TECHNICAL REPORT

TR-CS-NMSU-2018-09-02

Qixu Gong
Huiping Cao
Parth Nagarkar

Department of Computer Science
New Mexico State University

September 2, 2018

Skyline Queries Constrained by Multi-Cost Transportation Networks

Qixu Gong
Computer Science
New Mexico State University
Las Cruces, New Mexico
qixugong@nmsu.edu

Huiping Cao
Computer Science
New Mexico State University
Las Cruces, New Mexico
hcao@cs.nmsu.edu

Parth Nagarkar
Computer Science
New Mexico State University
Las Cruces, New Mexico
nagarkar@nmsu.edu

***Abstract*—Skyline queries are used to find the Pareto optimal solution from datasets containing multi-dimensional data points. In this paper, we propose a new type of skyline queries whose evaluation is constrained by a multi-cost transportation network (MCTN) and whose answers are *off* the network. This type of skyline queries is useful in many applications. For example, a person wants to find an apartment by considering not only the price and the surrounding area of the apartment, but also the transportation cost, time, and distance between the apartment and his/her work place. Most existing works that evaluate skyline queries on multi-cost networks (MCNs), which are either MCTNs or road networks, find interesting objects that locate on edges of the networks. Formally, our new type of skyline queries takes as input an MCTN, a query point q, and a set of objects of interest D with spatial information, where q and the objects in D are off the network. The answers to such queries are objects in D that are not dominated by other D objects when considering the multiple attributes of these objects and the multiple network cost from q to the solution objects. To evaluate such queries, we propose an exact search algorithm and its improved version by implementing several properties. The space of the exact skyline solutions is huge and can easily reach the order of thousands and incur long evaluation time. We further design much more efficient heuristic methods to find approximate solutions. We run extensive experiments using both real and synthetic datasets to test the effectiveness and efficiency of our proposed approaches. The results show that the exact search algorithm can be dramatically improved by utilizing several properties. The heuristic approaches to find approximate answers can largely reduce the query time and retrieve results that are comparable to the exact solutions.**

I. Introduction

Skyline queries are important in finding Pareto optimal solutions in multi-dimensional data. Conducting skyline queries on multi-cost networks (MCNs) has been studied [15], [17], [21], [26] in recent years. Examples of MCNs include road networks and multi-cost transportation networks (MCTNs). In an MCN, the cost of an edge is multi-dimensional in nature. For example, the cost of a road segment can represent the walking distance, driving time, and gasoline consumption. As far as we know, in existing works on evaluating skyline queries, the query points and the query results need to be simultaneously present on the edges of the given network.

We study the skyline query problem in a different real-world setting where the query points and/or the query results are off an MCTN while finding the solutions to the skyline queries need to utilize an MCTN. We denote such type of skyline queries as *MCTN-constrained skyline queries*. We now describe some of the real-world applications of MCTN-constrained skyline queries:

• *Application example 1*. Alice working at company X, which is the query point, wants to find an apartment, which is a query result (target object), with reasonable price and in a safe area. The apartment should be within reasonable distance from Alice's work place so that the commute time and the cost of using public transportation is acceptable. The desired apartment is one skyline solution by considering four factors: apartment price, safety of the apartment area, transportation time, and transportation cost. To find such apartments, we need to consider the travel distance and the cost of using the available public transportation network.

• *Application example 2*. Alice attending a conference, where the conference venue is a query point, wants to find a hotel (a query result) with good price and good service because the conference hotel is too expensive. Also, this hotel should not be too far away from the conference venue and the travel time between them should be reasonable. The hotel that meets Alice's requirements is a skyline solution of this conference venue (the query point) after taking into consideration the following factors: hotel price, hotel service, transportation time, and transportation cost. To find such hotels, we need to consider the travel distance and the cost of using the public transportation network or using road networks.

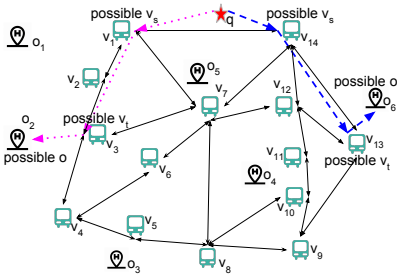


Fig. 1. Example of MCTN-constrained skyline queries & answers

When MCTNs are utilized to constrain skyline queries, there are several major challenges due to our problem letting: (i) the solution target object o (e.g., the apartment/hotel that Alice

finally decides to rent/book) are not known when the query is issued, and (ii) both the query point q and the target object o do not locate on the MCTN. One solution to a target object is a path containing three segments as shown by the three blue dash lines or the three purple dotted lines in Figure 1. The first segment is from q to a starting graph node v_s, the second segment is a graph path from v_s to an ending graph node v_t, and the third segment is from v_t to the target object o. When the query is issued, the starting graph node v_s, the ending graph node v_t, and the destination o are *all unknown*. The algorithm needs to find them in the search process. Once the graph nodes v_s and v_t are known, it is trivial to find the first and the last segments. The challenges lie in (i) finding the proper v_s and v_t and (ii) finding the graph paths from v_s to v_t that are not dominated by any other path between these two nodes. Every node in this MCTN can be v_s or v_t, thus a naive method needs to search paths between $N\times(N-1)$ node pairs, where N is the number of nodes in the MCTN. This calculation is prohibitively expensive.

Many skyline query processing techniques have been proposed (e.g., [2], [18], [19], [24]). However, these works do not take into consideration MCTNs for evaluating our proposed new queries. The work that is closest to our problem setting is [15], which presents an approach to find skyline paths between a *given* pair of source and destination graph nodes on an MCTN. The problem studied in [15] is much easier than our problem because both the source and destination graph nodes are known, while in our problem the starting and ending graph nodes (v_s and v_t in the above description) are all *unknown* since the query point q and the target object o are not on the MCTN. We discuss in more depth the differences between our work and [15] in Section II and propose a method that utilizes several heuristic rules in [15] in Section VI-B.

To evaluate the MCTN-constrained skyline queries and address the above mentioned challenges, we propose a baseline approach and several heuristics to improve upon the baseline. The contributions of this paper are as follows.

- We propose a new type of skyline queries whose answers are constrained by an MCTN. In particular, we consider the situation that the MCTN is stored in disk. This is different from most works that consider holding the graphs in memory (e.g., [8]).
- We propose a Best First Search (BFS) based baseline approach to evaluate such queries and find exact answers.
- We improve upon the baseline approach by utilizing several geometric-based properties that we observe.
- We improve the BFS-based exact search algorithm by utilizing heuristic rules to find approximate solutions. These heuristic methods further improve the query efficiency and are able to find answers that are comparable to the exact skyline solutions.
- We conduct extensive experiments using real and synthetic datasets. The results show that our improved methods can reduce the search space significantly and outperform the baseline.

The paper is organized as follows. In Section II, we discuss works that are related to our research problem. Section III formally defines the proposed problem. Sections IV and V present our proposed approaches. Section VI shows our experimental results. Finally, Section VII concludes our work.

II. Related Work

A. Skyline problem

The skyline problem is first proposed in [3], which introduces a Block Nested Loop (BNL) method and a Divide-and-Conquer approach. In general, there are two directions to solve the original skyline problem. The first direction is to design special index structures (e.g., variants of R-tree [9] or R^+-tree [20]) to accelerate query processing [18], [19]. The second direction is to pre-sort the source data to improve the efficiency of data scan [2], [6].

The original skyline problem has been extended to various applications. The k-dominant skylines are proposed in [4], which generalizes the dominance relationship by requiring that a point needs to be better than other points in at least k attributes. The dynamic skyline problem is introduced in [19], in which the dominance relationship between two points is defined based on an ad-hoc query point q.

Other works focus on proposing new methods to reduce the size of the skyline result set. In [16], [23], only k-representative results are returned. In [24], the authors define the score of a data point o by considering the total number of points it dominates and the distance between o and these dominated points. All these works do not involve any graph structured data, which we use.

B. Shortest path problem

The evaluation of MCTN-constrained skyline queries is highly related to path finding. Shortest path finding problem is one of the fundamental problems in the field of graph processing. The traditional Dijkstra [7] algorithm and the A^* algorithm [10] (and their subsequent extensions) are most widely used to find shortest paths in a graph. However, these traditional algorithms are not efficient in finding shortest paths from large graphs on the fly. To make online processing of big graphs faster, new index structures are proposed [1], [5], [11], [27]. These index structures cannot be directly utilized to solve our proposed problem because the size of these structures increases dramatically for the skyline setting (as apposed to shortest path finding).

C. Skyline queries on road networks

Given the fundamental importance of skyline queries, such queries have been proposed on road networks. As far as we know, the first work that considers skyline queries using road networks is [13], in which the in-route skyline problem is defined. This problem finds points of interest (POIs) on the edges of the network by considering multiple cost that are calculated using the location of a query (which is on one graph edge), a pre-defined route, and this route's destination.

A more often studied problem is the skyline path problem, which is introduced in [15]. In this problem, *given* a starting

node v_s and a destination node v_t in a multi-cost road network, a network path p from v_s to v_t dominates another path p' if and only if the cost on each dimension of p is better than that of p'. The search space of skyline paths is huge. To reduce the search space, Kriegel et al. [15] utilize the landmark index [14] to estimate the lower bound of the cost from any graph node v_i to the destination node v_t. The method in [15] also uses several heuristics: (h1) If a path p is dominated by one of the skyline paths found so far, p can be discarded. (h2) If the estimated cost of p is dominated by one of the skyline paths found so far, p can be discarded. (h3) A prefix sub-path p of a final skyline path must be a skyline path from v_s to p's ending node. Our work is very different from [15]. As analyzed in Section I, the target object in our problem setting is unknown and the query point is not on the graph. Because of these, the possible starting graph node v_s and the ending graph node v_t are unknown. A naive approach needs to compute skyline paths between N^2 node pairs (for possible v_s and v_t). The method in [15] only helps improve the search efficiency for *one pair of nodes*. Despite the intrinsic differences between our work and [15], we design an A*-based approach by utilizing several heuristics presented in [15] to compare with our proposed methods in Section VI.

