_1(p), \cdots, C_\kappa(p)]$ denotes as a κ-dimensional cost vector of $p(s, t)$. For any path $p_1(s, t) = p_2(s, v) \parallel p_3(v, t)$, we have $C(p_1) = C(p_2) \oplus C(p_3)$ where the l-th dimensional cost value is computed as

$$C_l(p_2) \oplus C_l(p_3) = \begin{cases} \min\{C_l(p_2), C_l(p_3)\} & \text{if } l \le k, \\ C_l(p_2) + C_l(p_3) & \text{otherwise.} \end{cases} \tag{2}$$

Next, we present "cost domination" and "path domination" as follows.

Definition 1 (**Cost Domination**). *For the l-th dimensional cost of two paths p_1 and p_2, we have $C_l(p_1) \prec C_l(p_2)$ if (1) $C_l(p_1) > C_l(p_2)$ with $l \le k$ or (2) $C_l(p_1) < C_l(p_2)$ with $l > k$.*

Definition 2 (**Path Domination** [10, 22]). *Let $p_1(s, t)$ and $p_2(s, t)$ be two different paths, where $C(p_1)$ and $C(p_2)$ are the corresponding cost vectors, respectively. We have p_1 dominates p_2, denoted by $C(p_1) \prec C(p_2)$, if there exists an $l \in [1, \kappa]$ such that*

- *$C_l(p_1) \prec C_l(p_2)$ and,*
- *$C_j(p_1) = C_j(p_2)$ or $C_j(p_1) \prec C_j(p_2)$ with $\forall j \in [1, \kappa] \setminus l$.*

Definition 3 (**Skyline Path** [10, 18, 19, 22, 26]). *Given a pair of vertices (s, t), $p(s, t)$ is a skyline path if there is no other path $p^*(s, t)$ to satisfy $C(p^*) \prec C(p)$.*

Without loss of generality, we use $SP(s, t)$ to denote the set of all skyline paths between s and t.
Problem Definition. Given a multi-attribute graph G and a query task $q(s, t)$, the skyline path enumeration problem aims to find all skyline paths between s and t.

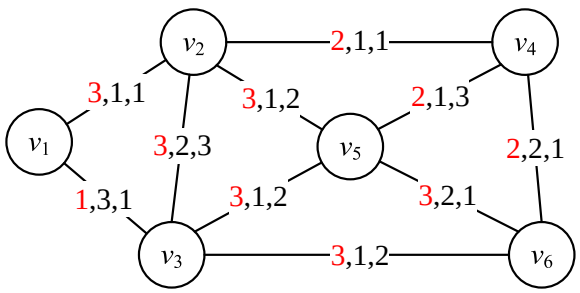


Fig. 2. Example of a multi-attribute graph where the "min" operator is used for the first dimension marked in red, whilst the rest attributes are equipped with the "sum" operator.

Example 1. *Figure 2 depicts a multi-attribute graph G, where each edge is equipped with a 3-dimensional real-valued attribute and $k = 1$. Given a vertex pair (v_1, v_6), there are two skyline paths $p_1 = \langle v_1, v_2, v_4, v_6\rangle$ and $p_2 = \langle v_1, v_2, v_5, v_6\rangle$ since $C(p_1) = [2, 4, 3]$ and $C(p_2) = [3, 4, 4]$ are not dominated by other cost vectors. In contrast, $p_3 = \langle v_1, v_3, v_6\rangle$ is not a skyline path since $C(p_3) = [1, 4, 3]$ is dominated by $C(p_1)$.*

2.2 Skyline Path Query Approaches

This part introduces the most related approaches to the skyline path query problem.

ARSC [18]. The authors focused on finding all preference shortest paths in a multi-attribute network graph, where the "preference distance" is defined as the weighted sum over all considered edge attributes. During the path-finding process, a route domination relationship is proposed to avoid traversing subpaths that are dominated by other results, thus reducing the processing time. However, considering that the cost function of each path is the weighted sum of edge attributes, the pruning strategies in ARSC cannot guarantee the discovery of all skyline paths based on Definition 3. Furthermore, it is inevitable that ARSC will produce massive fruitless explorations when extending to solve the SkyPE problem.

LSA [26]. The authors focused on finding top-k paths between a given node and facility points, where the rest of the paths cannot dominate the cost values of these paths. Then, they proposed LSA to perform the κ cost expansions concurrently on a multi-cost network and stop when none of them may lead to new skyline facilities.

However, we observe that the optimal paths in any single dimension can also be dominated by other results. For example, consider $q(s,t)$ in a 2-attribute graph with $k=0$. Let $p_1(s,t)$ and $p_2(s,t)$ be two paths with cost vectors $C(p_1)=[1,3]$ and $C(p_2)=[1,4]$. Apparently, p_2 is not skyline since it is dominated by p_1. Therefore, this strategy inevitably results in fruitless explorations, leading to significant time costs. More importantly, based on the path domination relationship in Definition 2, several skyline paths are not optimal in any dimension. Similarly, let $p_3(s,t)$ and $p_4(s,t)$ be two skyline paths with $C(p_3)=[1,3]$ and $C(p_4)=[3,1]$, respectively. Assuming that $p_5(s,t)$ exists in this graph with $C(p_5)=[2,2]$, we can conclude that p_5 is also a skyline path even though the cost in each dimension is not smallest. Therefore, the correctness of query results cannot be guaranteed when directly using LSA to resolve the SkyPE problem.

ParetoPrep [32]. ParetoPrep is a pre-processing method that collects the minimal values of all dimensions between each vertex to the target node. This strategy avoids exploring the redundant edges which are not located in the shortest paths. However, computing the minimal feature values of all vertices sequentially can be time-consuming, especially for large-scale graphs. Additionally, similar to LSA, this strategy cannot be used to enumerate all skyline paths, as some of these paths may not be optimal in all dimensions.

2.3 State-of-the-art Approaches on Hop-constrained Path Enumeration

In this part, we introduce the SOTA approaches to hop-constrained path enumeration (HcPE), and these methods can be extended to solve the SkyPE problem. Specifically, HcPE aims to enumerate all simple paths between the source and target vertices, where the hop number of each simple path is no larger than a given constraint.

BCDFS [29]. This method adopts a barrier-based pruning technique to update the hop number of vertices to the target node, thus avoiding exploring non-promising search branches in the future. However, considering that the hop number of each skyline path is uncertain, this strategy needs to sequentially enumerate all possible paths, leading to inevitable and extensive fruitless explorations. In addition, the pruning strategy may not work well in SkyPE since each exploration in BCDFS cannot provide an accurate bound for the unexplored branches.

PathEnum [33]. PathEnum introduces a lightweight index to rule out the redundant edges that cannot satisfy the hop constraint. Despite its effectiveness on HcPE, the extension of PathEnum exhibits two limitations. First, due to the constraint of multi-dimensional features, the redundant vertices and edges cannot be accurately ruled out, resulting in an ineffective reduction in the search spaces. Second, to ensure query result correctness, PathEnum must enumerate all simple paths, though this can be time-consuming due to the abundance of non-skyline paths.

EVE [5]. EVE improves upon PathEnum by eliminating redundant vertices and edges that do not participate in any simple path, thus compressing the search space of query tasks. Similar to

PathEnum, it is difficult for this strategy to precisely identify all candidate vertices and edges that are located in the corresponding skyline paths.

3 Extension of Hop-constrained path enumeration methods

Based on the analysis presented in Section 2.2, the existing skyline path query methods [10, 18, 19, 22, 26] face significant challenges in addressing the SkyPE problem due to limitations in query correctness. Conversely, HcPE can be readily extended to address the SkyPE problem by providing a precise maximal hop count across all skyline paths. Therefore, in this section, we primarily focus on introducing how to adapt two existing HcPE methods, namely BCDFS [29] and PathEnum [33], to solve the SkyPE problem. Before that, we outline a general pruning strategy based on skyline paths to eliminate unnecessary explorations, as detailed below.

Lemma 1 (**Skyline path-based pruning**). *Let $p(s,t)$ be a skyline path between s and t. Assuming that $p^*(s,v)$ is a subpath between s and v, we can deduce that all paths extended from $p^*(s,v)$ are not skyline if $C(p(s,t)) \prec C(p^*(s,v))$.*

Proof. Based on Equation 2, for each path $p^*(s,t)$ extended from $p^*(s,v)$, we have $C(p^*(s,t)) = C(p^*(s,v)) \oplus C(p^*(v,t))$. For the first k dimensions, $C_l(p^*(s,t)) = \min\{C_l(p^*(s,v)), C_l(p^*(v,t))\} \leq C_l(p(s,v))$ with $l \leq k$. Similarly, we have $C_l(p^*(s,t)) > C_l(p^*(s,v))$ with $l > k$. Based on Definition 2, we have $C(p^*(s,t)) \prec C(p^*(s,v))$, proving that $C(p^*(s,t)) \prec C(p(s,t))$. Therefore, all paths extended from $p^*(s,v)$ are not skyline paths. □

Example 2. *Take a query $q(v_1, v_6)$ in Figure 2 as an example. $p_1(v_1, v_6) = \langle v_1, v_2, v_4, v_6\rangle$ is a skyline path with $C(p_1) = [2,4,3]$. For the subpath $p_2(v_1, v_3)$, we find that each neighbor $u \in N(v_3)$ cannot be visited since $C(p_1) \prec C(p_2) \oplus X(v_3, u)$. Similarly, for the subpath $p_2(v_1, v_5) = \langle v_1, v_3, v_5\rangle$ in Figure 2, considering that $C(p_2) = [1,4,3]$ which is dominated by $C(p_1)$, we can conclude that all paths extended from p_2 are not skyline paths.*

For any given query task, two baselines require the maximum hop number d_{max} to ensure the completeness of query results and prevent unnecessary exploration. It is noted that this parameter can be calculated via our CAI index, where the details are shown in Section 5.

3.1 Extension of BCDFS

The essence of BCDFS hinges upon an innovative barrier-based pruning methodology, where the hop number of each candidate node towards the target node is dynamically adjusted. This mechanism is meticulously designed to steer clear of unfruitful search trajectories in future iterations. Yet, the challenge lies in the unpredictable nature of the hop counts associated with each potential skyline path, necessitating a rigorous, sequential examination of every conceivable route. This, in turn, results in an inevitable proliferation of exploratory endeavors that may ultimately yield limited dividends. Additionally, within the framework of SkyPE, the pruning strategy may not exhibit optimal efficacy, as the individual exploratory steps within BCDFS lack the precision to establish a definitive criterion for guiding subsequent actions.

