Spatial Skyline Query Problem in Euclidean and Road-network Spaces

by

Ruijia Mao

B.Sc., Simon Fraser University, 2018
B.Sc., Zhejiang University, 2018

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

Declaration of Committee

Name:	**Ruijia Mao**
Degree:	**Master of Science**
Thesis title:	**Spatial Skyline Query Problem in Euclidean and Road-network Spaces**
Committee:	**Chair:** Thomas Shermer Professor, Computing Science
	Binay Bhattacharya Supervisor Professor, Computing Science
	Abraham Punnen Committee Member Professor, Mathematics
	Ramesh Krishnamurti Examiner Professor, Computing Science

Abstract

With the growth of data-intensive applications, along with the increase of both size and dimensionality of data, queries with advanced semantics have recently drawn researchers' attention. Skyline query problem is one of them, which produces optimal results based on user preferences. In this thesis, we study the problem of spatial skyline query in the Euclidean and road network spaces. For a given data set P, we are required to compute the spatial skyline points of P with respect to an arbitrary query set Q. A point $p \in P$ is a spatial skyline point if and only if, for any other data point $r \in P$, p is closer to at least one query point $q \in Q$ as compared to r and has in the best case the same distance as r to the rest of the query points. We propose several efficient algorithms that outperform the existing algorithms.

Keywords: Spatial skyline query problem; Skyline point; Spatial skyline point; Euclidean d-dimensional space; Road network space

Acknowledgements

I would like to thank Simon Fraser University for offering me the opportunity to study here. I would like to express my sincere gratitude to my supervisor, Dr. Binay Bhattacharya, for being instructive, supportive and patient during my career as a Master student. Finally, huge thank you to my wife, Lin, for inspiring and accompanying me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the growth of decision support systems and data-intensive applications, along with the increasing volume of multi-dimensional data, queries with advanced semantics have recently drawn researchers' attention. Skyline query, which produces optimal results based on user preferences, contains abundant semantics and provides users with exciting insights. A skyline query locates a set of points based on multiple user-specified criteria that no other points are better than them.

In this thesis, we focus on efficient skyline query processing over spatial data. For a given data set P, we are required to compute the spatial skyline points of P with respect to an arbitrary query set Q. A point $p \in P$ is a spatial skyline point if and only if, for any other data point $r \in P$, p is closer to at least one query point $q \in Q$ as compared to r and has in the best case the same distance as r to the rest of the query points. Several algorithms are proposed to solve the spatial skyline query problem in d-dimensional Euclidean space and road network space. Our contributions can be summarized as follows:

- We propose several efficient algorithms that outperform the existing spatial skyline algorithm in the 2-dimensional Euclidean space.
- We study the spatial skyline query problem in the high dimensional Euclidean space.
- We show that an existing algorithm that solves the spatial skyline query problem in the road network space is incomplete.
- We provide a correct and efficient algorithm that outperforms the existing spatial skyline algorithm in the road network space.
- We validate the efficiency of our algorithms through extensive experiments.

The remainder of this chapter is organized as follows. In section 1.1, a brief survey is conducted on work related to the skyline query problem and several variants of skyline query problem. In section 1.2, we pay attention to the spatial skyline query problem. We also provide the formal definition of the spatial skyline query problem along with some useful concepts.

1.1 Theoretical Analysis of Skyline Query

The skyline query problem is also known as the problem of finding maxima/minima of a set of vectors. This problem is first defined and studied by Kung et al. [25]. We are given d totally ordered sets $D_1, D_2, \ldots, D_d$. Let $U = D_1 \times D_2 \times \cdots \times D_d$ denote the Cartesian product of the d sets mentioned above, and let $P \subseteq U$ be a subset of U of size n. Each element in P is a d-dimensional vector. For a vector $v = (v_1, v_2, \ldots, v_d) \in P$, let v_i be the i^{th} component of v. A partial ordering can be defined on P. For any $u, v \in P$, we say v is greater than u, denoted by $v \succ u$, if and only if, every component of v is greater than or equal to their counterparts in u, and at least one component of v is strictly greater than its counterpart component in u. This is equivalent to

$$\forall i \in \{1,2,3,\ldots,d\},\ v_i \geq_i u_i \quad \wedge \quad \exists j \in \{1,2,3,\ldots,d\},\ v_j >_j u_j$$

Here, $\geq_i$ and $>_i$ are defined using the order provided by i^{th} totally ordered set D_i. We can also say v dominates u. The problem of finding maxima of a set of vectors can then be defined as follows. We want to locate a set of elements $S \subseteq P$ of maximum size, such that,

$$\forall s \in S, \quad \forall v \in P \setminus \{s\}, \quad v \not\succ s$$

The worst case computational complexity of this problem can be defined as

$$C_d(n) = min_A \ max_P \ c_d(A, P)$$

where $c_d(A, P)$ represents the number of operations, used by algorithm A, to find all the maximal elements given a dataset P with d dimension. $C_d(n)$ denotes the maximum number of operations used to find all maximal elements for any data set of size n with the fastest algorithm. $C_d(n)$ represents the worst case theoretical bounds for finding maxima in a set of vectors problem. For $d = 1$, the dataset P is a totally ordered set. We can find the maximum element using one search over P in $O(n)$ operations. Kung et al. proves the following bounds of $C_d(n)$ [25]:

$$\begin{aligned}
&C_d(n) \leq O(n \log n) && \text{for } d = 2, 3 \\
&C_d(n) \leq O(n(\log n)^{d-2}) && \text{for } d \geq 4 \\
&C_d(n) \geq \Omega(\lceil \log n! \rceil) && \text{for } d \geq 2
\end{aligned}$$

Since $\lceil \log n! \rceil$ is $\Theta(n \log n)$, the bounds are tight for $d = 2, 3$. When it comes to $d \geq 4$, whether the bounds are tight remains an open problem.

The problem of finding the maxima of a set of vectors drew the attention of the database community, and they name it the skyline operator or the skyline query problem [4]. However, the skyline query problem tends to find the minima, instead of the maxima, of a set of vectors. The minima are also known as the skyline points. Researchers have proposed several algorithms to solve the skyline query problem efficiently. For example, Börzsönyi et al. [4] introduced a divide and conquer algorithm; Tan et al. [22] presented the index-based bitmap algorithm; Kossmann et al. [13] came up with the nearest neighbor algorithm; Papadias et al. [32] proposed the branch and bound skyline algorithm, and Bartolini et al. [1] introduced the sort and limit skyline algorithm.

Besides the general skyline query problem, variants of the skyline query problem draw researchers' attention as well. In constraint skyline query problem [11][16][31][32], users are interested in finding the skyline points of a subset of the original dataset. One or more constraints may define the subset. Each constraint is usually expressed as a range along a dimension of the original dataset. A group skyline query [19][28][49] focuses on groups of points instead of single points. It retrieves skyline groups that are not dominated by any other groups. Our thesis concentrates on the spatial skyline query problem, where we take advantage of the spatial relationship between different points to accelerate the skyline computation. The details of the spatial skyline query problem are covered in the next section.

1.2 Narrowing down: Spatial Skyline Query Problem

While the database community has extensively studied the skyline query problem, many failed to consider the spatial relationship between data objects. For example, consider the scenario of finding hotels for a trip to a city. A user wants to find a hotel that is close to the museum, the beach and the airport: Figure 1.1 displays several candidate hotels and 3 locations of interest.

If we treat this problem as a general skyline query problem, it is a 3-dimensional problem because, for a hotel, we need to compare its distances to three locations (museum, beach and airport) with other hotels. However, the algorithms solving the general skyline query problem ignore the geometric relationship between those locations and the hotels. Moreover, if we have more locations to consider, the dimensionality and computation time will increase dramatically. Consequently, the research community proposed the spatial skyline query problem, which takes advantage of the geometric relationship between data objects to simplify the computation. We now define the spatial skyline query problem and provide some related concepts.

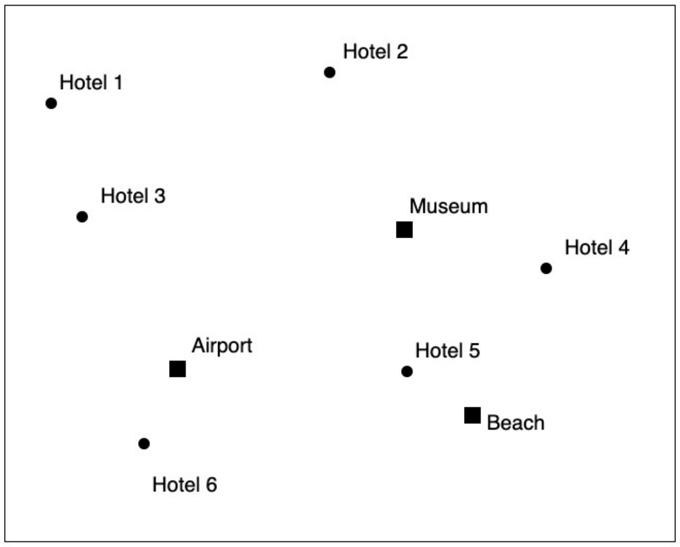


Figure 1.1: Hotel example

1.2.1 Problem Definition

In the spatial skyline query problem, a set of data points P is given so that data points of P can be preprocessed. For each spatial skyline query, a set of query points Q will be given. For each query set Q, we need to locate all the spatial skyline points $S \subseteq P$ with respect to Q.

The distance between any two points p_1 and p_2 can be computed using a distance function $d(p_1, p_2)$. In our thesis, the distance $d(\cdot, \cdot)$ is either the Euclidean distance or the shortest path distance in a road network. Before showing the formal definition of the spatial skyline query problem, we need to present several definitions.

Definition 1.1 (Domination)**.** For two data points $p_1, p_2 \in P$, p_1 dominates p_2 with respect to a query set Q, written as $p_1 \prec_Q p_2$, if and only if, distances from p_1 to all the query points are less than or equal to the corresponding distances from p_2 to all the query points, and for some $q_j \in Q$, the distance from p_1 to q_j is strictly less than the distance from p_2 to q_j. Put it into mathematical form:

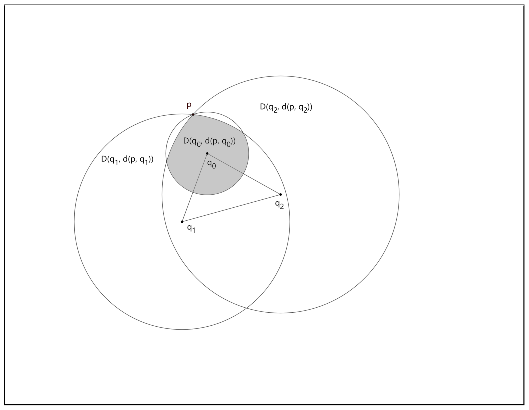
$$p_1 \prec_Q p_2 \iff \forall q_i \in Q,\ d(p_1, q_i) \le d(p_2, q_i)\ \wedge\ \exists q_j \in Q,\ d(p_1, q_j) < d(p_2, q_j).$$

Observation The dominance relationship is transitive, meaning that if $p_i \prec_Q p_j$, and $p_j \prec_Q p_k$, then $p_i \prec_Q p_k$. The proof of the transitivity of the dominance relationship is straightforward.

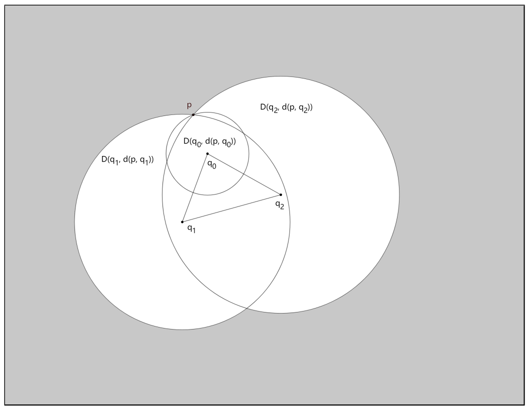
Note that we use $\not\prec_Q$ to indicate a point cannot dominate another point with respect to the query set Q. We use the term **dominance check** to represent the process of testing whether one data point dominates another data point with respect to Q.

If two data points p_1, p_2 are located at different positions, but have the same distances to all query points, p_1 and p_2 don't dominate each other. Take the hotel finding problem mentioned above as an example. If there are only two query points, for example, the museum and the beach, it is possible that two hotels have the same distances to the beach and the museum, but they are at different locations on the map. When searching for the optimal hotels, the algorithm should return both hotels.

Definition 1.2 (Dominator/dominating region)**.** The dominator region of a point $p \in P$ with respect to Q is a region such that for any point $p_d \in P$ in the region, p_d dominates p with respect to Q. Thus the dominator region is the intersection of disks $D(q,\ d(p,\ q))$ centred at all the query points q of Q with radius $d(p, q)$. The dominating region of p with respect to Q, on the other hand, is a region where points inside the region are dominated by p. The dominating region is unbounded and is formed by the complement of the union of all the disks $D(q,\ d(p,\ q))$ centred at all the query points q of Q with radius $d(p, q)$.



(a) Dominator region of p



(b) Dominating region of p

Figure 1.2: Dominator and dominating regions of p

The gray regions of Figure 1.2 are the dominator region and the dominating region of a point p with respect to three query points $\{q_0, q_1, q_2\}$. Note that the shapes of the two regions are irregular especially with the increase of the number of query points.

The definition of the spatial skyline point is now given:

Definition 1.3 (Spatial Skyline Point)**.** A data point $p \in P$ is a spatial skyline point with respect to Q if and only if p cannot be dominated by other data points of P. Put it into the mathematical form:

$$p \text{ is a spatial skyline point} \iff \forall p_i \in P \setminus \{p\},\ p_i \not\prec_Q p$$

We use the term **skyline check** to describe the procedure of determining whether a data point of P is a spatial skyline point with respect to Q.

Definition 1.4 (Spatial Skyline Query)**.** Given a data point set P and a query point set Q, a spatial skyline query locates all the spatial skyline points $S \subseteq P$ with respect to Q

In a spatial skyline query problem, S is used to denote the set that contains all the spatial skyline points of P with respect to Q.

1.2.2 Related Concepts

Various algorithms and data structures in computational geometry are used in this thesis. We briefly introduce them in this section.

Convex Hull A subset C of a Euclidean space $\mathbb{R}^d$ is convex, if and only if, for any pair of points $p, q \in C$, the line segment $\overline{pq}$ is completely contained in C. The *convex hull* $CH(P)$ of a set of points P is the smallest convex set that contains P. To be more precise, it is the intersection of all convex sets that contain P [10]. The convex hull of n points can be constructed in $O(n \log n)$ time in 2 and 3 dimensional spaces [10]. For $d \geq 4$, the convex hull of n points can be constructed in $\Theta(n^{\lfloor \frac{d}{2} \rfloor})$ time [10]. Figure 1.3 shows an example of the convex hull of a set of points.

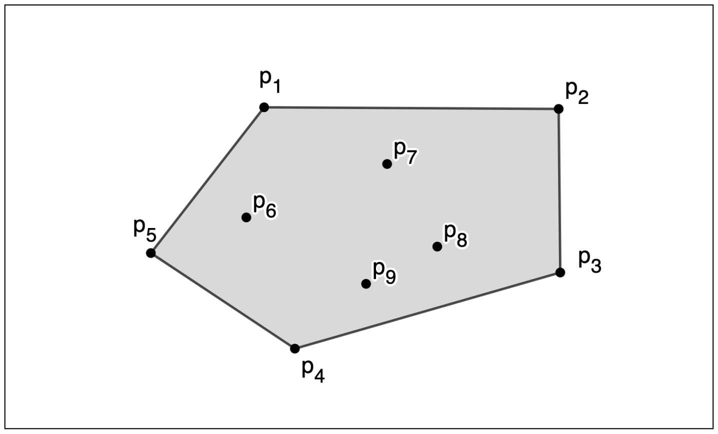


Figure 1.3: Convex hull of a set of points

Voronoi Diagram Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n distinct points in d-dimensional Euclidean space. We define the Voronoi diagram of P as the subdivision of the Euclidean space into n cells, one for each point in P, with the property that a point q lies in the cell corresponding to a point p_i if and only if $d(q, p_i) \leq d(q, p_j)$ for each $p_j \in P$ with $j \neq i$. We denote the Voronoi diagram of P by $Vor(P)$ [10]. Planar Voronoi diagram of a set of n points can be computed in $O(n \log n)$ time. For $d \geq 3$, the d-dimensional Voronoi Diagram of n points can be constructed in $O(n^{\lfloor \frac{d+1}{2} \rfloor})$ time [5][23]. This can be done by transforming the d-dimensional Voronoi diagram construction problem into a $(d+1)$-dimensional convex hull construction problem. Figure 1.4 shows the Voronoi diagram of a set of points along with the subdivision of the plane.

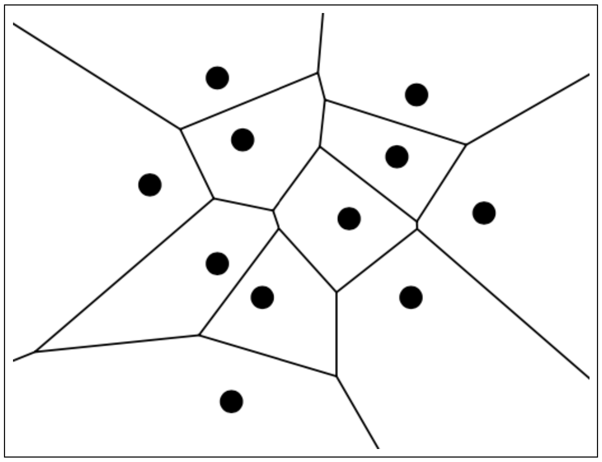
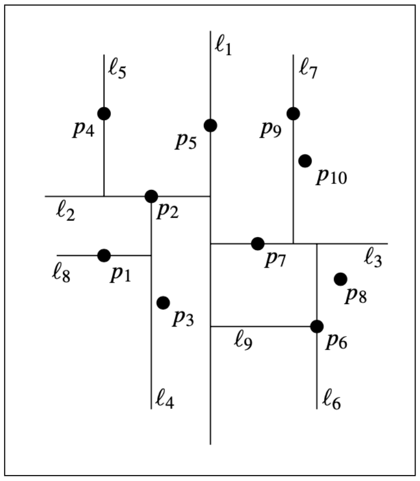
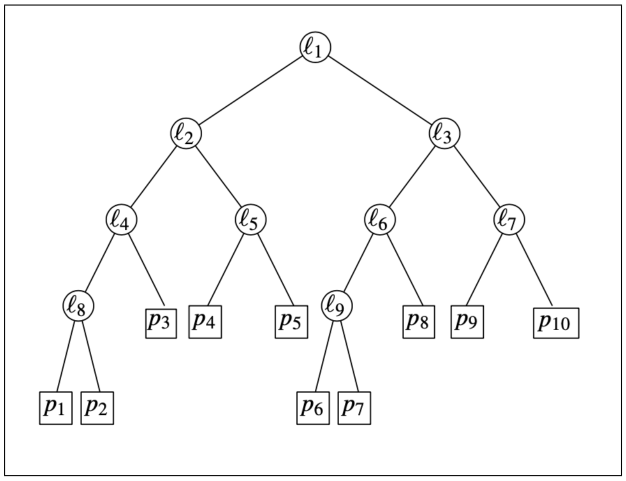


Figure 1.4: Voronoi diagram of a set of points [10]

KD-Tree KD-tree is a spatial index that supports point-location query and range query. A KD-tree for a set P of n points in the d-dimensional space uses $O(n)$ storage and can be built-in $O(dn \log n)$ time. A rectangular range query on the KD-tree takes $O(n^{1-1/d} + I)$ time in the worst case, where I is the number of reported points [2][10]. The nearest neighbor of a point can be computed by exploring $O(\log n)$ nodes on average [15]. Figure 1.5 shows an exmaple of the KD-tree of a set of points. Figure 1.5a shows the planar subdivision of the KD-tree and Figure 1.5b shows the corresponding tree structure.



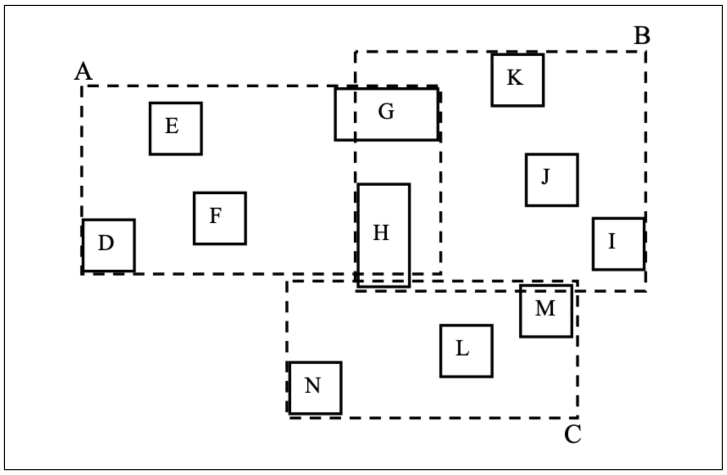
(a) Planar subdivision of the KD-tree



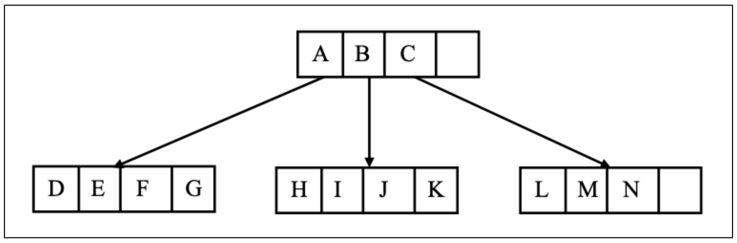
(b) Tree structure of the KD-tree

Figure 1.5: KD-tree of a set of points [10]

R-Tree R-tree is a dynamic spatial index based on B-tree. It is used for the dynamic organization of a set of d-dimensional geometric objects representing them by the minimum bounding d-dimensional rectangles (for simplicity, MBRs in the sequel). Each node of the R-tree corresponds to the MBR that bounds its children. The leaves of the tree contain pointers to the objects instead of pointers to the children nodes [45]. R-tree supports dynamic operations like insertion or deletion. Theoretical analysis shows $\Omega(n^{1-1/d})$ as the lower-bound for range query of R-tree in the worst case. [21] Recent studies show that logarithmic average search time can be achieved in the 2-dimensional case [18]. Figure 1.6 shows an example of the R-tree for a set of objects.