Following the initial skyline-path definition in [15], Yang et al. in [26] define the stochastically dominance relationship and use the reverse Dijkstra [7] search to estimate the lower-bound of the cost on each dimension from a network node v_i to the destination node v_t. We utilize skyline paths, but we work on a more challenging problem where the starting graph node v_s and the destination graph node v_t are unknown.

More recent related works focus on finding skylines when using moving objects in road networks as query points. Fu et al. in [8] find continuous skyline POIs for an object moving on a road network whose cost is one dimension. Xu et al. [25] further attempt to improve upon the above problem by considering complex relations between a moving object's state and the given query. None of the above works try to solve skyline queries whose answers are constrained by an MCTN.

III. Problem Definition

This section formalizes our proposed skyline queries and related terminologies.

A multi-cost transportation network (MCTN) is represented as a weighted direct graph $G=(V,E,W)$ where V (denoted as $G.V$) is the set of nodes and each node contains spatial information, $E \subset V \times V$ (denoted as $G.E$) is the set of edges, and $W \in \mathbb{R}^{d_G}$ is a set of d_G-dimensional positive weight vectors. Let N be the number of graph nodes, and w_i be the cost of the i-th dimension of an edge. In an MCTN, the nodes can represent bus stops or metro stations and the edges represent the segments of bus/metro lines.

Let D be a set of objects that are of users' interest, such as hotels, restaurants, and apartments. Each object $o \in D$ has spatial attributes and d_D non-spatial attributes $o.attr[1],\cdots,o.attr[d_D]$ that users are interested in (e.g., price, ranking of a hotel). The spatial attributes are used for distance calculation. An object in D may locate on the network or be off the network. We focus on the case that the objects in D are off the network because the case that D objects are on G is an easier special case.

Running example. This section uses the MCTN shown in Figure 1 as a running example to explain the different concepts. We assume that the MCTN edges have two cost attributes, travel time and travel expense, and D contains hotel objects, which have two non-spatial attributes, price and ranking.

A. Graph paths and graph-constrained paths

Definition 1 (A graph path in G)**.** *Given a start node $v_s \in G.V$ and a destination node $v_t \in G.V$, a graph path $p_G(v_s,v_t)$ is a sequence of nodes $(v_s,\cdots,v_i,v_j,\cdots,v_t)$ where $v_i \in G.V$, $(v_i,v_j) \in G.E$, and no node appears twice in a path.*

The cost of a graph path is the summation of the cost of all the edges of p_G.

Example 1. *Given the MCTN in Figure 1, one graph path is $p_G(v_1,v_{11})$=(v_1,v_7,v_{12},v_{11}), and its cost is the summation of the cost of edges (v_1,v_7), (v_7,v_{12}), and (v_{12},v_{11}).*

Definition 2 (A graph-constrained path)**.** *Given a start point $o_s \in D \cup G.V$, a target point $o_t \in D \cup G.V \setminus o_s$, and an MCTN G, a G-constrained path from o_s to o_t is $p_c(o_s,o_t) = (o_s, p_G(v_s,v_t), o_t)$, where $p_G(v_s,v_t)$ is a graph path from v_s to v_t in G.*

A graph-constrained path is called *constrained path* when there is no confusion in the context. When o_s and o_t are graph nodes, the constrained path $p_c(o_s,o_t)$ is the same as the graph path $p_G(v_s,v_t)$ where $v_s=o_s$ and $v_t=o_t$. Constrained paths are used to describe queries in real world.

Example 2. *A person may want to find a path from a hotel o_s to a restaurant o_t by taking buses. A path $p_c(o_s,o_t)$ = (o_s, $p_G(v_s,v_t)$, o_t) may indicate that this person walks from o_s to the bus stop v_s, takes a bus from v_s to another bus stop v_t, and walks from the bus stop v_t to the restaurant o_t.*

In the setting of utilizing transportation networks, the cost of a constrained path is multi-dimensional. The dimension of the cost of one constrained path is d_G+1. Formally, the cost of a constrained path is defined as

$$cost(p_c(o_s,o_t)) = (dist(o_s,v_s) + dist(v_t,o_t),\ cost(p_G(v_s,v_t))).$$

The function $dist(o_i,o_j)$ represents the distance (e.g., by walking or driving) from o_i to o_j where o_i and o_j are from D. It can take different distance measurements, such as Manhattan distance or Euclidean distance.

Definition 3 (Dummy Path)**.** *Given a start point $o_s \in D \cup G.V$, a target point $o_t \in D \cup G.V \setminus o_s$, and an MCTN G, a dummy path from o_s to o_t is a special case of a constrained path $p_c(o_s,o_t)$ = (o_s, $p_G(v_s,v_t)$, o_t) where $p_G(v_s,v_t)=\emptyset$.*

The cost of a dummy path is

$$cost(p_c(o_s,o_t)) = (dist(o_s,o_t), \underbrace{0,\cdots,0}_{d_G}) \quad (1)$$

Example 3. *In Figure 1, let q be the given query and o_2 be an object of interest. The path (q, v_1, v_2, v_3, o_2) is a graph-constrained path $p_c(q, o_2)$. The cost of $p_c(q, o_2)$ is $(dist(q, v_1)+dist(v_3, o_2), cost(p_G(v_1, v_3))$. The special case of $p_c(q, o_2)$ is when a user walks from q to o_2 directly.*

The *starting and ending nodes* of a graph path or a graph-constrained path p is denoted as $p.start$ and $p.end$ respectively. Given $p_G(v_s, v_t)$, $p_G.start=v_s$ and $p_G.end=v_t$. Given $p_c(o_s, o_t)$, $p_c.start=o_s$ and $p_c.end=o_t$. The *length* of a path is the number of nodes in the path sequence minus one.

B. Dominance relationship, skyline paths, and path constrained objects

Since the cost of a path (either graph path or constrained path) is multi-dimensional, it is possible that the cost of two paths are incomparable to each other. To compare the cost of paths, we define their dominance relationship as follows:

Definition 4 (Dominance relationship)**.** *Given two general elements e and e' with multi-dimensional cost $cost(e)$ and $cost(e')$ respectively, e dominates e' (denoted as $e \succ e'$) if and only if $\forall$dimension i, $cost(e)[i] \leq cost(e')[i]$ and $\exists$ dimension i, $cost(e)[i] < cost(e')[i]$.*

The dominance relationship is transitive. I.e., if $e_1 \succ e_2$ and $e_2 \succ e_3$, then $e_1 \succ e_3$.

The dominance relationship can be applied to two paths p and p' (to replace the general elements e and e') to define that a path p dominates p' (denoted as $p \succ p'$).

Given the dominance relationships defined on paths, the skyline paths from v_s to v_t are defined as below.

Definition 5 (Skyline paths)**.** *Given an MCTN G, a starting object $o_s \in D \cup G.V$, and a target object $o_t \in D \cup G.V$, the skyline paths from o_s to o_t form a set of constrained paths SP satisfying (1) $\forall p' \notin SP$, $\exists p \in SP$ s.t. $p \succ p'$, and (2) $\forall p \in SP, \nexists p' \in SP$ s.t. $p' \succ p$.*

Definition 6 (A path constrained object)**.** *Given an object $o \in D \cup G.V$ and a graph-constrained path $p_c(o_s, o)$, their corresponding constrained object, denoted as o^{p_c}, has $d_D + d_G + 1$ attributes*

$$(o.attr[1], \cdots, o.attr[d_D], cost(p_c(o_s, o)). \quad (2)$$

Let us denote the attributes of o^{p_c} as $o^{p_c}.attr$, and d_c represent the number of attributes for a path constrained object.

A given object o can have multiple corresponding constrained objects $\{o^p\}$, which are constrained by different paths. Even when several paths that constrain o have the same starting point o_s, the constrained objects for o can still be multiple because there can be multiple different paths from o_s to o.

Example 4. *Given the MCTN in Figure 1 and let q be the given query. For o_2, corresponding to two constrained paths $p_{c1}(q, o_2)$=(q, v_1, v_2, v_3, o_2) and $p_{c2}(q, o_2)$=(q, v_1, v_7, v_3, o_2), we can get two path constrained objects, $o_2^{p_{c1}}$ and $o_2^{p_{c2}}$. These constrained objects have five attributes: (i) hotel price and hotel ranking from o_2's attributes, (ii) the walking distance for two segments (q, v_1) and (v_3, o_2), and the (iii) network cost including network travel time and network travel expense.*

Given a constrained object o^{p_c}, we call o its *base object* and p_c its *constraining path*. We can represent this as $BaseObject(o^{p_c}) = o$ and $ConstrainingPath(o^{p_c}) = p_c$.

Skyline solutions are path constrained objects. We can apply the dominance relationship (Def. 4) to two constrained objects o_i^p and $o_j^{p'}$ (replacing e and e') to define o_i^p dominating $o_j^{p'}$, denoted as $o_i^p \succ o_j^{p'}$, by treating $cost(o_i^p) = o_i^p.attr$ and $cost(o_j^{p'}) = o_j^{p'}.attr$.

