Distribution Agreement

In presenting this thesis or dissertation as a partial fulfillment of the requirements for an advanced degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis or dissertation in whole or in part in all forms of media, now or hereafter known, including display on the world wide web. I understand that I may select some access restrictions as part of the online submission of this thesis or dissertation. I retain all ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Signature:

Jinfei Liu	Date

Efficient and Adaptive Skyline Computation

by

Jinfei Liu
Doctor of Philosophy

Computer Science and Informatics

Li Xiong, Ph.D.
Advisor

Vaidy Sunderam, Ph.D.
Committee Member

Michelangelo Grigni, Ph.D.
Committee Member

Jian Pei, Ph.D.
Committee Member

Accepted:

Lisa A. Tedesco, Ph.D.
Dean of the James T. Laney School of Graduate Studies

Date

Efficient and Adaptive Skyline Computation

by

Jinfei Liu
M.E., University of Science and Technology of China, 2012

Advisor: Li Xiong, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2017

Abstract

Efficient and Adaptive Skyline Computation
By Jinfei Liu

Skyline, also known as Maxima in computational geometry or Pareto in business management field, is important for many applications involving multi-criteria decision making. The skyline of a set of multi-dimensional data points consists of the points for which no other point exists that is better in at least one dimension and at least as good in every other dimension. Although skyline computation and queries have been extensively studied in both computational geometry and database communities, there are still many challenges need to be fixed, especially in this big data era. In this dissertation, we present several efficient and adaptive skyline computation algorithms. First, we show a faster output-sensitive skyline computation algorithm which is the state-of-the-art algorithm from the theoretical aspect. Second, traditional skyline computation is inadequate to answer queries that need to analyze not only individual points but also groups of points. To address this gap, we adapt the original skyline definition to the novel group-based skyline (G-Skyline), which represents Pareto optimal groups that are not dominated by other groups. Third, to facilitate skyline queries, we propose a novel concept Skyline Diagram, which given a set of points, partitions the plane into a set of regions, referred to as skyline polyominos. Similar to kth-order Voronoi Diagram commonly used to facilitate k nearest neighbor (kNN) queries, any query points in the same skyline polyomino have the same skyline query results.

Efficient and Adaptive Skyline Computation

by

Jinfei Liu
M.E., University of Science and Technology of China, 2012

Advisor: Li Xiong, Ph.D.

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2017

Acknowledgments

Five years ago, when I wrote the acknowledgment for my master's thesis, I imagined what I could write in the acknowledgment for my dissertation. How time flies! It's the time to write this acknowledgment now. I would like to express my gratitude to those who helped me reach this point.

Foremost among those is my advisor Professor Li Xiong. I cannot say you are the best advisor in the world, but I can say you are the "Skyline" advisor for me in the world. The most desired tacit understanding between advisor and student is that they can understand each other even they only say a half when discussing a problem. And I think we have achieved that. Thank you for seeing the potential in me and accepting me as a special standing student five years ago. Thank you for offering me the freedom so I can do what I love to do. Thank you for so many opportunities that you provided me, e.g., collaborating with researchers outside Emory, and supporting me for attending conferences. Your passion and devotion to work and your patience and encouragement to students have inspired me and are something I will strive to emulate in my future academic career.

I would like to thank Professor Vaidy Sunderam, Professor Michelangelo Grigni, and Professor Jian Pei for serving on my dissertation committee, and Professor Joyce Ho for serving on my qualifying exam committee. Thank you for all your time and efforts on supporting me.

I would like to thank my collaborator Professor Jian Pei. You are so intelligent and creative that always impress me. Your insight on unit-group and sweeping algorithm for skyline diagram make me know what the most top researchers should be. I learned a lot from you, not only in research but also about how to be a good mentor with patience and encouragement. I would also like to thank my collaborator Professor Jun Luo. Your passion and devotion to research always inspire me.

I cannot imagine what the four years' Ph.D. life would be without the supporting from my great friends: Guolan Lv, Jiulin Hu, Wenlu Ye, Yanhui Liang, Dejun Teng, Xin Chen, Haoyu Zhang, Jason Yang, Qiuchen Zhang, Xiaofeng Xu, Boyi Yang, Huanhuan Yang, Kai Yuan, Qingpo Cai, Yi Deng, Yuqi Sun, Yonghui Xiao, Haoran Li, Liyue Fan, Yousef Elmehdwi, Daniel Garcia Ulloa, Farnaz Tahmasebian, Layla Pournajaf, Yang Cao, Kevin Zhao, Tom Zhang, Michael Solomon, Luca Bonomi, Shengzhi Xu, Slawomir Goryczka.

I would like to thank Terry Ingram for all the paperwork and Edgar Leon for managing clusters in my experimental studies.

I would like to thank my middle school classmates, Jianyu and Huiquan to help me to take care of my family members.

Last but not least, I'd like to express my deepest gratitude to my parents for being the constant source of love and encouragement; my brother, Jinpeng, who always be a good example for me; and my wife, Jiao, who has given me a lifetime to look forward to.

To my family

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

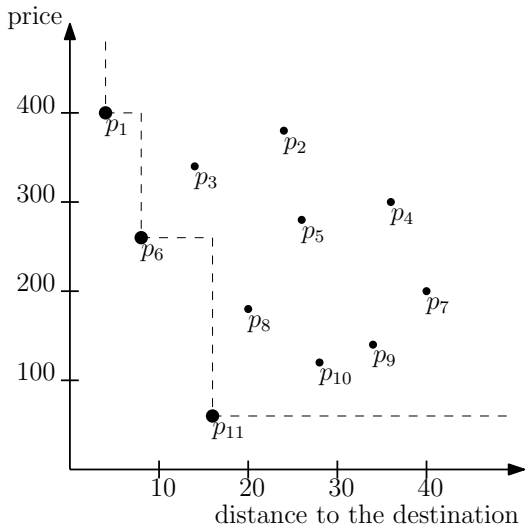
Skyline, also known as Maxima in computational geometry or Pareto in business management field, is important for many applications involving multi-criteria decision making. The skyline of a set of multi-dimensional data points consists of the points for which no other point exists that is better in at least one dimension and at least as good in every other dimension.

Assume that we have a dataset P of n points. Each point p of d real-valued attributes can be represented as a d-dimensional point $(p[1], p[2], ..., p[d]) \in \mathbb{R}^d$ where $p[i]$ is the i-th attribute of p. Given two points $p = (p[1], p[2], ..., p[d])$ and $p' = (p'[1], p'[2], ..., p'[d])$ in $\mathbb{R}^d$, p dominates p' if for every i, $p[i] \leqslant p'[i]$ and for at least one i, $p[i] < p'[i]$ $(1 \leqslant i \leqslant d)$. Given the set of points P, the skyline is defined as the set of points that are not dominated by any other point in P. In other words, the skyline represents the *best points* or Pareto optimal solutions from the dataset since the points within the skyline cannot dominate each other.

Figure 1.1(a) illustrates a dataset $P = \{p_1, p_2, ..., p_{11}\}$, each representing a hotel with two attributes: the distance to the destination and the price. Figure

hotel	distance	price
p_1	4	400
p_2	24	380
p_3	14	340
p_4	36	300
p_5	26	280
p_6	8	260
p_7	40	200
p_8	20	180
p_9	34	140
p_{10}	28	120
p_{11}	16	60

(a)



(b)

Figure 1.1: A skyline example of hotels.

1.1(b) shows the corresponding points in the two dimensional space where the x and y coordinates correspond to the attributes of distance to the destination and price, respectively. We can see that $p_3(14, 340)$ dominates $p_2(24, 380)$ as an example of dominance. The skyline of the dataset contains p_1, p_6, and p_{11}. Suppose the organizers of a conference need to reserve *one* hotel considering both distance to the conference destination and the price for participants, the skyline offers a set of best options or Pareto optimal solutions with various tradeoffs between distance and price: p_1 is the nearest to the destination, p_{11} is the cheapest, and p_6 provides a good compromise of the two factors. p_8 will not be considered as p_{11} is better than p_8 in both factors.

In this big data era, how to find skyline points is extremely important for many applications involving multi-criteria decision making. Although skyline computation and queries have been extensively studied in both computational geometry and database communities [37] [36] [24] [49] [41], there are still many challenges need to be fixed. From the theoretic aspect, the state-of-the-art output-sensitive algorithm had not been improved in past 29 years, and there

is no precomputation structure to enhance the queries of skyline. From the application aspect, traditional skyline computation is inadequate to answer queries that need to analyze not only individual points but also groups of points. To address those challenges, in this dissertation, we present several algorithms to efficiently and adaptively compute skyline as follows.

Faster Skyline Algorithm. We present the second output-sensitive skyline computation algorithm which is faster than the only existing output-sensitive skyline computation algorithm [24] in the worst case because our algorithm does not rely on the existence of a linear time procedure for finding medians. Traditional skyline computation algorithm achieves $O(n \log n)$ time complexity. The only existing output-sensitive skyline algorithm achieves $O(n \log k)$ time complexity, where k is the number of skyline points. However, this algorithm had not been improved in past 29 years. For each iteration, our algorithm requires $2nlogk$ comparisons which is much faster than $5.430\dot{5}n \log k$ comparisons in [24].

Group-based Skyline. Traditional skyline computation is inadequate to answer queries that need to analyze not only individual points but also groups of points. To address this gap, we adapt/generalize the original skyline definition to the novel group-based skyline (G-Skyline), which represents Pareto optimal groups that are not dominated by other groups. In order to compute G-Skyline groups consisting of k points efficiently, we present a novel structure that represents the points in a directed skyline graph and captures the dominance relationships among the points based on the first k skyline layers. We propose efficient algorithms to compute the first k skyline layers. We then present two heuristic algorithms to efficiently compute the G-Skyline groups: the point-wise algorithm and the unit group-wise algorithm, using various

pruning strategies. The experimental results on the real NBA dataset and the synthetic datasets show that G-Skyline is interesting and useful, and our algorithms are efficient and scalable.

Skyline Diagram. To facilitate skyline queries, we propose a novel concept Skyline Diagram, which given a set of points, partitions the plane into a set of regions, referred to as skyline polyominos. Any query points in the same skyline polyomino have the same skyline query results. Similar to kth-order Voronoi diagram that is commonly used to facilitate k nearest neighbor (kNN) queries, skyline diagram can be used to facilitate skyline queries and many other applications including reverse skyline queries, Private Information Retrieval (PIR) based skyline queries, and authentication of skyline queries. While the skyline diagram has many applications, it can be computationally expensive to build the diagram. By exploiting interesting properties of the skyline, we present several efficient algorithms for building the diagram with respect to three kinds of skyline queries, quadrant, global, and dynamic skyline. The experimental results on the real dataset and the synthetic datasets show that our algorithms are efficient and scalable.

Organization. The rest of the dissertation is organized as follows. Chapter 2 shows a faster output-sensitive skyline computation algorithm. The adaptive group-based skyline is presented in Chapter 3. Similar to kth-order Voronoi diagram commonly used to facilitate k nearest neighbor (kNN) queries, we present skyline diagram which can be used to facilitate skyline queries in Chapter 4. Chapter 5 concludes the dissertation and discusses the future directions.

Chapter 2

Faster Output-sensitive Skyline Computation

2.1 Related Work

The skyline computation problem has been extensively studied in the computational geometry field. The best of existing worst-case algorithms [2] [3] [4] [26] are based on divide-and-conquer paradigm which achieves $O(n \log n)$ time complexity in two dimensional space, where n is the number of points. The only existing output-sensitive skyline computation algorithm was presented in [24] by Kirkpatrick and Seidel. [24] illustrated an algorithm that achieves $O(n \log k)$ time complexity where k is the number of skyline points. Recently, Hu et al. [20] extended the output-sensitive algorithm to external memory. Unfortunately, both algorithms rely on the existence of a linear time median algorithm [5] in the first step which lead to more than $5.430\dot{5}n \log k$ comparisons. Recently, Chan and Lee [10] presented two output-sensitive algorithms that achieve expected comparisons of $n \log k + O(n\sqrt{\log k})$.