(a) MBRs of the R-tree



(b) Tree structure of the R-tree

Figure 1.6: R-tree of a set of objects [45]

Shortest Path Algorithms Dijkstra's algorithm is generally used to solve the single-source shortest-paths problem on a weighted, undirected graph $G = (V, E)$ for the case in which all edge weights are non-negative [9]. The runtime complexity for Dijkstra's algorithm is $O(min(|V| \log |V| + |E| \log |V|, |V|^2))$ [9].

There are several speed-up techniques for the Dijkstra's algorithm [43]. One technique is the bidirectional search algorithm [29][38] . The bidirectional search runs a forward Dijkstra's algorithm from the source vertex and a reverse Dijkstra's algorithm from the target vertex. The algorithm stops when the two searches meet. The bidirectional search algorithm works well in practice, even though it has the same worst case time complexity as the Dijkstra's algorithm. Another speed-up technique is A* algorithm [35][47]. It adds lower-bound heuristics to unvisited vertices so that the search wavefront can expand toward the target vertex. Additionally, if preprocessing is allowed, the landmark algorithm can be used [17]. The algorithm chooses a small fixed-sized subset of vertices $L \subset V$ as "landmarks". In the preprocessing step, for each landmark $l \in L$, the algorithm computes its distances to all the vertices $v \in V$. For each landmark $l \in L$, a lower-bound heuristic from a vertex v to the target vertex t is computed by $h_l(v) = d(l, t) - d(l, v)$. The heuristic for v is then defined to be the maximum of all heuristics $h(v) = max(max_l(h_l(v)),\ 0)$. Researches show that the landmark algorithm outperforms the A* algorithm [17].

1.2.3 Chapter Organization

The remainder of this thesis is organized as follows. In Chapter 2, several algorithms are provided to solve the 2-dimensional Euclidean spatial skyline query problem. In Chapter 3, algorithms are proposed that solve the spatial skyline query problem in high dimensional Euclidean space. In Chapter 4, we first point out a mistake in an existing algorithm for solving the spatial skyline query problem in road network space. We then propose a correct and more efficient algorithm that solves the spatial skyline query problem in road network space. In Chapter 5, conclusions are drawn showing our contributions in solving the spatial skyline query problem in different metric spaces.

Chapter 2

Spatial Skyline Query Problem in 2 Dimensional Euclidean Plane

In this chapter, we focus on the spatial skyline query problem in the 2-dimensional Euclidean plane. A point in 2-dimensional spatial skyline query problem has two attributes, representing the x value and the y value of a spatial location. The spatial skyline query problem retrieves all the skyline points S from the data set P with respect to an arbitrary query set Q. Sections in this chapter are organized as follows: Section 2.1 provides a survey on related researches. In section 2.2, we propose several new algorithms. In section 2.3, experiments are conducted to evaluate the efficiency of our algorithms.

2.1 Related Work

This section provides a brief survey on work related to the spatial skyline query problem in 2-dimensional Euclidean plane along with some useful theorems and algorithms.

Sharifzadeh and Shabi [36] first studied the spatial skyline query problem. They proposed the Voronoi-based Spatial Skyline Algorithm (VS^2) that takes advantage of the Voronoi diagram and the convex hull to accelerate the computation. Son et al. [39] pointed out an error in VS^2 and proposed a new algorithm called Enhanced Spatial Skyline Algorithm (ES) that correctly solves the spatial skyline query problem. Son et al [40][41] also studied the spatial skyline query problem in L1 space. They proposed an algorithm that can return a set of spatial skyline points in $O(|P| \log |P|)$ time where it is assumed that $|Q| \leq |P|$. Bhattacharya et al. [3] applied the compact Voronoi diagram to solve the spatial skyline query problem, which transforms the problem into finding sites with non-empty cells in an additive Voronoi diagram with a convex distance function. The complexity of their algorithm is $O((|Q| + |P|) \log(|Q| + |P|))$. Ji et al. [20] proposed an algorithm to solve the problem in parallel using MapReduce.

2.1.1 Useful Theorems

We now introduce several useful theorems. Sharifzadeh et al. [36] suggest a way to reduce the size of query points using the convex hull.

Theorem 2.1 (Convex Hull [36])**.** *The skyline points S of a set of data points P with respect to Q do not depend on any non-extreme query point $q \in Q$.*

Observation If a query point lies completely inside the convex hull of the query point set $CH(Q)$, it can be safely removed without affecting the final results of the spatial skyline query. In other words, if we spend $O(|Q|\log(|Q|))$ time to construct the convex hull of all the query points [10], the number of query points can be reduced from $|Q|$ to $|CH(Q)|$. Theorem 2.1 also implies that we can add new query points inside the $CH(Q)$ without affecting the final results of a spatial skyline query. Additionally, theorem 2.1 leads to the following corollary:

Corollary 2.1.1 (The Dominance Check Method [39])**.** *The bisector of two data points intersects the interior of $CH(Q)$ if and only if they do not spatially dominate each other.*

Observation If $CH(Q)$ is known, we can test whether a data point dominates another data point in $O(\log|CH(Q)|)$ time after constructing the convex hull of Q in $O(|Q|\log|Q|)$ time. The logarithmic time dominance test can be achieved by testing whether the perpendicular bisector of the two data points intersects with $CH(Q)$ using a binary-search-like technique [10].

Besides using the convex hull to reduce the number of the query points, some skyline points can be located without performing any dominance check [36].

Theorem 2.2 (Seed Skyline Points [36])**.** *If the Voronoi cell of $p \in P$ intersects with $CH(Q)$, p is a skyline point.*

Observation We can quickly determine a subset of skyline points using the Voronoi diagram of data points.

2.1.2 Existing Approaches

Based on the theorems mentioned above, we now go over the Voronoi-based Spatial Skyline Algorithm (VS^2) [36] and the Enhanced Spatial Skyline Algorithm (ES) [39]. ES is based on VS^2, and will be used later in the experiment section.

VS^2 first selects a random query point q_s. We call q_s the source query point since it is the query point the algorithm starts with. VS^2 then starts to traverse the Voronoi diagram of data points in sorted order from the source query point q_s. For each data point p, the algorithm first checks if p has at least one spatial skyline neighbor. If p has at least one spatial skyline neighbor, VS^2 checks if p is a skyline point. Meanwhile, the algorithm pushes the unvisited Voronoi neighbors of p along with their distances to q_s into a min-heap H. VS^2 stops when the heap H is empty. However, as first noted by Son et al. [39], VS^2 may fail to find all spatial skyline points. We present VS^2 in Algorithm 1 and will discuss why the algorithm is wrong and provide the corrections later.

Algorithm 1 VS^2 [36]

```
Input: Data Point Set $P$, Query Point Set $Q$
Output: Skyline Point Set $S$
1: Heap $H = \{(\text{NN}(q_s), dist(\text{NN}(q_s), q_s))\}$
2: Visited = $\{\text{NN}(q_s)\}$
3: Extracted = $\{ \}$
4: while $H$ is not empty do
5:   $(key, p) = H.top()$
6:   if $p \in$ Extracted then
7:     remove $(key, p)$ from $H$
8:     if Vor($p$) intersects with $CH(Q)$ or $p$ is not dominated by $S$ then
9:       add $p$ to $S$
10:    end if
11:  else
12:    add $p$ to Extracted
13:    if $S$ is empty or a Voronoi neighbor of $p$ is in $S$ then
14:      for each Voronoi neighbor $p'$ of $p$ do
15:        if $p' \notin$ Visited then
16:          add $p'$ to Visited, and add $(p', dist(p', q))$ to H
17:        end if
18:      end for
19:    end if
20:  end if
21: end while
```

In Algorithm 1, NN(q_s) denotes the data point in P that is closest to the source query point q_s, and $dist(\text{NN}(q_s), q_s))$ represents the Euclidean distance from NN(q_s) to q_s. The algorithm maintains a min-heap H, which keeps track of a set of data points currently being considered. The Visited array stores all the points visited, and the Extracted array stores all the points whose Voronoi neighbors have been visited. The algorithm continuously pops the top node (key, p) out of H. If the data point p is in Extracted, the algorithm performs dominance check for p against all the skyline points found so far. The data point p is accepted if no skyline point dominates it. If p is a skyline point or it has a neighbor who is a skyline point, VS^2 pushes the Voronoi neighbors of p into H. The algorithm stops when H is empty. In other words, the algorithm stops after it examines all the points whose neighbors have at least one skyline point.

However, as pointed out by Son et al. [39], VS^2 may fail to locate all the skyline points. A skyline point s can be found in a position where the Voronoi neighbors of s and the Voronoi neighbors of those Voronoi neighbors are all dominated by some other skyline points, as shown in Figure 2.1.

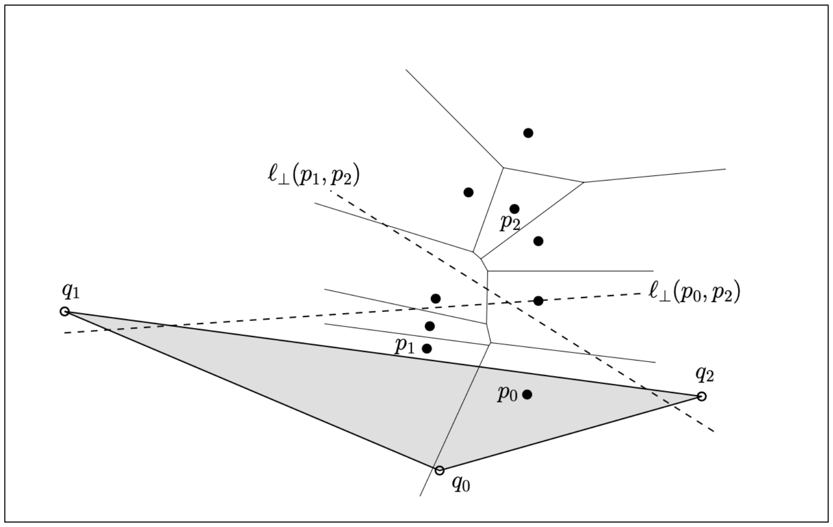


Figure 2.1: An example showing VS^2 may fail to find all skyline points [39]

In Figure 2.1, q_0, q_1, and q_2 are query points. Other points are data points. p_0, p_1, and p_2 are skyline points. All other data points are dominated by p_0 and p_1. If VS^2 is applied to this dataset, it will fail to locate p_2 as a skyline point, because the Voronoi neighbors of p_2 and the Voronoi neighbors' neighbors of p_2 are all non-skyline points. VS^2 simply stops without examining p_2.

After pointing out the problem of VS^2 in [39], the authors propose the ES algorithm to compute all the skyline points correctly. The ES algorithm comes from VS^2 with several changes to it. The main difference between VS^2 and ES is that, instead of using the Voronoi diagram to traverse the whole data point set, ES traverses data points according to their distances to the source query point. Before presenting the algorithm, we need to introduce the following lemma.

Lemma 2.3 (Domination Order [39])**.** *Suppose all data points are sorted according to their distances to a randomly selected query point $q \in CH(Q)$. Suppose the sorted data point set are $\{p_0, p_1, p_2, \ldots, p_{n-1}\}$ where $d(p_i, q) \leq d(p_j, q)$, $\forall i < j$. For an arbitrary data point p_i, points dominating p_i, if exists, can only appear from the set $\{p_0, p_1, p_2, \ldots, p_{i-1}\}$.*

Observation Suppose all data points are sorted by their distances to the source query point and stored in an array A. The number of dominance tests required to verify whether an arbitrary data point p_i is a skyline point can be reduced to the number of skyline points that appear before p_i in A.

Note that both VS^2 and ES take advantage of the bounding box of the dominating regions of the skyline points. In general, the number of the skyline points is much smaller than the number of data points. Moreover, skyline checks are costly. The number of points that go through skyline checks should be kept as low as possible. Observe that data points that lie inside the dominating regions of skyline points found so far can be pruned out. However, calculating the exact shape of dominating regions is a complex problem, because a dominating region of a point is shaped by the union of at most $|Q|$ number of disks. As a result, calculating the bounding box of the dominating region can be used as an approximation to prune out some non-skyline data points. Figure 2.2 shows an example of the dominating region of a point p with respect to three query points $\{q_0, q_1, q_2\}$. The dominating region of p is the gray area in the figure, which is shaped by three disks: $D(q_0, d(p, q_0))$, $D(q_1, d(p, q_1))$ and $D(q_2, d(p, q_2))$. The shape of the union of $|Q|$ disks is difficult to compute, especially when $|Q|$ is large. However, a bounding box enclosing the dominating region, as shown by the dashed boxes in the figure, is easy to compute.

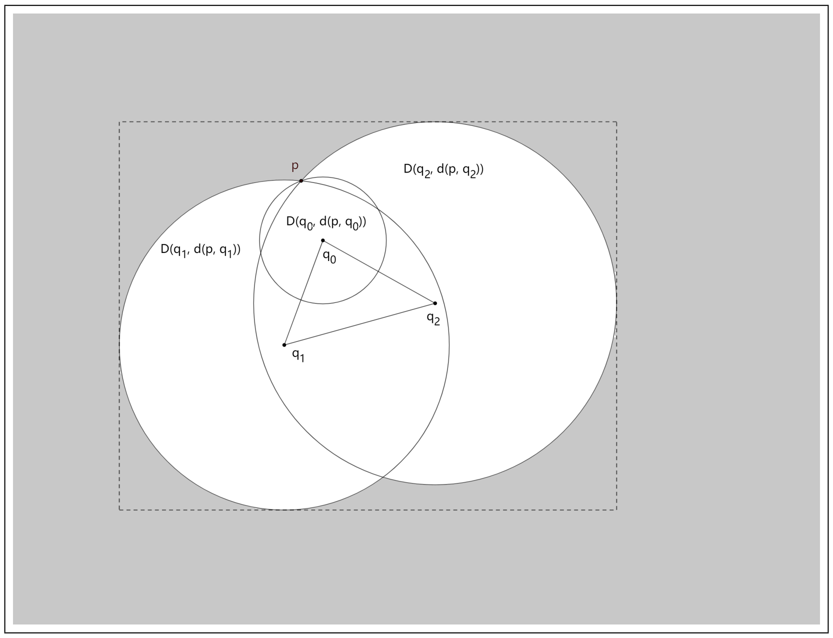


Figure 2.2: The gray region is the dominating region of p

The bounding box algorithm essentially computes the bounding boxes of dominating regions of all skyline points. Any data point outside the intersection bounding box of those bounding boxes can be pruned out safely. For those points that survive the pruning, skyline checks are needed to decide whether the points are skyline points. We now present two definitions: tentative point and candidate point.

Definition 2.1 (Tentative Point)**.** A data point $p \in P$ is called a tentative point if p is visited by an algorithm in a spatial skyline query.

We will use T_S to denote the set containing all the tentative points. The size of T_S is called tentative size. Note that although, in the worst case, T_S can be P, an algorithm usually visit, seen in the experimental results, a small subset of data points $T_S \subseteq P$ in one query.

Definition 2.2 (Candidate Point)**.** A data point $p \in P$ is called a candidate point if p goes through a skyline check.

We will use C_S to denote the set containing all the candidate points. The size of C_S is called candidate size. Note that C_S is a subset of T_S.

This paragraph briefly introduces the *ES* algorithm. The algorithm first locates a set of seed skyline points, as mentioned in Section 2.2. *ES* then calculates the bounding box B of the non-dominating regions of all seed skyline points. Seed skyline points dominate all the data points outside B. Examining the data points inside B is sufficient to locate all the yet to be computed skyline points. The algorithm then retrieves all the data points inside B and treats them as tentative points. *ES* then sorts all tentative points according to their distances to a randomly selected query point q_s. The bounding box B will become smaller during the spatial skyline computation, and tentative points inside the current B are considered as candidate points. The algorithm then performs dominance checks to each candidate point in sorted order with skyline points found so far. If skyline points found so far cannot dominate the candidate point, the candidate point is accepted as a skyline point and inserted into S. Whenever a new skyline point is added to S, the algorithm computes the bounding box of the new skyline point's non-dominating region. It then updates B by computing then intersection of B and the newly computed bounding box. *ES* stops when there is no unvisited candidate point inside B. Algorithm 2 below is the pseudocode of the *ES* algorithm.

Algorithm 2 *ES*: Enhanced Spatial Skyline

```
Input: Data Points P, Query Points Q
Output: Skyline Points S
 1: Find all data points in $P$ that are inside $CH(Q)$, or whose Voronoi cells intersect with
    $CH(Q)$, and put those points into $S$
 2: Generate a bounding box $B$ of the non-dominating region of $S$
 3: $P_n \leftarrow$ the distances from $q_s \in CH(Q)$ to data points in $B$
 4: Sort $P_n$ in ascending order
 5: for $i = 0$ to $|P_n|$ do
 6:     if $P_n[i]$ is not dominated by $\forall s \in S$ then
 7:         insert $P_n[i]$ into $S$ ($P_n[i]$ is a skyline point)
 8:         update $B$ using $P_n[i]$
 9:     end if
10: end for
```

Theorem 2.4. *The preprocessing time complexity of the ES algorithm is* $O(|P| \log |P|)$*, where* $|P|$ *is the size of the data point set. The query time complexity of the ES algorithm is* $O(|Q| \log |Q| + \sqrt{|P|} + |T_S| \log |T_S| + |C_S||S| \log |CH(Q)|)$ *in the worst case, where* $|T_S|$ *is* *the tentative size, $|C_S|$ is the candidate size, $|S|$ is the size of the spatial skyline point set, and $|Q|$ is the size of the query point set.*

Proof. In the preprocessing, the ES algorithm constructs the Voronoi diagram of all the data points, which takes $O(|P| \log |P|)$ time. It stores all the data points in an R-tree. Constructing R-tree for a set of $|P|$ points takes $O(|P| \log |P|)$ time [45].

For a spatial skyline query, $O(|Q| \log |Q|)$ is spent on constructing the convex hull of the query points of Q. Computing bounding boxes of seed skyline points costs $O(|S||CH(Q)|)$ time. Retrieving all the tentative points C_S inside the bounding box takes $O(\sqrt{|P|} + |T_S|)$ time in the worst case [45]. Sorting the tentative points costs $O(|T_S| \log |T_S|)$ time. Performing one dominance check for one candidate point against one skyline point takes $O(\log |CH(Q)|)$ time, as mentioned in Corollary 2.1.1. There are $|C_S|$ candidate points, and each of the candidate points needs at most $|S|$ dominance checks. Therefore, the total time spent on dominance checks is $O(|C_S||S| \log |CH(Q)|)$. Summing them up, we get the query time complexity $O(|Q| \log |Q| + \sqrt{|P|} + |T_S| \log |T_S| + |C_S||S| \log |CH(Q)|)$. □

2.2 Our Algorithms

In this section, we demonstrate our work on solving spatial skyline queries in a 2-dimensional plane. The bounded skyline check method (BSC) is first described in section 2.2.1. The index skyline check method (ISC) is then proposed in section 2.2.2. A geometric approach, called the bounding line method, which can further reduce the number of candidate points, is proposed in section 2.2.3.

2.2.1 Reducing the Number of Dominance Tests

In the ES algorithm, Son et al. [39] bound the number of dominance tests for a candidate point by $O(|S|)$, which is the size of skyline points found so far. However, if we view dominance checking in another perspective, dominance checking distinguishes skyline points from non-skyline points. We can achieve the same effect if we can compute the dominator region of a candidate point and test whether it contains any other skyline points. If the dominator region contains no skyline point, the candidate point is a skyline point.

Figure 2.3 demonstrates an example of a dominator region of p. In this figure, there are 3 query points, namely $\{q_0, q_1, q_2\}$. For a data point p, its dominator region, colored in gray, is the intersection of three disks: $D(q_0, d(p, q_0))$, $D(q_1, d(p, q_1))$ and $D(q_2, d(p, q_2))$. Any point in the dominator region of p dominates p with respect to Q.