To tackle these challenges, we introduce BCDFS*, an advanced extension of BCDFS that integrates sophisticated pruning strategies based on both hop number and skyline path considerations to effectively address the SkyPE problem. As shown in Algorithm 1, given a query $q(s,t)$ in G and a maximal hop number d_{max}, BCDFS* initializes the barrier value of each node $v \in V$ (denoted as $v.bar$) (Line 1). It then proceeds to recursively explore the graph to find all skyline paths (Lines 2-3). Specifically, the `BCSearch()` procedure is invoked from the source node s and gradually expanded to other vertices. During this process, Stk is used to collect the visited vertices, while Pth denotes a set of paths where the hop number of each path does not exceed d_{max}.

Algorithm 2 depicts the pseudo-code for the `BCSearch()` procedure. Specifically, if a subpath $p(s,v)$ in Stk ends at t ($v = t$), it is inserted into Pth (Line 3). In addition, the exploration branch from $p(s,v)$ to any neighbor $u \in N(v)$ is fruitless when satisfying

(1) $|p(s,v)| + 1 + u.bar > d_{max}$(Lines 6-7). The subpath $p(s,v)$ and the node u are not located in a simple path $p(s,t)$ with $|p(s,t)| \leq d_{max}$.
(2) $\exists p' \in Pth$ with $C(p') \prec c^*$ (Lines 8-9). Based on Lemma 1, all paths extended from $p(s,v)$ are dominated by p'.

In addition, when the hop number of subpath $p(s,v)$ stored in Stk is larger than d_{max}, we execute the UpdateBarrier() procedure to update the barrier values of vertices, thereby pruning the searching space of query task (Line 11). When finishing the BCSearch() procedure, each path $p^*(s,t)$ is removed if it is dominated by the other results in Pth (Lines 4-5 in Algorithm 1).

Algorithm 1: BCDFS*

Input: $G, q(s,t), Pth, d_{max}$
Output: all skyline paths between s and t
1 $v.bar \leftarrow 0$ with $\forall v \in V$
2 $Stk \leftarrow (s)$
3 BCSearch(t, Stk, Pth, d_{max})
4 **foreach** $p^* \in Pth$ **do**
5 Remove p^* from Pth if $C(p') \prec C(p^*)$ with $\exists p' \in Pth$
6 **return** Pth

Algorithm 2: BCSearch

1 **Procedure** BCSearch(t, Stk, Pth, d_{max})
2 $p(s,v) \leftarrow$ the subpath stored in Stk
3 **if** $v = t$ **then** Insert $p(s,v)$ into Pth;
4 **else if** $|p(s,v)| < d_{max}$ **then**
5 **foreach** $u \in N(v)$ *and* $u \notin Stk$ **do**
6 **if** $|p(s,v)| + 1 + u.bar > d_{max}$ **then**
7 continue // Hop-based constraint
8 $c^* \leftarrow C(p(s,v)) \oplus X(v,u)$
9 **if** $\exists p' \in Pth$ *with* $C(p') \prec c^*$ **then** continue // Lemma 1;
10 BCSearch($t, Stk \cup \{u\}, Pth, d_{max}$)
11 UpdateBarrier($v, d_{max} - |p(s,v)| + 1, Stk$)
12
13 **Procedure** UpdateBarrier(v, l, Stk)
14 **if** $v.bar > l$ *or* $v.bar = 0$ **then**
15 $v.bar = l$
16 **foreach** $u \in N(v)$ *with* $u \notin Stk$ **do**
17 UpdateBarrier($u, l + 1, Stk$)

Lemma 2 (**Correctness of BCDFS***). *With a proper d_{max}, the query result of BCDFS* in Algorithm 1 is correct.*

Proof. Given a query $q(s,t)$, let d_{max} be the maximal hop number of all paths in $SP(s,t)$, i.e., $d_{max} = \max_{p(s,t) \in SP(s,t)} |p(s,t)|$. We can conclude that $SP(s,t) \subset Pth$ based on Definition 3. Therefore, the rest of the paths in Pth are all skyline, thus guaranteeing the correctness of query results. □

Time complexity. Let C_j be the number of explored paths where the hop number of each path is j. In the worst case, each node v in an explored path p is visited at most $j-1$ times if $|p| = j$. Therefore, the time cost of BCDFS* can be upper-bounded by $O(\sum_{j=1}^{d_{max}} C_j \cdot j \cdot n)$.
Space complexity. In the worst, for a given query $q(s,t)$, BCDFS* needs to store all paths between s and t. Therefore, the space cost can be upper-bounded by $O(\sum_{j=1}^{d_{max}} C_j \cdot j)$.

3.2 Extension of PathEnum

The foundational principle of PathEnum lies in the introduction of a lean index structure, which serves to eliminate edges that fail to meet the hop constraint. When adapting PathEnum to the SkyPE problem, the accuracy of the query results becomes influenced by the specified hop number. Moreover, the complexities introduced by multi-dimensional features pose a challenge, rendering it difficult to precisely exclude redundant vertices and edges. Consequently, this leads to a suboptimal reduction in the searching space of any query.

To resolve these problems, we design the extension of PathEnum, named PathEnum*, where the pseudo-code is shown in Algorithm 3. Given a query $q(s,t)$ and a maximal hop number d_{max}, PathEnum* first computes the shortest distances of each node v inV to s and t (denoted as $dis(v,s)$ and $dis(v,t)$) and sorts the neighbors of v if $dis(v,s) + dis(v,t) \leq d_{max}$ (Lines 2-4). Then, the `Enumerate()` procedure is invoked from the source node s to find all skyline paths (Line 5) and Lemma 1 is applied to reduce the fruitless exploration of subpaths which are dominated by the existing results in Pth (Lines 17-21). In contrast to BCDFS*, PathEnum* benefits from precomputed distance information to avoid visiting redundant vertices that cannot satisfy the hop constraint (Lines 15-16). This approach significantly improves query performance by reducing unnecessary computations and explorations, making PathEnum* more efficient for finding skyline paths in graphs.

Lemma 3 (**Correctness of PathEnum***). *With a proper d_{max}, the query result of PathEnum* in Algorithm 3 is correct.*

Proof. Similar to Lemma 2, PathEnum can enumerate all simple paths where the hop number of each path is no larger than d_{max}, thus covering all skyline paths. Therefore, the query result of PathEnum is correct. □

Time complexity. Given a query $q(s,t)$ with the maximal hop number d_{max}, let n^* be the number of vertices that satisfy $dis(v,s) + dis(v,t) \leq d_{max}$. Then, the time cost of PathEnum* can be upper-bounded by $O(\sum_{j=1}^{d_{max}} C_j \cdot j \cdot n^*)$, where C_j is the number of explored paths with $|p| = j$.
Space complexity. For a given query $q(s,t)$, PathEnum* needs to store (1) the distance messages of each node to s and t and (2) all paths between s and t. Therefore, the space cost can be upper-bounded by $O(\sum_{j=1}^{d_{max}} C_j \cdot j + n)$.

3.3 Limitation of Extended Methods

Based on Lemmas 2 and 3, the correctness of query outcomes from the aforementioned two extension methods is guaranteed with a proper maximal hop number. However, these methods still suffer from substantial performance hurdles, particularly when confronted with query tasks characterized by higher maximal hop number values. The primary cause of this performance bottleneck stems from the expansive search space and the abundance of redundant outcomes. To elaborate, the quantity of feasible paths between two nodes escalates exponentially with the increase in hop number [5, 29, 33], yet a considerable proportion of these paths are dominated by others despite satisfying the hop constraint. As a result, the enumeration of these non-skyline paths gives rise to extra computations and substantial performance decline.

Algorithm 3: PathEnum*

Input: $G, q(s,t), Pth, d_{max}$
Output: all skyline paths between s and t

$Stk \leftarrow (s)$
Compute the shortest distances $dis(v,s)$ and $dis(v,t)$ with $\forall v \in V$
foreach $v \in V$ *with* $dis(v,s) + dis(v,t) \leq d_{max}$ **do**
  Sort $u \in N(v)$ based on $dis(u,t)$ // an ascending order
Enumerate(t, Stk, Pth, d_{max})
foreach $p^*(s,t) \in Pth$ **do**
  if $\exists p'(s,t) \in Pth$ *with* $C(p'(s,t)) \prec C(p^*(s,t))$ **then**
    Remove $p^*(s,t)$ from Pth
return Pth

Procedure Enumerate(t, Stk, Pth)
$v \leftarrow$ the last node in Stk
Get the subpath $p(s,v)$ stored in Stk
foreach $u \in N(v)$ *and* $u \notin Stk$ **do**
  if $dis(v,s) + dis(u,t) + 1 > d_{max}$ **then**
    break // Hop-based constraint
  $c^* \leftarrow C(p(s,v)) \oplus X(u,v)$
  if $\exists p'(s,t) \in Pth$ *with* $C(p') \prec c^*$ **then**
    continue // Lemma 1
  if $u \neq t$ **then** Insert $p(s,t)=p(s,v)\|e(v,t)$ into Pth ;
  else Enumerate$(t, Stk \cup \{u\}, Pth)$;

4 Core Attribute Index

In this section, we first introduce the structure of the core attribute index, called CAI, which is a task-oriented and lightweight index. Next, we design a CAI-based querying strategy to efficiently enumerate all skyline paths.

4.1 Index Structure

Based on the analysis in Sections 3.3, existing techniques always suffer from the inefficiency of fruitless explorations, causing significant time costs. To address this issue, we design CAI to collect two types of messages: (1) the cost values of skyline paths where each node is located and (2) the corresponding positions on these paths. This index can accurately rule out all redundant vertices and edges that are not located in any skyline path, as defined below.

Definition 4 (**core attribute index**). *Given a query $q(s,t)$, each node $v \in V$ collects a label set $L(v)$, where each label entry in $L(v)$ contains two κ-dimensional vectors $\{C(p(s,t)), C(p(s,v))\}$ with*

(1) $p(s,t)$ denoting a skyline path that passes through v, and
(2) $p(s,v)$ denoting a subpath in $p(s,t)$, i.e., $p(s,v) \subset p(s,t)$.

Specifically, $C(p(s,t))$ and $C(p(s,v))$ are the cost vectors of the skyline path $p(s,t)$ and its subpath $p(s,v)$, respectively. Then, the CAI index is the accumulation of label entries of each node, i.e., $CAI = \bigcup_{v \in V} L(v)$. Without loss of generality, for the path $p(u,v)$ with $u = v$, we have $C_l(p(u,v)) = \infty$ and $C_j(p(u,v)) = 0$ with $l \leq k$ and $j > k$, respectively.