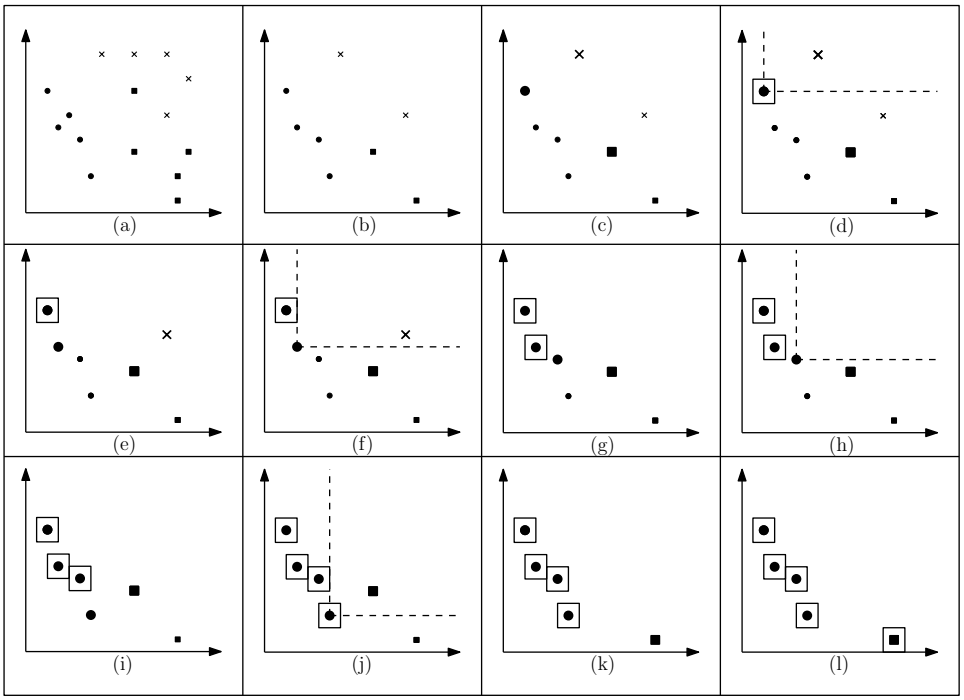


Figure 2.1: An example of Algorithm 2.1. (a) Step 1: partition P into subsets $P_1, P_2, ..., P_{\lceil n/K \rceil}$ randomly. (b) Step 2: compute skyline points of each subset. (c) Step 3: choose candidate skyline points. (d) Step 4: obtain one skyline point. (e) After eliminating non-skyline points (Step 5).

2.2 Output-Sensitive Skyline Computation Algorithm

Let $P \subset \mathbb{R}^2$ be a set of n points and K be the number of skyline points we expect. Since the number of skyline points k is not known in advance, we will show later how to use a sequence of K values to find all k skyline points, that is, increase K until $K \geqslant k$. The algorithm given K is shown in Algorithm 2.1. The overall algorithm to find all k skyline points is shown in Algorithm 2.2.

In Algorithm 2.1, Step 1 partitions the n points into $\lceil n/K \rceil$ subsets. Then Step 2 computes the skyline points of each subset in $\lceil n/K \rceil \times O(K \log K) = O(n \log K)$ time. We then find a global skyline point by selecting a candidate skyline point in each subset (Step 3) and selecting the point with the global

Algorithm 2.1: $O(nlogK)$ SKYLINE(P,K)

Input: a set of n points in two-dimensional space
Output: k skyline points or $\varnothing$

```
/*Step 1: partition*/
partition P into subsets P_1, P_2, ..., P_⌈n/K⌉ randomly, each of size at most K.
/*Step 2: compute skyline points of each subset*/
for j = 1, 2, ..., ⌈n/K⌉ do
    compute the skyline points of P_j using worst-case O(n log n) skyline
    algorithm [39].
end for
for i = 1, 2, ..., K do
    /*Step 3: choose the candidate skyline points*/
    for j = 1, 2, ..., ⌈n/K⌉ do
        choose point p_j with smallest value on first dimension value as a
        candidate skyline point.
    end for
    /*Step 4: obtain one skyline point*/
    compute the point p_i with smallest first dimension value from
    p_j, 1 ⩽ j ⩽ ⌈n/K⌉.
    /*Step 5: eliminate non-skyline points*/
    for j = 1, 2, ..., ⌈n/K⌉ do
        perform a binary search to delete those points whose second
        dimension value is equal to or greater than p_i[2].
    end for
    if no point in P then
        return SKYLINE={p_1, p_2, ..., p_i}.
    end if
end for
return ∅.
```

minimum value (Step 4). This point is used to eliminate all points dominated by this point in Step 5. Because of this elimination, we can guarantee that a skyline point is obtained in each subsequent iteration (Lemma 2.1). Step 3 to Step 5 are repeated for K iterations or until there is no remaining point, in which case all k ($k \leqslant K$) skyline points will be returned. If $k > K$, Algorithm 2.1 outputs an empty set since there are still remaining points after K skyline points are found with K iterations.

Example 2.1. Given 15 points in two-dimensional space, an example of Algorithm 2.1 is shown in Figure 2.1. For simplicity, we assume $k = 5$ is known in advance. In (a), the 15 points are partitioned into 3 subsets (circle, box, and cross) of 5 each. The skyline points of each subset are then computed, shown in (b). Then we choose the point with smallest first dimension value from each subset as the candidate skyline point, shown in (c). From these three candidate points, we choose the point with smallest first dimension value as the skyline point, highlighted in (d). Then all points that are dominated by this skyline point, i.e. the points whose second dimension is equal to or greater than the determined skyline point, are eliminated, shown in (e). Then the next skyline point from the remaining points is selected in next iteration and used to eliminate the dominated pointed, shown in (f). The algorithm continues until all 5 skyline points are found.

Lemma 2.1. We can obtain one skyline points in each iteration of Step 4.

Proof. It is easy to see that we can obtain a skyline point from the first iteration because no point can dominate p_i due to the smallest value in the first dimension. For the second iteration, because all the points dominated by p_i are eliminated, the point with smallest first dimension value of remaining points in P should be a skyline point as no other point can dominate it. The analysis for all other skyline points follows similarly. □

Theorem 2.1. Skyline can be computed in $O(n \log K)$ time in worst-case using Algorithm 2.1 where n is the number of points.

Proof. Step 1 requires $O(n)$ time to partition. Line 5 requires $O(K \log K)$ time, hence, the total time of Step 2 is $\lceil n/K \rceil \times O(K \log K) = O(n \log K)$. Line 10 requires $O(1)$ time since the skyline points of each subset are already sorted by

the first dimension, thus, Step 3 requires $\lceil n/K \rceil \times O(1) = O(\lceil n/K \rceil)$ time. Line 13 requires $O(\lceil n/K \rceil)$ time. Line 16 performs a binary search which requires $O(\log K)$ time. This is possible because if the skyline points of each subset are sorted in their first dimension, they are also sorted (reversely) in their second dimension. Thus, the total time of Step 5 requires $\lceil n/K \rceil \times O(\log K) = O(\lceil n/K \rceil \log K)$ time. In total, Step 3 to Step 5 require $O(\lceil n/K \rceil) + O(\lceil n/K \rceil) + O(\lceil n/K \rceil \log K)$ time. Since there are K iterations at most, the total time is

$$K \times (O(\lceil n/K \rceil) + O(\lceil n/K \rceil) + O(\lceil n/K \rceil \log K))$$

$$= O(n) + O(n) + O(n \log K) = O(n \log K)$$

□

Next, we extend Algorithm 2.1 by iteratively increasing K until all k skyline points are found. The final worst-case optimal $O(n \log k)$ time is shown in Algorithm 2.2. We note that Algorithm 2.2 is a classic paradigm for output-sensitive algorithms [24] [9].

Algorithm 2.2: O(nlogk) 2-D SKYLINE(P)

1 **Input:** a set of n points in two-dimensional space
2 **Output:** k skyline points
1: **for** $t = 1, 2, ...$ **do**
2: TEMP=$O(n \log K)$ SKYLINE(P,K), where K=min$\{2^{2^t}, n\}$.
3: **if** TEMP$\neq \varnothing$ **then**
4: **return** TEMP.
5: **end if**
6: **end for**

Theorem 2.2. Algorithm 2.2 requires $O(n \log k)$ time in worst-case.

Proof. Algorithm 2.2 stops with the list of skyline points as soon as the value of K in the for-loop reaches or exceeds k. The number of iterations in the loop

is $\lceil \log \log k \rceil$, and the t^{th} iteration takes $O(n \log K) = O(n \log 2^{2^t}) = O(n2^t)$ time. Hence, the total running time of the algorithm is

$$O(\sum_{t=1}^{\lceil \log \log k \rceil} n2^t) = O(n(2^1 + 2^2 + ... + 2^{\lceil \log \log k \rceil}))$$

$$= O(n(2^{\lceil \log \log k \rceil + 1} - 2)) = O(n \log k).$$

□

2.3 Analysis

Although the proposed algorithm has the same big O time complexity as [24], we analyze here that it has a significantly smaller constant factor and hence faster than [24]. Because we used the similar paradigm for Algorithm 2.2 with [24] and [20], we just focus on Algorithm 2.1. As noted by Luccio and Prparata [39], it is possible to find skyline by using no more than $S(n) + n$ comparisons for n points in two dimensional space, where $S(n)$ is the number of comparisons for sorting $n$numbers. If we use merge sort algorithm, then $S(n) = n \log n$. Therefore, we can compute skyline points within $n \log n + n$ comparisons. For step 2 in our algorithm, it requires $\frac{n}{k}(k \log k + k) = n \log k + n$ comparisons. For step 4, we use $\frac{n}{k}$ comparisons each time which leads to the total times for step 4 is $k \times \frac{n}{k} = n$. For step 5, we need $\log k$ comparisons to maintain the ordered structure. Hence, it requires $k \times \frac{n}{k} \log k = n \log k$ comparisons. In total, it requires $n \log k + n + n + n \log k = 2n \log k + 2n$ comparisons which is less than [24] (more than $5.430\dot{5}n \log k$ comparisons).

Chapter 3

Group-based Skyline

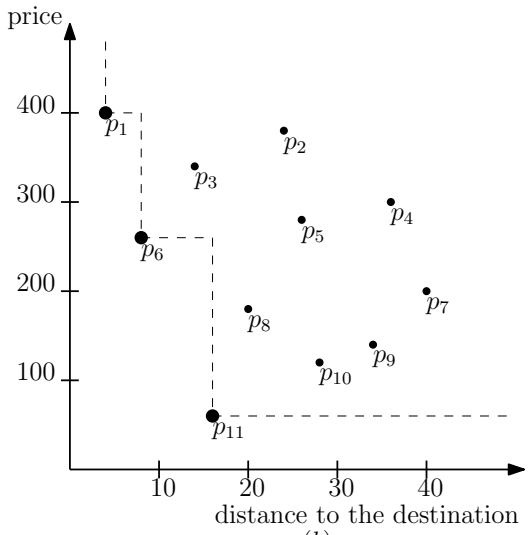
3.1 Introduction

Skyline, also known as *Maxima* in computational geometry or *Pareto* in business management field, is important for many applications involving multi-criteria decision making. The skyline of a set of multi-dimensional data points consists of the points for which no other point exists that is better in at least one dimension and at least as good in every other dimension.