The advantages and disadvantages of the dominance check method mentioned in Corollary 2.1.1 are the opposite of those of the dominator regions. In general, the dominator region of a candidate point only contains a small subset of skyline points. To complete a skyline check, we only need to know whether any skyline point lies inside the dominator region. The dominance check method needs to go over all the skyline points in order to

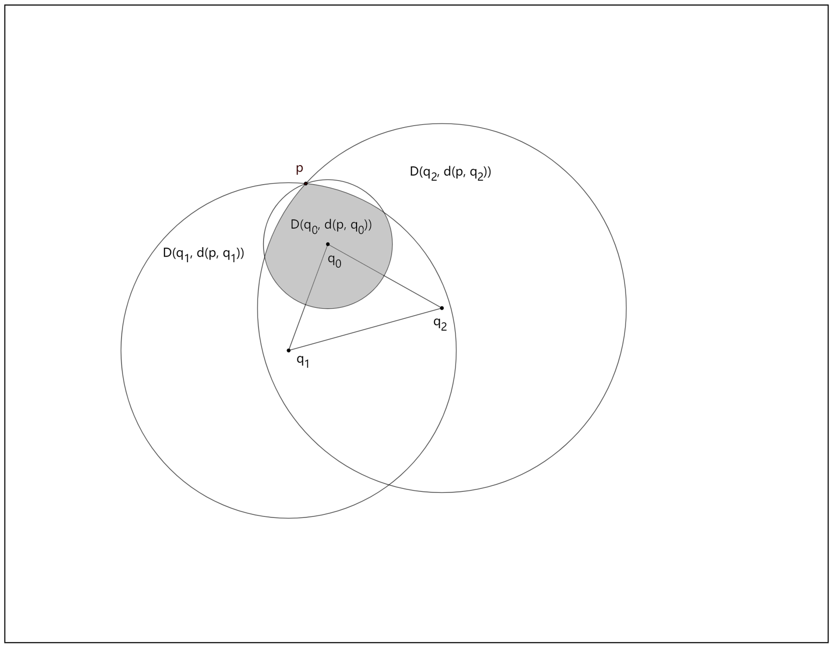


Figure 2.3: The gray region is the dominator region of p

complete a skyline check. However, the dominator region of a candidate point is hard to compute. The boundary of a dominator region of a candidate point consists of up to $|Q|$ number of circular arcs. Determining its shape and testing whether a point lies inside is difficult. In contrast, the dominance check method does not require complicated computation of the dominator region. It is natural for us to combine those two methods.

We now propose the bounded skyline check method (BSC) that combines the advantages of the dominance check method and the dominator region. Instead of computing the dominator region directly, BSC approximates it with a bounding box of the dominator region. For a candidate point p_c, if no skyline point lies inside the bounding box, p_c is a skyline point. Otherwise, the algorithm performs dominance checks between p_c and any skyline points lying inside the bounding box. If no skyline point dominates p_c, BSC accepts p_c as a skyline point. The pseudocode of BSC is provided in Algorithm 3.

Theorem 2.5 (BSC Time Complexity). *Suppose S' denotes the subset of the skyline points S, which lie inside the intersecting bounding box. A skyline check for a candidate point using BSC takes $O(|CH(Q)| + \sqrt{|S|} + |S'| \log |CH(Q)|)$ time in the worst case.*

Proof. $O(|CH(Q)|)$ time is spent on computing the bounding box of the dominator region. A range query locating skyline points inside the bounding box takes $O(\sqrt{|S|} + |S'|)$ time in the worst case, or $O(\log |S| + |S'|)$ time on average [18][21]. The dominance check method mentioned in Corollary 2.1.1 will be performed between p_c and the skyline points of S', which cost $O(|S'| \log |CH(Q)|)$ time. □

Algorithm 3 *BSC*: Bounded Skyline Check

```
Input: Candidate Point $p_c$, Query Point Set $Q$, Current Skyline Point Set $S$
Output: Whether $p_c$ is a skyline point
 1: for all $q_i \in Q$ do
 2:     Compute the orthogonal bounding box $B_i$ of the disk $D(q_i, d(q_i, p_c))$
 3: end for
 4: Bounding box of dominator region $B = \bigcap_{i=1}^{|CH(Q)|} B_i$
 5: if there is no skyline point of $S$ in $B$ then
 6:     return True ($p_c$ is a skyline point)
 7: else
 8:     if $p_c$ is not dominated by any skyline points in $B$ then
 9:         return True ($p_c$ is a skyline point)
10:     else
11:         return False ($p_c$ is not a skyline point)
12:     end if
13: end if
```

2.2.2 Another Method for Skyline Check

In this section, a new method for skyline check, the index skyline check method (ISC), is proposed. Instead of considering the data points in Euclidean space, they can be viewed in query space, where each dimension of a data point represents its distance to a distinct query point of $CH(Q)$. A data point $p = (x_p, y_p)$ in Euclidean space now becomes $p = (d(p, q_1), d(p, q_2), \ldots, d(p, q_{|CH(Q)|}))$ in query space. A data point can now be represented as a $|CH(Q)|$-dimensional vector. In query space, the dominator region of a point p is represented by the hypercube formed by p and the origin. If there is a point s inside the hypercube, s dominates p.

Figure 2.4 demonstrates an example of a data point in query space where there are only two query points $\{q_1, q_2\}$. The x-axis and y-axis represent the distance from the data points to q_1 and q_2, respectively. p'_i is the transformed data point of p_i, $i = 1, 2, 3, 4, 5$. Take p'_1 as an example. The gray area is the dominator region of p'_1, and the hatching area is the dominating region of p'_1. Since p'_5 is inside the gray area, p_5 dominates p_1.

The skyline check for a candidate point now becomes checking whether any skyline point lies inside the hypercube formed by the candidate point and the origin of the query space.

We now briefly introduce the ISC algorithm. ISC uses a spatial index, like KD-tree or R-tree, to store all the skyline points. When it encounters a candidate point, ISC transforms the candidate point from Euclidean space to query space. The method then checks whether any skyline point lies inside the $|CH(Q)|$-dimensional hypercube formed by the candidate point and the origin of the query space. If there is no skyline point inside, the candidate point is accepted as a skyline point and inserted into the spatial index. Otherwise, the

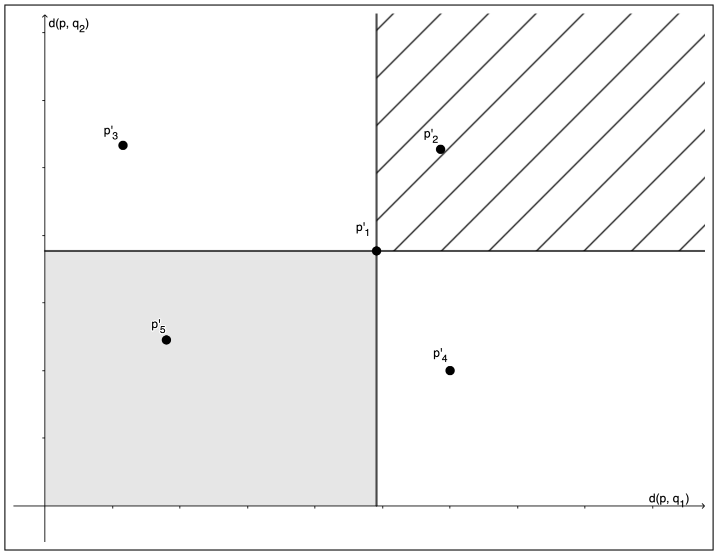


Figure 2.4: A demonstration of data points in query space

candidate point is dominated by some skyline points and can be discarded. The pseudocode of ISC is presented in Algorithm 4.

Algorithm 4 ISC: Index Skyline Check

```
Input: Candidate Point $p_c$, Query Point Set $Q$, Current Skyline Point Set $S$, Spatial Index
    of Skyline Point Set $I$
Output: Whether $p_c$ is a skyline point
1: for all $q_i \in CH(Q)$ do
2:     $d_i = d(p_c, q_i)$
3: end for
4: $p'_c = (d_1, \ldots, d_{|CH(Q)|})$
5: Check whether any skyline point lies inside the hypercube $c$ formed by $p'_c$ and the origin

6: if there is no skyline point in $c$ then
7:     Insert $d$ into $I$
8:     return True ($p_c$ is a skyline point)
9: else
10:    return False ($p_c$ is not a skyline point)
11: end if
```

In Algorithm 4, the inputs additionally include a spatial index I. The hypercube c, in line 5, is the hypercube formed by candidate point p' and the origin.

Theorem 2.6 (ISC Time Complexity)**.** *A skyline check for a candidate point using ISC can take* $O(|CH(Q)||S|^{1-\frac{1}{|CH(Q)|}})$ *time in the worst case.*

Proof. Computing distances from a candidate point to all the query points takes $O(|CH(Q)|)$ time. If a KD-tree or R-tree is used, a range query takes $O(|CH(Q)||S|^{1-\frac{1}{|CH(Q)|}})$ time in the worst case. □

2.2.3 Bounding the Number of Candidate Points

In this section, we provide a geometric method, the bounding line method, that can further reduce the number of candidate points and tentative points. The bounding line method, together with the bounding box method mentioned in Section 2.1, can prune out more data points. The bounding line method utilizes the geometric relationship between data points and the minimum enclosing disk of query points.

Theoretical Analysis

The main idea of the bounding line method is to approximate the dominating region of a data point. Instead of trying to directly compute the irregular shape of the dominating region with respect to a convex polygon. Our method tries to compute the dominating region with respect to a query disk. Comparing to a random convex shape, the shape of a disk is more regular and controllable.

As can be seen from Figure 2.5, the gray area is the dominating region for a point p with respect to D. Here, D is a query disk instead of a query convex polygon. Notice that if the bounding line L in Figure 2.5 can be calculated, it is a relatively tighter bound for the area lying on the right-hand side of L comparing with the bounding box algorithm. Moreover, if data points scatter around the disk, more bounding lines can appear around the circle, and a tighter convex bounding polygon can be obtained. We call line L, in Figure 2.5, the bounding line for p.

The question now becomes how to calculate the bounding line given a data point p and a disk D. This problem can be transformed to the following geometry problem.

We are given a disk $D(o, r)$ centering at o with radius r, and a point p outside $D(o, r)$. Since the disk is rotation-invariant, we can assume that the point p lies on the x-axis. Suppose q is a randomly selected point on the boundary of $D(o, r)$. A disk $D(q, \alpha)$ is constructed using q as the center and the distance between p and q, $d(p, q) = \alpha$, as the radius. Suppose the rightmost point of $D(q, p)$ is $j = (x_j, y_j)$. We want to find a point q on the boundary of $D(o, r)$ whose j has the largest x_j value. Figure 2.6 visualizes the problem. Some assumptions are made here. β represents the distance between o and p. θ is the angle between $\overline{op}$ and $\overline{oq}$. α is the distance between q and p, and x_q is the x value of q. This problem can be further formulated as given p, o, and r, find a $q = (x_q, y_q)$ that maximizes $x_q + \alpha$. It turns out that a point q on the boundary of $D(o, r)$ can be found to maximize $x_q + \alpha$.

Theorem 2.7. *Given a point p and a disk $D(o, r)$ centering at o with a radius r, $q_x + \alpha$ is maximized, when $d(p, q) = d(o, p)$.*

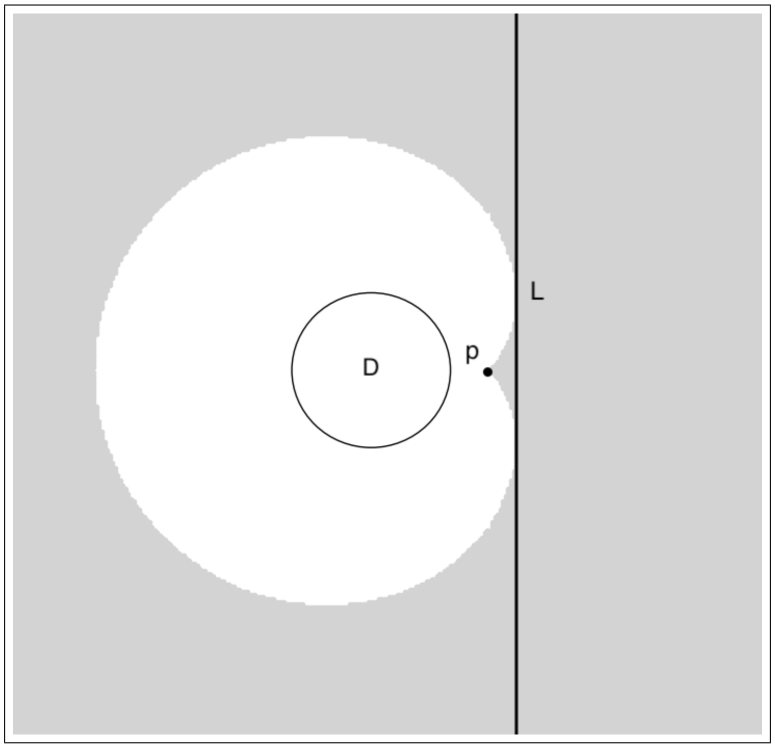


Figure 2.5: Dominating region of p with respect to D

Proof. Notice that $d(q, j) = d(q, p) = \alpha$, because q and j lie on the same circle centering at q. Since $\cos\theta = \frac{\beta^2+r^2-\alpha^2}{2r\beta}$,

$$q_x + \alpha = r\cos\theta + \alpha \tag{2.1}$$

$$= r(\frac{\beta^2 + r^2 - \alpha^2}{2r\beta}) + \alpha \tag{2.2}$$

$$= \frac{\beta^2 + r^2 - \alpha^2}{2\beta} + \alpha \tag{2.3}$$

Notice that β and r are constants. So, (2.3) is a function of α. If we take derivative of the function $F(\alpha) = \frac{\beta^2+r^2-\alpha^2}{2\beta} + \alpha$, we get $F'(\alpha) = 1 - \frac{\alpha}{\beta}$. This leads us to the conclusion that $q_x + \alpha$ will reach its maximum when $\alpha = \beta$. □

Notice that the bounding line method is still applicable in a higher dimension as well. We now present an algorithm that can reduce the number of candidate points using the bounding line method in the next section.

Maintaining Bounding Lines Using Symmetric Points

Suppose we have $|S|$ skyline points lying outside the minimum enclosing disk of the query points. The bounding lines of those skyline points form a convex polygon with at most $|S|$ edges. We want to test whether a data point p lies inside the convex polygon. If p

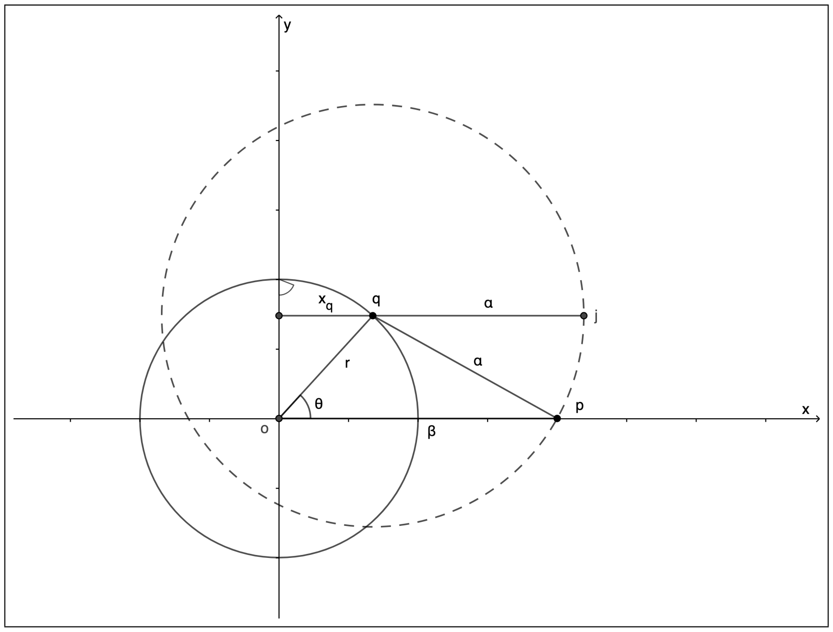


Figure 2.6: Finding the bounding line

lies inside, it is accepted as a candidate point. Otherwise, p is dominated by some current skyline point and can be discarded. Although testing whether a point is inside a convex polygon is easy, dynamically maintaining the convex polygon together with an efficient inclusion test is relatively tricky.

In order to maintain the convex polygon dynamically, we can transform the convex polygon maintaining problem into a nearest neighbor problem involving insertions and deletions. Instead of keeping track of the bounding lines, we can store a point symmetric to the center of the minimum enclosing disk o with respect to the bounding line. Figure 2.7 shows an example of the transformation. In the example, the convex polygon of five pints $CH(R)$ is transformed where $R = \{r_0, r_1, r_2, r_3, r_4\}$. Each edge $\overline{r_i r_j}$ of $CH(R)$ is represented by a point r'_{ij} symmetric to o with respect to $\overline{r_i r_j}$.

Symmetric points, along with the center o, can be stored using any spatial index data structure that supports fast dynamic nearest neighbor queries. For example, an R-tree containing n points can retrieve the nearest neighbor of an arbitrary point in $O(\log n)$ time on average [45].

In the bounding line algorithm, the center o is the minimum enclosing disk of the query points $D(Q)$. The edges are produced by the current skyline points outside $D(Q)$. When a tentative point p arrives, the algorithm searches for its nearest neighbor in the spatial index. If the nearest neighbor of p is o, p lies inside the convex polygon formed by bounding lines and is accepted as a candidate point. Otherwise, we can discard p because it lies outside the convex polygon, and it is dominated by some skyline points found so far.

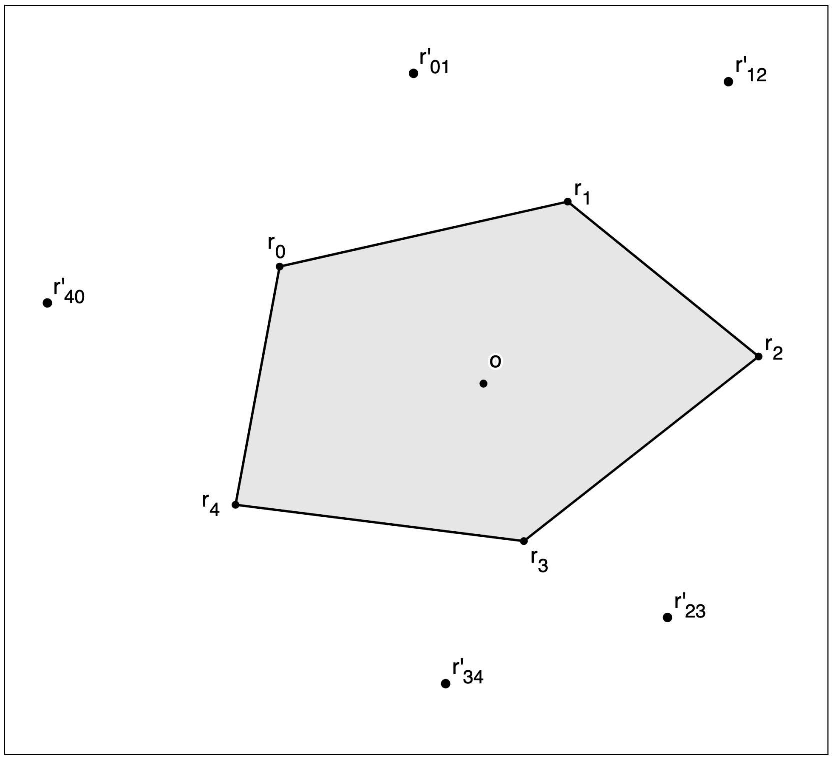


Figure 2.7: Transform the convex polygon maintaining problem into a nearest neighbor problem

Bounding Line Method

We now describe the bounding line method (BL) in detail. There are two operations in BL: Pruning operation decides whether a data point is a tentative point. Insertion operation adds a new symmetric point generated by the new skyline point to the existing spatial index. BL initializes the spatial index with the center of the minimum enclosing disk of the query points o.

The pruning operation is shown in Algorithm 5. For a data point p, BL checks whether p lies inside the bounding convex polygon by performing a nearest neighbor query. This operation will accept p as a candidate point if the nearest neighbor of p is o.

Algorithm 5 BL: Bounding Line Method (Pruning Operation)

```
Input: Data Point $p$, Symmetric Points $SP$
Output: Whether $p$ is a tentative point
1: Find the nearest neighbor $NN(p)$ of $p$ in $SP$
2: if $NN(p)$ is the disk center $o$ then
3:     return True ($p$ is a tentative point)
4: else
5:     return False ($p$ is not a tentative point)
6: end if
```

The insertion operation is shown in Algorithm 6. When a candidate point s is accepted as a new skyline point, the algorithm computes the symmetric point of the bounding line of s with respect to o, and stores the symmetric point in a specific data structure.

Algorithm 6 BL: Bounding Line Method (Insertion Operation)

```
Input: Skyline Point $s$, Minimum Enclosing Disk of $Q$ $D(Q)$ , Symmetric Points $SP$
Output: Symmetric Points $SP$
1: if $s$ is outside $D(Q)$ then
2:     Compute the symmetric point $s'$ of the bounding line of $s$ with respect to $o$
3:     Insert $s'$ into $SP$
4: end if
5: return $SP$
```

Since computing the symmetric point only takes constant time, the time complexity of the BL algorithm depends on the data structure used to perform the nearest neighbor. For example, if R-tree is used, pruning and insertion cost $O(\log |S|)$ time on average [45].