Lemma 4. *Given any query task $q(s,t)$, each node $v \in V$ is redundant when $L(v) = \emptyset$.*

Proof. Based on Definition 4, we can conclude that the node v is not located in any skyline path between s and t if $L(v) = \emptyset$. □

Example 3. *Given a query task $q(v_1, v_6)$ in Figure 2, the CAI index is shown in Table 2. Based on $L(v_1)$ and $L(v_6)$, there are two distinct types of skyline paths, where the corresponding cost vectors are $[2, 4, 3]$ and $[3, 4, 4]$, respectively. Here, the node v_3 is redundant since $L(v_3) = \emptyset$. In addition, the rest of the vertices are located in at least one skyline path based on Lemma 4. For example, due to $\{[2, 4, 3], [3, 1, 1]\} \in L(v_2)$, the node v_2 is located in the skyline path $p_1(v_1, v_6) = \langle v_1, v_2, v_4, v_6\rangle$.*

Table 2. The CAI index of $q(v_1, v_6)$ in Figure 2

V	**Label entries**
v_1	$\{[2, 4, 3], [\infty, 0, 0]\}, \{[3, 4, 4], [\infty, 0, 0]\}$
v_2	$\{[2, 4, 3], [3, 1, 1]\}, \{[3, 4, 4], [3, 1, 1]\}$
v_3	$\emptyset$
v_4	$\{[2, 4, 3], [2, 2, 2]\}$
v_5	$\{[3, 4, 4], [3, 2, 3]\}$
v_6	$\{[2, 4, 3], [2, 4, 3]\}, \{[3, 4, 4], [3, 4, 4]\}$

Space complexity. For the CAI index, the label size $|L(v)|$ of a node v is the number of entries in $L(v)$. Denote by δ the largest label size in all vertices, i.e., $\delta = \max_{v \in V} |L(v)|$, the space complexity of CAI is $O(n \cdot \delta)$.

4.2 CAI Query Processing

Although CAI can effectively identify all redundant vertices, direct visits to candidate vertices for enumerating skyline paths may still lead to instances of unproductive exploration. Specifically, for each subpath $p(s, u)$, where $v \in N(u)$ is a candidate node, the fruitless computation arises from two main aspects, which are (1) $e(u, v)$ is not located in any skyline path and (2) $p(s, u)$ and $e(u, v)$ are not located in a same skyline path. As shown in Table 2, the vertices v_4 and v_5 are located in two different skyline paths $p_1(v_1, v_6)$ and $p_2(v_1, v_6)$, where $C(p_1) = [2, 4, 3]$ and $C(p_2) = [3, 4, 4]$, respectively. Note that the edge $e(v_4, v_5)$ is redundant since it is not located in any skyline path. To avoid unnecessary computations, we design the following two pruning strategies to efficiently eliminate fruitless edge explorations.

Lemma 5. *Let $p(s, t)$ be a skyline path between s and t, we have $C(p(s, v)) \prec C(p(s, t))$ if $p(s, v)$ is the subpath of $p(s, t)$.*

Proof. Let $p(s, t) = p(s, v) \parallel p(v, t)$. For simplicity, $p(s, t)$, $p(s, v)$, and $p(v, t)$ are abbreviated as p_1, p_2, and p_3, respectively. For the first k dimensions, we have $C_l(p_1) = \min\{C_l(p_2), C_l(p_3)\} \leq C_l(p_2)$ with $l \leq k$. Similarly, for the rest $\kappa - k$ dimensions, we have $C_l(p_1) = C_l(p_2) + C_l(p_3) > C_l(p_2)$ with $l > k$. Based on Definition 1, we have $C(p(s, v)) \prec C(p(s, t))$. □

Lemma 6. *Let $p(s, t)$ be a skyline path that passes through the vertices v and u. We have $e(v, u) \in E$ is located in $p(s, t)$ if $\exists\{C_v^1, C_v^2\} \in L(v)$ and $\exists\{C_u^1, C_u^2\} \in L(u)$ satisfy (1) $C_v^1 = C_u^1 = C(p(s, t))$, (2) $C_v^2 = C(p(s, v))$ and $C_u^2 = C(p(s, u))$, and (3) $C_u^2 = C_v^2 \oplus X(v, u)$.*

Proof. We prove this lemma by contradiction. If $e(v, u) \in E$ is not located in $p(s, t)$, we can conclude that $p^*(s, t) = p^*(s, u) \parallel p(u, t)$ is not a skyline path, where $p^*(s, u) = p(s, v) \parallel e(v, u)$. Based on the definition of the CAI index, we have $p(s, t) = p(s, u) \parallel p(u, t)$ where $C(p(s, u)) = C_u^2$. Then, we have

$$\begin{aligned} C(p^*(s, t)) &= C(p^*(s, u)) \oplus C(p(u, t)) \\ &= C(p(s, v)) \oplus X(v, u) \oplus C(p(u, t)) \\ &= C_u^2 \oplus C(p(u, t)) \\ &= C(p(s, t)). \end{aligned} \tag{3}$$

This conclusion contradicts the hypothesis, proving that the edge $e(v,u) \in E$ is located in $p(s,t)$. □

Algorithm 4: The CAI-based querying algorithm

```
Input: G(V,E), (s,t), ⋃_{v∈V} L(v)
Output: all skyline paths between s and t
1  Stk ← (s)                                          // collect the visited vertices
2  Search(t, Stk, ⋃_{v∈V} L(v))
3  ---------------------------------------------------------------------
4  Procedure Search(t, Stk, ⋃_{v∈V} L(v))
5  v ← the last node in Stk
6  Get the subpath p(s,v) stored in Stk
7  if v = t then Output p(s,t), return;
8  foreach u ∈ N(v) and u ∉ Stk do
9      if L(u) ≠ ∅ then continue ;                    // Lemma 4
10     foreach label entry {C_u^1, C_u^2} ∈ L(u) do
11         if C(p(s,v)) ⊀ C_u^1 then continue;        // Lemma 5
12         if ∃{C_v^1, C_v^2} ∈ L(v) with C_v^1 = C_u^1 then
13             if C_u^2 = C(p(s,v)) ⊕ X(v,u) then     // Lemma 6
14                 Search(t, Stk ∪ {u}, ⋃_{v∈V} L(v))
```

Based on Lemmas 5 and 6, we present a CAI-index-based querying algorithm to efficiently enumerate all skyline paths, where the pseudo-code is shown in Algorithm 4. Here, Stk is used to collect the visited vertices (Line 1). Given a query task $q(s,t)$ in $G(V,E)$, the Search() procedure is performed recursively to output all skyline paths (Line 2). Specifically, if the last node in Stk is t, then we find a skyline path and output it (Line 7). Otherwise, we consider the neighbors u of v such that $e(v,u)$ and $p(s,v)$ are located in the same skyline path, where $p(s,v)$ is the subpath stored in Stk (Lines 8-14). During this procedure, the fruitless explorations about redundant vertices and edges can be avoided based on Lemmas 4 to 6, thus largely reducing the query time cost.

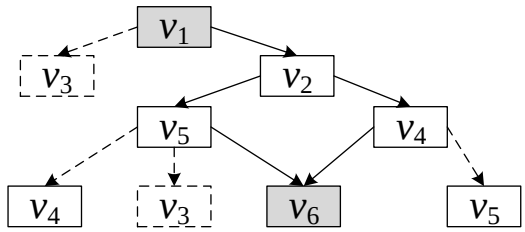


Fig. 3. Illustrating Algorithm 4 for $q(v_1, v_6)$ in Figure 2. Here, the arrow refers to the direction of path traversal and the dashed lines represent the redundant vertices and edges.

Example 4. *The query procedure of $q(v_1, v_6)$ is shown in Figure 3. The node v_2 is located in two different skyline paths based on $L(v_1)$ and $L(v_2)$ in Table 2 (Lemma 6). In contrast, the node v_3 is redundant since $L(v_3) = \emptyset$ (Lemma 4), which means that the edges $e(v_1, v_3)$, $e(v_2, v_3)$, and $e(v_5, v_3)$ are not required to be visited.*

Regarding the subpath $p(v_1, v_2)$, the node v_4 is further visited since (1) the vertices v_2 and v_4 are both located in the skyline path $p_1(v_1, v_6)$ with $C(p_1) = [2, 4, 3]$ and (2) $C(p(v_1, v_2)) \oplus X(v_2, v_4) = [2, 2, 2]$. Despite being a neighbor of v_4, the node v_5 is not visited since they are not located in the same skyline path.

Finally, we obtain two skyline paths $p_1 = \langle v_1, v_2, v_4, v_6 \rangle$ *and* $p_2 = \langle v_1, v_2, v_5, v_6 \rangle$*, where the cost vectors are* $C(p_1) = [2, 4, 3]$ *and* $C(p_2) = [3, 4, 4]$*, respectively.*

Time complexity. In the worst case, we assume that all vertices are not redundant. Specifically, $\forall v \in V$ takes $O(\delta)$ to visit the candidate neighbor u (Lines 8-14), where δ is the largest label sizes in all vertices. Therefore, the time complexity is $O(\sum_{v \in V} N(v) \cdot \delta) = O(m \cdot \delta)$.

5 CAI Index construction

Given a query $q(s, t)$, the task of building CAI poses two primary challenges: (1) computing the cost vectors of all skyline paths, i.e, $\{C(p(s,t)) | p(s,t) \in SP(s,t)\}$ and (2) determining the cost vectors of subpaths where each node resides, i.e., $\{C(p(s,v)) | p(s,v) \subseteq p(s,t)\}$. A straightforward method to address the first challenge involves traversing all paths which are not dominated by other results in each single dimension. Subsequently, the cost vectors of skyline paths can be derived by eliminating the vectors that are dominated by others. However, this strategy necessitates traversing a vast number of non-skyline paths and fails to guarantee the completeness of the cost vectors for the skyline paths containing each node, thereby inadequately addressing the second challenge. In this part, we propose a hop-dependency property to efficiently compute the cost values of all subpaths originating from the source node s. This approach not only mitigates the significant computational burden associated with traversing all paths but also facilitates the parallel construction of the CAI index.