Assume that we have a dataset P of n points. Each point p of d real-valued attributes can be represented as a d-dimensional point $(p[1], p[2], ..., p[d]) \in \mathbb{R}^d$ where $p[i]$ is the i-th attribute of p. Given two points $p = (p[1], p[2], ..., p[d])$ and $p' = (p'[1], p'[2], ..., p'[d])$ in $\mathbb{R}^d$, p dominates p' if for every i, $p[i] \leqslant p'[i]$ and for at least one i, $p[i] < p'[i]$ $(1 \leqslant i \leqslant d)$. Given the set of points P, the skyline is defined as the set of points that are not dominated by any other point in P. In other words, the skyline represents the *best points* or Pareto optimal solutions from the dataset since the points within the skyline cannot dominate each other.

hotel	distance	price
p_1	4	400
p_2	24	380
p_3	14	340
p_4	36	300
p_5	26	280
p_6	8	260
p_7	40	200
p_8	20	180
p_9	34	140
p_{10}	28	120
p_{11}	16	60

(a)



(b)

Figure 3.1: A skyline example of hotels.

Figure 3.1(a) illustrates a dataset $P = \{p_1, p_2, ..., p_{11}\}$, each representing a hotel with two attributes: the distance to the destination and the price. Figure 3.1(b) shows the corresponding points in the two dimensional space where the x and y coordinates correspond to the attributes of distance to the destination and price, respectively. We can see that $p_3(14, 340)$ dominates $p_2(24, 380)$ as an example of dominance. The skyline of the dataset contains p_1, p_6, and p_{11}. Suppose the organizers of a conference need to reserve *one* hotel considering both distance to the conference destination and the price for participants, the skyline offers a set of best options or Pareto optimal solutions with various tradeoffs between distance and price: p_1 is the nearest to the destination, p_{11} is the cheapest, and p_6 provides a good compromise of the two factors. p_8 will not be considered as p_{11} is better than p_8 in both factors.

Motivation. While the skyline definition has been extended with different variants and the skyline computation problem for finding the skyline of a given dataset has been studied extensively in recent years, most existing works focus

on skyline consisting of *individual* points. One important problem that has been surprisingly neglected to the large extent is the need to find groups of points that are not dominated by others as many real-world applications may require the selection of *a group of* points.

Hotels Example. Consider our hotel example again, suppose the organizers need to reserve *a group of* hotels (instead of one) considering both distance to the conference destination and the price for participants. In contrast to the traditional skyline problem which finds Pareto optimal solutions where each solution is a single point, we are interested in finding Pareto optimal solutions where each solution is a group of points. One may use the traditional skyline definition, and return all subsets from the skyline points p_1, p_6, and p_{11}. If the desired group size is 2, group $\{p_1, p_6\}$, $\{p_1, p_{11}\}$, and $\{p_6, p_{11}\}$ can be returned. However, we show that this definition does not capture all the best groups. For example, $\{p_{11}, p_{10}\}$ should clearly be considered a Pareto optimal group to users who use price as the main criterion, e.g. Ph.D. students with low travel budget, since p_{11} provides the best price and p_{10} the second best price. Note that p_{10} is only second best to p_{11} which is also part of the group, hence no other groups are better than this group in terms of price. As another example, $\{p_6, p_3\}$ also presents a Pareto optimal group, as both p_6 and p_3 provide a good tradeoff and no other groups are better than this group considering both price and distance. On the other hand, group $\{p_3, p_8\}$ is not a best group because $p_{11}(p_6)$ is better than $p_8(p_3)$, i.e., group $\{p_3, p_8\}$ is dominated by group $\{p_6, p_{11}\}$.

NBA Example. Consider another real example with NBA players. Table 1 shows the top five players on attribute PTS (Points). For other attributes, please see the experimental section 3.6 for detailed explanations. Suppose the coaches of NBA teams need to choose five players to compose a team. While

Table 3.1: Top five players on Attribute PTS.

Player	PTS	REB	AST	STL	BLK
Michael Jordan	33.4	6.4	5.7	2.1	0.9
Anthony Davis	30.5	8.5	2	1.5	3
Kyrie Irving	30	3	2	1	1
Allen Iverson	29.7	3.8	6	2.1	0.2
Jerry West	29.1	5.6	6.3	0	0
...	...	...	...	...	...

the traditional skyline will compute *best players* that are not dominated by other players, we need to compute *best teams* that are not dominated by other teams. For example, a coach may prefer PTS as the main selection criteria when building a team in order to maximize the overall points that can be earned by the team. In this case, the top five players on PTS, {Michael Jordan, Anthony Davis, Kyrie Irving, Allen Iverson, Jerry West}, should be considered a best team. However, if we only build groups from skyline players, this team will not be captured because Kyrie Irving is not a skyline player being dominated by Anthony Davis. In essence, taking only skyline players will not capture those teams which may include non-skyline players who are only dominated by another player in the team but are not dominated by any other players outside the team. In summary, we argue that there is a need to define a group skyline notion for group-based decision making such that we can find Pareto optimal groups.

Contributions. In this chapter, we formally define a novel group-based skyline, *G-Skyline*, for finding Pareto optimal groups. In order to find the best groups, i.e., groups not dominated by other groups, we will first define the dominance relationship between groups, group dominance. Given two different groups G and G' with k points, we say G g-dominates G', if for any point p'_i in G', we can find a distinct point p_i in G, such that p_i dominates p'_i or $p_i = p'_i$,

and for at least one i, p_i dominates p'_i. The G-Skyline are those groups that are not g-dominated by any other group with same size. Intuitively, if we consider the points in each group as a set of dimensions orthogonal to the attributes of each point, the definition of G-Skyline groups with the group dominance is in spirit similar to skyline definition, in that a group is a skyline group if no permutation of any other group exists that is better for at least one point and at least as good for every other point.

G-Skyline not only captures groups of points from traditional skyline points but also groups that may contain non-skyline points. Back to our hotel example, $\{p_{11}, p_{10}\}$ is a G-Skyline group as we discussed earlier even though p_{10} is not a skyline point. Group $\{p_6, p_3\}$ is also a G-Skyline. On the other hand, group $\{p_1, p_3\}$ is not as it is dominated by $\{p_1, p_6\}$. Group $\{p_3, p_8\}$ is also not as it is dominated by $\{p_6, p_{11}\}$. In summary, the G-Skyline in this example consist of all groups composed of skyline points, $\{p_1, p_6\}$, $\{p_1, p_{11}\}$, $\{p_6, p_{11}\}$, as well as groups that contain non-skyline points, $\{p_6, p_3\}$, $\{p_{11}, p_8\}$, and $\{p_{11}, p_{10}\}$.

It's non-trivial to solve G-Skyline problem efficiently. To find k-point G-Skyline groups from n points, there can be $\binom{n}{k}$ different possible groups. Unfortunately, the G-Skyline problem is significantly different from the traditional skyline problem, to the extent that algorithms for the latter are inapplicable. A brute force solution is to enumerate all $\binom{n}{k}$ possible groups, then for each group, to compare it with all other groups to determine whether it cannot be dominated. So there are $\binom{n}{k}^2$ comparisons. For each comparison, there are $k!$ possible permutations of the points, and for each permutation, it requires k comparisons. Therefore, the time complexity is in the order of $O(\binom{n}{k}^2 \times k! \times k)$.

In this chapter, we present a novel structure that represents the points in a directed skyline graph and captures all the dominance relationship among the

points based on the notion of *skyline layers*. Using the directed skyline graph, the G-Skyline problem can be formulated as the classic search problem in a set enumeration tree. We exploit the properties of G-Skyline groups and propose two algorithms with efficient pruning strategies to compute G-Skyline groups.

We briefly summarize our contributions as follows.

- For the first time, we generalize the original skyline definition (for individual points) to permutation group-based skyline (for groups) which is useful for finding Pareto optimal groups in practical applications.

- We present a novel structure for finding k-point G-Skyline groups by representing the points as a directed skyline graph based on the first k skyline layers. This directed skyline graph is significant as we show that we only need the points in the first k skyline layers (k is far less than n in the usual case) rather than the entire n points to compute k-point G-Skyline. We design efficient algorithms for computing the first k skyline layers with time complexity $O(n \log n)$ for two- and $O(n \log n + n\mathbb{S}_k)$ for higher-dimensional spaces, where $\mathbb{S}_k$ is the number of points in the first k skyline layers. This can be also of independent value and used as a preprocessing step for other skyline algorithms.

- Given the directed skyline graph, we present two efficient algorithms: the point-wise and the unit group-wise algorithms, to efficiently compute G-Skyline groups. We introduce a novel notion of unit-group for each point which represents the minimum number of points that have to be included with the point in a G-Skyline. Both algorithms employ efficient pruning strategies exploiting G-Skyline properties.

- We conduct comprehensive experiments on real and synthetic datasets.

The experimental results show that G-Skyline is interesting and useful, and our proposed algorithms are efficient and scalable.

- We also briefly discuss two variants of G-Skyline definition: One is AG-Skyline based on a more restrictive all-permutation group dominance. The other one is PG-Skyline based on a less restrictive partial group dominance, which can address the potential problem of large number of output groups of G-Skyline.

Organization. The rest of the chapter is organized as follows. Section 3.2 presents the related work. Section 3.3 introduces our G-skyline definitions as well as their properties. The algorithms of constructing directed skyline graph are shown in Section 3.4. Two algorithms for finding G-Skyline groups based on the directed skyline graph are discussed in Section 3.5. We report the experimental results and findings for performance evaluation in Section 3.6. Section 3.7 discusses two extensions to our work. Section 3.8 concludes the chapter.

3.2 Related Work

The problem of computing skyline (Maxima) is a fundamental problem in computational geometry field because the skyline is an interesting characterization of the boundary of a set of points. The skyline computation problem was firstly studied in computational geometry [26] which focused on worst-case time complexity. [24,35] proposed output-sensitive algorithms achieving $O(n \log v)$ in the worst-case where v is the number of skyline points which is far less than n in general. Several works [3,4,7,14] in both computational geometry and database

fields focused on how to achieve the best average-case time complexity. For a detailed survey both for worst-case and average-case, please see [19].

Since the introduction of the skyline operator by Börzsönyi et al. [7], skyline has been extensively studied in the database field. Many algorithms are proposed in the context of relational query engine and external memory model, for example, [19, 49]. Based on the traditional skyline definition, [1, 25] studied the parallel algorithms for skyline.

Many works also studied extensions or variants of the classical skyline definitions. Papadias et al. [42] studied *group-by skyline* which groups the objects based on their values in one dimension and then computes the skyline for each group, and *k-skyband* which computes objects dominated by at most k objects (the case $k = 0$ corresponds to the conventional skyline) based on individual dominance relationship. Skyline in subspace, i.e., a subset of the dimensions or points, was studied in [8, 45, 46, 51]. [17, 52] discussed the reverse skyline problem which is similar to the reverse k-nearest neighbor problem. [16] presented skyline-based statistical descriptors for capturing the distributions over pairs of dimensions. Some works defined and studied the skyline on different data types/domains. For example, [47] and [13] studied the spatial skyline and a more general metric skyline, respectively. [21] proposed the skyline for moving objects. [18, 29, 37, 44, 56] studied the skyline problem for uncertain data.