2.3 Experiments

This section compares our algorithms with the ES algorithm [39] on several aspects. In section 2.3.1, our algorithms using different methods mentioned in the previous sections are presented. In section 2.3.2, the experimental settings are reported. In section 2.3.3, the evaluation results show that our algorithms outperform the ES algorithm.

2.3.1 Algorithms and Implementation

In this section, we propose several algorithms that use different methods mentioned in the previous sections. Our algorithms have three components.

1. **traversal component**: determines the order with which the tentative points of $T_S \subseteq P$ are visited

2. **pruning component**: prunes a subset of non-skyline points quickly and produces the candidate point set C_S

3. **skyline check component**: performs a specific skyline check algorithm to all the candidate points and determines the skyline point set S

Take the ES algorithm [39] as an example. ES traverses tentative points after they are sorted from the source query point. The algorithm uses the bounding box method to prune out some non-skyline points quickly. It then uses the dominance check method to acquire all the skyline points. We now list several components that will be used by our algorithms.

Traversal Components There are two traversal components considered: sorting [39] and the Voronoi diagram [36]. The sorting method sorts the tentative points according to their distances to the source query point. It then traverses the tentative point set accordingly. The Voronoi diagram traversal method starts with the data point closest to the source query point. It pushes all the unvisited Voronoi neighbors of the data point into a min-heap, along with their distances to the source query point. Data points inside the heap are considered as tentative points. The Voronoi diagram traversal method stops when the heap is empty.

Pruning Components The bounding box method mentioned in Section 2.1.2 is one of the pruning components. For each skyline point s found so far, it computes an orthogonal bounding box enclosing the dominating region of s. The method keeps track of the intersection of the bounding boxes of all the skyline points found so far. Tentative points inside the bounding box are candidate points. The bounding box and bounding line method additionally apply the bounding line method mentioned in Section 2.2.3 to the bounding box method. It can further reduce the bounding region and thus decreases the size of the candidate set and the tentative set.

Skyline Check Components Three skyline check methods are available: the dominance check method mentioned in Section 2.1.2, BSC mentioned in Section 2.2.1, and ISC mentioned in Section 2.2.2.

Table 2.1 shows several algorithms using different components mentioned above. Although we only list five algorithms here, more algorithms are available by combining different components. In our experiments, we will compare the performance of three algorithms: ES, $BSCS$, and $ISCSBL$.

	ES	ESV	$BSCS$	$ISCS$	$ISCSBL$
Traversal component	Sorting	Voronoi diagram	Voronoi diagram	Voronoi diagram	Voronoi diagram
Pruning component	Bounding box	Bounding box	Bounding box	Bounding box	Bounding box & Bounding line
Skyline check component	Dominance check	Dominance check	BSC	ISC	ISC

Table 2.1: Configuration of different algorithms

2.3.2 Experimental Settings

In this section, we introduce the datasets and parameter settings of our experiments. The datasets are set up in a way similar to the datasets used by Son et al.[39]. Two types of datasets are used. One is a synthetic dataset. The other is a real-world point of interest (POI) dataset [27].

Synthetic dataset A synthetic dataset contains up to one hundred thousand uniformly distributed points in 2D. The domain of the data points is restricted to a unit square. The query points follow a normal distribution using a randomly chosen data point as their mean. Three parameters can be adjusted to thoroughly test the algorithms, namely, dataset size, query point set size, the standard deviation of the normal distribution of the query points. We use σ to represent the standard deviation of the normal distribution of the query points. Table 2.2 lists the configuration of the parameters.

Parameter	Setting
Data size	25K, 50K, 75K, 100K
Query size	10, 25
Standard deviation of normal distribution of query points (σ)	0.02, 0.04

Table 2.2: Configuration of synthetic datasets

As can be seen from Table 2.2, the synthetic dataset has four different sizes, 25K, 50K, 75K, and 100K. They also have five two query point sizes and two different σ. Twenty queries are generated for each specific configuration. The average response time, the size of the candidate point set, and the number of spatial skyline points are recorded.

Point Of Interest (POI) dataset The POI dataset contains 21058 locations in California [27] (dataset used by Son et al. [39]). We only use the vertex locations of the dataset. The edge data are not used in this chapter. The distance between any two points is the Euclidean distance. Figure 2.8 shows a visualization of the dataset. The configuration of the size of the query points and σ are the same as mentioned in Table 2.2.

Experimental Equipment The experiments are implemented using Python 3.7.4 and are carried out on a Macbook Pro with a 3.1GHz Intel i5 CPU and 8 GB memory.

2.3.3 Experimental Results

In this section, the experimental results are provided to show the comparison results of different algorithms. Table 2.3 and Table 2.4 show the experimental data for the synthetic dataset and the POI dataset, respectively. All the figures in this section are generated from these two tables.

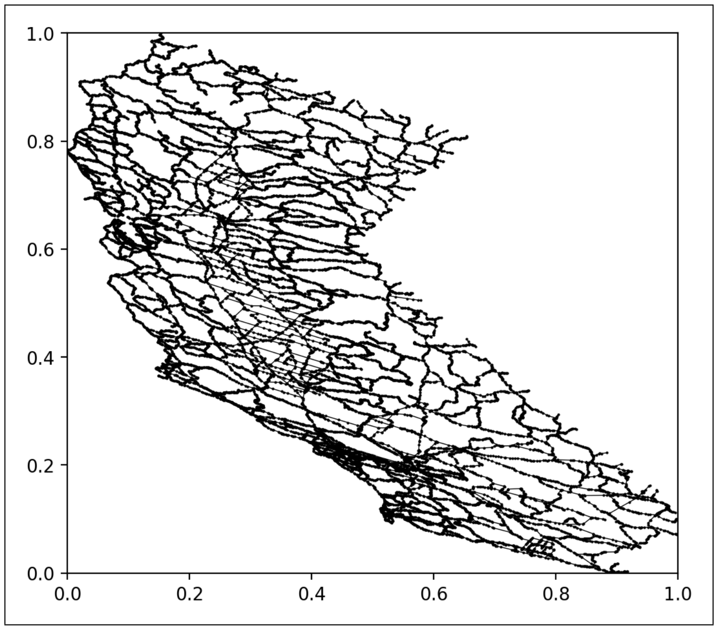


Figure 2.8: Visualization of POI dataset

Effect of data size

Both $BSCS$ and $ISCSBL$ outperform ES under various data sizes. $ISCSBL$ performs slightly better than $BSCS$. Note that the spatial skyline size, the candidate size and the tentative size are considerably smaller than the number of data points. If other parameters remain the same, the rates of increase in the spatial skyline size, the candidate size and the tentative size are almost the same as the rate of increase in the data size. When the data size becomes large, the total response time of ES increases dramatically, while the response times of $BSCS$ and $ISCSBL$ remain low. The reason is that the algorithms need to perform skyline checks on a larger candidate point set with the increase of data size. Additionally, the cost of a skyline check also increases because of the increase of spatial skyline size. The response time of ES increases dramatically because the skyline check component of ES examines a large portion of the spatial skyline points for each candidate point. As for $BSCS$ and $ISCSBL$, since they only examine a small subset of skyline points per skyline check, their response times do not grow as much as ES. The bounding box and bounding line method tends to prune out more data points with the growth of data size. The pruning effect will become more obvious in high dimensional Euclidean space (discussed in Chapter 3). Figure 2.9 shows the experimental results on the synthetic dataset with respect to different data sizes. The query size is 15 and σ is 0.04 in the figure.

Effect of query size

Both $BSCS$ and $ISCSBL$ outperform ES under different query sizes. $ISCSBL$ performs slightly better than $BSCS$. The increase of the query size has a smaller impact on the

Synthetic Dataset									
Data size		25000				50000			
Query size		5		15		5		15	
σ		0.02	0.04	0.02	0.04	0.02	0.04	0.02	0.04
Spatial skyline size		79.95	278.6	146.65	517.	151.95	595.65	257.15	887.85
Total time (s)	ES	0.3	3.96	0.91	12.36	1.14	17.55	2.82	36.32
	$BSCS$	0.05	0.27	0.1	0.51	0.12	0.76	0.21	1.04
	$ISCSBL$	**0.03**	**0.14**	**0.07**	**0.27**	**0.07**	**0.34**	**0.12**	**0.47**
Candidate Size	$ES/BSCS$	196.95	833.1	360.75	1402.	423.95	1885.95	658.95	2361.15
	$ISCSBL$	**189.8**	**810.8**	**351.85**	**1373.1**	**414.05**	**1860.05**	**638.05**	**2327.25**
Tentative Size	ES	545.95	2279.2	1004.05	3901.95	1235.5	5375.3	1864.4	6982.65
	$BSCS$	253.95	947.45	440.8	1554.35	510.8	2057.05	763.4	2543.95
	$ISCSBL$	**244.6**	**922.15**	**428.9**	**1520.45**	**498.45**	**2027.8**	**737.85**	**2504.75**
Data size		75000				100000			
Query size		5		15		5		15	
σ		0.02	0.04	0.02	0.04	0.02	0.04	0.02	0.04
Spatial skyline size		234.15	740.35	402.9	1459.85	303.35	930.2	498.4	1823.4
Total time (s)	ES	2.69	29.13	7.36	101.55	4.4	46.76	11.43	152.2
	$BSCS$	0.2	1.	0.38	2.17	0.29	1.56	0.49	2.89
	$ISCSBL$	**0.11**	**0.43**	**0.21**	**0.91**	**0.16**	**0.58**	**0.26**	**1.15**
Candidate Size	$ES/BSCS$	663.6	2263.85	1122.95	4352.75	902.5	3036.3	1366.45	5295.25
	$ISCSBL$	**649.05**	**2233.8**	**1084.45**	**4277.9**	**885.9**	**2987.15**	**1342.55**	**5189.2**
Tentative Size	ES	1967.9	6125.5	3114.1	12361.1	2551.8	8113.9	3687.65	14632.75
	$BSCS$	770.	2444.85	1254.95	4618.4	1028.25	3251.3	1512.35	5585.75
	$ISCSBL$	**752.85**	**2410.5**	**1211.55**	**4535.45**	**1008.05**	**3196.3**	**1484.15**	**5471.**

Table 2.3: Experimental results of algorithms in the 2-dimensional Euclidean space using synthetic dataset

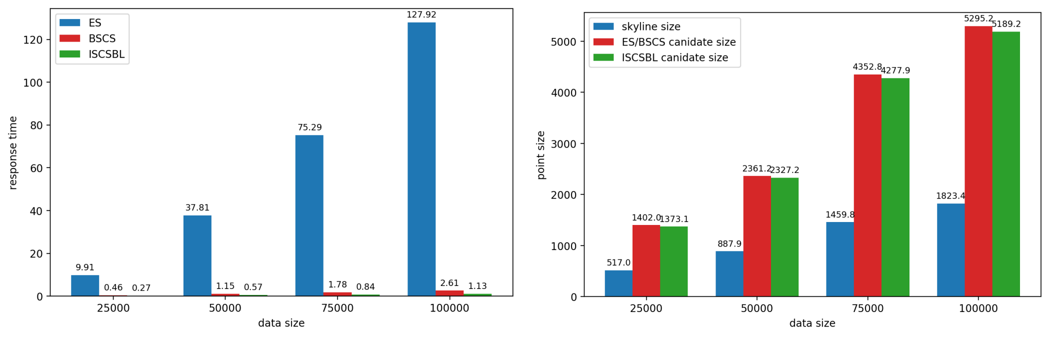
spatial skyline size and the candidate size comparing to the impact of data size. The reason is that the size of $CH(Q)$, not Q, affects the final results of a spatial skyline query. The query size has a similar impact on the POI dataset. $ISCSBL$ still has the best performance among the three algorithms. Its response time remains steady with the growth of the query size, while the response time of ES grows dramatically. Figure 2.10 shows the experimental results on the POI dataset with respect to the query size. The data size is 21058 and σ is 0.04 in the figure.

Effect of σ

Both $BSCS$ and $ISCSBL$ outperform ES under different standard deviations of the normal distribution of query points. $ISCSBL$ performs slightly better than $BSCS$. σ has a more significant impact than the data size or the query size on both the synthetic dataset and the POI dataset. The reason is that σ affects the size of the region covered by the query points. If the query points cover a broader region, the algorithms will visit more candidate points and retrieve more skyline points. The increase of the response time of $ISCSBL$ is the slowest among the three algorithms because it has the most efficient skyline check method.

POI Dataset					
Query size		5		15	
σ		0.02	0.04	0.02	0.04
Spatial skyline size		179.35	503.35	233.35	954.5
Total time (s)	ES	1.65	14.2	2.45	39.01
	$BSCS$	0.16	0.62	0.2	1.18
	$ISCSBL$	**0.08**	**0.26**	**0.11**	**0.46**
Candidate Size	$ES/BSCS$	463.	1386.3	579.1	2219.65
	$ISCSBL$	**446.65**	**1370.05**	**564.5**	**2186.05**
Tentative Size	ES	1281.4	3166.85	1693.45	5060.5
	$BSCS$	551.25	1522.85	676.65	2381.25
	$ISCSBL$	**531.75**	**1505.**	**658.25**	**2342.95**

Table 2.4: Experimental results of algorithms in the 2-dimensional Euclidean space using POI dataset



(a) Response time (b) Skyline/candidate size

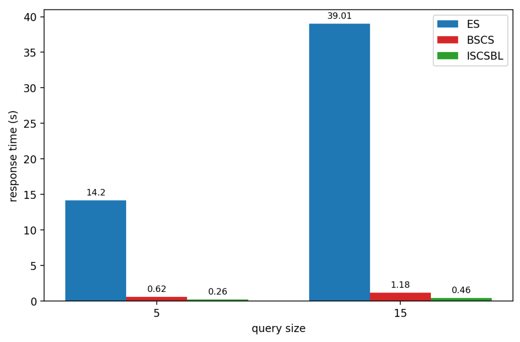
Figure 2.9: Experimental results on the synthetic dataset with respect to the data size

Figure 2.11 shows the experimental results on the POI dataset with respect to σ. The data size is 21058 and the query size is 15 in the figure.

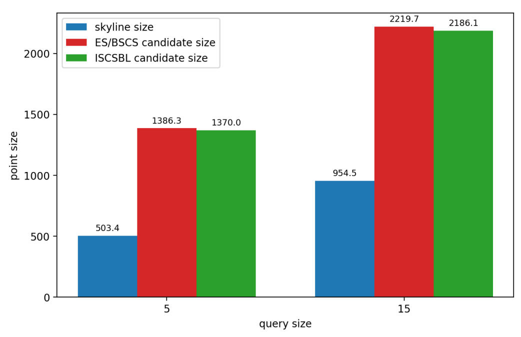
2.4 Conclusion

In this chapter, we have studied the spatial skyline query problem in the 2-dimensional Euclidean plane and present several algorithms that can improve the efficiency of the queries. Two efficient skyline check methods are proposed. BSC takes advantage of the dominator region, while ISC takes advantage of the spatial index. The bounding line method, which can reduce the candidate size, is also proposed here. The effect of the bounding line method is not very obvious for the 2-dimensional case. As we will see in the next chapter, the bounding line method will have a significant impact on reducing the candidate size in high dimensional space.

Our algorithms, $BSCS$ and $ISCSBL$, outperform the ES algorithm. ES takes $O(\sqrt{|P|} + |Q| \log |Q| + |T_S| \log |T_S| + |C_S||S| \log |CH(Q)|)$ time to complete a query in the worst case.

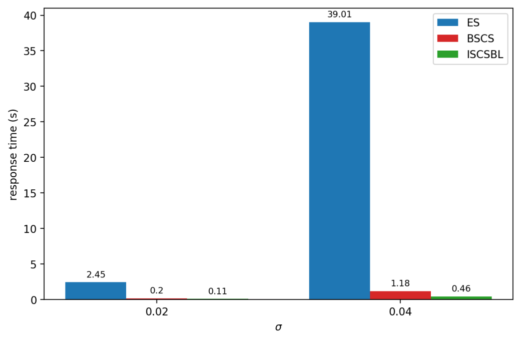


(a) Response time

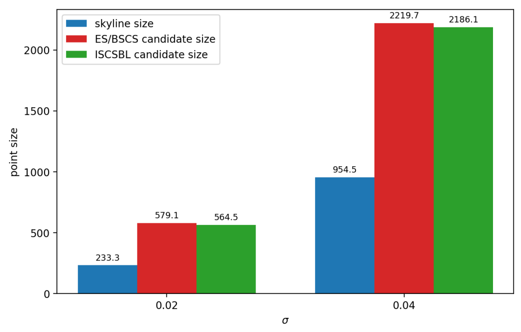


(b) Skyline/candidate size

Figure 2.10: Experimental results on the POI dataset with respect to the query size



(a) Response time



(b) Skyline/candidate size

Figure 2.11: Experimental results on the POI dataset with respect to the standard deviation of the normal distribution of the query points

The worst-case query time complexity of the $BSCS$ algorithm is $O(\sqrt{|P|} + |Q|\log|Q| + |T_S|\log|T_S| + |T_S|\sqrt{|S|} + |C_S||S'|\log|CH(Q)|)$, where $|S'|$ is the number of spatial skyline points used in BSC. The worst-case query time complexity for the $ISCSBL$ algorithm is $O(\sqrt{|P|} + |T_S|\log|T_S| + |T_S|\sqrt{|S|} + |C_S||CH(Q)||S|^{1-\frac{1}{|CH(Q)|}})$. In the worst case, $|T_S|$ and $|C_S|$ can be as large as $O(|P|)$, and $|S'|$ can be as large as $O(|S|)$. In the experiments, we notice that $|C_S|$ and $|T_S|$ are $O(|S|)$. We also show that the improvement in the response time of our algorithms mainly comes from the efficient skyline check methods. Instead of visiting the whole spatial skyline point set, our skyline check methods only visit a subset of the spatial skyline point set.

Chapter 3

Spatial Skyline Query Problem in High Dimensional Euclidean Space

In this chapter, we investigate the spatial skyline query problem in high dimensional Euclidean space. We are given a set of data points P. For an arbitrary query set Q in d-dimensional Euclidean space, the problem requires us to locate all the skyline points $S \subseteq P$ with respect to Q. Like in the 2-dimensional case, our algorithms have three components, namely the traversal component, the pruning component and the skyline check component.

This chapter is organized as follows: Section 3.1 discusses why the algorithms designed for the 2-dimensional Euclidean plane are not efficient in high dimensional space. In section 3.2, we discuss several solutions for the traversal components, pruning components, and skyline check components of our algorithms. Section 3.3 contains some experimental results comparing the solutions to different components. In section 3.4, conclusions are drawn.

3.1 Related Work

Son et al. [36] and Sharifzadeh et al. [39] mentioned in their papers that their algorithms could work in high dimensional space. However, some geometry structures used in their papers are complex and constructing them in high dimensional Euclidean space takes more time than the brute force spatial skyline algorithm. For example, in d-dimensional Euclidean space, the brute force algorithm compares each data point p against all other data points $p_i \in P \setminus \{p\}$ to see if any data point dominates p, which takes $O(|P|^2|Q|d)$ time. However, for a set of points P in d-dimensional Euclidean space, the Voronoi diagram of them has $\Theta(|P|^{\lceil \frac{d}{2} \rceil})$ combinatorial complexity [23]. The convex hull of those data points has $\Theta(|P|^{\lfloor \frac{d}{2} \rfloor})$ combinatorial complexity [10]. With the dimensionality growing, we want to avoid using geometric structures like Voronoi diagrams or convex hulls. Other papers focus on solving general skyline queries or variants of them in high dimensional space [6][7][26][33][48].

3.2 Our Algorithms

As mentioned in Section 2.3, a spatial skyline algorithm has three components: the traversal component, the pruning component, and the skyline check component. In this section, we propose several solutions to the components that can solve the high dimensional spatial skyline query problem. In Section 3.2.1, two traversal components, such as sorting and the index traversal method, are discussed. Section 3.2.2 presents the index traversal method with lazy construction. In Section 3.2.3, two pruning components, namely the bounding box method, and the bounding plane method, are proposed. In Section 3.2.4, skyline check components proposed in Chapter 2 are analyzed to show their potential usage in high dimensional Euclidean space.

3.2.1 Traversal Components in High Dimensional Space

Sorting, as a traversal component, works in high dimensional Euclidean space. Data points (tentative points) are visited according to their distances to the source query point, which is an arbitrarily selected query point. Sorting is the simplest and most straightforward traversal component.

Another traversal component is the index traversal method. In the index traversal method, data points are stored in a spatial index like a KD-tree. The method then keeps returning the next unvisited nearest neighbor of the source query point q_s, which is an arbitrarily selected query point. In this way, data points (called tentative points) are returned in sorted order. Additionally, as discussed later, the data structure used by the index traversal method allows pruning of data points during the computation of the spatial skyline points.