5.1 Hop-dependency Property

To efficiently compute the cost vectors of all skyline paths, we devise a hop-based label propagation mechanism. This mechanism allows direct computation of the cost values of each node, thereby avoiding the need to recursively traverse all possible paths. Given a query $q(s, t)$, assuming that $P^d(s, v)$ is a set of paths between s and v, where $\forall p(s, v) \in P^d(s, v)$ satisfies $|p(s, v)| = d$. For each vertex $v \in V \setminus \{s\}$, $CL^d(v)$ is a cost vector set of $P^d(s, v)$, i.e., $CL^d(v) = \{C(p^*) | p^* \in C^d(v)\}$. In addition, we have $CL^{<d}(v) = \bigcup_{j=0}^{d-1} CL^j(v)$. Then, we prove that the cost vectors of each node can be collected by gathering the cost vectors of its neighbors in Lemma 7.

Lemma 7 (**Hop-based label propagation**). *Given a query* $q(s, t)$*, for each cost vector* $c_1 \in C^d(v)$*, there is at least one cost vector* $c_2 \in C^{d-1}(u)$ *to satisfy* $c_1 = c_2 \oplus X(u, v)$*, where* $u \in N(v)$*.*

Proof. We proof this lemma by contradiction. For each cost vector $c_2 \in C^{d-1}(u)$ with $u \in N(v)$, we suppose that $c_1 \neq c_2 \oplus X(u, v)$. Considering that each subpath $p(s, v)$ must pass one neighbor of v at least, we have $C(p(s, v)) = C(p(s, u)) \oplus X(u, v)$ based Equation 2, where $p(s, v) = p(s, u) \parallel e(u, v)$. Therefore, for each subpath $p(s, v)$ with $|p(s, v)| = d$, we have $C(p(s, v)) \neq c_1$ that contradicts the premise. □

Based on Lemma 7, the cost values of each node originate from its neighbors. Next, we design a pruning strategy to reduce the redundant computations about non-skyline paths and ensure the correctness of query results.

Lemma 8. *Let* $p(s, t) = p_1(s, v) \parallel p_3(v, t)$ *and* $p^*(s, t) = p_2(s, v) \parallel p_3(v, t)$ *with* $C(p_1) \prec C(p_2)$*, we have (1)* $C_l(p) \prec C_l(p^*)$ *or (2)* $C_l(p) = C_l(p^*)$*.*

Proof. This conclusion can be derived via Equation 2 and Definition 1. □

Lemma 9 (**Subpath-based pruning**). *Let* $p(s, t) = p_1(s, v) \parallel p_3(v, t)$ *and* $p^*(s, t) = p_2(s, v) \parallel p_3(v, t)$ *with* $C(p_1) \prec C(p_2)$*. Then, we have* $C(p) = C(p^*)$ *if*

- $C_l(p_1) > C_l(p_2) \geq C_l(p_3)$ *with some* $\{l\} \subseteq [1, k]$ *and*
- $C_j(p_1) = C_j(p_2)$ *with* $\forall j \in [1, \kappa]$ *and* $j \notin \{l\}$*.*

Otherwise, we have $C(p(s, t)) \prec C(p^*(s, t))$*, proving that* $p^*(s, t)$ *is not a skyline path.*

Proof. We first prove the case of $C(p) = C(p^*)$. For the l-th dimensional cost with $l \leq k$, we have $C_l(p) = C_l(p^*) = C_l(p_3)$ since $C_l(p_1) > C_l(p_2) \geq C_l(p_3)$. Similarly, the cost vectors of other dimensions are consistent, proving that $C(p) = C(p^*)$. For the remaining cases, Lemma 8 establishes that $C(p(s,t)) \prec C(p^*(s,t))$, implying all paths extending from $p^*(s,v)$ are not skyline paths. □

Lemmas 1 and 9 both focus on reducing the fruitless computations about non-skyline paths, thus optimizing the processing time. In particular, Lemma 9 ensures that no valid skyline paths are incorrectly pruned by preventing the elimination of skyline paths that originate from dominated subpaths, thus maintaining the completeness of CAI. Note that these two lemmas can also be applied to reduce the redundant computations of subpaths between each node v and t.

Example 5. *Take a query $q(v_1, v_6)$ as an example. Let $p_1(v_1, v_5) = \langle v_1, v_2, v_5 \rangle$ and $p_2(v_1, v_5) = \langle v_1, v_2, v_4, v_5 \rangle$ be two subpaths between v_1 and v_5. For a subpath $p_3(v_5, v_6) = \langle v_5, v_6 \rangle$, we have $C(p_1 \parallel p_3) \prec C(p_2 \parallel p_3)$, proving that all paths extending from p_2 are not skyline paths.*

5.2 The Parallelized Labeling Method

Although the time cost of traversal recursion can be optimized by directly computing the cost vectors of all vertices, it is still time-consuming to sequentially compute the CAI index. To resolve this problem, we provide a practical labeling method to compute the label entries of all vertices in parallel.

The pseudo-code of this parallelized labeling method is shown in Algorithm 5. Similar to $CL^d(v)$, $CL_R^d(v)$ denotes a cost vector set of paths between t and other vertices. C_{sk} is used to collect the cost vectors of all skyline paths (Line 1). For simplicity, we have $C(p_1(s,v)) \prec^* C(p_2(s,v))$ if $C(p_1) \oplus C(p_3) \prec C(p_2) \oplus C(p_3)$, where $p_3(v,t)$ is a subpath between v and t.

Generally, we execute a bidirectional search to collect the cost vectors of all skyline paths and adopt a join-oriented operation to build the label entries of each node. First, we update $CL^1(v)$ and $CL_R^1(w)$ where $v \in N(s)$ and $w \in N(t)$ (Lines 2-3), respectively. Then, we execute the following two operations to update the cost vectors. Consider the d-th step as an example, with details outlined below.

- **Cost vector collection** (Line 7). In this part, we execute the `PathGet()` procedure to collect the cost vector of each path $p(s,t)$ with $|p(s,t)| = d$. Specifically, the new cost vector $c^* = c \oplus X(v,t)$ is collected if it is not dominated by existing values (Lines 2-8 in Algorithm 6). Here, we have $c \in CL^{d-1}(v)$ with $v \in N(t)$. Meanwhile, existing cost vectors are also deleted if they are dominated by the new elements (Line 7 in Algorithm 6).
- **Label computation** (Lines 8-14). In this part, we compute $CL^d(v)$ and $CL_R^d(v)$ of each node v in parallel based on Lemma 7. Specifically, for each node $v \in V$, the new cost vector $c^* = c \oplus X(u,v)$ is collected if it is not dominated by the existing results (Lines 11-12), where $u \in N(v)$ and $c \in CL^{d-1}(u)$ (Lines 10-13). Similarly, $CL_R^d(v)$ can be obtained by re-executing Lines 9-12 of Algorithm 5.

Algorithm 5 stops when each node no longer gets new cost vectors, i.e., $\bigcup_{v \in V} CL^d(v) = \emptyset$ or $\bigcup_{v \in V} CL_R^d(v) = \emptyset$ (Line 16). Finally, we perform the `LabelBuild()` procedure to get the CAI index (Line 17). Specifically, for any two cost vectors $c_l \in CL^{<d}(v)$ and $c_r \in CL_R^{<d}(v)$ with $v \in V \setminus \{s,t\}$, we can conclude that v is located in at least one skyline path if $c_l \oplus c_r \in C_{sk}$ (Lines 11-14 in Algorithm 6). Note that $L(s)$ and $L(t)$ can be directly computed based on C_{sk}.

Calculation of maximal hop number d_{max}. When building the CAI index for any query task $q(s,t)$, d_{max} is updated to $d (> d_{max})$ via the `PathGet()` procedure if a new path $p(s,t)$ fulfills two conditions: (1) $|p(s,t)| = d$ and (2) $C(p) = c^*$ is not dominated by any cost value in C_{sk}. Upon finishing the construction of the CAI index, we can conclude that d_{max} represents the maximal hop number of $q(s,t)$.

Algorithm 5: The Parallelized Labeling Method

Input: $G(V, E)$, $q(s, t)$
Output: the CAI index $\bigcup_{v \in V} L(v)$

1 Initial C_{sk} // collect the cost vectors
2 Insert $X(s, v)$ into $CL^1(v)$ with $v \in N(s) \setminus \{t\}$
3 Insert $X(t, w)$ into $CL_R^1(w)$ with $w \in N(t) \setminus \{s\}$
4 **if** $e(s, t) \in E$ **then** Insert $X(s, t)$ into C_{sk};
5 $d \leftarrow 2$
6 **while** *True* **do**
7 Execute PathGet(d, C_{sk}) // compute the cost vectors of skyline paths
8 **foreach** $v \in V \setminus \{s, t\}$ ***in parallel*** **do**
9 **foreach** $c \in CL^{d-1}(u)$ *with* $u \in N(v)$ **do**
10 $c^* \leftarrow c \oplus X(u, v)$
11 **if** $\exists c' \in C_{sk}$ *with* $c' \prec c^*$ **then** continue // Lemmas 1;
12 **if** $\exists c' \in CL^d(v)$ *with* $c' \prec^* c^*$ **then** continue // Lemmas 9;
13 Insert c^* into $CL^d(v)$ if $c^* \notin CL^d(v)$
14 Execute Lines 9-12 to update $CL_R^d(v)$
15 $d \leftarrow d + 1$
16 **if** $\bigcup_{v \in V} CL^d(v) = \emptyset$ *or* $\bigcup_{v \in V} CL_R^d(v) = \emptyset$ **then** break;
17 **return** $\bigcup_{v \in V} L(v) \leftarrow$ LabelBuild()

Algorithm 6: Two procedures in Algorithm 5

1 **Procedure** PathGet(d, C_{sk})
2 **foreach** $v \in N(t)$ **do**
3 **foreach** $c \in CL^{d-1}(v)$ **do**
4 $c^* \leftarrow c \oplus X(v, t)$, $flg \leftarrow 0$
5 **foreach** $c' \in C_{sk}$ **do**
6 **if** $c' \prec c^*$ **then** $flg \leftarrow 1$, continue;
7 **if** $c^* \prec c'$ **then** Delete c' from C_{sk} ;
8 **if** $flg = 0$ **then** Insert c^* into C_{sk};
9 ---
10 **Procedure** LabelBuild()
11 **foreach** $v \in V \setminus \{s, t\}$ ***in parallel*** **do**
12 **for** $\forall c_l \in CL^{<d}(v)$ *and* $\forall c_r \in CL_R^{<d}(v)$ **do**
13 **if** $\exists c' \in C_{sk}$ *satisfies* $c' = c_l \oplus c_r$ **then**
14 Insert (c', c_l) into $L(v)$
15 **foreach** $c \in C_{sk}$ **do**
16 Insert $(c, 0)$ and (c, c) into $L(s)$ and $L(t)$, respectively