The most related works to our group-based skyline are [22, 28, 40, 55]. [22,28,55] formulated and investigated the problem of computing skyline groups. However, the notion of dominance between groups in these works is defined by the dominance relationship between an "aggregate" or "representative" point of each group. More specifically, they calculate for each group a single aggregate point, whose attribute values are aggregated over the corresponding

attribute values of all points in the group. The groups are then compared by their aggregate points using traditional point dominance. While many aggregate functions can be considered in calculating aggregate points, they focus on several functions commonly used in database applications, such as, SUM, MIN, and MAX. In addition to the fact that it is difficult to choose a good or meaningful function, more importantly, it will not capture all the Pareto optimal groups. This is essentially similar to the multi-objective optimization or multi-attribute skyline problem where an aggregate function, such as weighted average, can be used to combine the multiple criteria to find a single optimal solution, but it also fails to capture all the Pareto optimal solutions.

In fact, the result of skyline groups under SUM dominance [22, 28, 55] is a subset of our G-Skyline groups. If group G is dominated by G' in G-Skyline definition, then G must be dominated by G' under SUM function, but not vice versa. Consider our hotel example, group $\{p_1, p_{11}\}$ dominates $\{p_3, p_6\}$ based on SUM function. However, we cannot conclude group $\{p_1, p_{11}\}$ is better than $\{p_3, p_6\}$ because the assumption of skyline is that we do not know users' attribute weights in advance. For users who consider both distance and price in their selection criteria, they may prefer group $\{p_3, p_6\}$, because $\{p_3, p_6\}$ provides a good compromise of distance and price, and neither p_1 or p_{11} can dominate p_3 or p_6. Hence, some Pareto optimal solutions are not captured by SUM dominance.

The work in [40] also defines a group dominance notion. However, their definition is based on the uncertain skyline definition by Pei et al. [44]. In our work, we define a deterministic dominance relationship between two groups in order to find "optimal" groups of objects.

3.3 G-Skyline Definitions

In this section, we introduce our G-Skyline definition and related concepts as well as their properties which will be used in our algorithm design. For reference, a summary of notations is given in Table 2.

Table 3.2: The summary of notations.

Notation	Definition
$P \backslash layer_i$	points in P but not in $layer_i$
$p \preceq p'$	p dominates or equals to p'
G-Skyline(i)	G-Skyline group with i points
$p_i.layer$	the skyline layer of p_i
$\|S\|_{p(u)}$	the point(unit group) size of set S

Definition 3.1. (**Skyline**). Given a dataset P of n points in d-dimensional space. Let p and p' be two different points in P, p dominates p', denoted by $p \prec p'$, if for all i, $p[i] \leqslant p'[i]$, and for at least one i, $p[i] < p'[i]$, where $p[i]$ is the i^{th} dimension of p and $1 \leqslant i \leqslant d$. The skyline points are those points that are not dominated by any other point in P.

G-Skyline. The key of skyline is that it consists of all the "best" points that are not dominated by other points. While a linear weighted sum function can be used to combine all the attribute values of each point as a scoring function to find the best points, the relative preferences (weights) for different attributes are not known in advance. The skyline essentially covers all the best points on all linear functions. Following this notion, in order to find all the "best" groups of points that are not dominated by other groups, we introduce group dominance definition as follows.

Definition 3.2. (**Group Dominance**). Given a dataset P of n points in a d-dimensional space. Let $G = \{p_1, p_2, ..., p_k\}$ and $G' = \{p'_1, p'_2, ..., p'_k\}$ be two

different groups with k points of P, we say group G **g-dominates** group G', denoted by $G \prec_g G'$, if we can find two permutations of the k points for G and G', $G = \{p_{u_1}, p_{u_2}, ..., p_{u_k}\}$ and $G' = \{p'_{v_1}, p'_{v_2}, ..., p'_{v_k}\}$, such that $p_{u_i} \preceq p'_{v_i}$ for all i $(1 \leqslant i \leqslant k)$ and $p_{u_i} \prec p'_{v_i}$ for at least one i.

Given the group dominance definition, we define group-based skyline, G-Skyline, as follows.

Definition 3.3. (**G-Skyline**). The k-point G-Skyline consists of those groups with k points that are not g-dominated by any other group with same size.

Example 3.1. Consider the dataset in Figure 3.1 and $k = 3$. For group $G = \{p_8, p_{10}, p_{11}\}$ and group $G' = \{p_4, p_5, p_7\}$, G g-dominates G' because we can find two permutations, $G = \{p_8, p_{10}, p_{11}\}$ and $G' = \{p_5, p_4, p_7\}$ such that $p_8 \prec p_5$, $p_{10} \prec p_4$, and $p_{11} \prec p_7$. Therefore, $G' = \{p_4, p_5, p_7\}$ is not a G-Skyline group. G is one of the G-Skyline groups as no other group with 3 points can g-dominate G.

Next, we present a few properties of G-Skyline groups and related concepts that will be used in our algorithm design for computing G-Skyline groups.

Property 3.1. (**Asymmetry**). Give two groups G and G' with same size. If $G \prec_g G'$, then $G' \not\prec_g G$.

Property 3.2. (**Transitivity**). Given three groups G_1, G_2, and G_3 with same size. If $G_1 \prec_g G_2$ and $G_2 \prec_g G_3$, then $G_1 \prec_g G_3$.

Lemma 3.1. A point in a G-Skyline group cannot be dominated by a point outside the group.

Proof. By contradiction, assume a point p_i in a G-Skyline group G is dominated by a point p_j outside the group. If we use p_j to replace p_i in G, the new group

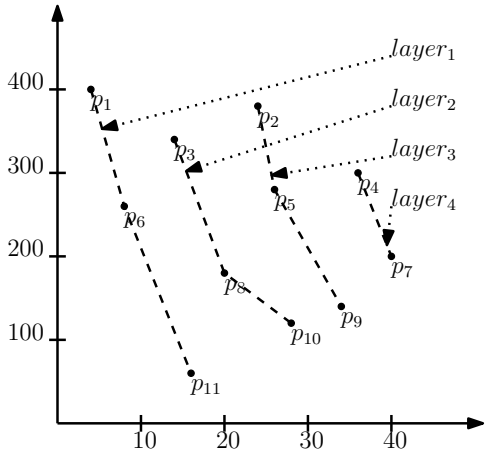


Figure 3.2: Skyline layers.

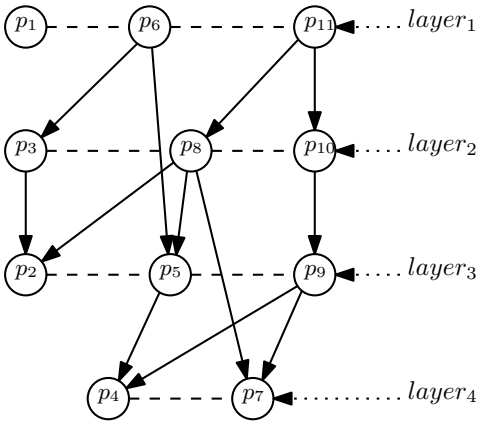


Figure 3.3: Directed skyline graph.

will g-dominate G since p_j dominates p_i and all the other points are the same, which contradicts the G-Skyline definition. □

Skyline Layers. Motivated by Lemma 3.1, we present a structure representing the points and their dominance relationships based on the notion of skyline layers. A formal definition is presented as follows.

Definition 3.4. (**Skyline Layers**). Given a dataset P of n points in a d-dimensional space. The set of skyline layer $layer_1$ contains the skyline points of P, i.e., $layer_1 = skyline(P)$. The set of $layer_2$ contains the skyline points of $P \backslash layer_1$, i.e., $layer_2 = skyline(P \backslash layer_1)$. Generally, the set of $layer_j$ contains the skyline points of $P \backslash \bigcup_{i=1}^{j-1} layer_i$, i.e., $layer_j = skyline(P \backslash \bigcup_{i=1}^{j-1} layer_i)$. The above process is repeated iteratively until $P \backslash \bigcup_{i=1}^{j-1} layer_i = \varnothing$.

An example of skyline layers of Figure 3.1 is shown in Figure 4.5. It is easy to see from Definition 3.4 that for a point p, if there is no point in $layer_{i-1}$ that can dominate p, p should be in $layer_{i-1}$ or a lower layer.

Property 3.3. For a point p in $layer_i$, where $2 \leqslant i \leqslant l$ and l is the maximum layer number, there must be at least one point in $layer_j$ $(1 \leqslant j \leqslant i-1)$ that

dominates p.

We then make an observation that in order to compute k-point G-Skyline groups, we only need to examine the points from the first k skyline layers. We formally present the theorem below.

Theorem 3.1. If a group $G = \{p_1, p_2, ..., p_k\}$ is a k-point G-Skyline group, then all points in G belong to the first k skyline layers.

Proof. We prove by contradiction. Assume a point of G is from the j^{th} skyline layer where $j \geqslant (k+1)$. Without loss of generality, we assume this point is p_k. Since there are k points in G, there is at least one layer $layer_i, 1 \leqslant i \leqslant k$, that has no point in G. According to Property 3.3, p_k should be dominated by at least one point in $layer_i$, denoted by t. Then it is easy to see that group $G' = \{p_1, p_2, .., p_{k-1}, t\}$ by replacing p_k in G with t can dominate group $G = \{p_1, p_2, ..., p_{k-1}, p_k\}$. Then group G is not a G-Skyline group which is a contradiction. □

We note that the skyline layers are closely related to the k-skyband [42] we discussed in the related work. In fact, $(k-1)$-skyband is the subset of the points in the first k skyline layers. We can alternatively compute k-point G-Skyline groups from the $(k-1)$-skyband, as versus the entire set of n points. The reason we use skyline layers instead of skyband in our preprocessing is that we can design efficient algorithms to compute the skyline layers as we will show in Section 3.4 and we can leverage the layers to compute skyline groups efficiently as we show in Section 3.5.

Directed Skyline Graph. We now present a definition of directed skyline graph, a data structure we use to represent the points from the first k skyline

layers as well as their dominance relationships, in order to compute k-point G-Skyline groups.

Definition 3.5. (**Directed Skyline Graph (DSG)**). A directed skyline graph is a graph where a node represents a point and an edge represents a dominance relationship. Each node has a structure as follows.

$$[layer\ index, point\ index, parents, children]$$

where layer index ranging from 1 to k indicates the skyline layer that the point lies on, point index ranging from 0 to $\mathbb{S}_k - 1$ uniquely identifies the point and $\mathbb{S}_k$ is the number of points in the first k skyline layers, parents include all the points that dominate this point, and children include all the points that are dominated by the point.

Example 3.2. Figure 4.6 shows the DSG corresponding to the skyline layers in Figure 2. Note that $p_6 \prec p_5$ and $p_5 \prec p_4$ imply $p_6 \prec p_4$. For visualization clarity, we omit all indirect dominance edges such as $p_6 \prec p_4$.

Lemma 3.2. Given a point p, if p is in a G-Skyline group, p's parents must be included in this G-Skyline group.

Proof. By Lemma 3.1, a point p in a G-Skyline group cannot be dominated by a point outside the group, i.e., all the points that dominate p must be included in this G-Skyline group. □

Verification of G-Skyline. Motivated by Lemma 3.2, we define a concept of Unit Group and then formally state a theorem for verifying whether a group is a G-Skyline group based on unit group.

Definition 3.6. (**Unit Group**). Given a point p in DSG, p and its parents form the unit group for p.

Example 3.3. The unit group of p_5 in Figure 4.6, denoted by u_5, contains p_5 and its parents p_6, p_{11}, p_8. Thus, $u_5 = \{p_6, p_{11}, p_8, p_5\}$.

Based on Lemma 3.2, we have the verification of G-Skyline theorem as follows.