C. Constrained skyline queries

Definition 7 (MCTN-constrained skyline query)**.** *Given an MCTN G, a set of objects of interest D, and a query point q, an MCTN-constrained skyline query returns a set $\mathcal{R}$ of constrained objects $\{o^{p_c}\}$ and their corresponding constraining paths $\{p_c(q, o)\}$ such that (i) $\forall o'^{p'} \notin \mathcal{R}$, $\exists o^{p_c} \in \mathcal{R}$ s.t. $o^{p_c} \succ o'^{p'}$, and (ii) $\forall o^{p_c} \in \mathcal{R}$, $\nexists o'^{p'} \in \mathcal{R}$ s.t. $o'^{p'} \succ o^{p_c}$.*

Note that skyline queries are defined in a similar way as skyline paths (Def. 5).

IV. Evaluate MCTN-constrained skyline queries

This section presents our baseline approach and its improved version to find the exact answers for our newly defined MCTN-constrained skyline queries.

A. ExactAlg-baseline*: Baseline method to find exact answers*

Several naive approaches can be used to find exact answers. One naive method can directly find skyline paths between every pair of graph nodes (as discussed in Section I). Another method is to introduce a dummy source node (the query point) and a dummy destination node (one object of interest), and find skyline paths between the dummy source and destination nodes. To avoid missing any solution, both the dummy source and destination nodes need to connect to *all* the nodes on the MCTN. The methods based on the above ideas incur expensive computations because there are $N \times (N-1)$ graph-node pairs and the number of skyline paths from one graph node to another graph node is exponential to the length of paths.

Due to the expensive computations of the naive approaches, we think of utilizing heuristics to solve the problem. One approach is to design an A*-based algorithm as in [15] by estimating lower-bound cost in the search process. For our problem, because the target object of interest is unknown, an A*-based method needs to consider every object in D as a possible target object. Even with a fixed target object, we still need to consider every MCTN node as a starting graph node and an ending graph node in the skyline path. This method also needs to find paths between $N \times (N-1)$ node pairs and the overall computation requires us to apply the method in [15] $|D| \times N \times (N-1)$ times to get the exact solutions. This method shares the similar complexity as the naive method although it can benefit from the heuristics in [15] to reduce the search space when the starting/ending graph nodes are fixed.

Considering the expensive computation of finding the exact solutions, we design a method (Section VI-B) to reduce the factors of $N \times (N-1)$ to find approximate solutions.

After analyzing the nature of our problem and the different possible naive approaches, we take a Best First Search (BFS)-based strategy to solve this problem because BFS only needs to explore the research space when necessary. We propose a BFS-based baseline method (*ExactAlg-baseline*, Algorithm 1) to evaluate an MCTN-constrained skyline query. This method utilizes a property of skyline paths. Before describing this property, we first introduce the concept of path prefix.

Definition 8 (Constrained prefix path)**.** *Given a constrained path $p_c(o_s, o_t) = (o_s, p_G(v_s, v_t), o_t)$ where $o_s \in D, o_t \in D$, its constrained prefix path is $(o_s, p_G(v_s, v_t))$, which is denoted as $p_c(o_s, v_t)$.*

A constrained prefix path is also called *prefix path* when no confusion is caused in the context. The cost $cost(p_c(o_s, v_t))$ is $(dist(o_s, v_s), cost(p_G(v_s, v_t)))$.

Example 5. *In the scenario of taking buses, a constrained prefix path means that a user knows the starting bus stop, the ending bus stop, and the bus line that he/she can take from the starting bus stop to the ending bus stop. However, this user does not know which target object he/she can reach from the ending bus stop.*

Property 1 (Property of skyline paths)**.** *Given an MCTN G, a query q, and a constrained path $p_c(q, o_t) = (q, p_G(v_s, v_t), o_t)$, if $p_c(q, o_t)$ is a skyline path from q to o_t, then its prefix path $p_c(q, v_t) = (q, p_G(v_s, v_t))$ must be a constrained skyline path from q to v_t.*

Proof. This property can be proved by contradiction. Assume that the constrained prefix path $p_c(q, v_t) = (q, p_G(v_s, v_t))$ is not a skyline path from q to v_t. Then, there must exist another prefix path $p'_c(q, v_t)$ that dominates $p_c(q, v_t)$. Concatenating $p'_c(q, v_t)$ with the segment (v_t, o_t), we get a constrained path $p'_c(q, o_t) = (p'_c(q, v_t), o_t)$. Then, $p'_c(q, o_t)$ must dominate $p_c(q, o_t)$ because (i) $p'_c(q, o_t) = (p'_c(q, v_t), o_t)$, (ii) $p_c(q, o_t) = (p_c(q, v_t), o_t)$, and (iii)$p'_c(q, v_t) \succ p_c(q, v_t)$. Then, p_c should not be a skyline path from q to o_t. This contradicts the given condition. ∎

This property generalizes the heuristic rule (h3) in [15].

This property is utilized in Algorithm 1, which shows the framework of our proposed baseline exact search algorithm. This framework utilizes a priority queue to keep the graph nodes that have been processed and have the potential to be in a skyline path from q to a base object of a skyline solution. An element in the priority queue is a graph node. For each such graph node v, we keep its spatial information, a flag *visited* to denote whether the node has been visited, and a structure *skypaths* to keep all the skyline paths from q to this node. The spatial information of the node v is used to calculate the distance from the query point q to v. This distance is used to rank the elements in the priority queue. The distance is utilized here because we can use it to conduct several improvements

Algorithm 1: Method *ExactAlg-baseline*

```
Input  : an MCTN G, a query point q, the set of objects of interest D
Output: The set of skyline solutions R
1  begin
2      Initialize a priority min-queue Q to be empty;
3      Initialize the result set R = ∅;
4      // Step 1: Graph traversal
5      v_nearest = the nearest graph node to q;
6      Q.enqueue(v_nearest);
7      while Q is not empty do
8          v = Q.pop() ;
9          if v is not visited before then
10             Create a dummy path dp;
11             v.visited = true;
12             addToSkyline(dp, v.skypaths) ;
13         foreach path p ∈ v.skypaths do
14             if p.expanded = fasle then
15                 foreach v_next ∈ neighbors(v) of G.V do
16                     p_next = path(p, v_next);
17                     if p_next is a new skyline path from q to v_next then
                           // Property 1
18                         v_next.Skypaths.add(p_next);
19                         Q.enqueu(v_next) ;
20     // Step 2: Create path constrained objects and put
          them to result set
21     Initialize the candidate result set D_cand to contain the objects in D that
         are not dominated by q ;
22     foreach v is visited do
23         foreach p_c ∈ v.skypaths do
24             foreach o ∈ D_cand do
25                 Create o^{p_c} with attributes updated using o's attributes,
                     cost(p_c), and dist(p_c.end, o);
26                 addToSkyline(o^{p_c}, R) ;
27 return R;
```

using Lemmas 2-4. For each path in v's skyline path set, we keep its current cost and an *expanded* flag to denote whether this path has been expanded in the traversal process. Every newly created path has the flag *expanded* set to be false.

The *ExactAlg-baseline* method runs in two steps, graph traversal and creating result set. In graph traversal, it first finds the graph node nearest to q and puts it to the priority queue (Lines 5-6). Then, it pops out the next best node v from the priority queue (Line 8) and expands its skyline paths. If the node v has not been visited before, this algorithm creates a dummy path dp (Line 10). The cost of the first dimension of dp is set to be the distance from q to v and the cost in all the other dimensions is set to be zero.

The second step of the algorithm (Lines 21-25) creates all the constrained objects that can be skyline solutions. Such objects are denoted as skyline candidates. In particular, it first finds the objects of interest that are not dominated by q (Line 21). These objects are possible skyline candidates. This step utilizes the R-tree structure [19] to index all the objects in D. Then, for every visited node v, each of its skyline path $p_c(q, v)$ can be combined with a base object $o \in D_{cand}$ to form a skyline candidate o^{p_c}. o^{p_c} consists of d_D attributes from o and $1 + d_G$ attributes from the cost of p_c (Line 25). In particular, $o^{p_c}.attr[i] = o.attr[i]$ for $1 \leq i \leq d_D$, $o^{p_c}.attr[d_D + 1] = cost(p_c)[1] + dist(p_c.end, o)$, and $o^{p_c}.attr[i] = cost(p_c)[i - d_D]$ for $d_D + 2 \leq i \leq d_D + d_G + 1$. Let the average number of skyline paths for each visited node be $|SP|$, Lines 22-25 have complexity $O(|G.V_{visited}| \times |SP| \times |D_{cand}|)$.

A very important step in the algorithm is to add a candidate

Function 2: Function *addToSkyline*

Input : a new object obj_{new} (which can be a path or a constrained object), a set of skyline objects $\mathbb{S}_{skyline}$
Output: Updated $\mathbb{S}_{skyline}$

```
1  begin
2      if S_skyline is ∅ then
3          S_skyline.insert(obj_new);
4      else
5          can_insert = true;
6          i = 0;
7          while the i-th object in S_skyline (obj) is not null do
8              if checkDominance(obj, obj_new) then
9                  can_insert = false;
10                 break;
11             if checkDominance(obj_new, obj) then
12                 S_skyline.remove(obj);
13                 continue;
14             i++;
15         if can_insert is true then
16             S_skyline.insert(obj_new);
17 return S_skyline;
```

constrained object to the result set $\mathcal{R}$, which may be huge. The details of this step are presented in Function 2 (*addToSkyline*).

The function *addToSkyline* checks whether it can add a new object obj_{new} (which can be a path or a constrained object) to the skyline object set $\mathbb{S}_{skyline}$. This algorithm utilizes the following Property 2.