Example 6. *Given a query $q(v_1, v_6)$ in Figure 2, we show the process of Algorithm 5. Table 3 records the whole process of collecting the skyline path cost vectors. In addition, Tables 4 and 5 record the process of constructing $CL(v)$ and $CL_R(v)$ with $\forall v \in V \setminus \{v_1, v_6\}$. Specifically,*

- *When $d = 1$, we have*
  (1) ***Cost vector collection.*** *We have $C_{sk} = \emptyset$ since there is no 1-hop path between v_1 and v_6.*

Table 3. The cost vectors of skyline paths of $q(v_1, v_6)$

		$d=1$	$d=2$	$d=3$
C_{sk}	Insert	$\emptyset$	$[1,4,3]$	$[3,4,4]$, $[2,4,3]$
	Delete	$\emptyset$	$\emptyset$	$[1,4,3]$
	Final	$[3,4,4]$, $[2,4,3]$		

Table 4. The cost vector set $CL()$ of $q(v_1, v_6)$

V	$d=1$	$d=2$	$d=3$	**Total**
v_2	$[3,1,1]$	$\emptyset$	$\emptyset$	$[3,1,1]$
v_3	$[1,3,1]$	$[3,3,4]$	$\emptyset$	$[1,3,1]$, $[3,3,4]$
v_4	$\emptyset$	$[2,2,2]$	$\emptyset$	$[2,2,2]$
v_5	$\emptyset$	$[3,2,3]$	$\emptyset$	$[3,2,3]$

Table 5. The cost vector set $CL_R()$ of $q(v_1, v_6)$

V	$d=1$	$d=2$	$d=3$	**Total**
v_2	$\emptyset$	$[3,3,3]$, $[2,3,2]$	$\emptyset$	$[3,3,3]$, $[2,3,2]$
v_3	$[3,1,2]$	$\emptyset$	$\emptyset$	$[3,1,2]$
v_4	$[2,2,1]$	$\emptyset$	$\emptyset$	$[2,2,1]$
v_5	$[3,2,1]$	$\emptyset$	$\emptyset$	$[3,2,1]$

(2) ***Label computation.*** *We update the $CL^1(v)$ and $CL^1_R(w)$ respectively, where $v \in N(s) \setminus \{t\}$ and $w \in N(t) \setminus \{s\}$. As shown in Table 4, due to $e(v_1, v_2) \in E$ and $e(v_1, v_3) \in E$, two cost vectors $[3,1,1]$ and $[1,3,1]$ are inserted into $CL^1(v_2)$ and $CL^1(v_3)$, respectively. Similarly, we have $CL^1_R(v_3) = \{[3,1,2]\}$, $CL^1_R(v_4) = \{[2,2,1]\}$, and $CL^1_R(v_5) = \{[3,2,1]\}$.*

- *When $d = 2$, we have*
  (1) ***Cost vector collection.*** *Considering that $[1,3,1] \in CL^1(v_3)$, the cost vector $c^* = [1,3,1] \oplus X(v_3, v_6) = [1,4,3]$ is inserted into C_{sk} since this cost is not dominated by the existing results in C_{sk}.*
  (2) ***Label computation.*** *We update $CL^2(v)$ and $CL^2_R(v)$ with $\forall v \in V$. For example, considering that $[3,1,1] \in CL^1(v_2)$ and $X(v_2, v_4) = [2,1,1]$, the node v_4 can collect a cost vector $c^* = [3,1,1] \oplus [2,1,1] = [2,2,2]$, which means that there is a subpath $p_2(v_1, v_4)$ with $C(p_2) = [2,2,2]$. Similarly, we have $CL^2_R(v_2) = \{[3,3,3], [2,3,2]\}$.*
- *When $d = 3$, we have*
  (1) ***Cost vector collection.*** *We collect the cost vectors of paths from each node $u \in N(v_6)$ with $CL^2(u) \neq \emptyset$. Accordingly, we collect two vectors $C(p_3) = [2,4,3]$ and $C(p_4) = [3,4,4]$, where $p_3(v_1, v_6) = \langle v_1, v_2, v_4, v_6 \rangle$ and $p_4(v_1, v_6) = \langle v_1, v_2, v_5, v_6 \rangle$, respectively. Note that the cost vector $[1,4,3]$ is deleted from C_{sk} since it is dominated by $C(p_3)$.*
  (2) ***Label computation.*** *We update $CL^3(v)$ and $CL^3_R(v)$ with $\forall v \in V$. For example, the node v_5 can collect the cost vectors $[3,4,6] = [3,3,4] \oplus [3,1,2]$ and $[2,3,5] = [2,2,2] \oplus [2,1,3]$ from v_3 and v_4, respectively. However, these two vectors cannot be inserted into $CL^3(v_5)$ since they are both dominated by the existing results.*

Due to $\bigcup_{v \in V} CL^3(v) = \emptyset$ and $\bigcup_{v \in V} CL^3_R(v) = \emptyset$, the process of cost vector collection is terminated in this round.

Example 7. *Figure 4 describes the process of* `LabelBuild()` *for $q(v_1, v_6)$ in Figure 2. After collecting all cost vectors, for each node $v \in V \setminus \{s, t\}$, we execute a join-oriented operation to compute the CAI index. The essence of the join-oriented operation lies in identifying all cost vector pairs (c_l, c_r) with $c_l \in CL(v)$ and $c_r \in CL_R(v)$. Specifically, the node v is located in the skyline path $p(s,t)$ if $C(p(s,t)) = c_l \oplus c_r$. For example, the node v_2 is located in two kinds of skyline paths where the cost vectors are*

$[3, 4, 4]$ *and* $[2, 4, 3]$*, respectively. Therefore, we have* $L(v_2) = \{([3, 4, 4], [3, 1, 1]), ([2, 4, 3], [3, 1, 1])\}$*. In contrast, we have* $L(v_3) = \emptyset$ *since the corresponding cost vectors of* v_3 *are* $[1, 4, 3]$ *and* $[3, 4, 6]$ *which are dominated by* $[2, 4, 3]$ *and* $[3, 4, 4]$*, respectively.*

V	$CL(v) \oplus CL_R(v)$	$L(v)$
v_2	$[3, 1, 1] \oplus [3, 3, 3] = [3, 4, 4]$	$\{[3, 4, 4], [3, 1, 1]\}$
	$[3, 1, 1] \oplus [2, 3, 2] = [2, 4, 3]$	$\{[2, 4, 3], [3, 1, 1]\}$
v_3	$[1, 3, 1] \oplus [3, 1, 2] = [1, 4, 3]$	$\emptyset$
	$[3, 3, 4] \oplus [3, 1, 2] = [3, 4, 6]$	$\emptyset$
v_4	$[2, 2, 2] \oplus [2, 2, 1] = [2, 4, 3]$	$\{[2, 4, 3], [2, 2, 2]\}$
v_5	$[3, 2, 3] \oplus [3, 2, 1] = [3, 4, 4]$	$\{[3, 4, 4], [3, 2, 3]\}$

Fig. 4. Example of the `LabelBuild()` procedure in Algorithm 3 for $q(v_1, v_6)$ in Figure 2.

Time complexity. Let δ be the maximal number of label entries among all vertices. In the worst case, each node $v \in V$ takes $O(\sum_{u \in N(v)} \delta \cdot \log \delta)$ and $O(\delta \cdot \log \delta)$ to collect the label entries and compute the CAI index. Therefore, the time complexity of Algorithm 5 is $O(\sum_{v \in V}((|N(v)| + 1) \cdot \delta \cdot \log \delta))$ which can be reduced to $O(m \cdot \delta \cdot \log \delta)$.

Discussion for single dimension cost. In this part, we analyze how to effectively handle the query tasks in the single-dimensional graphs.

- **Single "minimized sums" property**. This special case is equivalent to all edges possessing identical "maximized minimums" attribute values. Our method can directly address this kind of query task since the index construction process can be terminated normally, guaranteeing CAI-based querying.
- **Single "maximized minimums" property**. We observe that an infinite path exists when a cycle with the largest maximized minimum exists between the source and target vertices. To address this issue, a straightforward strategy involves introducing an additional dimension with the "minimized sums" property, where the numerical value of each edge is uniformly set to 1. Following this strategy, each non-simple path will be dominated by the corresponding skyline simple path.

6 Experiments

In this part, we introduce the experimental setup and conduct extensive experiments to evaluate the performance of our methods.

6.1 Setup

Datasets. In the experiments, we employ several real-life datasets (Table 6) that are downloaded from Stanford Network data [1], Network Repository [2], and Laboratory for Web Algorithmic [3]. All directed data graphs have been converted to undirected graphs. Note that our method can be adapted to answer any query task in directed graphs.

In this part, we compare the following algorithms:

- **BCDFS***. The polynomial delay method in Algorithm 1 with a given hop constraint.
- **PathEnum***. The DFS-based method in Algorithm 3 with a given hop constraint.
- **EVE***. The adapted version of EVE [5]

[1] http://snap.stanford.edu/data/
[2] http://networkrepository.com/index.php
[3] https://law.di.unimi.it/index.php

Table 6. **Statistic of Real-world Graphs**

Alias	Dataset	$\|V\|$	$\|E\|$	deg_m	deg_{avg}	Type
CA	California	1.9M	2.7M	12	3	Road
TK	WikiTalk	2.4M	4.7M	100029	4	Social
YT	Youtube	3.2M	9.3M	91751	6	Video
UA	USA	20.4M	28.8M	9	2	Road
LJ	LiveJournal	3.9M	34.6M	14815	17	Social
SJ	SocLiveJ	4.8M	42.8M	20333	17	Social
OK	Orkut	2.9M	106.3M	27466	71	Social
WB	Webbase	100M	725.4M	772,198	15	Web
IT	IT2004	41.3M	1.03B	1,326,744	49	Web
TW	Twitter	52.6M	1.61B	3,691,240	75	Social
SK	SK2005	50.6M	1.81B	8,563,816	71	Web
FD	Friender	65.6M	1.81B	4,531,243	55	Social
U6	UK2006	77.4M	2.6B	5,896,421	39	Web
U7	UK2007	109.4M	3.4B	6,366,528	41	Web
UN	UK0607	131.5M	4.7B	6,100,318	71	Web

- **CAI**. Our method with a single core.

The details of EVE*. Given any query task $q(s,t)$, EVE* executes the following three steps.

(1) Calculating the maximum hop number d_{max} of $q(s,t)$ via Algorithm 5.
(2) Utilizing EVE to generate the simple path graph G^* of $q(s,t)$ based on the parameter d_{max}.
(3) Conducting $PathEnum^*$ (Algorithm 3) on G^* to get all skyline paths.