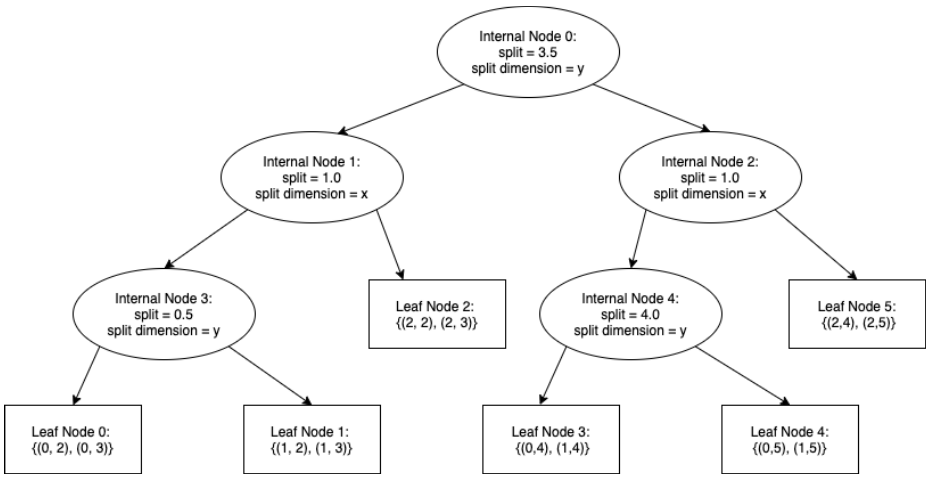
In our thesis, KD-tree is used to store all the data points. There are two types of nodes in a KD-tree [2]: the internal node and the leaf node. The internal nodes store the splitting value and splitting dimension, while the leaf nodes store the actual data points. A leaf node has a size parameter that specifies the maximum number of data points sorted in a leaf node.

For example, suppose we have a set of data points:

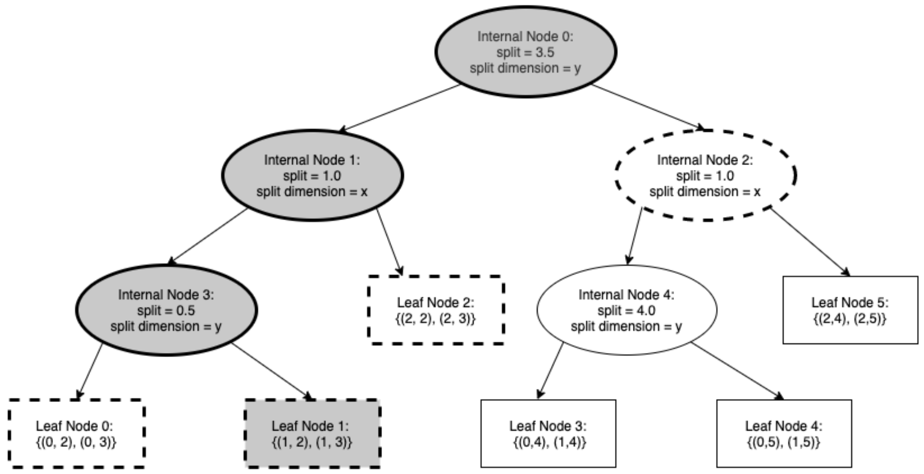
$$\begin{aligned}\{&(0,2),(0,3),(0,4),(0,5),\\ &(1,2),(1,3),(1,4),(1,5),\\ &(2,2),(2,3),(2,4),(2,5)\}\end{aligned}$$

Figure 3.1a shows the newly constructed KD-tree for the data points. Note that the elliptical nodes are the internal nodes, and the rectangular nodes are the leaf nodes.

Each node also maintains a bounding box and an enclosing hypersphere in order to support the pruning methods mentioned later in Section 3.2.3. The bounding box is the minimum orthogonal bounding box that contains all the data points in the subtree rooted



(a) Newly constructed KD-tree



(b) Traversal using KD-tree

Figure 3.1: KD-tree example

at the node. The bounding box is represented by a vector of size $2d$, where d is the dimension. The distance from q_s to a node is the shortest distance from q_s to the bounding box of the node. If q_s lies inside the bounding box of a node, the distance between them is 0.

We also store, at each node v, an enclosing hypersphere, not necessarily of minimum radius, to contain all the points in v's subtree. The minimum radius spanning hypersphere of n points in d dimension can be computed in $O(dn)$ time [14][30]. Note that a spanning hypersphere of v can easily be computed using the spanning hyperspheres stored at v's two children. This way we can easily compute all the spanning hyperspheres of the KD-tree nodes bottom up in $O(dn)$ time.

We now briefly introduce the index traversal method. The method continuously outputs the next unvisited nearest neighbors of q_s. The index traversal method uses two heaps:

- Node heap H_N: A min-heap stores the nodes of the KD-tree that are being visited. The keys of H_N are the distances from nodes of the KD-tree to q_s. Initially, H_N only contains the root node of the KD-tree.

- Point heap H_P: A min-heap stores data points yet to be output by the index traversal method. The keys of H_P are the distances from data points to q_s. Initially, H_P is empty.

The index traversal method keeps checking the top entry (d_{min}, N_v) of H_N. If N_v is an internal node, two children of N_v are pushed into H_N. Otherwise, N_v is a leaf node, and the index traversal method pushes all the data points stored in N_v into H_P. Additionally, if H_P is not empty, the method compares the keys of the top nodes of H_N and H_P. If the key of the top node of H_P is smaller than that of H_N, the top node of H_P is returned as the next nearest neighbor of the source query point. The index traversal method stops when H_N is empty. The pseudocode of the index traversal method is shown in Algorithm 7.

Algorithm 7 Index Traversal Method

```
Input: KD-Tree of all data points $T$, Source Query Point $q_s$
Output: Next unvisited nearest neighbor $z$
1: Initialize the node Heap $H_N = [\ ]$
2: Initialize the point Heap $H_P = [\ ]$
3: $H_N.push(d(T.root, q_s), T.root)$
4: while $H_N$ is not empty do
5:     while $H_P$ is not empty and $H_P.top().key < H_N.top().key$ do
6:         yield $z = H_P.pop()$
7:     end while
8:     $(d_{min}, N_v) = H_N.pop()$
9:     if $N_v$ is a leaf node then
10:        Compute the distances from points in $N_v$ to $q_s$ and push those points along with
           their distances to $H_P$
11:    else
12:        Push the children of $N_v$ into $H_N$ along with their distances to $q_s$
13:    end if
14: end while
```

Notice that the **yield** keyword in line 6 means that the top node of H_p is returned as the next nearest neighbor, and when the next nearest neighbor is required again, the algorithm resumes and starts from line 7.

An example of the index traversal method is now provided. We want to find the next nearest neighbors of the point $(0.75, 2)$ with respect to the KD-tree in Figure 3.1. The method traverses from the root node down to the nearest leaf node. Figure 3.1b displays the status when the method is going to pop out the first leaf node from H_N. The gray nodes with the bold border in the figure are visited nodes. The white nodes with the dashed border are the nodes currently contained in H_N. The gray leaf node with the dashed border

Step	H_N: (key, node)	H_P: (key, point)
1	[(0, Internal Node 0)]	[ ]
2	[(0, Internal Node 1), (2, Internal Node 2)]	[ ]
3	[(0, Internal Node 3), (1.25, Leaf Node 1), (2, Internal Node 2)]	[ ]
4	[(0.25, Leaf Node 1), (0.75, Leaf Node 0), (1.25, Leaf Node 2), (2, Internal Node 2)]	[ ]
5	[(0.75, Leaf Node 0), (1.25, Leaf Node 2), (2, Internal Node 2)]	[(0.25, (1,2)), (1.03, (1,3))]

Table 3.1: Index traversal algorithm detailed procedure

is then popped out. The data points in the leaf node are pushed into H_P. For now, the top node of H_N is $[0.75, \{\text{Leaf Node 0}:(0,2),(0,3)\}]$, and the top node of H_P is $[0.25,(1,2)]$. The key of the top node of H_N and the key of the top node of H_P are then compared. Because the top node of H_P has a smaller key, $(1,2)$ is returned as the next nearest neighbor. The process repeats whenever the next nearest neighbor is required. Table 3.1 shows the status of H_N and H_P before the first nearest neighbor is returned.

Theorem 3.1. *In the preprocessing step, for a set of data points P in d-dimensional Euclidean space, the index traversal method takes $O(d|P|\log|P|)$ to construct the KD-tree [10].*

3.2.2 Index Traversal Method with Lazy Construction

In the previous section, we show that the index traversal method can be used to traverse the data points in sorted order. However, the index traversal method must spend $O(d|P|\log|P|)$ preprocessing time to construct the KD-tree before outputting the first skyline point [9]. The lazy construction technique can be incorporated into the index traversal method. The lazy construction technique builds a KD-tree node only when the method is going to visit the node. It can reduce the preprocessing time. If the pruning component is incorporated into the index traversal method, we can avoid constructing some nodes of the KD-tree with lazy construction.

The lazy construction version of the index traversal method assembles most parts of the original version except for two differences. The first difference is that the method initializes the KD-tree with only one leaf node containing all the data points. The second difference appears when the method pops a node out of the node heap H_N. In this scenario, if the popped node is a leaf node with a size larger than the user-specified leaf size, the method will replace the leaf node with a newly constructed internal node. Data points stored in the original leaf node are split and stored in the two newly constructed children leaf nodes of the internal node. Note that the enclosing hyperspheres of the lazy KD-tree nodes are computed in a top-down manner, since the lazy KD-tree starts with only one leaf node and the tree structure is built on-the-fly.

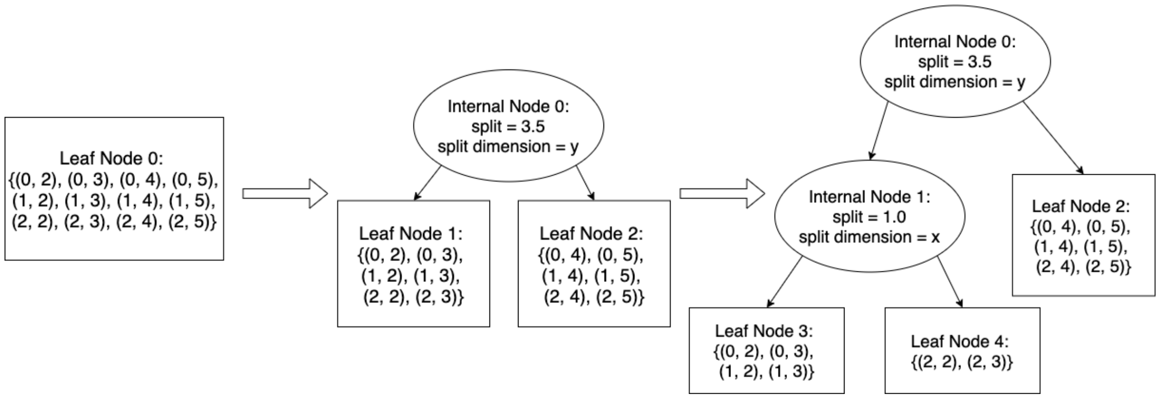

Leaf Node 0:
{(0, 2), (0, 3), (0, 4), (0, 5), (1, 2), (1, 3), (1, 4), (1, 5), (2, 2), (2, 3), (2, 4), (2, 5)}
Internal Node 0: split = 3.5 split dimension = y
Leaf Node 1: {(0, 2), (0, 3), (1, 2), (1, 3), (2, 2), (2, 3)}
Leaf Node 2: {(0, 4), (0, 5), (1, 4), (1, 5), (2, 4), (2, 5)}
Internal Node 0: split = 3.5 split dimension = y
Internal Node 1: split = 1.0 split dimension = x
Leaf Node 2: {(0, 4), (0, 5), (1, 4), (1, 5), (2, 4), (2, 5)}
Leaf Node 3: {(0, 2), (0, 3), (1, 2), (1, 3)}
Leaf Node 4: {(2, 2), (2, 3)}

Figure 3.2: Index traversal method with lazy construction

Figure 3.2 contains an example showing the first three steps of applying the index traversal method with lazy construction. We want to locate the nearest neighbor of the point $(0.75, 2)$. Initially, all the data points are stored in the Leaf Node 0. When we visit the Leaf Node 0, it is replaced by an internal node (Internal Node 0). Data points in Leaf Node 0 are split into two children leaf nodes: Leaf Node 1 and Leaf Node 2. We then visit the left child of the Internal Node 0. The index traversal method with lazy construction transforms the Leaf Node 1 into Internal Node 1 by computing the splitting value and splitting dimension. Leaf Node 3 and Leaf Node 4 are created to store the data points. This process ends if the number of data points is smaller than the leaf size. The pseudocode of the index traversal method with lazy construction is presented below in Algorithm 8. Algorithm 8 is similar to Algorithm 7 except for line 3, which is the initialization of lazy KD-tree ,and lines 11-14, which are the KD-tree construction steps.

3.2.3 Pruning Components in High Dimensional Space

The bounding box method and the bounding line method described in Chapter 2 can be used in higher dimensional space. The bounding box method, which uses the bounding box of the non-dominating region of the skyline points to prune out non-skyline data points, becomes less efficient in high dimensional space because the ratio between the volume of the unit hypersphere and the volume of the unit hypercube approaches zero with the growth of dimensionality [44].

Theorem 3.2. *For a given query set, the index traversal method along with the bounding box method can visit all the tentative points T_S of the KD-tree in sorted order and produce the candidate set C_S in $O(d|P|^{1-\frac{1}{d}} + d|S||Q| + d|T_S| + |T_S|\log|T_S|)$ time.*

Proof. Each skyline point generates a bounding box, which costs $O(|Q|d)$ time. Maintaining the intersection bounding box takes the same time. The index traversal method can retrieve all the tentative points T_S inside the bounding box using a range-query-like technique, which costs $O(d|P|^{1-\frac{1}{d}} + d|T_S|)$ time. Computing distances from all the tentative points to

Algorithm 8 Index Traversal Method with Lazy Construction

```
Input: Data Point Set $P$, Source Query Point $q_s$
Output: Next unvisited nearest neighbor $z$
1: Initialize the node Heap $H_N = [\ ]$
2: Initialize the point Heap $H_P = [\ ]$
3: Initialize the KD-tree $T.root = LeafNode(P)$
4: $H_N.push(d(T.root, q_s), T.root)$
5: while $H_N$ is not empty do
6:     while $H_P$ is not empty and $H_P.top().key < H_N.top().key$ do
7:         yield $z = H_P.pop()$
8:     end while
9:     $(d_{min}, N_v) = H_N.pop()$
10:    if $N_v$ is a leaf node then
11:        if $N_v.size > leaf\_size$ then
12:            Transform $N_v$ to a inner node by computing the split and the split dimension of
               $N_v$
13:            Create two leaf nodes containing data points after the splitting and make them
               two children of $N_v$
14:        else
15:            Compute the distances from points in $N_v$ to $q_s$ and push those points along with
               their distances to $H_P$
16:        end if
17:    else
18:        Push the children of $N_v$ into $H_N$ along with their distances to $q_s$
19:    end if
20: end while
```

the source query point takes $O(d|T_S|)$ time. Visiting all the tentative points of T_S in sorted order takes $O(|T_S|\log|T_S|)$ time. The total time complexity is $O(d|P|^{1-\frac{1}{d}} + d|S||Q| + d|T_S| + |T_S|\log|T_S|)$. □

The bounding line method mentioned in section 2.2.3 now becomes the bounding plane method in higher dimensional space, since the bounding line in the 2-dimensional plane now becomes a $(d-1)$-dimensional hyperplane in d-dimensional space where d is the dimension of the data points. Each $(d-1)$-dimensional hyperplane indicates a d-dimensional half-space where yet to be computed spatial skyline points can lie. The intersection of the half-spaces determined by the skyline points computed so far forms a d-dimensional polyhedron. Any data point outside the polyhedron cannot be a skyline point. The bounding plane method, in combination with the bounding box method, performs much better in high dimensional space.

Theorem 3.3. *For a given query set, the bounding box and bounding plane method along with the index traversal method can visit all the tentative points T_S in sorted order and produce the candidate set C_S in $O(d|P|^{1-\frac{1}{d}} + d|S||Q| + |T_S|\log|T_S| + d|T_S||S|^{1-\frac{1}{d}})$ time.*

Proof. The bounding plane method adds an additional $O(d|T_S||S|^{1-\frac{1}{d}})$ to Theorem 3.2 because it checks whether a tentative point produced by the bounding box method lies inside the convex polyhedron produced by the bounding plane method. $\square$

The pruning components can be incorporated into the index traversal method. Whenever a new node of the KD-tree is met, the node can be checked against the bounding region of the current skyline points (either a bounding box or a bounding polyhedron). If the node is completely outside the bounding region, the whole subtree of the node can be pruned out.

3.2.4 Skyline Check Components in High Dimensional Space

The dominance check method in 2-dimensional space is not applicable in the high dimensional space since we want to avoid computing the convex hull of all the query points. An alternative is to compute the distances from the candidate point to all the query points and compare the distances with current skyline points. The new dominance check method takes $O(d|Q| + |Q||S|)$ time.

The Index skyline check method (ISC) mentioned in Section 2.2.2 is applicable and efficient in high dimensional space. Recall that ISC puts all skyline points found so far into a spatial index I. A candidate point p_c is transformed from the Euclidean space to the query space. The method then checks whether any skyline point lies inside the $|Q|$-dimensional hypercube formed by p_c and the origin of the query space. If there is no skyline point inside the hypercube, p_c is accepted as a skyline point. The time complexity of ISC is $O(d|Q| + |Q||S|^{1-\frac{1}{|Q|}})$.

3.3 Experiments

In this section, several experiments are conducted to compare the efficiency of different algorithms in solving the spatial skyline query problem in high dimensional space. In section 3.3.1, we describe our algorithm settings in conducting the experiments. In section 3.3.2, experimental settings are described. In section 3.3.3, experimental results are shown to demonstrate the efficiency of different algorithms.

3.3.1 Algorithm Settings

In this section, we provide several algorithms that take advantage of different components we described in the previous sections. The names of the algorithms consist of the starting alphabet of their components in the following order: the traversal component, the pruning component and the skyline check component. For example, if an algorithm is named IB^2I, it means that the algorithm uses the index traversal method as their traversal component, the bounding box and bounding line method as its pruning component and ISC as their skyline check component.

	SBD	IB^2D	IB^2I	ILB^2I
Traversal component	Sorting	Index traversal method	Index traversal method	Index traversal method with lazy construction
Pruning component	Bounding box	Bounding box & Bounding plane	Bounding box & Bounding plane	Bounding box & Bounding plane
Skyline check component	Dominance check	Dominance check	ISC	ISC

Table 3.2: Configuration of different algorithms

Note that ES algorithm [39] in 2-dimensional Euclidean plane now becomes the SBD algorithm in high dimensional Euclidean space by using the new dominance check method as its skyline check component.

3.3.2 Experimental Settings

The dataset used in this chapter is a synthetic dataset. Data points are generated uniformly in a unit hypercube. The query points follow a normal distribution using a randomly chosen data point as their mean. We investigate the effect of data size, dimension, query size, and the standard deviation of the normal distribution of the query points on the performance of the algorithms. Three different data sizes, three different dimensions, two different query sizes, and two different σ are considered. Table 3.3 shows the detailed dataset settings.

Parameter	Setting
Data point set size	100K, 175K, 250K
Dimension	3, 4, 5
Query point set size	5, 20
σ	0.05, 0.125

Table 3.3: Configuration of the dataset in high dimensional space

KD-tree is used in the index traversal method to store data points. It is also used in ISC to store skyline points computed so far. The leaf size of the KD-trees used in our algorithms is 16. As for the insertion of a skyline point s to the KD-tree, we insert s to the leaf node if the number of the skyline points in the leaf node is smaller than the leaf size. When the number of the skyline points becomes greater than the leaf size, we replace the leaf node with a newly constructed internal node. Skyline points stored in the original leaf node are split and stored into two newly constructed children leaf nodes of the internal node. The expected height of the KD-tree built in this way is $O(\log n)$ [8].

3.3.3 Experimental Results

Table 3.4 contains the experimental data. Note that we only evaluate the tentative sizes of SBD and IB^2I in the table because ILB^2I visits all the data points in the first few queries and IB^2D has the same tentative size as IB^2I. Figures in this section are generated from Table 3.4.