Property 2. *Given a new object obj_{new}, if obj_{new} dominates an object $obj \in \mathbb{S}_{skyline}$, then obj_{new} must be a skyline object and needs to be added to the result set $\mathbb{S}_{skyline}$.*

Proof: We prove this by contradiction. Assume that obj_{new} is not a skyline object. There must $\exists obj' \in \mathbb{S}_{skyline}$ such that obj' dominates obj_{new}. Since obj_{new} dominates obj (given), then obj' dominates obj. This means that obj cannot be in $\mathbb{S}_{skyline}$. However, the given fact is that both obj and obj' are in $\mathbb{S}_{skyline}$. Thus, the assumption that obj_{new} is not a skyline object is not correct. ∎

Utilizing this property, the function *addToSkyline* works as follows. If the result set $\mathbb{S}_{skyline}$ is empty, the new object obj_{new} is directly added to $\mathbb{S}_{skyline}$ (Line 2). If the set $\mathbb{S}_{skyline}$ is not empty, the algorithm checks the dominance relationship of the new object obj_{new} and existing object obj in $\mathbb{S}_{skyline}$. In this step, we keep a flag can_insert, with initial value $true$, to denote whether the new object obj_{new} is dominated by any existing object. If it is dominated by one object, then it should not be inserted to $\mathbb{S}_{skyline}$ and the value of can_insert is set to $false$ (Line 8). If an existing object obj is dominated by obj_{new}, the algorithm removes obj from the set $\mathbb{S}_{skyline}$. After we scan every object $obj \in \mathbb{S}_{skyline}$, if the flag can_insert is true, we insert the obj_{new} into the set $\mathbb{S}_{skyline}$ (Line 16). The major computation of this function is the checking of the dominance relationship between the new object obj_{new} and each object $obj \in \mathbb{S}_{skyline}$. The function *checkDominance(obj_i, obj_j)* is used to check whether the object obj_i dominates another object obj_j.

B. ExactAlg-improved*: Improved exact search algorithm*

Two major expensive computation steps in Method *ExactAlg-baseline* are traversing the graph (Lines 7-19) and constructing constrained objects from the large D_{cand} to update the result set $\mathcal{R}$ (Lines 22-26). In this section, we propose several lemmas that can help us improve the baseline search method by reducing the queue size and the space of D_{cand}. Let us denote the method that utilizes these several lemmas as *ExactAlg-improved*.

1) Improvement to reduce queue size:

Lemma 1. *Let q be one query point and o be an object in D. The dummy path $p_c(q, o)$ must be a skyline path from q to o.*

Proof. Given that the cost of the dummy path p_c, $cost(p_c) = (dist(q, o), 0, \cdots, 0)$ (Eq. (1)), has d_G 0s, and no other graph paths have cost less than 0 in different dimensions. p_c must be a skyline path from q to o according to the definition of the skyline path (Def. 5). ∎

Utilizing this lemma, we can directly add a new dummy path to the set of skyline paths of one graph node. This Lemma is implemented in Line 12 of the baseline method (Algorithm 1).

Lemma 2. *Given a query q and two graph nodes v_i and v_j, if $dist(q, v_j) > dist(q, v_i)$, then the prefix path $p_c(q, v_i) = (q, p_G(v_j, v_i))$ cannot be a skyline path from q to v_i.*

Proof. Let us compare $p_c(q, v_i) = (q, p_G(v_j, v_i))$ and the dummy path $dp(q, v_i)$. The cost of $p_c(q, v_i)$ is $(dist(q, v_j), cost(p_G(v_j, v_i)))$. The cost of $dp(q, v_i)$ is $(dist(q, v_i), 0, \cdots, 0)$ (Eq. (1)). Because $dist(q, v_j) > dist(q, v_i)$ and each dimension of $cost(p_G(v_j, v_i))$ is bigger than 0, $dp(q, v_i) \succ p_c(q, v_i)$. Then, $p_c(q, v_i)$ is not a skyline path from q to o_i according to the definition. ∎

This lemma can be implemented before Line 17 in the baseline method (Algorithm 1). A condition can be added to check the relationship between $dist(q, p.start)$ and $dist(q, v_{next})$. If $dist(q, p.start) > dist(q, v_{next})$, the new path p_{next} is not a skyline path from q to v_{next} based on this lemma. Thus, we do not need to put it to the priority query.

2) Improvement to reduce skyline candidates: In the baseline algorithm, every object $o \in D_{cand}$ is utilized to form skyline candidates by using the constrained path $p_c(q, o) = (p_c(q, v), o)$ (Lines 24-26). We propose strategies to improve this step by eliminating the construction of skyline candidates for some objects in D_{cand}.

The new strategies utilize two lemmas. Let q be a query and let $p_c = (q, p_G(v_s, v_t), o)$ be a constrained path where $p_G(v_s, v_t)$ is not empty. We present two lemmas as follows.

Lemma 3. *Given q and p_c, if p_c is a skyline path from q to o, then $cost(p_c)[1] = (dist(q, v_s) + dist(v_t, o)) < dist(q, o)$.*

Proof. Let us compare p_c and the dummy path $dp(q, o)$. The cost of p_c is $(dist(q, v_s) + dist(v_t, o), cost(p_G(v_s, v_t)))$. The cost of the dummy path $dp(q, o)$ is $(dist(q, o), 0, \cdots, 0)$. Since $p_G(v_s, v_t)$ is not empty, each dimension of $cost(p_G(v_s, v_t))$ is bigger than 0. In order for both p_c and dp to be skyline paths, the first dimension of p_c must be smaller than that of dp. I.e., $dist(q, v_s) + dist(v_t, o) < dist(q, o)$ ∎

Lemma 4. *Given q and p_c, if p_c is a skyline path from q to o and $dist(v_t, o) \geq min\{dist(v_t, o_x)|o_x \succ o\}$, o^{p_c} is not a skyline solution.*

Proof. Let $p'_c(q, o_x) = (q, p_G(v_s, v_t), o_x)$ be a graph-constrained path from q to o_x where $o_x \succ o$. According to Eq. (2), the cost of $o_x^{p'_c}$ is

$$(o_x.attr[1], \cdots, o_x.attr[d_D], dist(q, v_s) + dist(v_t, o_x), cost(p_G(v_s, v_t)).$$

We know that the cost of o^{p_c} is

$$(o.attr[1], \cdots, o.attr[d_D], dist(q, v_s) + dist(v_t, o), cost(p_G(v_s, v_t)).$$

Let us compare the cost of o^{p_c} and $o_x^{p'_c}$. First, since $o_x \succ o$, we get $o.attr[i] \geq o_x.attr[i]$ for $1 \leq i \leq d_D$ and there must $\exists i \in [1, d_D]$ s.t. $o.attr[i] > o_x.attr[i]$. Second, $cost(p_G(v_s, v_t))$ is the same for both constrained objects. Then, if $dist(v_t, o) \geq dist(v_t, o_x)$, o^{p_c} is dominated by $o_x^{p'_c}$. Thus, o^{p_c} can not be a solution for the skyline query q. ∎

Algorithm 3: Function *addToSkylineImproved*

Input : a query point q, a new path np, a set of skyline solutions $\mathcal{R}$ candidate object set D_{cand}
Output: updated $\mathcal{R}$

```
1  begin
2      v_s = np.start; v_t = np.end;
3      foreach o ∈ D_cand do
4          dist_min = MIN{dist(v_t, o_x)|o_x ≻ o} ;
5          if ((dist(q, v_s) + dist(v_t, o) < dist(q, o))
6          & (dist(v_t, o) < dist_min) then ;          // Lemmas 3& 4
7
8              Create o^{p_c} with attributes updated using o's attributes,
                 cost(np), and dist(np.end, o);
9              addToSkyline(o^{p_c}, R) (Function 2);
10 return R;
```

We create a new function *addToSkylineImproved* (Algorithm 3) by utilizing Lemmas 3 and 4 to reduce the time of updating skyline results. This function creates new skyline candidate o^{p_c} only when the distances from q to v_s and from v_t to o meet the given conditions in both lemmas. These two conditions limit the creation of candidate skylines. With this function, *ExactAlg-improved* rewrites Lines 24-26 in the baseline method to

$$addToSkylineImproved(q, p_c, R, D_{cand}).$$

C. How much space to improve

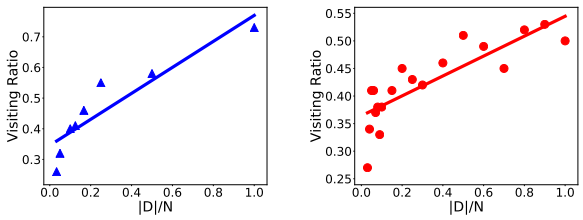
We utilize different Lemmas to improve the exact search algorithms in Sections IV-B1 and IV-B2. Do we still have much space to improve the baseline algorithm? We propose a measurement, called *visiting ratio*, to quantify this.

For a given query, the visiting ratio is defined as follows.

$$\text{Visiting Ratio} = \frac{|\{Nodes \in p_c | o^{p_c} \in \mathcal{R}\}|}{|\{\text{Nodes in } G \text{ that are visited}\}|}$$

A higher ratio means that larger number of graph nodes that are visited are also in the constrained paths of the final results. Thus, less graph traversal effort is wasted.

We examine the visiting ratio by plotting the ratios for different settings of $\frac{|D|}{N}$ in Figure 2. The figure shows that the visiting ratio is very high. Even when the number of objects



(a) Fix $|D|$=1000 and vary N from 1000 to 30,000 (b) Fix N=10,000 and vary $|D|$ from 300 to 10,000

Fig. 2. Visiting ratio vs. $\frac{|D|}{N}$

is only 20% of the graph size N, the visiting ratio is more than 40%, which means that more than 40% of the nodes that are visited in the query process is a node in the constrained path of a result. These results show that there is little space to improve the exact search algorithm.

V. Heuristic approaches to find approximate solutions

The space of exact skyline answers is huge. It can easily reach thousands, thus incur very expensive calculation. This section proposes strategies to reduce the unnecessarily huge space of results based on two intuitions in real applications.