The given hop constraint of each query in BCDFS*, PathEnum*, and EVE* is computed based on Algorithm 5. For each query task, the process time of CAI is the sum of indexing time (Algorithm 5) and query time (Algorithm 4).

Environment. All algorithms are deployed in a Linux server which has Intel(R) Xeon(R) Silver 4210R with 20 computing cores and 512 GB of main memory. All algorithms are compiled with O3-level optimization. The parallel optimization is supported by the OpenMP framework.

Edge attributes setting. The multi-attribute graphs are widely used to model many real-world scenarios. Drawing upon existing works [19, 26], three prevalent types of attribute distribution are typically employed in each network below:

- **Independence**. The numerical attributes are generated independently using a uniform distribution.
- **Anti-correlation**. If an edge is good in one dimension, then it is bad in all the other dimensions.
- **Correlation**. An edge is good in one dimension, and it is also good in all the other dimensions.

Given that the networks with anti-correlated attributes are common cases (e.g., faster routes often have higher costs in transportation networks), we have designated the anti-correlation distribution as the default in our paper. Without specifying, the maximal numerical value of each dimension is 10.

Query task. Referring to [33], we generate disjoint sets V^* and $V^\#$ based on the vertex degrees: (a) V^* is the set of vertices within the top 10% in the descending order of their degrees, and (b) $V^\# = V - V^*$. Then, we have a setting according to the locations of s and t: $V^* \times V^\#$ and $V^\# \times V^\#$. We generate 1000 queries by choosing s and t uniformly at random. Note that in our experiments, CAI and the three baseline methods use the same set of queries on each dataset. The processing time is set as INF when an algorithm cannot finish in 10^6 seconds. Without specifying, the dimension number of attributes in each edge is set as 2.

6.2 Processing time and memory cost

In this section, we mainly examine the processing time and memory cost of all methods.

• **Processing time on all datasets.** In this part, we evaluate the processing time of all algorithms on all datasets. As shown in Fig. 5, CAI achieves up to 4, 4, and 3 orders of magnitude faster process times compared to BCDFS*, PathEnum*, and EVE*, respectively The superior performance of CAI can be mainly attributed to its well-suited index structure and effective pruning strategies, significantly reducing fruitless explorations by eliminating all redundant vertices and edges. Additionally, all computations can be optimized based on the parallel computing strategy, thus further improving the query performance. In contrast, BCDFS* and PathEnum* struggle to complete query tasks with large maximal distances within a reasonable time limit. This is because these two methods involve traversing redundant paths during the enumeration process, leading to increased execution time. Despite EVE*'s effort to eliminate some redundant vertices and edges, it still necessitates traversing non-skyline simple paths, therefore increasing its execution time.

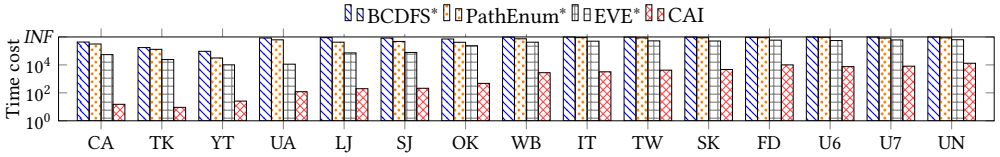


Fig. 5. Process time (s) on all datasets

Figure 6 shows the comparison of time cost between CAI construction and skyline path enumeration. We can observe that the time cost of index construction is much larger than that of skyline path enumeration.

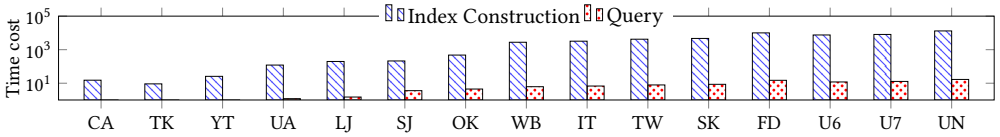


Fig. 6. Comparison of CAI index construction and skyline path enumeration

• **Memory cost.** In this part, we analyze the memory cost associated with three methods across various datasets. As depicted in Figure 7, BCDFS* emerges as the most memory-efficient approach due to its direct execution within the original graph, which necessitates minimal additional storage. Furthermore, the pre-computed index in PathEnum* is designed solely based on the maximal hop number constraint, thereby eliminating the need for gathering multi-dimensional attributes, which contributes to its relatively low memory usage. Similar to PathEnum*, EVE* also exhibits a slightly lower memory cost compared to our method. CAI incurs a slightly higher memory cost compared to the other two methods. This can be attributed to the necessity of constructing an index structure in CAI to minimize redundant computations, which inherently require additional storage space.

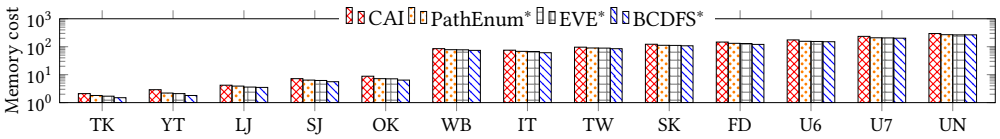


Fig. 7. Memory cost (GB) on all datasets

6.3 Scalability evaluation

In this section, we evaluate the scalability of CAI with respect to the process time in different scenarios.

• **Scalability w.r.t the graph size.** In this part, we evaluate the processing time of three methods as the graph size scales from 20% to 100%. Notably, BCDFS* is excluded from this comparison due

to its inefficiency. Due to the page limitation, we only list the results on 8 datasets, noting that the trends observed in these datasets are consistent with those from other experiments.

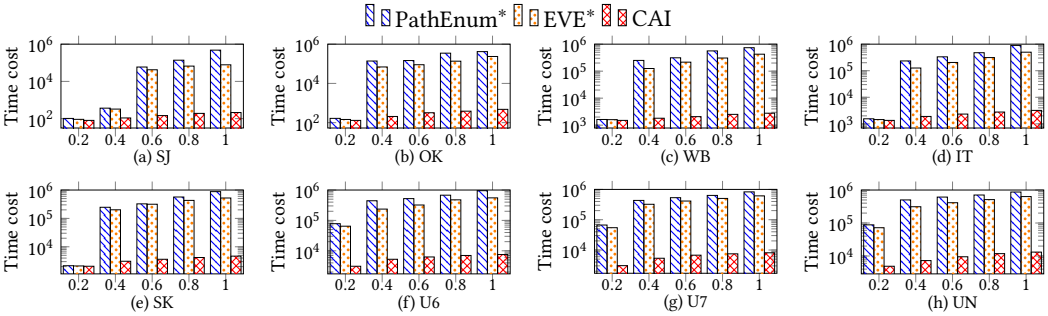


Fig. 8. Process time (s) when varying the graph size

As illustrated in Fig. 8, as the data graphs are enlarged, the process time for all three methods increases. This is due to the increased workload associated with each query task. Consequently, it requires more time for these methods to retrieve all skyline paths. We also observe that CAI demonstrates superior performance compared to PathEnum* and EVE* in nearly all scenarios, with the performance disparity widening as the graph size expands. Specifically, CAI achieves processing speeds that are up to 3 and 2 orders of magnitude faster than PathEnum* and EVE*, respectively. This significant advantage stems from CAI's ability to substantially minimize the number of explorations required and effectively harness available computing resources. In contrast, the other methods incur significant time costs due to the necessity of traversing numerous redundant paths, particularly when handling query tasks in large-scale graphs. This redundancy leads to inefficiencies that hinder their performance, especially as the graph size increases.

• **Scalability w.r.t the maximal weight value.** In this experiment, we evaluate the efficiency of CAI by adjusting the maximal edge weight from 10 to 30. Due to the inefficiency, BCDFS* and PathEnum* are excluded in this part.

As shown in Fig. 9, the average process time of CAI increases with the increase of the weight, which is caused by two aspects. First, the computational cost of index construction is enlarged when involving a larger maximal weight. Second, higher maximal edge weights can lead to an expansion of the search spaces for query tasks. Specifically, the variety and number of potential skyline paths may increase, making it more challenging and time-consuming for CAI to identify and evaluate these paths.

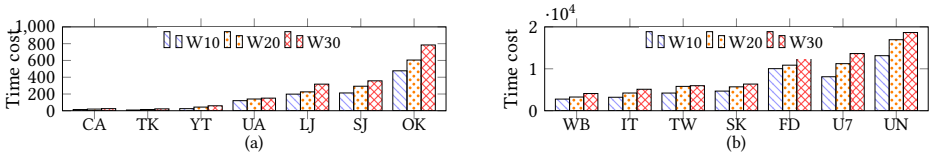


Fig. 9. Process time (s) vs. the maximal weight

• **Scalability w.r.t the number of cores.** In this experiment, we assess the scalability of CAI by varying the number of cores from 1 to 16. Note that the other two algorithms are omitted since they cannot be parallelized.

As shown in Fig. 10, CAI exhibits substantial performance gains when leveraging multiple cores. Compared to a single-core execution, CAI achieves up to 14.2× (on average 11.2×) acceleration in terms of process time on all datasets when utilizing 16 cores. This is because (1) the inherent computational complexity of the skyline path enumeration problem in multi-attribute networks

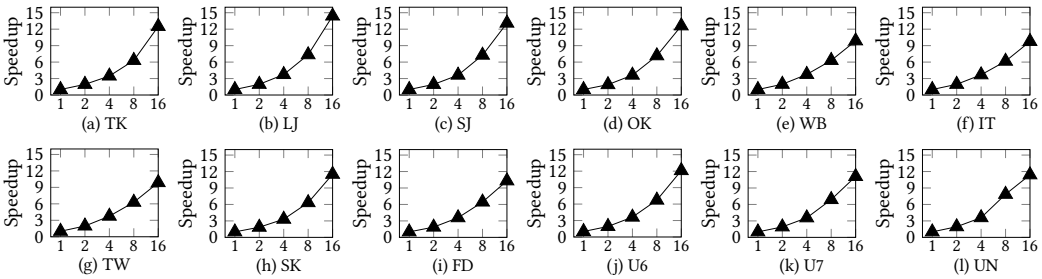


Fig. 10. Speedup vs. the number of cores

is significant and (2) CAI employs a hop-dependency label propagation strategy to distribute the workload across multiple cores, thus reducing the overall process time.

• **Scalability w.r.t the dimension of attribute.** In this experiment, we evaluate the process time of CAI by varying the dimension of attributes from 2 to 4. Due to the inefficiency, BCDFS* and PathEnum* are excluded in this part. As shown in Fig. 11, the average process time of CAI also basically increases when equipping more attributes. The increased time cost is mainly caused by the large amount of computation in the index construction process.