Theorem 3.2. (**Verification of G-Skyline**). Given a group $G = \{p_1, p_2, ..., p_k\}$, it is a G-Skyline group, if its corresponding unit group set $S = u_1 \cup u_2 \cup ... \cup u_k$ contains k points, i.e., $|S|_p = k$.

This theorem is significant because given a group G, in order to check whether it is a G-Skyline group, we do not need to compare G with all other candidate groups any more. Instead, we only need to check whether its corresponding unit group set S has k points.

3.4 Constructing Directed Skyline Graph

In this section, we first present our algorithms for computing the first k skyline layers in two- and higher-dimensional space, and then briefly discuss how to construct the DSG based on skyline layers. In next section, we will present our algorithms for finding G-Skyline groups using DSG.

A straightforward way of computing skyline layers is to iteratively compute (and remove) the skyline points of each layer using any $O(n \log n)$ time complexity skyline algorithms [26]. Since a dataset may exhibit a linear number of layers, this leads to an $O(n^2 \log n)$ worst-case running time. Existing work proposed space-efficient algorithms for computing all skyline layers simultaneously

with $O(n \log n)$ time complexity for two dimensions [6] and $O(n^2)$ for higher dimensions [38]. In this chapter, we present an efficient $O(n \log n)$ algorithm for two dimensional space based on the ideas briefly mentioned in [6] (they did not provide any algorithm details as their focus is on designing in-place algorithms). In addition, since we only need the first k skyline layers for computing k-point G-Skyline groups, as we have shown in Theorem 1, we present more efficient output-sensitive algorithms with $O(n + \mathbb{S}_k \log k)$ time complexity for two- and $O(n\mathbb{S}_k)$ for higher-dimensional space after the points are sorted.

Computing Skyline Layers for Two Dimensions. For two dimensional space, the main intuition of the algorithm is motivated by the monotonic property of skyline points in two dimensional space, that is, if we sort the skyline points with increasing x-coordinate, their y-coordinates decrease monotonically, since they cannot dominate each other. This applies to each skyline layer as shown in Figure 4.5. We refer to the point with minimum y-coordinate in $layer_i$ as the *tail point* of $layer_i$. We derive the following two properties for skyline layers in a two dimensional space which will motivate our algorithm design.

Property 3.4. Given a skyline layer $layer_i$ with its tail point denoted as p_{layer_i}, and a point p, if $p[x] \geqslant p_{layer_i}[x]$ and p is not dominated by p_{layer_i}, then p cannot be dominated by any other point in $layer_i$.

Proof. For a point p with $p[x] \geqslant p_{layer_i}[x]$, if it is not dominated by p_{layer_i}, then we have $p[y] < p_{layer_i}[y]$. Because p_{layer_i} has the smallest value on y-coordinate in $layer_i$, then all other points in $layer_i$ cannot dominate p. □

Property 3.5. Given l layers for the n points in P with their tail points denoted as p_{layer_1}, p_{layer_2}, ..., p_{layer_l}, the y-coordinates of those points are in

ascending order, i.e., $p_{layer_1}[y] \leqslant p_{layer_2}[y] \leqslant ... \leqslant p_{layer_i}[y]$.

Proof. We prove by contradiction. Suppose $p_{layer_i}[y] < p_{layer_{i-1}}[y]$. Since $p_{layer_{i-1}}$ is the tail point in $layer_{i-1}$, we know that the y-coordinates of all the points in $layer_{i-1}$ are larger than $p_{layer_{i-1}}[y]$, hence no points of $layer_{i-1}$ can dominate the tail point of $layer_i$. This is contradictory to the skyline layer definition. □

As an example for Property 3.4 in Figure 4.5, we can see that $layer_1$ has tail point p_{11}, given a point p with x-coordinate greater than p_{11}, if p is not dominated by p_{11}, then p cannot be dominated by p_1 and p_6. For Property 3.5, we can see that the tail points p_{11}, p_{10}, p_9, and p_7 are in ascending order on their y-coordinates.

Algorithm 3.1: Skyline layers algorithm in two-Ds.

input : a set of n points in two dimensional space.
output: l skyline layers.

1 If the points are not sorted already, sort the n points on the first dimension in ascending order $P = \{p_{u_1}, p_{u_2}, ..., p_{u_n}\}$;
2 $p_{u_1}.layer = 1$;
3 $maxlayer = 1$;
4 tail point of $layer_1 = p_{u_1}$;
5 **for** *i = 2 to n* **do**
6 **if** *the tail point of* $layer_1$ *cannot dominate* p_{u_i} **then**
7 $p_{u_i}.layer = 1$;
8 tail point of $layer_1 = p_{u_i}$;
9 **else if** *the tail point of* $layer_{maxlayer}$ *dominate* p_{u_i} **then**
10 $p_{u_i}.layer = ++maxlayer$;
11 tail point of $layer_{maxlayer} = p_{u_i}$;
12 **else**
13 use binary search to find $layer_j$ $(1 < j \leqslant maxlayer)$ such that the tail point of $layer_j$ cannot dominate p_{u_i} and the tail point of $layer_{j-1}$ dominates p_{u_i};
14 $p_{u_i}.layer = j$;
15 tail point of $layer_j = p_{u_i}$;

Based on the above properties, our key idea of the algorithm, shown in Algorithm 3.1, is to sort all the points in ascending x-coordinates if they are not sorted already and process them in that sequence by either adding them into an existing layer it belongs to or starting a new layer. Line 2-4 adds the first point (with minimum x value) into the first skyline layer. For each new point p_{u_i}, its x-coordinate is larger than all the points in existing layers. Based on Property 3.4, we only need to compare point p_{u_i} with the (current) tail point of each layer to determine if it cannot be dominated by any of the points in that layer. Based on Property 3.5, the tail points are sorted by y-coordinates in ascending order so we can perform a binary search to quickly find the layer that the point belongs to. If the tail point of the first layer cannot dominate p_{u_i} (line 6), we insert the point to the first layer. If the tail point of the last layer dominates p_{u_i} (line 9), we add a new layer for the point. Line 13 performs such a binary search and finds $layer_j$ to insert the point, i.e., the tail point of $layer_j$ cannot dominate p_{u_i} but the tail point of $layer_{j-1}$ dominates p_{u_i}. Once the point is inserted, it becomes the new tail point of that layer.

Since we just need the points in the first k skyline layers (Theorem 3.1), we can slightly modify the algorithm as follows. Once the k^{th} layer is established, for each of the remaining points p, we can compare p with the tail point of the k^{th} layer. If p is dominated by it, it means p lies outside the first k layers, and we can drop p directly. If not, we can then use binary search on the first k layers.

Running time. For each point in the first k skyline layers, we need at most $O(\log k)$ time to determine its layer because we only need to maintain k layers. This part costs $O(\mathbb{S}_k \log k)$. For those points not in the first k layers, we only need to compare it with the tail point of the k^{th} layer. Therefore, the algorithm

requires $O(n + \mathbb{S}_k \log k)$ time in total for computing the first k skyline layers in two dimensional space after the points are sorted in one dimension.

Higher dimensional space. For a higher dimensional space, we can use an algorithm similar to Algorithm 3.1. It processes each point in order and finds an existing skyline layer to insert it or starts a new layer. However, the difference is that Properties 3.4 and 3.5 do not hold for the higher dimensional case anymore. So in order to find an existing skyline layer the point belongs to, we need to compare the point with all existing points, as versus only the tail points in each layer in two-dimensional case. For each point, we need at most $O(\mathbb{S}_k)$ time to determine its position because there are at most $\mathbb{S}_k$ points in the first k layers. Therefore, the algorithm for computing the first k skyline layers in higher dimensional space requires $O(n\mathbb{S}_k)$ time after the data are sorted in one dimension.

Constructing Directed Skyline Graph. Once we build the skyline layers, we can build a DSG to capture all the dominance relationships between the layers which will then be used to compute G-Skyline groups. Building DSG using the skyline layers is straightforward: the points are processed in the order of their skyline layers. For each point p_i, we scan all points in the previous layers and find those points that dominate p_i, add p_i to their children list, and add those points that dominate p_i as p_i's parents. It is easy to see such an algorithm takes $O({\mathbb{S}_k}^2)$ time.

3.5 Finding G-Skyline Groups

In this section, we present our algorithms for efficiently finding G-Skyline groups given the DSG built from the first k skyline layers. We first present a point-

wise algorithm which builds G-Skyline groups from points (adding one point at a time), then present a unit group-wise algorithm which builds G-Skyline groups from unit groups (adding one unit group at a time). Before beginning the discussion of the two algorithms, we first show a preprocessing step similar to that in [22, 28, 55] to further prune points from the first k skyline layers.

Preprocessing. Theorem 3.2 shows that a k-point group is a G-Skyline group, if for each point p_i in the group, its parents are also in the group, i.e., the unit group u_i is a subset of the group. Therefore, for a point p_i, if the point size of its unit group is greater than k, i.e., $|u_i|_p > k$, it will not be in any k-point G-Skyline group, and we can remove p_i directly from the DSG without having to consider it. If $|u_i|_p = k$, we can output u_i as one of the G-Skyline groups, and p_i will not be considered either as it will not contribute to any other G-Skyline groups.

Example 3.4. If we set $k = 4$ (we will use $k = 4$ in all the remaining examples of the chapter), the node p_2, p_4, p_7 in Figure 4.6 can be removed directly because $|u_2|_p = 5, |u_4|_p = 7, |u_7|_p = 5$. Unit group $u_5 = \{p_6, p_{11}, p_8, p_5\}$ can be output as a G-Skyline group. As a result, p_2, p_5, p_4, p_7 will not be considered in our algorithms.

3.5.1 The Point-Wise Algorithm

The problem of finding G-Skyline groups can be tackled by the classic set enumeration tree search framework. The idea is to expand possible groups over an ordered list of points as illustrated in Figure 4.12. Each node in the set enumeration tree is a *candidate* group. The first level contains the root node which is the empty set, while the ith level contains all i-point groups. Naively,

we can enumerate all $\binom{\mathbb{S}_k}{k}$ candidates and employ Theorem 3.2 to check each candidate. However, this baseline method is too time-consuming which will be verified in our experiments.

The main idea of our algorithm is to dynamically generate the set enumeration tree of candidate groups one level at a time while pruning the non-G-Skyline candidates as much as possible without having to check them. For each node, we store a tail set that consists of the points with point index larger than the points in current node. Each node can be expanded to create a set of new nodes at the next level, each by adding a new point from its tail set. The root node contains an empty set with a tail set composed of all remaining points from the first k skyline layers after the preprocessing. We present our tree expansion and pruning strategies in detail below.

Subtree Pruning. We observe a property of superset monotonicity which allows us to do subtree pruning when a node is not a G-Skyline group.

Theorem 3.3. (**Superset Monotonicity**). If a group G_i with i points is not a G-Skyline group, by adding a new point from its tail set, the new group G_{i+1} with $i+1$ points is not a G-Skyline group either.

Proof. If G_i is not a G-Skyline group, there is a group G_i' with i points that dominates G_i. For any superset of G_i with a new point p_j added from G_i's tail set, we denote it by $G_i \cup p_j$. Since the points are ordered by skyline layers in our DSG, and any point in G_i's tail set has a larger point index than the points in G_i, hence p_j will not dominate any points in G_i, and will not be in G_i'. It is then easy to see that the group $G_i' \cup p_j$ dominates $G_i \cup p_j$, i.e., $G_i' \cup p_j \prec G_i \cup p_j$. □

This theorem implies that if a candidate group G is not a G-Skyline group,

we do not need to expand it further or check its subtree.