The first intuition comes from how users utilize search results. Given a query, people tend to utilize the first tens of answers [22]. The thousands of answers returned to users may not really help much. The second intuition is related to how much users care whether a solution is an exact solution or not. In the applications of utilizing transportation networks, when a non-skyline answer is close to a skyline answer (e.g., the travel time differs from the exact travel time (which is thirty minutes) by two minutes and all the other dimensions are the same), the non-skyline answers are generally acceptable to users. Based on the above intuitions, we propose two heuristic approaches to find approximate solutions. These approximate solutions are comparable to the exact solutions, while the heuristic methods can dramatically reduce the result space.

A. Heuristic approach by using approximate range search

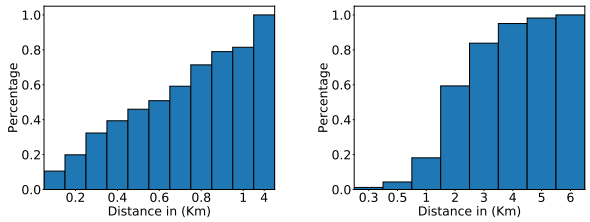
The first heuristic targets to reduce the number of starting and ending nodes during graph traversal by using approximate range search. We denote this method as ***Approx-range***.

Observations: In real applications of utilizing transportation networks, many bus/metro stops are far away from a query point q. To get results for the query q, it is not reasonable to use those faraway bus/metro stops as starting graph nodes to traverse the graph. Also, if a bus/metro stop is far away from the target object, people may not want to walk to such target object from the bus/metro stop. A similar scenario is observed by [12] which limits the distance from a query point to the target result.

Making use of these observations, we try to find a reasonable distance threshold that can be utilized to decide whether a bus/metro stop is too far from a given query q or any target object. If a graph node v_s is too far from the query point, this

v_s does not need to be a starting graph node during traversal. Similarly, if a target object is too far from a graph node v_t, this object cannot be a result object constrained by any graph path ending at v_t.

We use the statistics of real datasets to find such threshold. From the real data (see Section VI for detailed descriptions), we run a random query and get the result set $\mathcal{R}$ of exact skyline solutions. From $\mathcal{R}$, we extract all the constrained paths. From such paths, we get the set of distinct starting graph nodes $\{v_s\}$ and distinct ending graph nodes $\{v_t\}$. For each starting



(a) Distribution of the distance from q to the starting nodes of graph paths

(b) Distribution of the distance from the ending nodes of graph paths to the base objects of skyline solutions

Fig. 3. Distribution of distance

node v_s, we calculate the distance from q to v_s and plot the distribution of such distance in Figure 3(a). For each ending node v_t, we calculate the distance from v_t to its corresponding paths' constrained objects and plot the distance distribution in Figure 3(b). The figures show that more than 80% of graph starting nodes are within 1 Kilometer (Km) of the query point, and more than 20% of objects of interest are within 1Km of a graph ending node. The distance from the ending nodes to constrained objects is larger than the distance from q to the starting nodes of the graph paths. This is because $\{v_t\}$ are more constrained: the objects constrained by graph paths ending at v_t are skyline answers considering both the distance of graph paths and the non-spatial attributes of the objects.

Based on these statistics, we set a parameter τ to limit the range search of starting and ending graph nodes. For a given query q, we find the graph nodes that are within distance τ from q and treat them as starting nodes to traverse the graph. If τ is 1Km, then we can find 80% of the actual starting graph nodes. Similarly, for each graph node v, which can be a potential ending node of a graph path, we find objects in D that are within distance τ from v. Such objects have the potential to form a skyline answer.

B. Heuristic approach by using limited prefix paths

Another factor that impacts the performance of the exact search algorithms is the number of skyline paths. As shown in [15], when the length of a path increases, the number of skyline paths between two nodes increases dramatically. When a path is long, this number becomes prohibitively huge.

The large number of skyline paths incurs expensive calculation in the exact search methods. Our second heuristic approach targets to reduce the factor of $|skypaths|$ of each graph node. Our approach is inspired by the existing work [17] which defines the skyline candidates by considering only the shortest path on each dimension from the query point to each target object. Utilizing a similar idea, this heuristic chooses the skyline paths that have the minimum value on one dimension to expand. This heuristic reduces the number of skyline paths that need to be expanded for each node to d_G.

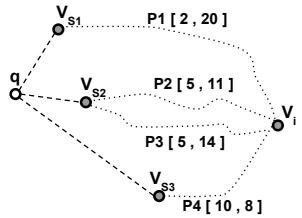


Fig. 4. Four skyline paths from q to v_i

Figure 4 shows a simple example of the path expansion process for a node v_i where the graph d_G is two. $p1$, $p2$, $p3$, and $p4$ represent the constrained prefix paths that need to be expanded. The exact search algorithm needs to expand all the four paths. This heuristic only needs to expand $p1$ and $p4$ because they have the minimum cost on dimension $d1$ and $d2$ respectively.

Issue caused by dummy paths. The heuristic approach described above always chooses the dummy path to expand because dummy paths only have one non-zero dimension and have zero cost (minimum cost value) on all the other dimensions. Thus, when we expand the skyline paths at each node, the dummy path for this node is always chosen to be expanded. This way, too much information is lost.

We propose to leverage range searches to this heuristic to avoid the issue. When we use approximate range search, for a graph node v that is too far away from q (beyond the threshold τ), we can avoid creating dummy paths from q to v. Then, node v does not have a dummy path as a skyline path to be expanded. We denote the heuristic that utilizes both the range searches and the limited skyline-path expansion as ***Approx-mix***.

C. Indexed search algorithm

Lemma 4 shows that we can eliminate candidate objects by utilizing only the attributes of objects in D and the distance from graph nodes to these objects. Suppose that we have two objects o_i and o_j, and $o_i \succ o_j$. For a graph node v, if $dist(v, o_j) > dist(v, o_i)$, then o_j cannot form a skyline candidate while o_i has the possibility to form a candidate solution. For each graph node v, we calculate a set of objects that have the possibility to form candidate solutions. Let $\mathbb{S}_v$ be a set with such objects. I.e., $\mathbb{S}_v = \{o_i | \exists o_j, (o_i \succ o_j) \wedge (dist(v, o_j) > dist(v, o_i))\}$. The set $\mathbb{S}_v$ can be pre-calculated and be used to calculate distance in Line 4 of Function *addToSkylineImproved*.

To facilitate the fast calculation of distance. We create an index structure to organize these sets of $\mathbb{S}_v$. This index structure is denoted as *LSO* to represent local skyline objects. The index structure organizes the objects in three layers. The first layer contains all the objects in D in disk. The second

Algorithm 4: Algorithm to construct the LSO index

Input : the set of objects D, an MCTN G
Output: the LSO index

```
1  begin
2      Initialize LSO to be empty;
3      S = findSkyline(D);
4      foreach v ∈ G.V do
5          Create B_v for node v as a block for the second layer of the index;
6          Add all the objects in S to B_v;
7          foreach pair (o, s) where s ∈ S and o ∈ D \ S do
8              if ((s ≻ o) ∧(dist(v, o) < dist(v, s)) ∧(dist(v, o) < τ) )
                 then
9                  B_v.add(o);
10                 Break;
11         LSO.add(v, B_v);
12 return LSO;
```

layer has N blocks where the i-th block B_i contains the pointers pointing to the objects $\mathbb{S}_{v_i}$ in the first layer. The third layer keeps N pointers, where the i-th pointer points to block B_i in the second layer.

Utilizing the index, we can save calculations in two steps. First, we do not need to calculate D_{cand} for each query. Instead, we replace D_{cand} with $\mathbb{S}_{np.end}$ in Algorithm 3. Second, the condition in Line 6 of Algorithm 3 does not need to be checked because the way we build the index guarantees that this condition is satisfied.

Algorithm 4 describes the process to construct the LSO index. The algorithm creates the second layer of the index. For each graph node v, it creates a block B_v with pointers pointing to (i) all the skyline objects of D and (ii) base objects for skyline candidates. The skyline objects in D by considering only the non-spatial attributes of $o \in D$ can be found using the function $findSkyline(D)$(Line 3). Any state-of-the-art skyline finding algorithm (e.g., [19]) can be applied here. The base objects of skyline candidates are constrained using the conditions in Line 8. When an object o is dominated by a skyline object s (i.e., all the attributes of o is larger than or equal to the attributes of s), but the distance from a graph node v to o is less than the distance from v to s, the object o has the possibility to form a skyline candidate according to Lemma 4. Furthermore, we utilize the first approximate heuristic to control that such objects' distance to v need to be less than the approximate range τ.

D. Goodness of approximate results

To evaluate the quality of an approximate result set $\mathcal{R}_{approx}$, we define a goodness score for $\mathcal{R}_{approx}$, $score(\mathcal{R}_{approx}, \mathcal{R})$, where $\mathcal{R}$ is the solution set returned from the exact algorithm.

Let $D_{\mathcal{R}}$ and D_{approx} be the set of distinct base objects in $\mathcal{R}$ and $\mathcal{R}_{approx}$. Given an object $o \in D_{\mathcal{R}} \cap D_{approx}$, let $\mathbb{P}(o, \mathcal{R})$ and $\mathbb{P}(o, \mathcal{R}_{approx})$ contain all the graph-constrained paths of o in $\mathcal{R}$ and $\mathcal{R}_{approx}$ respectively. We can define the goodness of approximate result set by considering several intuitions. First, if the approximate result set share more common base objects with the exact result set, $\mathcal{R}_{approx}$ is better. To represent this intuition, we calculate the score using the base objects that are in both the exact and approximate result sets. The second intuition is that, for a base object o in $D_{\mathcal{R}}$, we prefer to see that its graph-constrained paths are the same or similar to the graph-constrained paths of o in D_{approx}. To represent this intuition, we define a score for each object o as

$$score(o) = max_{p \in \mathbb{P}(o,\mathcal{R}) \wedge p' \in \mathbb{P}(o,\mathcal{R}_{approx})}(sim(p, p'))$$

If $\mathbb{P}(o, \mathcal{R})$ or $\mathbb{P}(o, \mathcal{R}_{approx})$ is empty, this score is 0, which means that the base object is not in both result sets. The similarity score of two paths $sim(p, p')$ is defined to be the cosine similarity of their path cost.