Data size		100000											
Dimension		3				4				5			
Query size		5		20		5		20		5		20	
σ		0.05	0.125	0.05	0.125	0.05	0.125	0.05	0.125	0.05	0.125	0.05	0.125
skyline size		109.25	808.	442.45	3569.4	30.6	283.	144.7	1854.2	15.85	106.1	75.2	796.35
t (s)	SBD	0.7	4.02	1.25	43.88	0.61	1.35	0.71	13.69	0.61	0.92	0.66	6.12
	IB^2D	**0.1**	2.42	0.73	31.8	**0.05**	**0.6**	**0.26**	11.11	**0.05**	**0.45**	**0.23**	5.21
	ILB^2I	0.17	1.63	0.67	5.98	0.09	0.91	0.42	6.58	0.11	0.8	0.46	6.16
	IB^2I	0.13	**1.47**	**0.56**	**5.68**	0.05	0.74	0.29	**5.85**	0.05	0.58	0.25	**5.12**
$\lvert C_S \rvert$	SBD	900.	9815.6	2759.3	23709.7	340.4	6522.4	1446.	24457.8	269.4	5055.5	1122.8	18904.6
	Others	**858.1**	**9626.1**	**2498.8**	**23097.9**	**303.1**	**6366.8**	**1083.3**	**22741.**	**220.8**	**4619.4**	**669.6**	**15448.2**
$\lvert T_S \rvert$	SBD	3744.6	29349.7	7432.9	54961.5	1462.8	25779.2	5100.9	58208.4	1362.5	24180.6	4395.4	51920.5
	IB^2I	**2358.4**	**13382.4**	**5039.6**	**27099.6**	**1921.**	**10513.9**	**3611.1**	**30692.7**	**2364.4**	**11177.7**	**4479.4**	**27459.7**

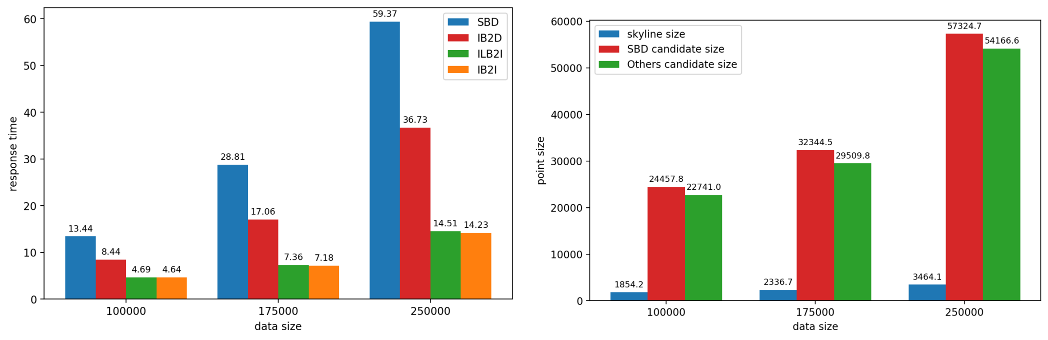
Data size		175000											
Dimension		3				4				5			
Query size		5		20		5		20		5		20	
σ		0.05	0.125	0.05	0.125	0.05	0.125	0.05	0.125	0.05	0.125	0.05	0.125
skyline size		155.25	1013.35	692.4	6622.95	56.45	413.	210.4	2336.65	21.	180.35	87.55	1283.15
t (s)	SBD	1.21	7.	2.52	159.21	1.09	2.9	1.26	23.81	1.06	1.79	1.11	13.06
	IB^2D	**0.17**	4.17	1.26	113.44	**0.1**	**1.44**	**0.46**	17.51	**0.07**	**0.94**	**0.32**	12.52
	ILB^2I	0.27	2.41	0.98	13.38	0.22	1.92	0.74	8.09	0.18	1.67	0.66	13.79
	IB^2I	0.2	**2.19**	**0.82**	**12.74**	0.12	1.59	0.5	**7.21**	0.07	1.22	0.34	**11.64**
$\lvert C_S \rvert$	SBD	1559.	14675.2	4150.9	46050.4	818.2	13549.2	2172.4	32344.5	419.2	9785.4	1488.	36632.8
	Others	**1522.5**	**14428.8**	**3865.5**	**44719.3**	**731.8**	**13116.9**	**1640.**	**29509.8**	**353.1**	**9141.5**	**909.4**	**31065.8**
$\lvert T_S \rvert$	SBD	6485.6	42519.4	13639.6	100416.4	4300.4	54716.4	7704.8	84464.2	1945.2	49860.3	3975.3	98459.4
	IB^2I	**3738.8**	**19815.2**	**7416.6**	**51661.2**	**3390.4**	**22222.9**	**5526.8**	**39621.1**	**3019.7**	**21141.5**	**5599.2**	**51464.2**

Data size		250000											
Dimension		3				4				5			
Query size		5		20		5		20		5		20	
σ		0.05	0.125	0.05	0.125	0.05	0.125	0.05	0.125	0.05	0.125	0.05	0.125
skyline size		241.25	1456.8	924.95	8578.85	54.7	468.9	260.9	3364.1	22.55	232.8	119.6	1627.7
t (s)	SBD	1.83	13.54	4.76	264.06	1.53	3.91	1.84	58.53	1.51	2.68	1.64	17.17
	IB^2D	**0.34**	8.44	2.57	202.04	**0.11**	**1.77**	**0.66**	38.11	**0.08**	**1.48**	**0.49**	14.44
	ILB^2I	0.49	3.98	1.63	18.56	0.24	2.37	1.	12.7	0.22	2.46	0.97	13.92
	IB^2I	0.38	**3.65**	**1.39**	**17.89**	0.13	1.94	0.7	**11.53**	0.09	1.88	0.53	**11.74**
$\|C_S\|$	SBD	2985.6	24770.4	6824.3	60772.9	906.7	15867.9	3067.4	57324.7	475.6	14577.6	2152.3	47254.2
	Others	**2943.2**	**24350.1**	**6382.2**	**58677.6**	**843.2**	**15332.6**	**2457.2**	**54166.6**	**399.7**	**13597.5**	**1332.3**	**41695.1**
$\|T_S\|$	SBD	12148.1	67771.1	24204.4	142141.2	4945.8	65499.4	11842.9	143059.6	2929.2	77429.7	9971.	143590.4
	IB^2I	**6634.1**	**31976.8**	**11610.**	**66576.9**	**4036.9**	**25768.7**	**6883.**	**68667.8**	**3953.3**	**27064.5**	**7520.7**	**66171.6**

Table 3.4: Experimental results of algorithms in the high dimensional Euclidean space

Effect of data size

Our algorithms outperform SBD under various data sizes. The spatial skyline size and the candidate size increase as the data size increases. When the spatial skyline size is small, IB^2D has the best performance because the dominance check method is simple and straightforward. When the data size becomes large, IB^2I starts to outperform IB^2D, and the response times of IB^2D and SBD increase significantly. The reason is that ISC can complete a skyline check by only visiting a relatively small subset of skyline points, while the dominance check method needs to visit all the skyline points in the worst case. ILB^2I performs slightly worse than IB^2I because it spends more time computing the enclosing hyperspheres of its KD-tree nodes. As for the candidate size and tentative size, all the algorithms perform better than SBD. More data points are pruned out with the increase in data size. Figure 3.3 shows the experimental results on different data sizes. The figure shows the case when the query size is 20, σ is 0.125, and the dimension is 4.



(a) Response time (b) Skyline/candidate size

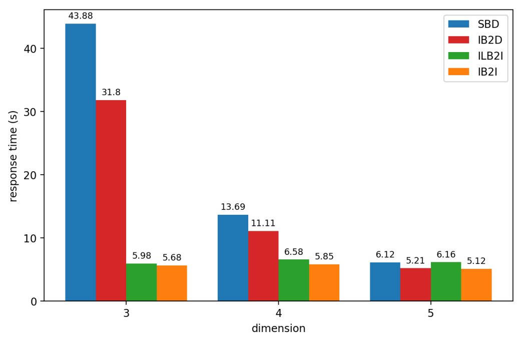
Figure 3.3: Experimental results on the data size

Effect of dimension

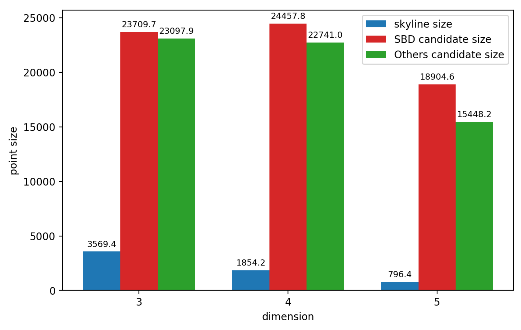
Our algorithms outperform SBD under different dimensions. Note that the spatial skyline size, the candidate size and the tentative size become smaller with the increase in the dimension. The response times of all the algorithms decrease as the dimension increases. The IB^2D has a better performance because the brute force dominance check method becomes more efficient as the spatial skyline size becomes very small. The bounding box and bounding plane method tends to prune out more data points. Figure 3.4 shows the experimental results on different dimensions. The figure shows the case when the data size is 100000, the query size is 20, and σ is 0.125.

Effect of query size

Our algorithms outperform SBD under various query sizes. The response time of SBD and IB^2D increases dramatically with the increase of the query size, while the increase of the



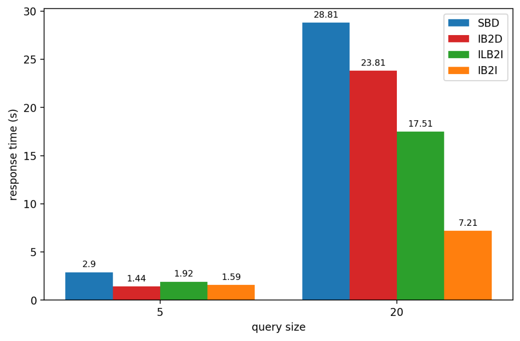
(a) Response time



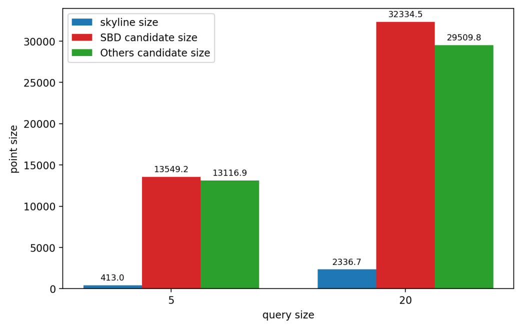
(b) Skyline/candidate size

Figure 3.4: Experimental results on the dimension

response time of IB^2I and ILB^2I is relatively small. The reason is that the dominance check method needs $O(|Q||S|)$ comparisons to complete one skyline check, while ISC can bypass some skyline points by taking advantage of the KD-tree. Figure 3.5 shows the experimental results on different query sizes. The figure shows the case when the data size is 175000, σ is 0.125, and the dimension is 4.



(a) Response time

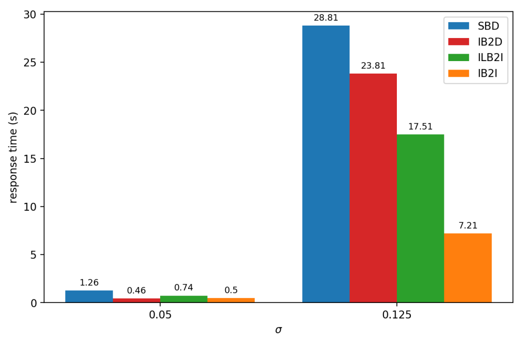


(b) Skyline/candidate size

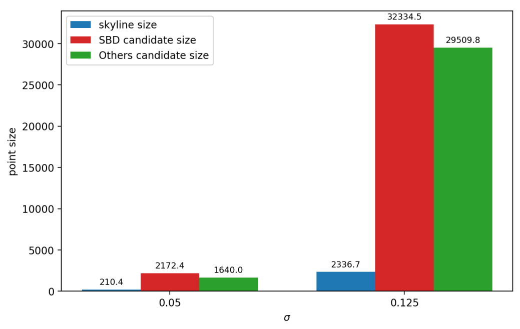
Figure 3.5: Experimental results on the query size

Effect of σ

Our algorithms outperform SBD under different standard deviation of the normal distribution of the query points. The response time of SBD increases dramatically with the increase of σ. However, the increase of the response time of our algorithms is relatively small. The σ has a significant impact on the spatial skyline size, the candidate size and the tentative size because it affects the area covered by the query points in d-dimensional Euclidean space. When σ is small, the bounding box and bounding plane method can prune out many more points than the bounding box method alone. Figure 3.5 shows the experimental results on different σ. The figure shows the case when the data size is 175000, the query size is 20, and the dimension is 4.



(a) Response time



(b) Skyline/candidate size

Figure 3.6: Experimental results on the standard deviation of the normal distribution of the query points

3.4 Conclusion

In this chapter, we have studied spatial skyline query problem in high dimensional space. We incorporated the lazy construction technique into the index traversal method. Lazy construction method is efficient when the number of queries is small. We showed how to incorporate the bounding box and the bounding plane method into the index traversal method using the KD-tree. In the experiments, we show that our algorithms outperform the algorithm by Son et al [39] whose worst case query time complexity is $O(d|P|^{1-\frac{1}{d}} + d|T_S| + |T_S|\log|T_S| + d|S||Q| + d|C_S||Q| + |C_S||Q||S|)$. The IB^2I algorithm has the best performance in general. The runtime complexity of the IB^2I algorithm is $O(d|P|^{1-\frac{1}{d}} + d|T_S| + |T_S|\log|T_S| + d|S||Q| + d|T_S||S|^{1-\frac{1}{d}} + d|C_S||Q| + |C_S||Q||S|^{1-\frac{1}{|Q|}})$. When the skyline set size is small or the dimension is high, the IB^2D algorithm has the best performance. The worst case runtime complexity of the IB^2D algorithm is $O(d|P|^{1-\frac{1}{d}} + d|T_S| + |T_S|\log|T_S| + d|S||Q| + d|T_S||S|^{1-\frac{1}{d}} + d|C_S||Q| + |C_S||Q||S|)$.

Chapter 4

Spatial Skyline Query Problem in Road Network Space

In this chapter, we focus on the spatial skyline query problem in road network space. Sections in this chapter are organized as follows: In Section 4.1, we introduce the problem definition and some related concepts. In Section 4.2, we discuss some related work on the spatial skyline query problem in road network space. Section 4.3 discusses the lower-bound constraint algorithm (LBC) in detail [12]. We then point out and correct a mistake in LBC. In Section 4.4, we present our algorithms. Section 4.5 contains experiments to compare our algorithms with the corrected version of LBC. Section 4.6 draws some conclusions.

4.1 Problem Definition and Related Concepts

A road network can be considered as a weighted undirected graph $G = (V, E)$ where V is a set of vertices in a 2-dimensional plane and $E \subseteq V \times V$ is a set of edges connecting two vertices of V. A road network is generally a planar graph. However, the existence of flyovers and tunnels may make the graph to be non-planar. We will assume in this thesis that our graphs can be arbitrary, but has $O(|V|)$ edges (i.e. $|E|$ is $O(|V|)$).

Given two points p_s and p_t in a road network, $d_E(p_s, p_t)$ denotes the **Euclidean distance** between p_s and p_t, and $d_N(p_s, p_t)$ denotes the shortest path distance or the **network distance** from p_s to p_t. We will assume, in a road network, $d_E(p_s, p_t) \leq d_N(p_s, p_t)$.

Unlike in Euclidean space where the distance between two points can be computed in constant time, the shortest distance computation in the road network space takes $O(|V| \log |V|)$ time in the worst case, if $|E|$ is $O(|V|)$ [9]. Two representative algorithms for computing the shortest path distance are Dijkstra's algorithm and A* algorithm. Both algorithms maintain a **search wavefront** from the source vertex. The search wavefront contains points whose shortest path distances are relaxed but not yet decided.

When it comes to the spatial skyline query problem, both the query point set Q and the data point set P belong to V, which means $P \subseteq V$ and $Q \subseteq V$. If some query point $q \in Q$

lie on the edges, we can still expend V to include those query points. A spatial skyline query problem in road network space locates all the network spatial skyline points $S \subseteq P$ with respect to Q, i.e., each point of S cannot be dominated by any other data points of P with respect to Q. Some concepts that will be used in the future are mentioned below.

Definition 4.1 (Points in Euclidean space)**.** For a point p and a set of query points Q, we say that p is in Euclidean space when the Euclidean distances, instead of shortest path distances, from p to query points $q_i \in Q \quad \forall i \in \{1, 2, 3, \ldots, |Q|\}$ are computed and used in an operation.

Definition 4.2 (Points in network space)**.** For a point p and a set of query points Q, we say that p is in network space when the shortest path distances from p to query points $q_i \in Q$ $\forall i \in \{1, 2, 3, \ldots, |Q|\}$ in a road network are computed and used in an operation.

Definition 4.3 (Domination in a road network)**.** Given two points p_1 and p_2 and a set of query points Q in a road network, p_1 dominates p_2 with respect to Q in network space, written as $p_1 \prec_Q p_2$, if and only if, the network distances from p_1 to all the query points $q_i \in Q \quad \forall i \in \{1, 2, 3, \ldots, |Q|\}$ are less than or equal to the corresponding network distances from p_2 to all the query points, and for some $q_j \in Q$, the network distance from p_1 to q_j is less than the network distances from p_2 to q_j. Put it into a mathematical form:

$$p_1 \prec_Q p_2 \iff \forall q_i \in Q, d_N(p_1, q_i) \leq d_N(p_2, q_i) \quad \wedge \quad \exists q_j \in Q, d_N(p_1, q_j) < d_N(p_2, q_j)$$

4.2 Related Work

After studying the spatial skyline query problem in Euclidean space, researchers paid attention to the spatial skyline query problem in the constraint-based space where an environment dataset is used for distance calculation [12]. For example, in road network space, besides reducing the candidate size, one additional goal is to reduce the number of network distance calculations. Several papers have studied the spatial skyline query problem in the network space [12] [34] [37]. Other papers focus on some variants of the problem [24] [42] [46].

In our thesis, we mainly focus on the paper by Deng et al. [12]. The authors proposed three algorithms to solve the spatial skyline query problem in road network space: the collaborative expansion algorithm (CE), the Euclidean distance constraint algorithm (EDC), and the lower-bound constraint algorithm (LBC). The authors analyzed and compared the three algorithms. They concluded that LBC outperforms CE and EDC in all aspects, ranging from the response time to the candidate size. LBC is the best algorithm among the three algorithms, and we will pay a closer look at this algorithm.

4.3 Lower-bound Constraint Algorithm and a Correction to it

4.3.1 Existing Approach

The name of *LBC* comes from a concept by Deng et al. [12], namely the path distance lower-bound. It refers to the fact that when computing the shortest path distance $d_N(p_s, p_t)$ from p_s to p_t, a lower-bound of $d_N(p_s, p_t)$ can be obtained as used in A* algorithm. Initially, the lower-bound is the Euclidean distance from p_s to p_t. When an intermediate point p_v is reached, the lower-bound increases to $d_l = d_N(p_s, p_v) + d_E(p_v, p_t)$, where $d_N(p_s, p_v)$ is the network shortest path distance between p_s and p_v, and $d_E(p_v, p_t)$ is the Euclidean distance between p_v and p_t. The path distance lower-bound gradually increases during the shortest path computation. We now present an important theorem:

Theorem 4.1 (Domination transitivity in network space [12])**.** *Given two points $p_1, p_2 \in P$, if p_1 in network space dominates p_2 in Euclidean space, p_1 also dominates p_2 in network space*

Proof. The proof follows from the definition of domination (Definition 4.3) and the fact that the Euclidean distance between two points in a road network is a lower-bound of the network distance between those points. We thus have the following inequalities:

$$\begin{aligned} d_N(p_1, q_i) \le d_E(p_2, q_i) \le d_N(p_2, q_i) \quad \forall i \in \{1, 2, 3, \ldots, |Q|\} \\ d_N(p_1, q_j) < d_E(p_2, q_j) \le d_N(p_2, q_j) \quad \exists j \in \{1, 2, 3, \ldots, |Q|\} \end{aligned}$$

The inequalities above prove that p_1 dominates p_2 by the definition of the domination. □

We now introduce the *LBC* algorithm. The algorithm will be explained using the concept of three components mentioned earlier. We will first discuss the traversal component of *LBC*. The algorithm first randomly selects a query point q_s as the source query point. In order to traverse all the data points according to their network distances, *LBC* locates the, not yet pruned, next Euclidean space nearest neighbor p of q_s. The network distance from q_s to p is computed using the A* algorithm. All points along the shortest path between p and q_s are added to the tentative set T_S. The algorithm then retrieves the network nearest neighbor from T_S.

Lemma 4.2. *The time complexity of the traversal component of LBC is $O(\sqrt{|P|} + |T_S| \log |T_S|)$ in the worst case.*

Proof. Suppose $|T_S|$ vertices are visited in a spatial skyline query instance. In the worst case, *LBC* retrieves $|T_S|$ Euclidean space nearest neighbors of the source query point, which takes $O(\sqrt{|P|})$ time by performing a nearest neighbor query to the source query point first, and

Algorithm 9 Traversal Component of *LBC*

Input: Data point set P, Source query point q_s

Output: Next network nearest neighbor z

```
1: Initialize the candidate point heap $C_S = [\ ]$
2: while there are some data points in the undominated area do
3:     Find the next Euclidean space nearest neighbor $p_e$
4:     Compute the shortest network distance from $q_s$ to $p_e$, and push all the data points
       met along the way to $C_S$
5:     yield $z = C_S.pop()$
6: end while
```

then searching the surrounding KD-tree nodes of the nearest neighbor [10]. Computing network distances from a vertex to $O(|T_S|)$ vertices in a road network takes $O(|T_S| \log |T_S|)$ time if the Dijkstra's algorithm is used [10]. □

The pruning component of *LBC* utilizes Theorem 4.1 to prune out data points. Note that the shortest path distances from the network skyline points to each query point are already known. The skyline check using the dominance check method are performed between data points in Euclidean space and the network skyline points found so far. Data points are discarded if they are dominated by any skyline point in network space. Data points in Euclidean space that pass the skyline check becomes candidate points, and the algorithm will compute their network distances in the skyline check component.