Tail Set Pruning. Each node in our tree can be expanded to a set of new nodes by adding a point from its tail set. However, not all tail points need to be considered. Recall Lemma 3.1, a point in a G-Skyline group cannot be dominated by a point outside the group. In other words, if a point p is to be added to a group G to form a new group that is a G-Skyline group, its parents must be in the group already. This means that p must be either a skyline point (with no parent), or a child of some points in G. Note this is a necessary condition but not sufficient. Once the candidate group is built, we still need to check whether p's parents are all in G (Theorem 3.2). However, this necessary condition allows us to quickly prune those points from the tail set of G that are not skyline points or not a child of points in G. In addition, given a candidate group G, if all its points are in the first i skyline layers, p must come from the first $i+1$ skyline layers. Otherwise, we can find a point outside G that lies on the $(i+1)^{th}$ layer to dominate p. Hence we can also prune the points beyond the $(i+1)^{th}$ layer. Once a point is pruned from the tail set, it will not be used to expand the current node. This is because based on the superset monotonicity, any node in the subtree of the new node will not be a G-Skyline group either.

Algorithm. Given the above two pruning strategies, we show our complete point-wise algorithm in Algorithm 2.2. Line 1 initializes the root node and its tail set. Tail set pruning is implemented in Line 4 to Line 10. For each node G_j at level i, it is expanded with a new point p from its tail set to form a new candidate group only if p is a child of some points in the current node or a skyline point and p is in the first $i+1$ layers. The for loop in Line 4 and Line 6 can be finished in linear time $O(|CS|)$ and $O(|CS|+|TS|)$, respectively, by

Algorithm 3.2: The point-wise algorithm for computing G-Skyline groups.

input : a DSG and group size k.
output: G-Skyline(k) groups.

```
1  initialize the G-Skyline(0) group at root node as an empty set and its
   tail set as all points from DSG after preprocessing;
2  for i=1 to k do
3      for each G-Skyline(i-1) group G do
4          for each point p_l in G do
5              add p_l's children to Children Set CS;
6          for each point p_j in Tail Set(TS) of G do
7              if p_j is not in CS && p_j is not a skyline point then
8                  delete p_j;
9              if p_j.layer − max_l{p_l.layer} ⩾ 2 then
10                 delete p_j;
11         for each remaining point p in tail set of G do
12             add p to G to form a new candidate G-Skyline(i) group;
13             if the new candidate group is not a G-Skyline group then
14                 delete;
```

employing the idea of merge sort, where $|CS|$ and $|TS|$ are the size of children set and tail set. The candidate group is verified in Line 13. If it is not a G-Skyline group, it will be pruned from the tree (subtree pruning).

Example 3.5. We show a running example of Algorithm 2.2 in Figure 4.12 based on Figure 4.6. The root node at level $|S|_p = 0$ has an initial tail set $\{p_1, p_6, p_{11}, p_3, p_8, p_{10}, p_9\}$. Points p_3, p_8, p_{10}, p_9 can be pruned immediately because they are not skyline points, i.e., their parents are not in the root node. The remaining points p_1, p_6, p_{11} are used to create the new nodes at level $|S|_p = 1$. Similarly, for node $\{p_1, p_{11}\}$ at level $|S|_p = 2$, its tail set is $\{p_3, p_8, p_{10}, p_9\}$. Point p_3 can be pruned because p_3 is not a child of either p_1 or p_{11}. Point p_9 also can be pruned as $p_9.layer - max\{p_1.layer, p_{11}.layer\} = 2$. Hence, the remaining points p_8, p_{10} are used to create the new candidate group-

s at level $|S|_p = 3$, namely $\{p_1, p_{11}, p_8\}$ and $\{p_1, p_{11}, p_{10}\}$. As a result, level $|S|_p = 4$ shows all candidate 4-point groups that need to be checked. After checking, the ones with blue slash are not G-Skyline groups while the remaining ones are G-Skyline groups. Based on our pruning strategies, only 31 candidate groups are generated and checked while the baseline approach needs to enumerate and check $\binom{8}{4} = 70$ candidate groups.

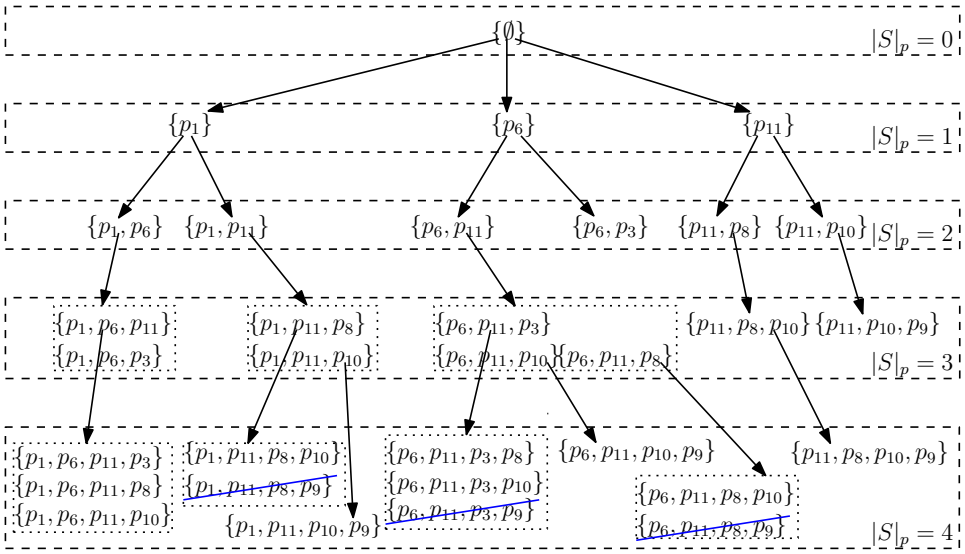


Figure 3.4: The point-wise algorithm for finding G-Skyline groups when $k = 4$.

3.5.2 The Unit Group-Wise Algorithm

The point-wise algorithm expands candidate groups one point at a time. We already showed (in Lemma 3.1) that a point in a G-Skyline group cannot be dominated by a point outside the group. In other words, for a point in a G-Skyline group, its unit group must be in the group. This motivates our unit group-wise algorithm which expands candidate groups by unit groups, adding one unit group at a time. Similar to the point-wise algorithm, we can represent the entire search space of candidate groups as a set enumeration tree, where

each node is a set of unit groups. We can dynamically generate the tree while pruning as much non-G-Skyline groups as possible. We can use similar pruning strategies as in the point-wise algorithm.

Superset Pruning.

Given a candidate group G with at least k points, the candidate groups in G's subtree will have more than k points. Hence, we can prune G's subtree directly.

Example 3.6. Figure 3.5 shows the dynamically generated set enumeration tree for the unit group-wise algorithm. At level $|S|_u = 2$, $|u_3 \cup u_8|_p = 4$, hence we do not need to check the candidate groups in its subtree, e.g., $u_3 \cup u_8 \cup u_{10}$. This is because $|u_3 \cup u_8 \cup u_{10}|_p = |\{p_6, p_{11}, p_3, p_8, p_{10}\}|_p = 5$.

Tail Set Pruning.

For a candidate group G at Level $|S|_u = i$, we do not need to add the children of the unit groups in G to form a new candidate group at Level $|S|_u = i + 1$. Therefore, the children of the unit groups in G can be pruned from the tail set.

Example 3.7. We show a running example for the unit group-wise algorithm with the two pruning strategies in Figure 3.5. The number on each candidate group represents the number of points in this candidate group. All candidate groups with one unit group are checked at Level $|S|_u = 1$. For candidate group u_6 at Level $|S|_u = 1$, its tail set is $\{u_{11}, u_3, u_8, u_{10}, u_9\}$. However, u_3 can be pruned by Tail Set Pruning because u_3 is the child of u_6. From Level $|S|_u = 2$ to Level $|S|_u = 3$, candidate group $u_3 \cup u_8$'s subtree does not need to be checked since $|u_3 \cup u_8|_p = 4$, thanks to Superset Pruning. Based on the two pruning strategies, only 35 candidate groups need to be checked as shown.

The resulting G-Skyline(4) groups are shown in red (solid) boxes and we can see that they come from different levels while the point-wise algorithm outputs all G-Skyline(4) groups at Level $|S|_p = 4$.

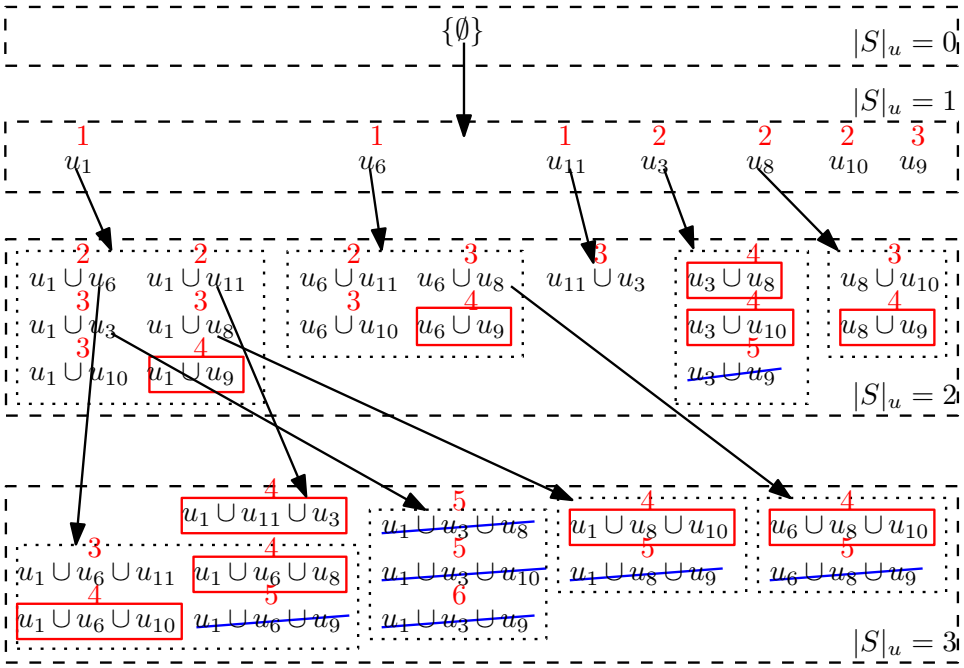


Figure 3.5: The basic unit group-wise algorithm for finding G-Skyline groups when $k = 4$.

In addition to the above strategies, we show a few additional refinements that further improve the algorithm.

Unit Group Reordering.

We observe that Superset Pruning is important to reduce the candidate groups. This motivates us to reorder the unit groups in order to increase the effectiveness of Superset Pruning. Recall that Superset Pruning can be applied when a candidate group G has $|G|_p \geqslant k$. Therefore it would be beneficial if we build large candidate groups first which allows us to prune more non-G-Skyline candidate groups at an early stage. A good heuristic for accomplishing this is to reorder the unit groups for each point in the reverse order of their point index.

This is because a point with a larger index, i.e., at a higher skyline layer, tends to have more parents, and hence a larger unit group size. By ordering the unit groups this way, they are likely in the (although not monotonically) decreasing order in their unit group size. We simply need to modify the Tail Set Pruning strategy such that for each node G, the parents (instead of the children) of the unit groups of G are pruned from its tail set.

Subset Pruning.

Given a candidate group G_i, if $|G_i|_p \leqslant k$, then any candidate groups that are G_i's subset have at most k points. This motivates us to employ Subset Pruning. For each candidate group G_i with $|G_i|_p < k$ at Level $|S|_u = 1$, we can check a new candidate group by adding the entire tail set of G_i to G_i, i.e., the last or deepest leaf candidate group G_i^{last} in G_i's subtree. If $|G_i^{last}|_p \leqslant k$, the entire subtree of G_i can be pruned directly and G_i^{last} can be output if $|G_i^{last}|_p = k$. Furthermore, we do not need to check those candidate groups in the subtree of G_i's right siblings G_{sib} because the last candidate group in G_{sib}'s subtree, G_{sib}^{last}, is a subset of G_i^{last}, hence, $|G_{sib}^{last}|_p < k$ too. While both depth-first and breadth-first expansion of the tree will work equally well for Superset Pruning, Subset Pruning benefits from a depth-first expansion such that more siblings can be pruned.