Definition 9 (Goodness of approximate result set)**.** *The goodness of $\mathcal{R}_{approx}$ is defined as*

$$score(\mathcal{R}_{approx}, \mathcal{R}) = \sum_{o \in (D_{\mathcal{R}} \cap D_{approx})} \frac{score(o)}{|D_{\mathcal{R}}|}. \quad (3)$$

Example 6. *Assume that $\mathcal{R}$ contains four path constrained objects, $o_1^{p_{11}}$, $o_2^{p_{21}}$, $o_2^{p_{22}}$, and $o_3^{p_{31}}$, and $\mathcal{R}_{approx}$ consists of three path constrained objects, $o_2^{p'_{21}}$, $o_3^{p'_{31}}$, and $o_4^{p'_{41}}$. Let $p_{21} = p'_{21}$. Then, $D_{\mathcal{R}} = \{o_1, o_2, o_3\}$, $D_{approx} = \{o_2, o_3, o_4\}$, and $D_{\mathcal{R}} \cap D_{approx} = \{o_2, o_3\}$.*

$$score(o_2) = max\{sim(p_{21}, p'_{21}), sim(p_{22}, p'_{21})\} = sim(p_{21}, p'_{21})$$

which is 1 since p_{21}=p'_{21}. For o_3, $score(o_3) = sim(p_{31}, p'_{31})$. Thus, the overall
$score(\mathcal{R}_{approx}, \mathcal{R}) = \frac{score(o_2)+score(o_3)}{2} = \frac{1+sim(p_{31},p'_{31})}{2}$.

The way that we define the goodness score guarantees that it is in the range of [0,1].

VI. Experiments

The algorithms have been implemented using JAVA 1.8. The experiments are conducted on a desktop equipped with an Intel(R) CPU with 3.60 GHz and 32 GB RAM. The transportation network is stored using the Neo4j graph database (https://neo4j.com). Neo4j is adopted because it has shown to be one of the most popular graph databases according to DB-Engines ranking (https://db-engines.com/en/ranking/graph+dbms). The default page size and cache size of the Neo4j database are set to 2 KB and 2 GB respectively. The JAVA APIs of Neo4j are used to manage and access the graph data.

A. Data and query

We utilize both *synthetic* and *real* data to test our proposed methods. For *synthetic* data, we first generate the synthetic graphs to simulate the public transportation networks. The default average degree is set to four to simulate the real-world application where an intersection generally has four different road segments. The range of the degree is 1 to 5. The adjacent graph nodes in a public transportation line are generated such that the distance between them is in a given range and the edge direction does not differ much from the previous edge's direction in the same transportation line. The attribute values on each edge are generated following a normal distribution. Next, we generate the synthetic objects of interest D. For each object $o \in D$, the number of non-spatial attributes is set to be three. For each attribute, the values follow a uniform

distribution in the range of [0,1]. The spatial information of an object represents the x and y coordinates. The synthetic objects are generated by letting the number of graph nodes close to an object follow a Beta distribution. This is to simulate the real application scenario where the number of bus stops within a given range of an object forms a curve similar to the probability density function (PDF) of Beta distribution. Each synthetic graph has a corresponding set of synthetic objects.

The *real data* is obtained from three cities, New York (NY), Los Angeles (LA), and San Francisco (SF), using *rideschedules* (https://rideschedules.com) and Google Maps API. In total, we get 298,022 bus stops and 25,954 objects of interest (e.g., hotels and restaurants) from the three cities. For the objects of interest, we remove the objects that do no have location information and get a set D with 25,854 objects of interest (14,155 for LA, 9,589 for SF, and 2,110 for NY). The MCTN is formed using the bus stops. Each graph node represents a bus stop. Note that close bus stops (e.g., the distance is within fifty meters) are treated as one node. Each graph edge represents a segment in a bus line. The value of each dimension of the graph edge is generated using a uniform distribution with the range of [0, 1]. For the objects of interest, we extract three non-spatial attributes (rating, price, and interestingness) from the information crawled using Google Maps. We find the corresponding city for each bus stop based on its spatial information. Finally, we get 5,127 nodes and 11,152 edges for NY, 9,041 nodes and 13,615 edges for SF, and 12,433 nodes and 22,752 edges for LA.

A query point is a randomly chosen object from D. For each setting in our experiments, we choose 30 queries and report the averaged results.

For experiments on synthetic datasets, the results may show fluctuations because the objects of interest D are generated randomly. To avoid getting unstable results, for the same experiment setting (e.g., D with 10K objects), we generate five sets of D with the same size and report the averaged running results from the five sets.

B. Comparison methods and performance metrics

Since no other methods can be directly applied to solve our proposed problem, we cannot compare our methods with other existing approaches. We compare the two exact search algorithms, *ExactAlg-baseline* and *ExactAlg-improved*. We also compare the two heuristic methods (*Approx-range* and *Approx-mix*) and their corresponding versions that utilize indexes to find approximate solutions. For comparison purpose, we design and implement an A*-based algorithm by using range search to find approximate solutions. This algorithm is denoted as **Approx-A*-range** and is explained below. We did not implement an A*-based exact search algorithm due to its expensive computation (as analyzed in Section IV). Our experimental results in Section VI-F show that even the *Approx-A*-range* cannot outperform our exact search algorithm.

A*-based algorithm using range search. *Approx-A*-range* finds approximate solutions by using range search. It needs to consider every object in $D \setminus q$ as a target object. For a fixed target object o, this method limits the starting graph nodes to be the nodes that are within a distance (τ) from the query point, and the ending graph nodes to be within a distance of o by utilizing the heuristic of approximate range search (Section V-A). This method utilizes a priority queue to keep the graph nodes that have been explored. For a given node v popped out from the priority queue, if the cost of its constrained skyline paths and the estimated cost from v to the target object o is dominated by an existing skyline path, this node is not expanded. The lower bound of the cost from v to a possible ending graph node is calculated using a landmark index as in [15]. Note that for different target objects, we are not naively repeating this process. Instead, we use the solutions that are found so far (may be from different target objects) to conduct pruning.

Algorithm 5: Method *Approx-A*-range*

Input : an MCTN G, a query point q, the set of objects of interest D, the distance threshold τ
Output: the set of skyline solutions $\mathcal{R}$

```
1  begin
2      Initialize the result set R = ∅;
3      Initialize the candidate result set D_cand to contain the objects in D that
         are not dominated by q;
4      foreach o ∈ D_cand do
5          queryResultSourceDestination(q,o,R,τ,G)
6  return R;
```

Algorithm 6: Function *queryResultSourceDestination*

Input : a query point q, a target object o, current skyline solutions $\mathcal{R}$, the distance threshold τ, an MCTN G
Output: updated skyline solutions $\mathcal{R}$

```
1  Initialize a priority min-queue Q to be empty;
2  v_nearest= the nearest graph node to q;
3  Q.enqueue(v_nearest);
4  while Q is not empty do
5      v = Q.pop() ;
6      if v is not visited && dist(q, v) < τ then
7          Create a dummy path dp;
8          v.visited = true;
9          addToSkyline(dp, v.skypaths);
10     foreach path p ∈ v.skypaths do
11         if p.expanded = fasle then
12             foreach v_next ∈ neighbors(v) do
13                 p_next = path(p, v_next);
14                 if the lower-bound cost from p_next to o is not dominated
                     by any solution ∈ R && p_next is a new skyline path
                     from q to v_next then
15                     v_next.Skypaths.add(p_next);
16                     Q.enqueu(v_next);
17 foreach v is visited && dist(v, o) < τ do
18     foreach p_c ∈ v.skypaths do
19         Create o^{p_c} with attributes updated using o's attributes, cost(p_c), and
             dist(p_c.end, o);
20         addToSkyline(o^{p_c}, R) ;
21 return R;
```

The detailed algorithm is shown in Figure 5. We explain how we calculate the lower-bound cost of the constrained prefix path ($p_{next} = path(p, v_{next})$) for a potential target object o (Line 14 in Algorithm 6) in what follows.

Let V_t be the set of the graph nodes that are within distance τ from a possible target object o. Let v'_t be the nearest graph

node to o. The lower bound of the distance from p_{next} to o is $dist(v'_t, o)$.