If data points in the Euclidean space are stored in a spatial index, like a KD-tree, *LBC* performs skyline checks between the bounding box of the internal nodes of the KD-tree and the network skyline points found so far, instead of performing skyline checks on the data points. The algorithm discards an internal node along with the sub-tree rooted at the node, if the internal node is dominated by at least one network skyline point.

Lemma 4.3. *The worst case time complexity of the pruning component of LBC is* $O(\sqrt{|P|}|S||Q|)$.

Proof. In the worst case, the dominance check method is performed between all the Euclidean space KD-tree nodes visited by the nearest neighbor query and all the skyline points of S in network space. □

We now discuss the skyline check component of *LBC*. The algorithm maintains a sorted list $q.L$ for each query point q, storing all the network skyline points found so far. Network skyline points in the list are sorted in ascending order by their network distances to q. When *LBC* meets a new candidate point p, it inserts p into all the sorted lists using the Euclidean distances from p to all the query points in Q. If there is a network skyline point s in network space that dominates p in Euclidean space, p is discarded. Otherwise, *LBC* starts to compute the network distance between p and the query point q who has smallest

Euclidean distance to p. Along with the computation of the network distance, the algorithm moves p up in the sorted list of q. During the movement, if p is found to be dominated by a network skyline point s, the network distance computation stops immediately, and p is discarded. The pseudocode of the pruning component and skyline check component of *LBC* is presented in Algorithm 10.

Algorithm 10 Pruning and Skyline Check Component of *LBC*

```
Input: Data point $p$, Query point set $Q$, Network skyline point set $S$
Output: Whether $p$ is a skyline point
1: $p' = (d_E(p, q_1), d_E(p, q_2), d_E(p, q_3), d_E(p, q_4), \ldots, d_E(p, q_{|Q|}))$
2: while No network skyline point dominates $p'$ do
3:    Choose a non-source query point with the minimum path distance lower-bound to $p$,
      and expand the search wavefront of it
4:    if All the network distances are computed then
5:       return True ($p$ is a network skyline point)
6:    end if
7: end while
8: return False ($p$ is not a network skyline point)
```

Lemma 4.4. *The skyline check component of LBC takes* $O(|Q||C_S|\log|C_S| + |C_S||Q|^2|S|)$ *time in the worst case to perform skyline checks on all the candidate points.*

Proof. $O(|Q||C_S|\log|C_S|)$ is spent on computing the network distances from all the candidate points of C_S to all the query points of Q.

A dominance check takes $O(|Q|)$ time to decide whether one point dominates another point. For each sorted list of the query points, a candidate point p_c can move from the bottom to the top of the sorted list during the network distance computation. Along its way in a sorted list, p_c can meet at most $O(|S|)$ network skyline points, and thus $O(|S|)$ dominance checks between p_c and the network skyline points found so far are needed. There are $|Q|$ lists in total. The time spent on one candidate point by the skyline check component of *LBC* is $O(|Q|^2|S|)$. The total time complexity for all the candidate points is then $O(|C_S||Q|^2|S|)$. □

Theorem 4.5. *The worst case runtime complexity of the LBC algorithm is* $O(\sqrt{|P|}|Q||S| + |T_S|\log|T_S| + |Q||C_S|\log|C_S| + |C_S||Q|^2|S|)$.

Proof. The runtime complexity is computed by combining Lemma 4.2, Lemma 4.3 and Lemma 4.4 together. □

4.3.2 A Mistake in LBC

In this section, a mistake in the traversal component of *LBC* is pointed out. In the traversal component, *LBC* uses the A* algorithm to compute the shortest path distance between the source query point q_s and the next Euclidean space nearest neighbor to q_s. *LBC* pushes all the data points along the shortest path from q_s to its Euclidean nearest neighbor into a candidate heap. The algorithm then selects a point with the shortest network distance from the candidate heap to be the next candidate point to process. We argue that candidate points visited in this way may accept some non-skyline points as skyline points.

An example is shown in Figure 4.1 to illustrate an erroneous case in the traversal component in *LBC*. In the example, there are two query points q_1 and q_2 and five data points $\{p_1,\ p_2,\ p_3,\ p_4,\ p_5\}$. The solid lines are the network edges with distance showing by numbers beside them. The dashed line are helper edges showing the Euclidean distance between different points. The network distances from data points to q_1 and q_2 are:

$$p_1 : (5,\ 16.18) \quad p_2 : (10,\ 11.18) \quad p_3 : (17.92,\ 19,1) \quad p_4 : (9,\ 10.3) \quad p_5 : (17.57,\ 18.75)$$

Only p_1 and p_4 are network skyline points. However, if *LBC* is applied to the above example, and q_1 is selected as the source query point, the algorithm will visits p_2 before visiting p_4. *LBC* will wrongly accept p_2 as a skyline point, which is dominated by p_4. The Euclidean distances from data points to the source query point q_1 are:

$$p_1 : 5 \quad p_2 : 10 \quad p_3 : 2.61 \quad p_4 : 9 \quad p_5 : 4$$

The algorithm first locates the Euclidean nearest neighbor to q_1, which is p_3. A* algorithm is then applied between q_1 and p_3. Because the estimated network distance from p_3 to p_4 and p_3 to p_5 are large, p_4 and p_5 stay in the search wavefront, and p_1, p_2 and p_3 eventually appear in the heap. At the end of the first iteration, p_1 is selected as the next candidate point and then becomes the first skyline point. Since p_5 is the second Euclidean space nearest neighbor of q_1 and p_5 is already in the search wavefront, *LBC* adds p_5 to the heap and p_2 is popped out of heap at the end of the second iteration. p_2 is then wrongly accepted as a skyline point even though it is dominated by p_4. p_4 never appears in the heap during the first two iterations.

In our implementation of *LBC*, Dijkstra's algorithm is used in the traversal component instead of the A* algorithm.

4.4 Our Algorithm

We now propose our algorithm, the network index skyline check with bounding box algorithm (*NISCB*). This section introduces the three components of our algorithms respectively.

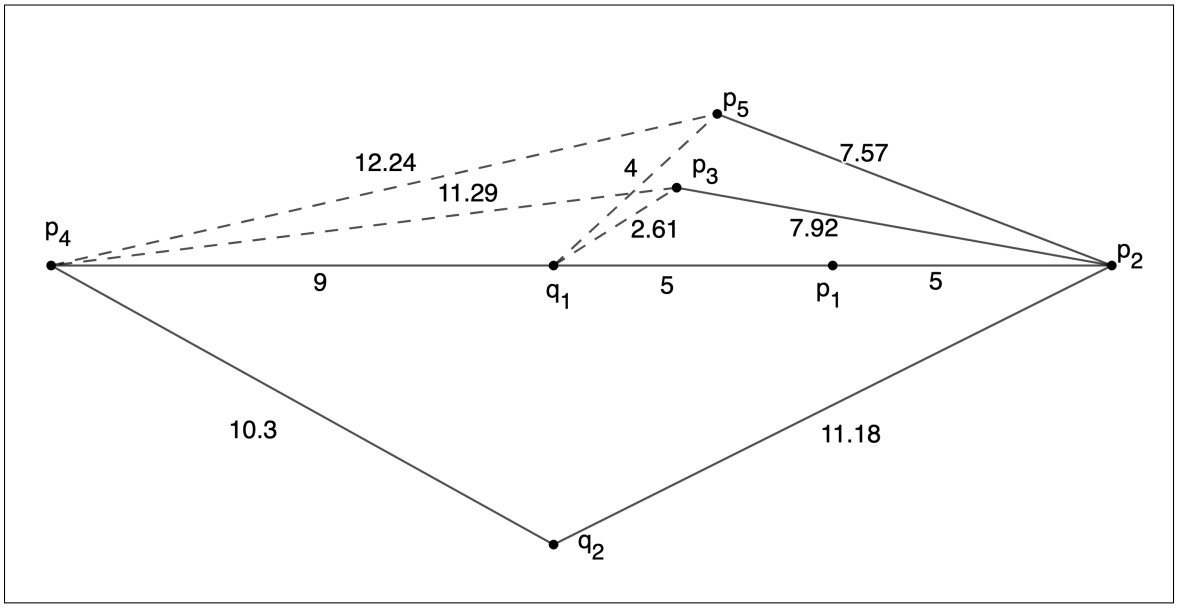


Figure 4.1: An Erroneous Case of *LBC*

4.4.1 Traversal component

Our algorithm creates a dummy point as the source query point. The dummy point has zero distance edges to all the query points of Q. $NISCB$ expands a Dijkstra's search from the source query point. In this way, data points are visited according to their network distances to the query point set within one Dijkstra's search.

An example of the traversal component of our algorithm is shown in Figure 4.2. The example has four data points $\{p_1,\ p_2,\ p_3,\ p_4\}$ and three query points $\{q_1,\ q_2,\ q_3\}$. The solid lines are network edges, and their distances are indicated by the number besides them. q_s is the dummy source query point who has three edges with zero distance connecting to three query points. The dashed lines are edges with zero distance. Our algorithm will initiate the Dijkstra search from q_s. Data points are visited in the following order: $\{p_1,\ p_2,\ p_3,\ p_4\}$.

The traversal component of our algorithm is simple and straightforward. The pruning component of our algorithm ensures that our algorithm stops earlier without traversing all the data points.

4.4.2 Pruning component

The bounding box method mentioned in Section 2.1.2 is modified to be the pruning component of our algorithm. The method computes a bounding box for every network skyline point. For a network skyline point $s \in S$, a minimum orthogonal bounding box B is computed to contain $|Q|$ disks $D(q_i, d_N(q_i, s))\ \ \forall q_i \in Q$. $D(q_i, d_N(q_i, s))$ is a disk centering at q_i with a radius of length $d_N(q_i, s)$. Note that the radius of a disk is the network distance from q_i to s.

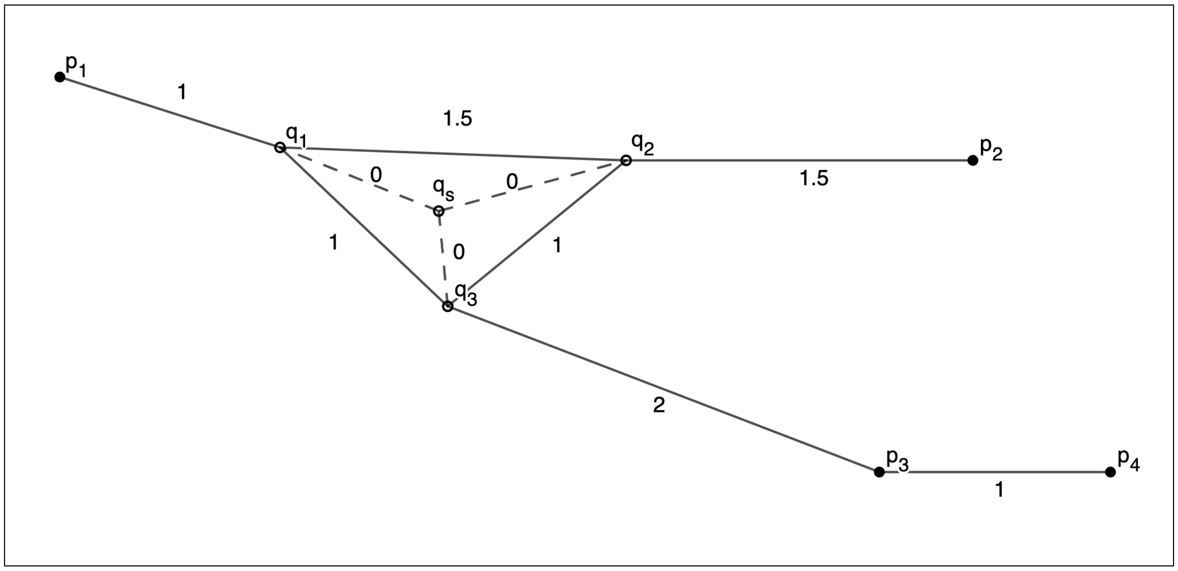


Figure 4.2: An example of the traversal component of $NISCB$

Figure 4.3 shows an example of the bounding box B of a skyline point s. There are three query points $\{q_0, q_1, q_2\}$ in the example. The solid straight lines are the network edges connecting s and the query points. The disks with solid borders are disks centred at different query points. The radii of those disks are the network distances from s to the query points. The dashed bounding box B, which contains all the disks, is the bounding box computed by the bounding box method.

The algorithm keeps track of the intersection bounding box B_S of the bounding boxes B of all the network skyline points computed so far. The algorithm stops when there is no unvisited data points inside B_S. The Lemma 4.6 below proves the correctness of the bounding box algorithm.

Lemma 4.6. *The bounding box algorithm along with the Dijkstra's search from the source query point visits all the skyline points.*

Proof. Assume $s \in S$ is a skyline point that has already been located by the algorithm. Suppose a bounding box B is computed for s in the way described above. Suppose p_o represents a randomly chosen data point that is outside B.

We will first prove that p_o is dominated by s, thus cannot be a skyline point. The bounding box B is computed using the network distances from s to all the query points of Q. For any data point outside B, their Euclidean distances to all the query points are greater than the network distances from s to all the query points. This is equivalent to

$$\forall q_i \in Q, \quad d_N(s, q_i) < d_E(p_o, q_i) \leq d_N(p_o, q_i)$$

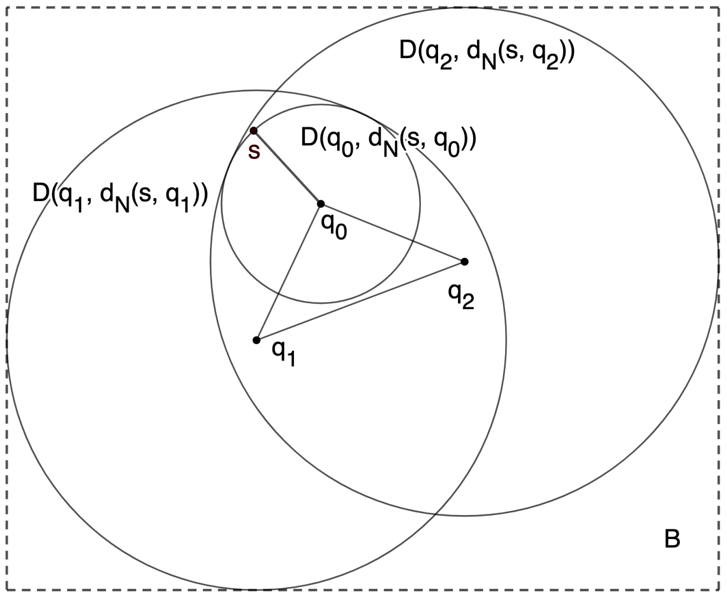


Figure 4.3: An example of the bounding box B of s

Equation above indicates that s in network space dominates p_o in the network space. Any data point outside B cannot be a skyline point.

We will then prove that all the, not yet located, skyline points inside the bounding box can be visited by Dijkstra's search starting from the dummy source query point q_s. For any not yet located skyline point s_n inside B, if there is a path from s_n to q_s that is fully contained in B, s_n is visited when the searching wavefront of Dijkstra's algorithm reaches it. The only concern is that all the paths from s_n to q_s go outside the B so that the algorithm stops before reaching s_n. We claim that this case will never happen. To prove the claim, notice that in the case with multiple query points, the source query point is a dummy point who has paths to all the query points with distance 0. In the case described above, s_n is not visited if and only if the shortest paths from s_n to all the query points Q go outside B, which means s_n can be treated as a point outside B in network space. Some current skyline points dominates s_n. As a result, s_n cannot be a skyline point. $\square$

We present the pseudocode of the traversal and pruning components of our algorithm below in Algorithm 11.

Lemma 4.7. *The time complexity of the traversal and pruning components of NISCB is* $O(|T_S| \log |T_S| + |S||Q|)$ *in the worst case.*

Proof. It takes $O(|T_S| \log |T_S|)$ time to expand a Dijkstra's search from q_s to all the tentative points in the road network. $O(|S||Q|)$ time is spent to compute the bounding boxes of all the skyline points. Pruning of a data point takes only constant time. $\square$

Algorithm 11 Traversal and pruning components of $NISCB$

```
Input: Road network data points $P$, Query point set $Q$
Output: Next candidate point $c$
 1: Construct a dummy source query point $q_s$ with zero-weighted edges to all the query
    points
 2: Initialize a heap $H = [(0, q_s)]$
 3: Initialize an array $Closed = [\ ]$ containing all the outputted points
 4: Initialize an array $Dist$ of size $|P|$ with values $\infty$ // This array contains the network
    distances from $q_s$ to all the data points
 5: Initialize a bounding box $B_S = [(-\infty, -\infty), (\infty, \infty)]$ // [bottom left point, upper right
    point]
 6: while $H$ is not empty do
 7:     $(d, p) = H.pop()$
 8:     if $p$ is in $Closed$ then
 9:         continue
10:     end if
11:     Add $p$ into $Closed$
12:     yield $c = p$ // Output the next candidate point
13:     Update $B_S$ according to the results of the skyline check component
14:     for all network neighbor $N(p)$ of $p$ do
15:         if $N(p)$ lies inside $b$ then
16:             Update $d(N(p), q_s) = d_N(p, q_s) + d_N(N(p), p)$
17:             if $d(N(p), q_s) < Dist[N(p)]$ then
18:                 $Dist[N(p)] = d(N(p), q_s)$
19:                 Push $(d(N(p), q_s), N(p))$ into $H$
20:             end if
21:         end if
22:     end for
23: end while
```

4.4.3 Skyline check component

We modify the index skyline check method (ISC) mentioned in 2.2.2 to apply it to the road network space. We call this new algorithm Network Index Skyline Check method ($NISC$).

Like ISC, $NISC$ also maintains a spatial index, a KD-tree in our thesis, storing all the skyline points in the query space. Instead of storing the Euclidean distances from skyline points to the query points, $NISC$ stores skyline points using their network distances to the query points.

When $NISC$ meets a new candidate point p, the method will perform a range query to the KD-tree, checking if any network skyline point lies inside the hypercube w formed by p in Euclidean space and the origin. $w = [(0, 0, \ldots, 0), (d_E(p, q_1), d_E(p, q_2), \ldots, d_E(p, q_{|Q|}))]$ is represented by its bottom left vertex and upper right vertex. Starting from the root node of the KD-tree, $NISC$ goes down the KD-tree. Let v be an internal node of the KD-tree. Let $x_i(v)$ be the splitting value of v in the dimension i. There are two cases:

1. $x_i(v) \geq d_E(p, q_i)$: we only need to search the left subtree of v, since any node in the right subtree of v stay outside w.
2. $x_i(v) < d_E(p, q_i)$: we need to search both left and right subtree of v.

When we reach a leaf node, skyline checks are performed to the skyline points contained in the node using the dominance check method. If any skyline point in the leaf node dominates p, we can stop immediately.

After we examined the data points in the first leaf node, if no skyline point inside the leaf node dominates p, $NISC$ starts the back-up procedure. $NISC$ goes back to the parent internal node of the leaf node. It now computes the network distance between p and q_i where i is the splitting dimension of the parent internal node. The window of the range query now becomes $w = [(0, 0, \ldots, 0), (d_E(p, q_1), d_E(p, q_2), \ldots, d_N(p, q_i), \ldots, d_E(p, q_{|Q|}))]$. There are three cases:

1. $x_i(v) \geq d_N(p, q_i)$: we only need to search the left subtree of v again, since any node in the right subtree of v still stays outside w.
2. $x_i(v) < d_E(p, q_i) < d_N(p, q_i)$: we only need to search the right subtree of v.
3. $d_E(p, q_i) \leq x_i(v) < d_N(p, q_i)$: we need to search both the left and the right subtree of v.

Once the searching procedure is done, and there is no skyline point in the subtree rooted at v dominates p, the back-up procedure can continue. $NISCB$ stops when the network distances to all the query points are computed and the range query with the hypercube $w = [(0, 0, \ldots, 0), (d_N(p, q_1), d_N(p, q_2), \ldots, d_N(p, q_{|Q|}))]$ is complete.