Algorithm.

Given the Superset Pruning, Unit Group Reordering, Subset Pruning, and Tail Set Pruning strategies, the complete unit group-wise algorithm is shown in Algorithm 3. Line 1 applies Unit Group Reordering. Subset Pruning is applied to the 1-unit groups in Lines 3 and 6. We also note that while Subset Pruning can be employed at every node, it also adds additional cost for checking the leaf node. So we only apply them at Level $|S|_u = 1$ to gain the most pruning

benefit, as k is far less than n in the usual case. Tail Set Pruning is applied in Line 11 to Line 14 to prune those candidate groups that do not need to be checked. Line 19 applies Superset Pruning to prune those candidate groups with more than k points.

Example 3.8. We show a running example of Algorithm 3 in Figure 3.6. The 1-unit groups are built in reverse order of point index at Level $|S|_u = 1$. For candidate group u_9, we first check the last candidate group (linked by red (thin) arrow) which has $|u_9 \cup u_{10} \cup u_8 \cup u_3 \cup u_{11} \cup u_6 \cup u_1|_p = |u_9 \cup u_8 \cup u_3 \cup u_1|_p = 7 > 4$. So Subset Pruning cannot be applied, and we still need to check u_9's subtree. We build each branch with a depth-first strategy. At candidate group u_3, the last candidate group of its subtree has $|u_3 \cup u_{11} \cup u_6 \cup u_1|_p = |u_3 \cup u_{11} \cup u_1|_p = 4$, so it is a G-Skyline(4) group. Based on the Subset Pruning strategy, the algorithm is terminated because there are no more candidate groups that need to be checked. As a result, only 27 candidate groups need to be checked as shown in Figure 3.6.

3.6 Experiments

In this section, we present experimental studies evaluating our approach.

3.6.1 Experiment Setup

We first present a small user study using the example hotel dataset (as shown in Figure 1) to verify the motivation of G-Skyline. We then evaluate the algorithm for computing the skyline layers, and then perform an extensive empirical study to examine the point-wise and unit group-wise algorithms using both synthetic and real datasets.

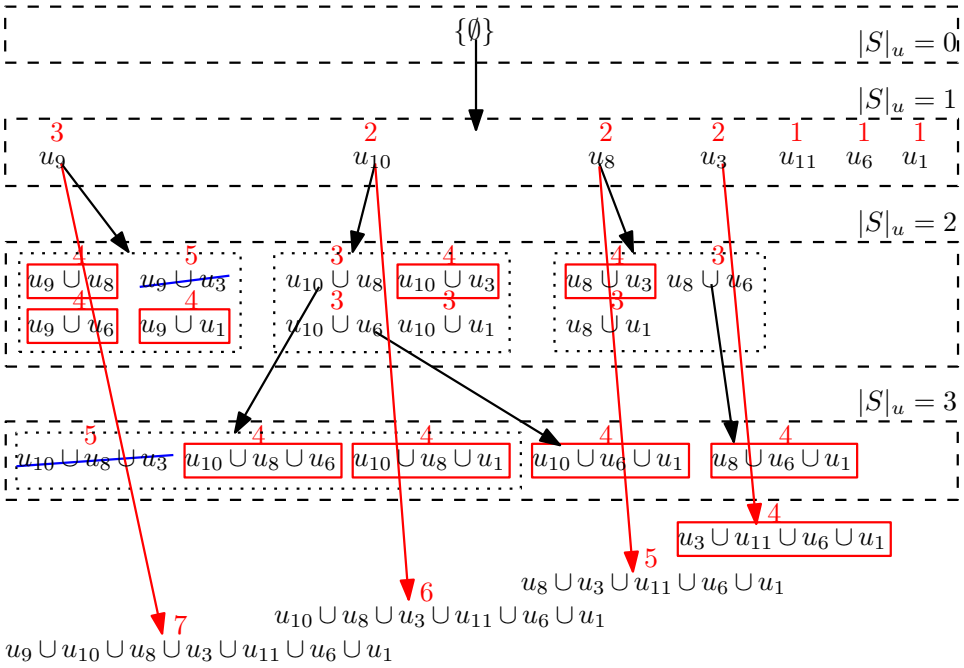


Figure 3.6: The enhanced unit group-wise algorithm for finding G-Skyline groups when $k = 4$.

Since this is the first work for group-based skyline with the new definition of G-Skyline, our performance evaluation was conducted against the enumeration method as a baseline. We implemented the following algorithms in Java and ran experiments on a machine with Intel Core i7 running Ubuntu with 8GB memory.

- **PWise**: Point-wise algorithm presented in Subsection 3.5.1.
- **UWise**: Basic unit group-wise algorithm with Superset Pruning and Tail Set Pruning.
- **UWise+**: Refined unit group-wise algorithm with Unit Group Reordering and Subset Pruning.
- **BL**: We enumerate all $\binom{\mathbb{S}_k}{k}$ candidates, and use Theorem 3.2 to verify each candidate.

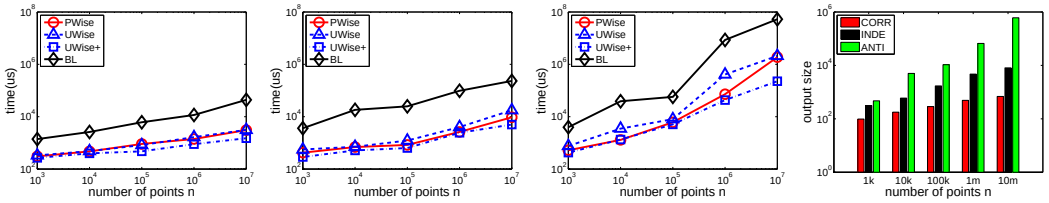
We used both synthetic datasets and a real NBA dataset in our experiments. To study the scalability of our methods, we generated independent (INDE), correlated (CORR), and anti-correlated (ANTI) datasets following the seminal work [7]. We also built a dataset that contains 2384 NBA players who are league leaders of playoffs. The data was extracted from http://stats.nba.com/leaders/alltime/?ls=iref:nba:gnav on 04/15/2015. Each player has five attributes that measure the player's performance. Those attributes are Points (PTS), Rebounds (REB), Assists (AST), Steals (STL), and Blocks (BLK).

3.6.2 Case Study

We performed a small user study using the hotel example dataset (Figure 1). We posted a questionnaire using the conference scenario to ask 38 students and staff members in our department and 30 workers from Amazon Mechanical Turk. We asked them to answer with groups of 2 hotels that they think are the best and provide the reasons of their selections when possible. We received 61 responses in total. Table 3 shows the number of answers for each hotel combinations. The results indeed showed that our group skyline definition covers all the returned groups, while taking any k-skyline points and other existing SUM based group skyline definitions [22, 28, 55] will miss a number of groups that are perceived relevant by the users, such as $\{p_6, p_3\}$ and $\{p_{11}, p_{10}\}$.

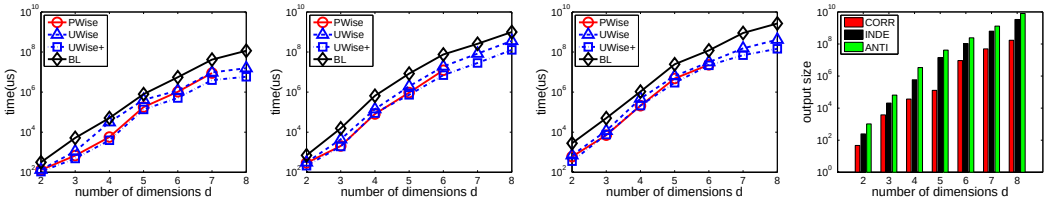
3.6.3 Computing Skyline Layers

We first evaluate our algorithms for computing skyline layers. A baseline approach (BL) is to iteratively compute and then remove the skyline points for each layer. We compare our binary search algorithm (BS) that builds all skyline



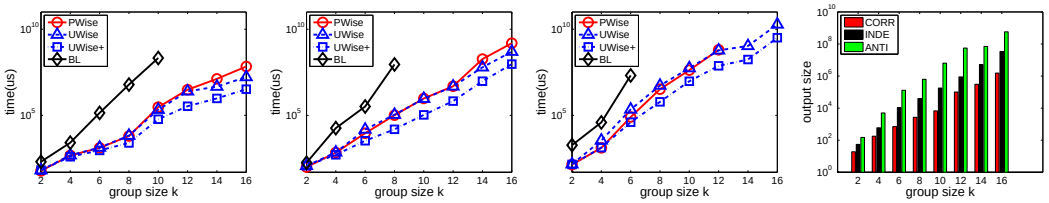
(a) time cost of COR-R (b) time cost of INDE (c) time cost of ANTI (d) output size

Figure 3.7: Computing G-Skyline groups on synthetic datasets of varying n.



(a) time cost of COR-R (b) time cost of INDE (c) time cost of ANTI (d) output size

Figure 3.8: Computing G-Skyline groups on synthetic datasets of varying d.

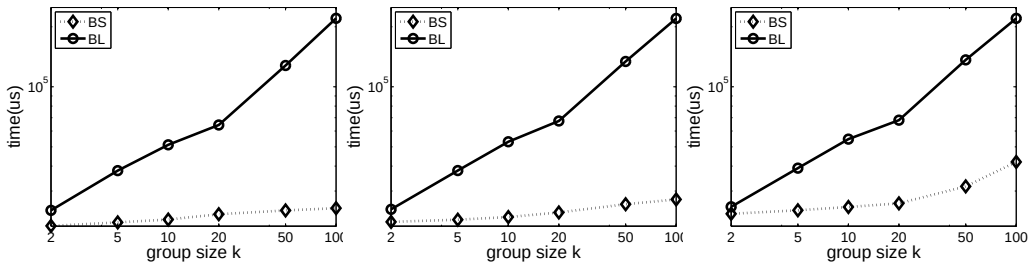


(a) time cost of COR-R (b) time cost of INDE (c) time cost of ANTI (d) output size

Figure 3.9: Computing G-Skyline groups on synthetic datasets of varying k.

layers simultaneously for two dimensional space to the baseline approach.

Figure 3.10 shows the runtime of our binary search algorithm and the Baseline algorithm for varying group size k on the three different datasets (n=10k) respectively. The runtime for the baseline algorithm is not significantly different for the three datasets because the number of skyline points (which differ in the three datasets) has minimal impact on the skyline algorithms in two-dimensional space. For each dataset, the time of the baseline algorithm almost linearly increases with the increase of group size k because the algorithm iter-



(a) CORR (b) INDE (c) ANTI

Figure 3.10: Computing skyline layers on synthetic datasets of varying k.

atively computes the skyline points for each layer. Different from the baseline algorithm, the running time of our binary search algorithm is affected by the different datasets and shows little growth from CORR to INDE, and from INDE to ANTI dataset. The reason is, in our algorithm, only the points in the first k skyline layers will trigger the binary search on the existing layers while the points not in the first k skyline layers are dropped directly. Because of the distribution or correlation patterns of the datasets, the average number of points in each skyline layer (a) for the datasets follows $CORR.a < INDE.a < ANTI.a$, which explains the runtime difference among the datasets. Finally, our binary search algorithm significantly outperforms the baseline algorithm on all datasets. We also implemented and evaluated the higher dimensional case. Even though both algorithms are sensitive to the data distribution in higher-dimensional space, BS still significantly outperforms BL. We did not report them here due to limited space.