For a graph node $v_t \in V_t$, the minimum cost for a path from $p_{next}.end$ to v_t on dimension i is calculated using a landmark index. This cost is denoted as $cost_i(p_{next}.end, v_t)$. For the set V_t, the lower-bound cost for a dimension i from $p_{next}.end$ to V_t (denoted as $min\_cost[i]$) is the minimum of $cost_i(p_{next}.end, v_t)$ for all the nodes in V_t. I.e.,

$$min\_cost[i] = MIN\{cost_i(p_{next}.end, v_t) | v_t \in V_t\}.$$

For o, the lower bound of the cost from $p_{next}.end$ to it consists of three components: (i) the attribute values of the object o, (i) the total cost from q to $p_{next}.start$ and from v'_t (o's nearest graph node) to o, and (iii) the estimated lower-bound cost from $p_{next}.end$ to V_t. I.e.,

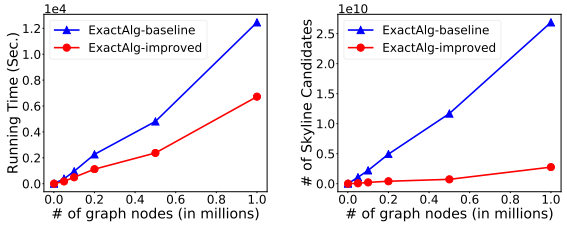
$$\begin{aligned} lb(p_{next}, o) = (&o.attr[1], \cdots, o.attr[d_D], \\ &dist(q, p_{next}.start) + dist(v'_t, o), \\ &cost(p_{next})[1] + min\_cost[1], \cdots, \\ &cost(p_{next})[d_G] + min\_cost[d_G]). \end{aligned}$$

Running time are reported to show the efficiency of the different methods. We do not report the disk I/Os as these algorithms are very computation heavy. Disk I/Os are not as representative as the total query time to evaluate the efficiency of different methods.

Goodness of approximate solutions. For the approximate solution sets, we report their goodness scores.

C. Performance of the exact search methods

In this section, we test the performance of the proposed exact search methods, *ExactAlg-baseline* and *ExactAlg-improved*, using synthetic data. These results are to show the pruning power of the improved algorithm *ExactAlg-improved*.



(a) Query time vs. N (b) # of candidates vs. N

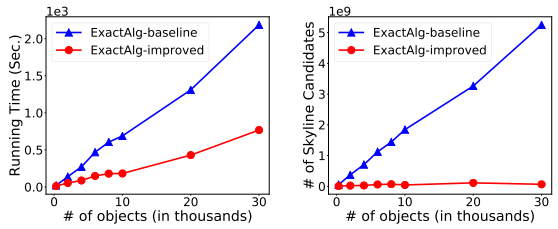
Fig. 5. Exact search algorithms ($|D|$ = 1,000)

# of graph nodes (in millions)	0.001	0.05	0.1	0.2	0.5	1
Running time Speed-up ratio	2.46	2.16	1.94	2.02	2.03	1.85
# of candidates Reduction ratio	22.71	16.89	10.44	12.24	16.25	9.75

TABLE I

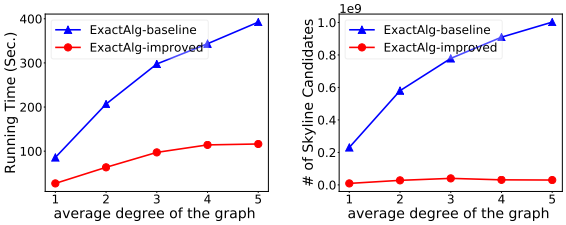
SPEED-UP OF THE *ExactAlg-improved* OVER *ExactAlg-baseline*

The first set of experiments compare these two algorithms using different graphs where the number of graph nodes vary from 1,000 to 1,000,000. In these experiments, we fix the synthetic dataset with 1,000 objects of interest (i.e., $|D|$ = 1,000). The running time of the two algorithms are shown in Figure 5(a). The results show that the *ExactAlg-improved* algorithm can speed up *ExactAlg-baseline* more than 1.8 times (Table I). This is mainly because it reduces the number of skyline candidates (Section IV-B2). The reduction of the skyline candidates can be verified using the results in Figure 5(b) and Table I, which shows that the improved exact search algorithm can reduce the number of skyline candidates to be $\frac{1}{10}$ to $\frac{1}{22}$ of the candidates in the baseline algorithm. The reduction of the running time is less than the decrease of the candidate number because the running time is also affected by other factors. In particular, graph traversal (Step 1 of *ExactAlg-baseline*) is expensive; also, the dominance relationship checking in *addToSkylineImproved* is not a constant, it grows with the number of the candidates.



(a) Query time vs. $|D|$ (b) # of candidates vs. $|D|$

Fig. 6. Exact search algorithms (N=10,000)



(a) Query time vs. d_G (b) # of candidates vs. d_G

Fig. 7. Exact search algorithms (N=10,000, $|D|$=5000)

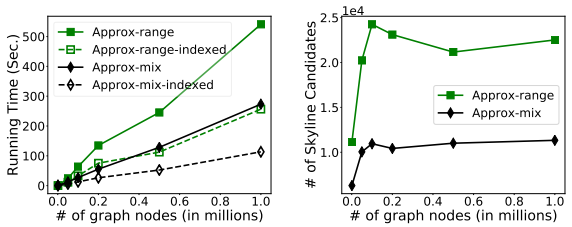
We also conduct experiments by varying the number of data objects (the number of graph nodes N is fixed), and varying the average degree of graph nodes (the number of graph nodes N and the objects of interest D are fixed). Figures 6 and 7 show the running time and the number of skyline candidates for the above settings. The results of these experiments show similar trends as that in Figure 5.

D. Performance of the heuristic approaches

This section evaluates the performance of the heuristic approaches to find approximate solutions.

1) Query time: This set of experiments compares the efficiency (query time) of the different heuristic methods. We run the experiments using two different settings. First, we fix the number of objects of interest to be 1,000 and vary the graph size from 1K to 1M. The results are shown in Figure 8.

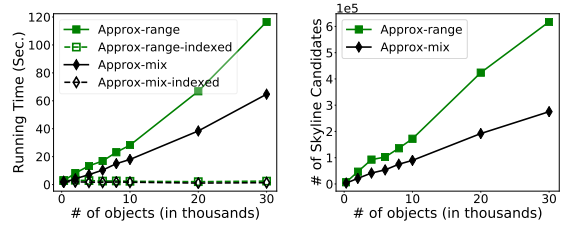
The query time of *Approx-mix* is faster than *Approx-range*. This is because less number of prefix paths are expanded using



(a) Query time vs. N (b) # of candidates vs. N

Fig. 8. Heuristic approaches to find approximate solutions ($|D|$=1,000)

Approx-mix. Due to the same reason, *Approx-mix-indexed* uses less time than *Approx-range-indexed*. The indexed version of the approaches, *Approx-mix-indexed* and *Approx-range-indexed*, use much less time than their non-indexed counterparts. This shows that the index can help improve the efficiency dramatically.



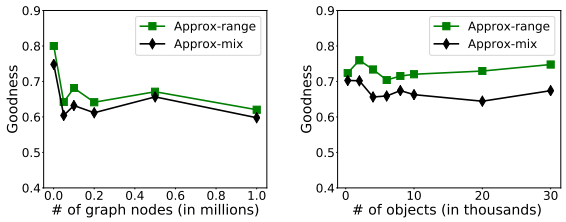
(a) Query time vs. $|D|$ (b) # of candidates vs. $|D|$

Fig. 9. Heuristic approaches to find approximate solutions (N=10,000)

We further compare the heuristic methods by using different settings, fixing the graph (N=10,000) and varying the number of objects of interest ($|D|$). Figure 9 shows the results. Similar to the above setting, *Approx-mix* is faster than *Approx-range*, the indexed version of the methods greatly outperform the non-indexed version, and *Approx-mix-indexed* uses less time than *Approx-range-indexed*.

2) Goodness of approximate solutions: An important measurement of the heuristic methods is the goodness of the approximate solutions. This set of experimental results shows the goodness scores of the approximate solution sets found by the heuristic approaches. Note that, we do not include the results for the indexed version because the indexed and the non-indexed versions return the same result set $\mathcal{R}_{approx}$ for the same query. We use two settings: (i) fixing the number of objects of interest to be 1,000 and varying the graph size from 1,000 to 1,000,000, and (ii) fixing the graph size (N=10,000) and varying the number of objects of interest ($|D|$).

Figure 10 plots the goodness scores of the approximate solution sets. This figure shows clearly that the results returned by *Approx-range* has higher goodness score than those from *Approx-mix*. This is consistent with our intuition that *Approx-mix* removes more valid results. Despite these differences, both algorithms achieve higher than 60% of goodness. Figure 10(a) shows that the goodness is slightly worse for larger graphs (larger N). This is because the skyline paths in larger graphs are typically longer and contain more information than the



(a) $score$ vs. N (b) $score$ vs. $|D|$

Fig. 10. Goodness of $\mathcal{R}_{approx}$, $score(\mathcal{R}_{approx}, \mathcal{R})$

paths used in a smaller graph. Thus the heuristic approaches have higher probability to lose information.

3) Size of indexes: We show the size of indexes that are created for graphs with different size. The index size is

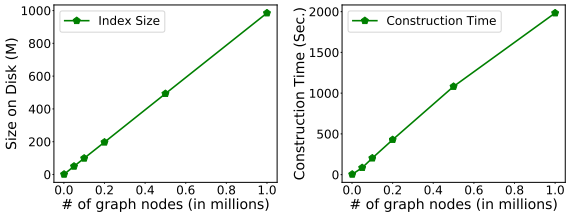
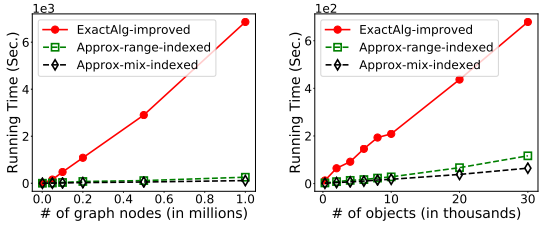


Fig. 11. Index Construction vs. N ($|D|$=1,000)

calculated using the data from the second and the third layers of the index. Figure 11 shows that the index size on disk is linear to the number of nodes in graphs. This is because the the number of pointers in the second layer is the same as the number of objects in the first layer. The number of pointers in the third layer is a fixed ratio of the number of objects.

E. Compare the approaches to find exact and approximate solutions

This section reports experimental results that compare the exact search algorithm *ExactAlg-improved* with the indexed version of the heuristic methods. We use the same experimental setting as Section VI-D by varying N (fix $|D|$) and varying $|D|$ (fix N).