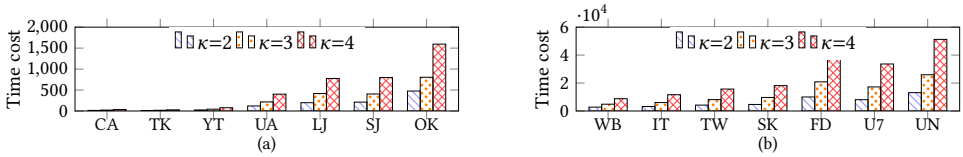


Fig. 11. Process time (s) vs. the dimensions of attributes

Figure 12 illustrates the process time of our method across three scenarios featuring diverse ratios of properties. Note that the query tasks in SK and IT were accelerated using 20 cores. Figure 12 (a) shows the trend when varying #dimensions of maximized minimums and fixing #dimensions of minimized sums to be 1, Figure 12 (b) shows the trend when fixing #dimensions of maximized minimums to be 1 and varying #dimensions of minimized sums. We can find that the average process time of our method generally increases with the addition of more dimensions. Figure 12 (c) demonstrates that the impact of varying combinations of dimensions on process time is indeterminate when fixing the total number of dimensions.

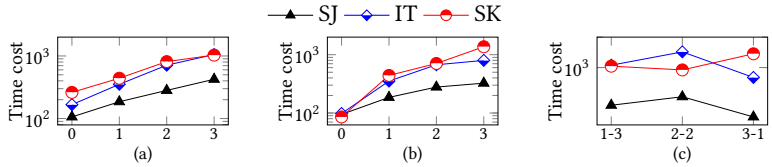


Fig. 12. Process time (s) vs. the number of dimensions

6.4 Ablative evaluation

In this section, we evaluate the effect of pruning and searching strategies in CAI.

• **Effectiveness of hop-dependency strategy.** In this experiment, we evaluate the effectiveness of the hop-dependency strategy by setting the hop constraint of all query tasks as a larger value for BCDFS* and PathEnum* (denoted as BCDFS# and PathEnum#).

As shown in Fig. 13, BCDFS* achieves up to 5.4× (on average 1.24×) speedup compared to BCDFS# in terms of process time. Similarly, PathEnum* achieves up to 12.97× (on average 1.97×) speedup

compared to PathEnum$^{\#}$ in terms of process time. Specifically, the hop-dependency strategy in CAI provides accurate maximal hop numbers for all query tasks, thus avoiding fruitless explorations without satisfying the hop constraint in BCDFS* and PathEnum*.

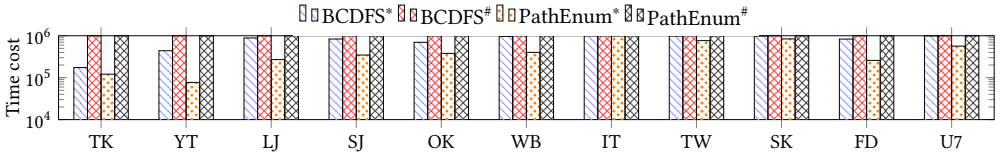
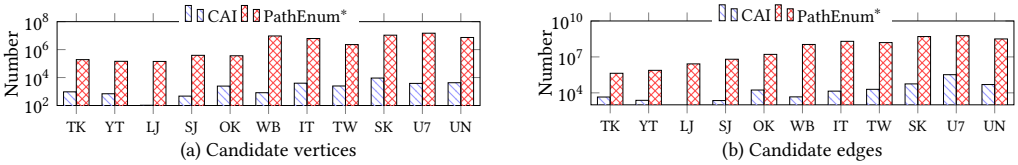


Fig. 13. Comparison of process time (s)

• **Effectiveness of redundant vertices/edges elimination.** In this experiment, we evaluate the scale of candidate vertices and edges in CAI and PathEnum*. Here, BCDFS* is not set as a competitor since the search process is executed in the whole graph.

As shown in Fig. 14, CAI achieves up to 11232× (on average 2611×) reduction compared to PathEnum* in terms of the number of vertices on all datasets. Similarly, it also achieves up to 32532× (on average 7722×) reduction in terms of the number of edges. Specifically, PathEnum* only eliminates the redundant vertices and edges which cannot satisfy the constriction of hop number. When facing the query tasks with large maximal path distances, this method inevitably involves massive redundant results that are actually not located in any skyline path. By contrast, CAI can rule out all redundant vertices and edges, thus improving the query performance.



(a) Candidate vertices (b) Candidate edges

Fig. 14. Comparison of #candidate vertices and edges

• **Effectiveness of different search paradigms.** In this part, we evaluate the effectiveness of the CAI-based searching method in Algorithm 4 by comparing it with a simple DFS method without pruning strategies, where all redundant vertices and edges have been eliminated.

As shown in Fig. 15, CAI achieves up to 9.76× (on average 2.84×) reduction compared to DFS in terms of process time on all datasets. Considering that two adjacent edges may be not located on the same skyline path, it is inevitable for the simple DFS strategy to explore massive fruitless explorations, especially for the tasks with large-scale search spaces, thus largely increasing the process time. By contrast, our proposed search technique can avoid all fruitless explorations by introducing effective pruning strategies.

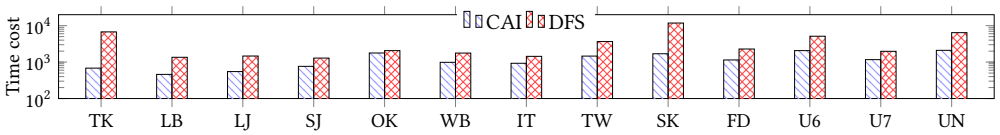


Fig. 15. Process time (s) vs. different search strategies

6.5 Performance evaluation on directed graphs

We have implemented our CAI method and performed some experiments on directed graphs. Figure 16 (a) and (b) depict the comparison of running time as the number of dimensions and the maximum attribute value vary, respectively, with a fixed core number of 20. As the number of dimensions grows from 1 to 4, we observe a gradual increase in the running time of our method.

Similarly, Figure 16 (b) shows a minor increase in the running time of our method as the maximum attribute value is incremented from 10 to 40. Furthermore, Figure 17 demonstrates the speedup achieved by our method. When utilizing 16 cores, our method exhibits an acceleration of up to 9.7× (with an average of 9.4×) in running time across all datasets, compared to single-core execution. In summary, the trends in those three experiments are consistent with those on undirected graphs.

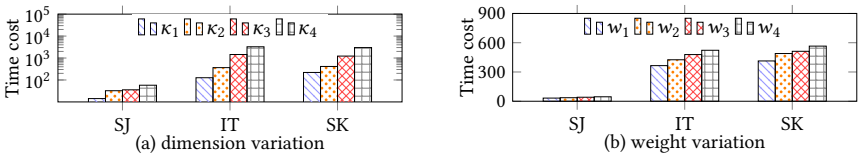


Fig. 16. Process time (s) on three directed graphs

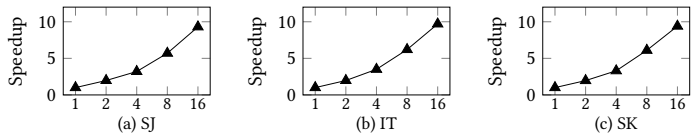


Fig. 17. Speedup on three graphs vs. the number of cores

7 RELATED WORK

In this section, we discuss the related work on s-t path enumeration. We also provide a brief review of other path-related problems, including reachability queries, distance queries, and skyline queries.

7.1 Path Enumeration

We begin with the simple path enumeration problem and next discuss the hop-constrained path enumeration.

Simple Path (or Cycle) Enumeration. Existing researches [4, 27, 31, 38] concentrate on efficiently listing the simple paths or cycles for the query tasks. These methods leverage representation structures to avoid explicitly storing each individual result, but they can incur significant memory costs. More importantly, it is time-consuming for these methods to effectively prune the search space of each query in SkyPE. Additionally, some works [2, 3, 16] focus on detecting the existence of cycles in dynamic graphs rather than enumerating the results.

Hop-constrained Path Enumeration. HybridEnum [17] proposed a hybrid search paradigm to enumerate simple paths to solve the HcPE problem in a distributed setting. In addition, the algorithm incorporated mechanisms such as work-stealing and caching to handle unbalanced workloads and optimize communication costs, respectively. Compared to HybridEnum, DistriEnum [40] employs a core search paradigm on a sketch graph to enable fast enumeration of simple paths while maintaining well-bound memory consumption. In addition, DistriEnum introduces the strategies of task division and vertex migration to enhance query efficiency and scalability.

Although designing effective pruning strategies to enumerate simple paths, these methods face a non-negligible performance bottleneck. Considering that the maximal hop number of skyline paths is an agnostic parameter, the corresponding query results are inaccurate when selecting a smaller hop number. In contrast, the search space will be enormous when choosing a larger hop number value. Due to the path domination relationship, these methods inevitably take massive fruitless explorations when directly applied to handle SkyPE, incurring huge processing time.

7.2 Skyline Queries

In the realm of skyline path queries, several approximation and optimization methods have been proposed to address the computational challenges associated with finding exact skyline paths in

large graphs. However, each of these methods has its limitations, particularly when it comes to finding all exact skyline paths. In [22], the authors designed an approximation method α-FHL which uses tree decomposition to hierarchically assign approximation ratios, supporting to find an approximate constrained skyline path. In [14], the authors proposed a novel hierarchical index and clusters to abstract the original graph to several summarized graphs, thus reducing the searching space.

Other research efforts have focused on applying skyline queries to specific domains, such as road networks. In [18], the authors computed skylines on routes, considering multiple preferences like distance, driving time, and gas consumption. They employed graph embedding techniques to enable a best-first-based graph exploration and proposed pruning techniques to reduce the search space. In [26], the authors designed progressive and incremental methods to solve skyline and top-k queries in multi-cost transportation networks, respectively. The goal of these methods is to search for optimal paths in a road network to the underlying optimization criteria. While these methods are effective for searching for optimal paths in road networks, they are not suitable for solving the SkyPE problem as they do not provide complete results. The detailed analysis is shown in Section 2.2 and more works about skyline queries have been summarized in [25].

7.3 Other Related Queries

Reachability Queries. Reachability queries involve determining the existence of directed paths between two vertices in a graph. Many works [15, 29, 34, 41, 42] improve query efficiency by building effective indexes. In [9, 29, 36], the reachability queries can be resolved based on the 2-hop index without the data graph. However, the simple paths of each vertex pair are not recorded in the 2-hop index. Therefore, it is impossible to only use this index to enumerate all simple paths between the given two vertices. More details have been summarized in [43].