Lemma 4.8. *The worst case runtime complexity of $NISC$ for a set of candidate points C_S is $O(|Q||C_S| \log |C_S| + |C_S||Q|^2|S|^{1-\frac{1}{|Q|}})$.*

Proof. $O(|Q||C_S| \log |C_S|)$ time is spent on computing the network distances from all the query points of Q to all the candidate points of C_S. $O(|C_S||Q|^2|S|^{1-\frac{1}{|Q|}})$ time is spent on performing skyline checks to all the candidate points. Each skyline check consists of at most $O(|Q|)$ range queries to the KD-tree that stores the network skyline points found so far. Each range query takes at most $O(|Q||S|^{1-\frac{1}{|Q|}})$ time in the worst case. □

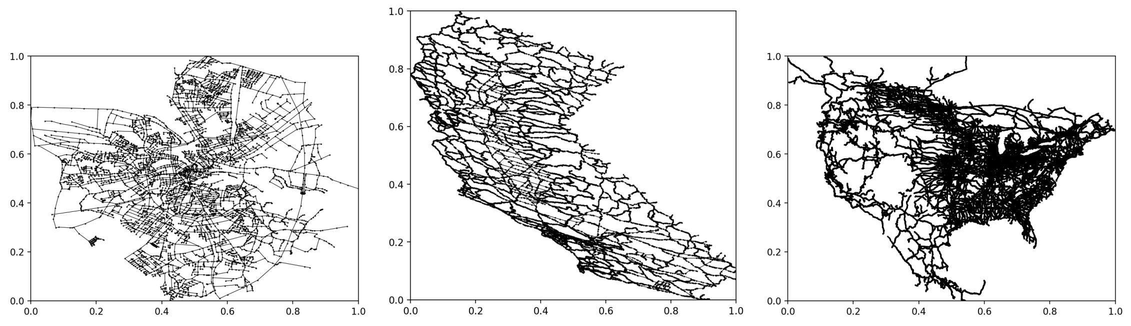
Theorem 4.9. *The worst case runtime complexity of the $NISCB$ algorithm is $O(|T_S| \log |T_S| + |Q||C_S| \log |C_S| + |C_S||Q|^2|S|^{1-\frac{1}{|Q|}})$.*

Proof. The time complexity of $NISCB$ is computed by combining Lemma 4.7 and Lemma 4.8 together. □

4.5 Experiments

4.5.1 Experimental Settings

The dataset used in the experiment is the same as the Point of Interest (POI) dataset [27] used in section 2.3.2. We use both the node data and the edge data of the POI dataset in this chapter. Three road network datasets with different numbers of data points and edges are used. They are the road network of California (CA), the road network of Oldenburg (OL), and the road network of North America (NA). All the data points are normalized to lie within the unit square to simulate different network density. Figure 4.4 visualizes the three datasets. Table 4.1 contains the parameter configuration of the experiments.



(a) OL (b) CA (c) NA

Figure 4.4: Visualization of three road network datasets

Parameter	Setting
Dataset (point size, edge size)	OL(6105, 7037), CA(21058, 21697), NA(175813, 179198)
Query point set size	5, 15
Region size	0.025, 0.05

Table 4.1: Configuration of road network data

As for the generation of query points, a randomly selected data point p is chosen, and a square region is computed using p as its center. The side length of the square region is 2 times the value of the region size. The query points are generated on the randomly selected network edges within the rectangular region.

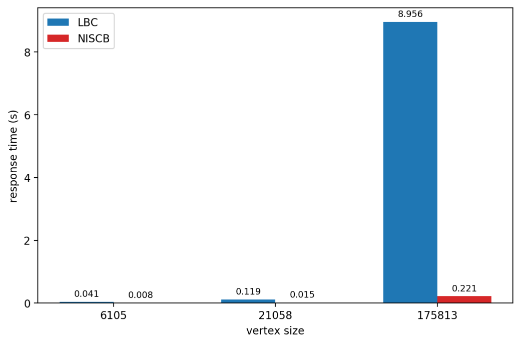
4.5.2 Experimental Results

In this section, the experimental results are provided and analyzed to show that $NISCB$ outperforms LBC on the datasets considered here. Table 4.2 contains the experimental results.

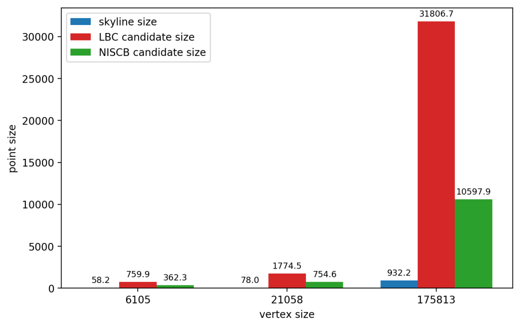
Dataset		OL				CA				NA			
Query size		5		15		5		15		5		15	
Region size		0.025	0.05	0.025	0.05	0.025	0.05	0.025	0.05	0.025	0.05	0.025	0.05
skyline size		14.47	24.85	27.	58.2	20.98	46.6	36.17	77.97	304.07	559.48	540.88	932.23
Total time (s)	*LBC*	0.04	0.05	0.18	0.68	0.04	0.87	0.22	2.27	8.15	30.75	30.19	163.13
	NISCB	**0.03**	**0.04**	**0.14**	**0.51**	**0.02**	**0.23**	**0.17**	**0.93**	**1.21**	**4.88**	**7.07**	**24.12**
candidate size	*LBC*	305.2	366.1	390.5	759.9	118.5	1048.5	774.	1774.5	5358.5	16920.8	7355.3	31806.7
	NISCB	**79.7**	**128.9**	**124.3**	**362.3**	**79.2**	**616.6**	**162.7**	**754.6**	**2855.3**	**7245.9**	**4253.5**	**10597.9**
tentative size	*LBC*	310.	370.9	405.5	774.7	123.2	1053.	787.6	1788.5	5363.2	16925.8	7368.6	31821.1
	NISCB	**95.8**	**150.1**	**156.8**	**406.9**	**89.7**	**632.1**	**186.3**	**786.1**	**2884.7**	**7287.7**	**4303.4**	**10651.5**
traversal pruning (s)	*LBC*	0.007	0.011	0.013	0.041	0.011	0.187	0.02	0.119	1.695	3.514	1.617	8.956
	NISCB	**0.002**	**0.003**	**0.003**	**0.008**	**0.002**	**0.011**	**0.003**	**0.015**	**0.055**	**0.144**	**0.087**	**0.221**
skyline check (s)	*LBC*	0.03	0.04	0.17	0.64	0.03	0.68	0.2	2.15	6.44	27.2	28.55	154.1
	NISCB	**0.03**	**0.04**	**0.14**	**0.5**	**0.02**	**0.21**	**0.17**	**0.91**	**1.15**	**4.71**	**6.98**	**23.86**

Table 4.2: Experimental results of algorithms in the road network space

Comparing traversal and pruning components The traversal and pruning components of $NISCB$ outperform the corresponding components of LBC with respect to different datasets, query size, and region size. $NISCB$ has a much shorter response time than LBC. The reason is that checking whether points lie inside a bounding box (the bounding box algorithm) is much faster than performing skyline checks to the bounding boxes of the KD-tree nodes. The candidate size and tentative size of $NISCB$ are only $\frac{1}{4}$ to $\frac{1}{2}$ of those of LBC. The bounding box method tends to prune out more data points than the pruning component of LBC. Figure 4.5 shows the experimental results on different road network datasets with respect to traversal and pruning time, and skyline/candidate sizes. The figure shows the case when the query size is 15 and the region size is 0.05.



(a) Traversal and pruning time



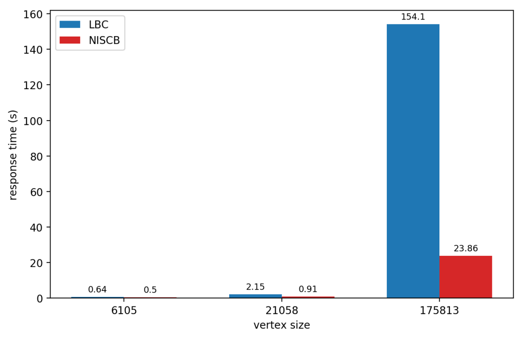
(b) Skyline/candidate size

Figure 4.5: Experimental results on the road network datasets

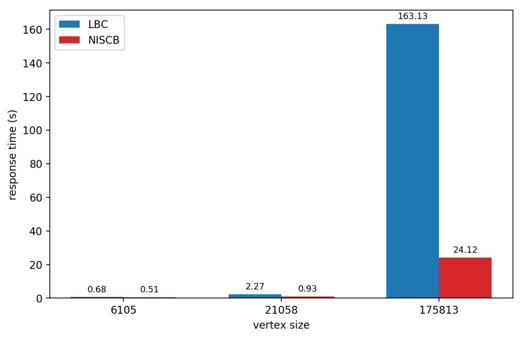
Comparing skyline check component The skyline check component of $NISCB$ outperforms the skyline check component of LBC under different parameter settings. Note that when the data size becomes large, the algorithm spent most of the time on performing skyline checks. When the data size or the region size becomes very large, $NISCB$ can finish the skyline checks quickly while the time spent on the skyline checks by the LBC algorithm increases dramatically. The reason is that, when performing a skyline check, $NISCB$ only examines a small subset of the network skyline points by taking advantage of the spatial index. Figure 4.6 shows the experimental results on different road network datasets with respect to skyline check time and total response time. The figure shows the case when the query size is 15 and the region size is 0.05.

4.6 Conclusion

In this chapter, we first analyze an existing algorithm, LBC, in solving spatial skyline query problem in road network space. The worst case runtime complexity of LBC is $O(\sqrt{|P|}|Q||S| + |T_S|\log|T_S| + |Q||C_S|\log|C_S| + |C_S||Q|^2|S|)$. A mistake in the traversal method of LBC is then pointed out. We propose a correct and efficient algorithm, $NISCB$.



(a) Skyline check time



(b) Total response time

Figure 4.6: Experimental results on the road network datasets

The worst case run-time complexity of $NISCB$ is $O(|T_S|\log|T_S|+|Q||C_S|\log|C_S|+|C_S||Q|^2|S|^{1-\frac{1}{|Q|}})$. Extensive experiments are conducted to show that $NISCB$ outperforms LBC component by component.

Chapter 5

Conclusion

In this thesis, we have studied spatial skyline query problem in 2-dimensional Euclidean space, high dimensional Euclidean space, and road network space. For a given data set P, we are required to compute the spatial skyline points of P with respect to an arbitrary query set Q. A point $p \in P$ is a spatial skyline point if and only if, for any other data point $r \in P$, p is closer to at least one query point $q \in Q$ as compared to r and has in the best case the same distance as r to the rest of the query points.

Our proposed algorithms consist of three components: the traversal component, the pruning component, and the skyline check component. The traversal component determines the order with which the tentative points are visited. The pruning component quickly prunes out a subset of non-skyline tentative points using the dominating region of the skyline points. The rest of the tentative points become candidate points. The skyline check component performs skyline checks to the candidate points against the skyline points found so far. The skyline check component retrieves all the skyline points from the candidate points.

We studied the spatial skyline query problem in the 2-dimensional Euclidean plane in Chapter 2. We proposed two efficient skyline check methods, namely BSC and ISC. BSC is based on the previously proposed dominance check method. Additionally, it takes advantage of the bounding box of the dominator region. ISC takes advantage of the spatial index to complete a skyline check with only a subset of the skyline points. Two algorithms, that use the two skyline check methods respectively, are $BSCS$ and $ISCSBL$. Experiments show that our algorithms outperform the ES algorithm by Son et al. [39]. We also proposed the bounding box and the bounding line method as a solution to the pruning component.

We also studied the spatial skyline query problem in high dimensional Euclidean space in Chapter 3. We incorporated the lazy construction technique into the index traversal method. We showed how to incorporate the bounding box and the bounding plane method into the index traversal method using the KD-tree. The bounding box and bounding plane method can significantly prune out more tentative points than the bounding box method with the increase of dimension. We also showed that ISC can be used in high dimensional Euclidean space. Two algorithms, IB^2I and IB^2D, are proposed that use different components. They are compared with the SBD algorithm, which is the high dimensional version of ES algorithm. Experiments show that our algorithms outperform SBD algorithm. IB^2I has the best performance in general. IB^2D has the best performance when the candidate size is small.

We studied the spatial skyline query problem in road network space in Chapter 4. We analyzed the LBC algorithm by Deng et al. [12], and corrected a mistake in its traversal component. We then proposed our algorithm, $NISCB$, which uses Dijkstra's search from a dummy query point as its traversal component, the bounding box method as its pruning component, and $NISC$ as its skyline check component. Experiments show that $NISCB$ outperforms LBC with respect to candidate size and response time under various scenario.

Bibliography

[1] Bartolini, I., Ciaccia, P., & Patella, M. (2006). SaLSa: Computing the skyline without scanning the whole sky. *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, 405-414.

[2] Bentley, J. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM Vol.18*(9), 509-517.

[3] Bhattacharya B., Bishnu A., Cheong O., Das S., Karmakar A., & Snoeyink J. (2010) Computation of Non-dominated Points Using Compact Voronoi Diagrams. *In: Rahman M.S., Fujita S. (eds) WALCOM: Algorithms and Computation. WALCOM 2010. Lecture Notes in Computer Science, vol 5942. Springer, Berlin, Heidelberg*

[4] Borzsony, S., Kossmann, D., & Stocker, K. (2001). The Skyline operator. *Proceedings 17th International Conference on Data Engineering*, 421-430.

[5] Brown, K. (1979). Voronoi diagrams from convex hulls. *Information Processing Letters, 9*(5), 223-228.

[6] Chan, C., Jagadish, H., Tan, K., Tung, A., & Zhang, Z. (2006). On High Dimensional Skylines. In *Advances in Database Technology - EDBT 2006: 10th International Conference on Extending Database Technology, Munich, Germany, 26-31 March, 2006* (Vol. 3896, Lecture Notes in Computer Science, pp. 478-495). Berlin, Heidelberg: Springer Berlin Heidelberg.

[7] Chan, C., Jagadish, H., Tan, K., Tung, A., & Zhang, Z. (2006). Finding k-dominant skylines in high dimensional space. *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, 503-514.*

[8] Chanzy, P., Devroye, L., & Zamora-Cura, C. (2001). Analysis of range search for random k-d trees. *Acta Informatica, 37*(4), 355-383.

[9] Cormen, & Cormen, Thomas H. (2009). *Introduction to algorithms (3rd ed.).* Cambridge, Mass.: MIT Press.

[10] De Berg, M., Cheong, Van Kreveld, & Overmars. (2008). *Computational Geometry: Algorithms and Applications* (Third ed.). Berlin, Heidelberg: Springer Berlin Heidelberg.

[11] Dellis, E., Vlachou, A., Vladimirskiy, I., Seeger, B., & Theodoridis, Y. (2006). Constrained subspace skyline computation. *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, 415-424.

[12] Deng, K., Zhou, X., & Shen, H. (2007). Multi-source Skyline Query Processing in Road Networks. *2007 IEEE 23rd International Conference on Data Engineering*, 796-805.

[13] Donald, K., Frank, R., & Steffen, R. (2002) Shooting stars in the sky: an online algorithm for skyline queries. In *Proceedings of the 28th international conference on Very Large Data Bases, (VLDB '02).* VLDB Endowment, 275–286.

[14] Fischer K., Gärtner B., Kutz M. (2003). *Fast Smallest-Enclosing-Ball Computation in High Dimensions. In: Di Battista G., Zwick U. (eds) Algorithms - ESA 2003. ESA 2003.* Lecture Notes in Computer Science, vol 2832. Springer, Berlin, Heidelberg

[15] Friedman, J., Bentley, J., & Finkel, R. (1977). An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software (TOMS), 3*(3), 209-226.

[16] Gao, Y., Miao, X., Cui, H., Chen, G., & Li, Q. (2014). Processing k-skyband, constrained skyline, and group-by skyline queries on incomplete data. *Expert Systems with Applications*, 41(10), 4959-4974.

[17] Goldberg, A., & Harrelson, C. (2005). Computing the shortest path: A search meets graph theory. *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 156-165.

[18] Göbel, R., & Sobh, T. (2007). Towards Logarithmic Search Time Complexity for R-Trees. *In Innovations and Advanced Techniques in Computer and Information Sciences and Engineering* (pp. 201-206). Dordrecht: Springer Netherlands.

[19] Im, H., & Park, S. (2012). Group skyline computation. *Information Sciences, 188*(C), 151-169.

[20] Ji, Z., Xunfei, J., Wei-Shinn, K., & Xiao, Q. (2016). Efficient Parallel Skyline Evaluation Using MapReduce. *IEEE Transactions on Parallel and Distributed Systems*, 27(7), 1996-2009.

[21] Kanth, K.V.R., & Singh, A. (1999). Optimal Dynamic Range Searching in Non-replicating Index Structures. *Proc. International Conference on Database Theory*, LNCS 1540, 257-276.

[22] Kian-Lee, T., Pin-Kwang, E., & Beng C. O. (2001) Efficient Progressive Skyline Computation. *In Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 301–310.

[23] Klee, V., & Washington Univ Seattle Dept Of Mathematics. (1979). *On the Complexity of D-Dimensional Voronoi Diagrams.*

[24] Kriegel, H., Renz, & Schubert. (2010). Route skyline queries: A multi-preference path planning approach. *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, 261-272.

[25] Kung, H., Luccio, & Preparata. (1975). On Finding the Maxima of a Set of Vectors. *Journal of the ACM (JACM)*, 22(4), 469-476.

[26] Lee, J., You, G., & Hwang, S. (2009). Personalized top-k skyline queries in high-dimensional space. *Information Systems*, 34(1), 45-61.

[27] Li, F., Cheng, D., Hadjieleftheriou, M., Kollios, G., & Teng, S. (2005). On Trip Planning Queries in *Spatial Databases. In Advances in Spatial and Temporal Databases: 9th International Symposium, SSTD 2005, Angra dos Reis, Brazil, August 22-24, 2005. Proceeding* (Vol. 3633, Lecture Notes in Computer Science, pp. 273-290). Berlin, Heidelberg: Springer Berlin Heidelberg.

[28] Liu, J., Xiong, L., Pei, J., Luo, J., & Zhang, H. (2015). Finding Pareto optimal groups: Group-based skyline. *Proceedings of the VLDB Endowment, 8*(13), 2086-2097.

[29] Luby, M., & Ragde, P. (1989). A bidirectional shortest-path algorithm with good average-case behavior. *Algorithmica, 4*(1), 551-567.

[30] Megiddo, N. (1983). Linear-Time Algorithms for Linear Programming in R^3 and Related Problems. *SIAM Journal on Computing, 12*(4), 759-776.

[31] Papadias, D., Tao, Y., Fu, G., & Seeger, B. (2003). An optimal and progressive algorithm for skyline queries. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 467-478.

[32] Papadias, D., Tao, Y., Fu, G., & Seeger, B. (2005). Progressive skyline computation in database systems. *ACM Transactions on Database Systems (TODS), 30(1)*, 41-82.

[33] Pei, J., Yuan, Y., Lin, X., Jin, W., Ester, M., Liu, Q., . . . Zhang, Q.. (2006). Towards multidimensional subspace skyline analysis. *ACM Transactions on Database Systems (TODS), 31*(4), 1335-1381.

[34] Safar, M., El-Amin, D., & Taniar, D. (2011). Optimized skyline queries on road networks using nearest neighbors. *Personal and Ubiquitous Computing*, 15(8), 845-856.

[35] Sedgewick, R., & Vitter, J. (1986). Shortest paths in euclidean graphs. *Algorithmica, 1*(1-4), 31-48.

[36] Sharifzadeh, M., Shahabi, C. (2006). The spatial skyline queries. *In Proceedings of the 32nd international conference on Very large databases (VLDB '06).* VLDB Endowment, 751–762.

[37] Sharifzadeh, M., Shahabi, C. & Kazemi, L. (2009). Processing spatial skyline queries in both query spaces and spatial network databases. *ACM Transactions on Database Systems* (TODS), 34(3), 1-45.

[38] Sint, L., & De Champeaux, D. (1977). An Improved Bidirectional Heuristic Search Algorithm. *Journal of the ACM (JACM)*, 24(2), 177-191.

[39] Son, W., Lee, M., Ahn, H., & Hwang, S. (2009). Spatial Skyline Queries: An Efficient Geometric Algorithm.

[40] Son, W., Hwang, S., & Ahn, H. (2014). MSSQ: Manhattan Spatial Skyline Queries. *Information Systems*, 40(C), 67-83.

[41] Son, W., Stehn, F., Knauer, C., & Ahn, H. (2017). Top-k Manhattan spatial skyline queries. *Information Processing Letters*, 123, 27-35.

[42] Tang, Y., & Chen, S. (2018). Supporting Continuous Skyline Queries in Dynamically Weighted Road Networks. *Mathematical Problems in Engineering*, 2018, 14.

[43] Wagner, D., & Willhalm, T. (2007). Speed-Up Techniques for Shortest-Path Computations. In *STACS 2007: 24th Annual Symposium on Theoretical Aspects of Computer Science, Aachen, Germany, February 22-24, 2007. Proceedings* (Vol. 4393, Lecture Notes in Computer Science, pp. 23-36). Berlin, Heidelberg: Springer Berlin Heidelberg.

[44] Wang, J. (2012). *Geometric Structure of High-Dimensional Data and Dimensionality Reduction by Jianzhong Wang.* (1st ed. 2012. ed.). Springer Berlin Heidelberg : Imprint: Springer

[45] Yannis, M., Alexandros, N., Apostolos N. P., Yannis, T. (2006). R-Trees: Theory and Applications. 1st ed. *Advanced Information and Knowledge Processing.*

[46] Yuan-Ko, H. (2017). Within Skyline Query Processing in Dynamic Road Networks. *ISPRS International Journal of Geo-Information*, 6(5), 137.

[47] Zeng, W., & Church, R. (2009). Finding shortest paths on real road networks: The case for A*. *International Journal of Geographical Information Science*, 23(4), 531-543.

[48] Zhang, Z., Guo, X., Lu, H., Tung, A., & Wang, N. (2005). Discovering strong skyline points in high dimensional spaces. *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, 247-248.

[49] Haoyang, Z., Peidong, Z., Xiaoyong, L., Qiang, L., & Peng. X. (2017). Parallelization of group-based skyline computation for multi-core processors: Parallelization of Group-based Skyline Computation for Multi-core Processors. *Concurrency and Computation: Practice and Experience, 29*(18), 1-20.