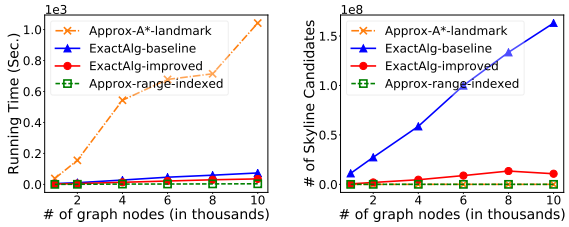
(a) Query time vs. N (b) Query time vs. $|D|$

Fig. 12. Comparison of methods to find exact and approximate solutions

Figure 12 shows the query time of the three methods. The results show that the methods to find approximate solutions dramatically outperform the improved exact search algorithm *ExactAlg-improved*. This is consistent with the design of these heuristic methods. The results of the skyline-candidate number have the same trend in the previous sections. We do not include such results due to space limitation.

F. Compare Approx-A*-range with other methods

This section compares the performance of *Approx-A*-range* with our proposed exact and heuristic search methods, *ExactAlg-baseline*, *ExactAlg-improved*, and *Approx-range-indexed*, using synthetic data.



(a) Query time vs. N (b) # of candidates vs. N

Fig. 13. Comparison of the exact search algorithms and the heuristic approaches that use indexes to find approximate solutions (The tests are on smaller datasets because *Approx-A*-range* takes very long time to finish even for smaller graphs; e.g., it uses more than 8 hours to finish for a graph with $10K$ nodes and $D = 30K$.)

Figure 13 displays the running time and the number of skyline candidates of different algorithms for different graph sizes. Figure 13(a) shows that *Approx-A*-range* runs much slower than all of our proposed methods. It is even slower than the baseline exact search method. This is because that it needs to examine each object of interest as a possible target object and the calculation of the lower bound incurs more computation.

Interestingly, the number of skyline candidates is not proportional to the running time when *Approx-A*-range* is involved. Figure 13(b) shows that both *Approx-range-index* and *Approx-A*-range* could reduce the number of skyline candidates dramatically when compared with the exact search algorithms. This shows that the *Approx-A*-range* heuristic algorithm indeed can reduce the search space by using the lower-bound estimation and the pruning strategy although its running time is still high due to reasons stated above.

G. Experimental results using real datasets

Besides running experiments on synthetic data to test the performance of our proposed methods in different settings, we also test our proposed methods on the real datasets collected for three cities, LA, SF, and NY. For the *Approx-range* and *Approx-mix* methods, the range search range τ is set to 1 Km.

Table II shows the query time of different methods and the goodness scores of the sets of approximate solutions. For the smaller NY dataset, the exact search algorithm can finish running queries using reasonable amount of time (3.34 seconds). For the larger SF and LA datasets, the exact search algorithms run slowly while the heuristic methods are 5 to 8 times faster. We observe that the goodness score of the approximate solutions for *Approx-range* is high (0.79 and 0.93) for SF and LA respectively, but is low (0.39) for NY dataset. This is because the same τ is utilized for all the datasets. A range search using the fixed τ on a larger graph loses less information (i.e., starting nodes for graph traversal) than the search over a smaller graph.

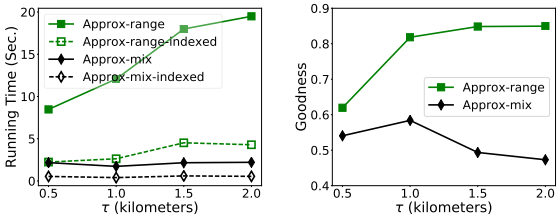
	Query Time (in Sec.)			# of Skyline Candidates		
	NY	SF	LA	NY	SF	LA
ExactAlg-baseline	9.47	175.06	-	3.8×10^7	83×10^7	-
ExactAlg-improved	3.34	60.22	4207.08	318855	0.7×10^7	105×10^7
Approx-range	1.06	11.67	591.59	10199	122510	1.8×10^7
Approx-range-indexed	0.24	**0.23**	14.40	10199	122510	1.8×10^7
Approx-mix	0.09	1.67	75.63	830	19444	995924
Approx-mix-indexed	**0.08**	2.54	**3.60**	**830**	**19444**	**995924**

(a) Running time of different methods on real datasets (For the LA dataset, *ExactAlg-baseline* does not report any results within 5 hours.)

	NY	SF	LA
Approx-range	**0.39**	**0.79**	**0.93**
Approx-mix	0.21	0.56	0.65

(b) Goodness of approximate solution sets

TABLE II
COMPARISON OF DIFFERENT METHODS ON REAL DATASETS



(a) Query time vs. τ (b) Goodness vs. τ

Fig. 14. Query evaluation on SF data with varying τ

We further show the effect of τ to the different search algorithms and show the results in Figure 14. Figure 14(a) shows that *Approx-range* uses more time for bigger τ. This is because bigger τ values allow more graph nodes to be the starting nodes for graph traversal. The goodness values increase with τ for *Approx-range*. However, the goodness values decrease with τ for *Approx-mix*. This is because a larger τ allows more objects to have dummy paths in their skyline paths and this worsens the dummy path issue (Section V-B) when we expand limited number of skyline paths.

VII. CONCLUSIONS

In this paper, we introduce a new variant of skyline queries, which are constrained by MCTNs. The major challenge to address this type of queries comes from the large search space of the network and the huge number of candidates. We propose two exact search algorithms to evaluate such queries. The first exact algorithm *ExactAlg-baseline* can find exact skyline answers, but suffers from expensive calculations. The second exact search algorithm *ExactAlg-improved* improves *ExactAlg-baseline* by implementing several Lemmas. Besides these, we further propose two heuristic methods to find approximate solutions for such queries. The heuristic methods utilize a range search to narrow the space of graph traversal (*Approx-range*) and expand limited number of intermediate paths to reduce the number of candidates (*Approx-mix*). The experimental results on both the synthetic and real data show that *ExactAlg-improved* outperforms *ExactAlg-baseline*. The approximate solutions are reasonably comparable to the exact

solutions, and the methods to find the approximate solutions run much faster than the exact search algorithms.

REFERENCES

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

[2] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Salsa: computing the skyline without scanning the whole sky. In *CIKM*, pages 405–414. ACM, 2006.

[3] Stephan Borzsony, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430. IEEE, 2001.

[4] Chee-Yong Chan, HV Jagadish, Kian-Lee Tan, Anthony KH Tung, and Zhenjie Zhang. Finding k-dominant skylines in high dimensional space. In *SIGMOD*, pages 503–514. ACM, 2006.

[5] James Cheng, Yiping Ke, Shumo Chu, and Carter Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *SIGMOD*, pages 457–468. ACM, 2012.

[6] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with presorting: Theory and optimizations. In *Intelligent Information Processing and Web Mining*, pages 595–604. Springer, 2005.

[7] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[8] Xiaoyi Fu, Xiaoye Miao, Jianliang Xu, and Yunjun Gao. Continuous range-based skyline queries in road networks. *World Wide Web*, 20(6):1443–1467, 2017.

[9] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[10] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[11] Hao He, Haixun Wang, Jun Yang, and Philip S Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316. ACM, 2007.

[12] Ji Hu, Zidong Yang, Yuanchao Shu, Peng Cheng, and Jiming Chen. Data-driven utilization-aware trip advisor for bike-sharing systems. In *Data Mining (ICDM), 2017 IEEE Intl. Conf. on*, pages 167–176, 2017.

[13] Xuegang Huang and Christian S Jensen. In-route skyline querying for location-based services. In *International Workshop on Web and Wireless Geographical Information Systems*, pages 120–135. Springer, 2004.

[14] Hans-Peter Kriegel, Peer Kröger, Peter Kunath, Matthias Renz, and Tim Schmidt. Proximity queries in large traffic networks. In *Proc. of the ACM Intl. symp. on Advances in geographic information systems*, page 21. ACM, 2007.

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

[16] Xuemin Lin, Yidong Yuan, Qing Zhang, and Ying Zhang. Selecting stars: The k most representative skyline operator. In *ICDE*, pages 86–95. IEEE, 2007.

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

[18] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478. ACM, 2003.

[19] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems (TODS)*, 30(1):41–82, 2005.

[20] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. Technical report, 1987.

[21] Michael Shekelyan, Gregor Jossé, and Matthias Schubert. Linear path skylines in multicriteria networks. In *ICDE*, pages 459–470. IEEE, 2015.

[22] Craig Silverstein, Hannes Marais, Monika Henzinger, and Michael Moricz. Analysis of a very large web search engine query log. In *ACm SIGIR Forum*, volume 33 (1), pages 6–12. ACM, 1999.

[23] Yufei Tao, Ling Ding, Xuemin Lin, and Jian Pei. Distance-based representative skyline. In *ICDE*, pages 892–903. IEEE, 2009.

[24] Xike Xie, Hua Lu, Jinchuan Chen, and Shuo Shang. Top-k neighborhood dominating query. In *International Conference on Database Systems for Advanced Applications*, pages 131–145. Springer, 2013.

[25] Bin Xu, Jun Feng, and Jiamin Lu. Continuous skyline queries for moving objects in road network based on mso. In *Proc. of the 12th Intl. Conf. on Ubiquitous Information Management and Communication*, IMCOM, pages 53:1–53:6. ACM, 2018.

[26] Bin Yang, Chenjuan Guo, Christian S Jensen, Manohar Kaul, and Shuo Shang. Multi-cost optimal route planning under time-varying uncertainty. In *ICDE*, 2014.

[27] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. G-tree: An efficient index for knn search on road networks. In *CIKM*, pages 39–48. ACM, 2013.