Distance Queries. A distance query asks about the distance between two vertices in a graph, which receives a lot of research interests [1, 9, 12, 13, 20, 21, 28, 30, 35]. For example, the authors in [1] constructed a landmark-based index to serve all queries and evaluated the query with the pre-computed results. In [20], the authors proposed a parallel method to accelerate the construction of the 2-hop index whilst keeping the minimal property of distance labels. However, these methods or indexes cannot be used to efficiently resolve the path enumeration problem.

8 Conclusion

In this paper, we address the skyline path enumeration problem. Specifically, we design a core attribute index to rule out redundant vertices and edges that are not located in any skyline path. In addition, we propose effective pruning strategies to further reduce the fruitless explorations during the enumeration. Moreover, a hop-dependency label propagation strategy is designed to accelerate the construction of CAI. The comprehensive experiments demonstrate that our method achieves great improvements in terms of query time and scalability while taking a well-bounded memory consumption.

Acknowledgements

This work was supported in part by NSFC under Grant 62302421, Basic and Applied Basic Research Fund in Guangdong Province under Grant 2023A1515011280, 2025A1515010439, Shenzhen Research Institute of Big Data under grant SIF20240002, SIF20240004, Guangdong Talent Program under Grant 2021QN02X826, Shenzhen Science and Technology Program under Grants JCYJ20220530143602006 and ZDSYS20211021111415025, Ant Group through CCF-Ant Research Fund, the Guangdong Provincial Key Laboratory of Big Data Computing, and The Chinese University of Hong Kong, Shenzhen.

References

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*. ACM, 349–360.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. 2016. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Trans. Algorithms* 12, 2 (2016), 14:1–14:22.

[3] Sayan Bhattacharya and Janardhan Kulkarni. 2020. An Improved Algorithm for Incremental Cycle Detection and Topological Ordering in Sparse Graphs. In *SODA*. SIAM, 2509–2521.

[4] Katerina Böhmová, Luca Häfliger, Matús Mihalák, Tobias Pröger, Gustavo Sacomoto, and Marie-France Sagot. 2018. Computing and Listing st-Paths in Public Transportation Networks. *Theory Comput. Syst.* 62, 3 (2018), 600–621.

[5] Yuzheng Cai, Siyuan Liu, Weiguo Zheng, and Xuemin Lin. 2023. Towards Generating Hop-constrained s-t Simple Path Graphs. *Proc. ACM Manag. Data* 1, 1 (2023), 61:1–61:26.

[6] Yankai Chen, Yixiang Fang, Reynold Cheng, Yun Li, Xiaojun Chen, and Jie Zhang. 2018. Exploring communities in large profiled graphs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 31, 8 (2018), 1624–1629.

[7] Yankai Chen, Yixiang Fang, Yifei Zhang, Chenhao Ma, Yang Hong, and Irwin King. 2024. Towards Effective Top-N Hamming Search via Bipartite Graph Contrastive Hashing. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* (2024).

[8] Yankai Chen, Jie Zhang, Yixiang Fang, Xin Cao, and Irwin King. 2021. Efficient community search over large directed graphs: An augmented index-based approach. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence (IJCAI)*. 3544–3550.

[9] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[10] Linlin Ding, Gang Zhang, Ji Ma, and Mo Li. 2023. An Efficient Index-Based Method for Skyline Path Query over Temporal Graphs with Labels. In *DASFAA, 2023 (Lecture Notes in Computer Science, Vol. 13945)*. Springer, 217–233.

[11] Yixiang Fang, Reynold Cheng, Yankai Chen, Siqiang Luo, and Jiafeng Hu. 2017. Effective and efficient attributed community search. *The VLDB Journal* 26 (2017), 803–828.

[12] Muhammad Farhan, Henning Koehler, Robert Ohms, and Qing Wang. 2023. Hierarchical Cut Labelling - Scaling Up Distance Queries on Road Networks. *Proc. ACM Manag. Data* 1, 4 (2023), 244:1–244:25.

[13] Muhammad Farhan, Qing Wang, and Henning Koehler. 2022. BatchHL: Answering Distance Queries on Batch-Dynamic Networks at Scale. In *SIGMOD, 2022*. ACM, 2020–2033.

[14] Qixu Gong and Huiping Cao. 2022. Backbone Index to Support Skyline Path Queries over Multi-cost Road Networks. In *EDBT, 2022*. OpenProceedings.org, 2:325–2:337.

[15] Sairam Gurajada and Martin Theobald. 2016. Distributed Set Reachability. In *SIGMOD*. ACM, 1247–1261.

[16] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert Endre Tarjan. 2012. Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance. *ACM Trans. Algorithms* 8, 1 (2012), 3:1–3:33.

[17] Kongzhang Hao, Long Yuan, and Wenjie Zhang. 2021. Distributed Hop-Constrained s-t Simple Path Enumeration at Billion Scale. *Proc. VLDB Endow.* 15, 2 (2021), 169–182.

[18] Hans-Peter Kriegel, Matthias Renz, and Matthias Schubert. 2010. Route skyline queries: A multi-preference path planning approach. In *ICDE, 2010*. IEEE Computer Society, 261–272.

[19] Rong-Hua Li, Lu Qin, Fanghua Ye, Jeffrey Xu Yu, Xiaokui Xiao, Nong Xiao, and Zibin Zheng. 2018. Skyline Community Search in Multi-valued Networks. In *SIGMOD, 2018*. ACM, 457–472.

[20] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling Distance Labeling on Small-World Networks. In *SIGMOD*. ACM, 1060–1077.

[21] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2020. Scaling Up Distance Labeling on Graphs with Core-Periphery Properties. In *SIGMOD*. ACM, 1367–1381.

[22] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2024. Approximate Skyline Index for Constrained Shortest Pathfinding with Theoretical Guarantee. In *ICDE, 2024*. IEEE, 4222–4235.

[23] Chenhao Ma, Reynold Cheng, Laks V. S. Lakshmanan, and Xiaolin Han. 2022. Finding Locally Densest Subgraphs: A Convex Programming Approach. *Proc. VLDB Endow.* 15, 11 (2022), 2719–2732.

[24] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks V. S. Lakshmanan, and Xiaolin Han. 2022. A Convex-Programming Approach for Efficient Directed Densest Subgraph Discovery. In *SIGMOD, 2022*. ACM, 845–859.

[25] Kyriakos Mouratidis, Keming Li, and Bo Tang. 2021. Marrying Top-k with Skyline Queries: Relaxing the Preference Input while Producing Output of Controllable Size. In *SIGMOD, 2021*. ACM, 1317–1330.

[26] Kyriakos Mouratidis, Yimin Lin, and Man Lung Yiu. 2010. Preference queries in large multi-cost transportation networks. In *ICDE, 2010*. IEEE Computer Society, 533–544.

[27] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. 2017. Compiling Graph Substructures into Sentential Decision Diagrams. In *AAAI*. 1213–1221.

[28] You Peng, Zhuo Ma, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Xiaoshuang Chen. 2023. Efficiently Answering Quality Constrained Shortest Distance Queries in Large Graphs. In *ICDE, 2023*. IEEE, 856–868.

[29] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2019. Hop-constrained s-t Simple Path Enumeration: Towards Bridging Theory and Practice. *Proc. VLDB Endow.* 13, 4 (2019), 463–476.
[30] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast shortest path distance estimation in large networks. In *CIKM*. ACM, 867–876.
[31] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888.
[32] Michael Shekelyan, Gregor Jossé, and Matthias Schubert. 2015. ParetoPrep: Efficient Lower Bounds for Path Skylines and Fast Path Computation. In *SSTD, 2015 (Lecture Notes in Computer Science, Vol. 9239)*. Springer, 40–58.
[33] Shixuan Sun, Yuhang Chen, Bingsheng He, and Bryan Hooi. 2021. PathEnum: Towards Real-Time Hop-Constrained s-t Path Enumeration. In *SIGMOD*. ACM, 1758–1770.
[34] Lucien D. J. Valstar, George H. L. Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *SIGMOD*. ACM, 345–358.
[35] Ye Wang, Qing Wang, Henning Koehler, and Yu Lin. 2021. Query-by-Sketch: Scaling Shortest Path Graph Queries on Very Large Networks. In *SIGMOD, 2021*. ACM, 1946–1958.
[36] Dong Wen, Yilun Huang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2020. Efficiently Answering Span-Reachability Queries in Large Temporal Graphs. In *ICDE*. IEEE, 1153–1164.
[37] Haoxuan Xie, Yixiang Fang, Yuyang Xia, Wensheng Luo, and Chenhao Ma. 2023. On Querying Connected Components in Large Temporal Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 170:1–170:27.
[38] Norihito Yasuda, Teruji Sugaya, and Shin-ichi Minato. 2017. Fast Compilation of s-t Paths on a Graph for Counting and Enumeration. In *AMBN (Proceedings of Machine Learning Research, Vol. 73)*. PMLR, 129–140.
[39] Lyuheng Yuan, Da Yan, Wenwen Qu, Saugat Adhikari, Jalal Khalil, Cheng Long, and Xiaoling Wang. 2023. T-FSM: A Task-Based System for Massively Parallel Frequent Subgraph Pattern Mining from a Big Graph. *Proc. ACM Manag. Data* 1, 1 (2023), 74:1–74:26.
[40] Yuanyuan Zeng, Yixiang Fang, Chenhao Ma, Xu Zhou, and Kenli Li. 2024. Efficient Distributed Hop-Constrained Path Enumeration on Large-Scale Graphs. *Proc. ACM Manag. Data* 2, 3 (2024), 22:1–22:25.
[41] Yuanyuan Zeng, Kenli Li, Xu Zhou, Wensheng Luo, and Yunjun Gao. 2022. An Efficient Index-Based Approach to Distributed Set Reachability on Small-World Graphs. *IEEE Trans. Parallel Distributed Syst.* 33, 10 (2022), 2358–2371.
[42] Yuanyuan Zeng, Wangdong Yang, Xu Zhou, Guoqin Xiao, Yunjun Gao, and Kenli Li. 2022. Distributed Set Label-Constrained Reachability Queries over Billion-Scale Graphs. In *ICDE*. IEEE.
[43] Chao Zhang, Angela Bonifati, and M. Tamer Özsu. 2023. An Overview of Reachability Indexes on Graphs. In *SIGMOD, 2023*. ACM, 61–68.

Received October 2024; revised January 2025; accepted February 2025