3.6.4 G-Skyline Groups in the Synthetic Data

In this subsection, we report the experimental results for computing G-Skyline groups based on synthetic data.

Figures 3.7(a)(b)(c) present the time cost of UWise, UWise+, PWise, and

BL with varying number of points n for the three datasets ($d = 2, k = 4$). Figure 3.7(d) shows the output size with varying n on the three datasets. Because only the points in the first k skyline layers (total number is $\mathbb{S}_k$) are used to compute G-Skyline groups and $\mathbb{S}_k \ll n$ in general, we can see that the time cost and output size are not significantly impacted by n. Figures 3.7(a)(b)(c)(d) show that the time cost and output size grow approximately linearly with n.

Figures 3.8(a)(b)(c) show the time cost of UWise, UWise+, PWise, and BL with varying number of dimensions d on the three datasets ($n = 10000, k = 3$). Figure 3.8(d) shows the output size with varying d on the three datasets. The time cost and output size increase exponentially with respect to the increasing d. This is largely due to the increasing number of points in the first k skyline layers. We did not report the result of the PWise algorithm in some figures due to the high space cost of PWise since it needs to generate much more candidates than UWise.

Figures 3.9(a)(b)(c) show the time cost of UWise, UWise+, PWise, and BL with varying group size k on the three datasets ($n = 10000, d = 2$). Figure 3.9(d) shows the output size with varying k on the three datasets. We did not report the result of the BL algorithm in some figures due to the high cost when k is big. The time cost increases exponentially with respect to the increasing k. Furthermore, the group size k has a significant impact on the output size because there are $\binom{\mathbb{S}_k}{k} \approx {\mathbb{S}_k}^k$ candidate groups. Empirically, the result shows that the output size also grows exponentially with k as shown in Figure 3.9(d).

From the viewpoint of different datasets, the time cost and output size are in increasing order for CORR, INDE, and ANTI, due to the increasing number of points in the first k skyline layers. Comparing different algorithms, UWise,

UWise+, and PWise significantly outperform BL, which validates the benefit of our pruning strategies. PWise is better than UWise when k is small but worse when k is big. Furthermore, PWise is highly space-consuming. Both UWise and UWise+ outperform PWise when k is big, which shows the benefit of the unit group notion. UWise+ outperforms UWise, thanks to the Unit Group Reordering and Subset Pruning strategies.

Discussion. We note that the large output size is indeed a challenging problem for our G-Skyline definition as well as the other group skyline definitions and even the original skyline definition, especially for the ANTI datasets. We provide some discussions as follows. First, the essence of skyline is arguably not to fully help users to choose points given the assumption that the users' attribute weights or preferences are unknown in advance. Rather skyline can be particularly useful to prune those points that are certain to be inferior or dominated by others given any attribute weights. Hence, in a way, we can consider skyline as a preprocessing step for multi-criteria decision making. In this regard, the (relative) output ratio in our results is significantly small compared to the number of all possible groups. Second, if the output size is too large to be consumed by users, additional steps can be performed to choose meaningful representative points. Several existing works [8, 33, 50] investigated this challenging problem. Finally, we also show a weaker group dominance relationship definition, PG-Skyline, which alleviates this issue in Section 3.7.

3.6.5 G-Skyline Groups in the NBA Data

In this subsection, we report the experimental results on the NBA real data.

Figure 3.11(a) shows the time cost with varying n when $d = 5, k = 5$. For each value of $n = 500, 1000, 1500, 2000$, we took the average result based on

100 experiment runs and for each experiment, we randomly chose n out of 2384 players. UWise, UWise+, and PWise again significantly outperform BL due to the efficient pruning strategies and UWise+ performs the best. However, varying n does not have a significant impact on the runtime and output size, as shown in Figure 3.11(b). The reason is that only the number of points in the first k layers is used to compute the G-Skyline groups and $\mathbb{S}_k \ll n$ in general.

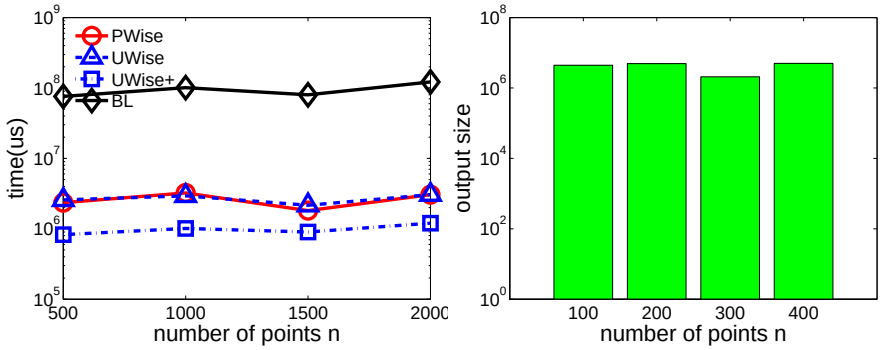
Figures 3.12(a) and (b) show the time cost and output size for different d when $n = 2384, k = 5$. We took the average result of all possible dimension combinations. We see the number of dimensions d has a large impact on both the runtime and output size which increase with increasing d.

Figure 3.13 shows the time cost and output size for different k when $n = 2384, d = 5$. k also has a large impact since the number of points in the first k skyline layers increases significantly as k increases while our approaches are less impacted than the Baseline.

We also report a sample of the final G-Skyline groups in Table 3. There are $\binom{2384}{5} \approx 6.4 \times 10^{14}$ candidate groups, but our algorithm only returns 4865073 G-Skyline groups, i.e., 1 out of 1.3×10^8. We can see the sample groups are formed by elite players with different strengths. For example, G3 is excellent in PTS, REB, AST, STL, and BLK while G1 excels in PTS, and G4 is a good balanced group.

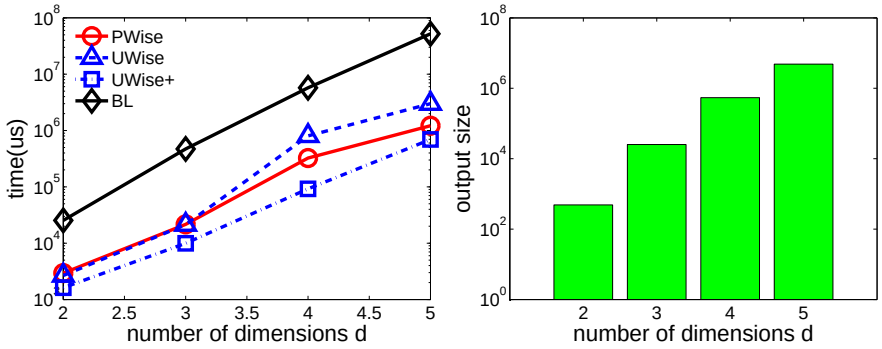
3.7 Extensions

In this section, we discuss two interesting extensions of our proposed work: 1) an AG-Skyline definition based on a more restrictive all-permutation group dominance than G-Skyline, and 2) a PG-Skyline definition based on a less



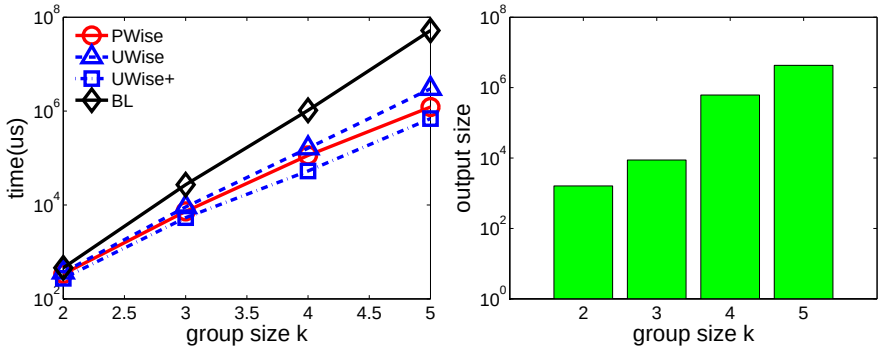
(a) impact of n. (b) impact of n.

Figure 3.11: G-Skyline on NBA dataset of varying n.



(a) impact of d. (b) impact of d.

Figure 3.12: G-Skyline on NBA dataset of varying d.



(a) impact of k. (b) impact of k.

Figure 3.13: G-Skyline on NBA dataset of varying k.

restrictive partial group dominance.

3.7.1 AG-Skyline

Our G-Skyline definition is based on the group dominance defined between a pair of permutations of the points in each group. We formulate an alternative definition AG-Skyline as follows.

Definition 3.7. (**AG-Skyline**). Given a dataset P of n points in a d-dimensional space. Let $G = \{p_1, p_2, ..., p_k\}$ and $G' = \{p'_1, p'_2, ..., p'_k\}$ be two different groups with k points, we say group G ag-dominates group G' if for all (i, j) pairs, $p_i \preceq p'_j$, and for at least one pair (i, j), $p_i \prec p'_j$. The AG-Skyline are those groups that are not ag-dominated by any other groups with same size.

While g-dominance only requires point-wise domination between two groups for *one* permutation of the points in each group (*one-to-one* point domination), ag-dominance requires each point in one group dominates all points in the other group (*one-to-all* point domination). In other words, ag-dominance requires point-wise domination between two groups for *all* permutations of the points. Because the ag-dominance relationship of AG-Skyline is more strict than G-Skyline, less candidate groups can be dominated by other groups, that is, the output of AG-Skyline is a superset of the output of G-Skyline with same group size on the same input dataset.

3.7.2 PG-Skyline

A potential limitation of our proposed definition is the large number of output groups. As the number of dimensions and group size increase, the chance for one group to g-dominate another group is low. As such, the number of G-Skyline groups becomes significantly large, especially for anti-correlated datasets. A potential solution to circumvent this issue is to define an alternative, more

relaxed group-dominance relationship. We define PG-Skyline which is similar to the notion in [8] for individual skyline points. The key idea is to relax the dominance requirement from point-wise dominance for *all* points in each group to (*partial*) p points where $p \leqslant k$.

Definition 3.8. (**PG-Skyline**). Given a dataset P of n points in a d-dimensional space. Let $G = \{p_1, p_2, ..., p_k\}$ and $G' = \{p'_1, p'_2, ..., p'_k\}$ be two different groups with k points of P, we say group G pg-dominates group G' if for p points ($p \leqslant k$) in G and G', we can find two permutations of the p points, $G = \{p_{u_1}, p_{u_2}, ..., p_{u_p}\}$ and $G' = \{p'_{v_1}, p'_{v_2}, ..., p'_{v_p}\}$, such that $p_{u_i} \preceq p'_{v_i}$, for all i ($1 \leqslant i \leqslant p$) and $p_{u_i} \prec p'_{v_i}$ for at least one i. The PG-Skyline are those groups that are not pg-dominated by any other group with same size.

Because the dominance relationship of PG-Skyline is less strict than G-Skyline, more candidate groups can be dominated by other groups, that is, the output of PG-Skyline is a subset of the output of G-Skyline with the same group size on the same input dataset.

3.8 Conclusions

In this chapter, we proposed the problem of G-Skyline groups for finding Pareto optimal groups, instead of finding Pareto optimal points in the classic definition of skylines. Our definition is based on a dominance relationship between groups with same number of points. This is the first work to extend the original skyline definition to group level which captures the quintessence of original skyline definition. To compute the G-Skyline groups efficiently, we presented a novel structure based on skyline layers that not only partitions the points efficiently but also captures the dominance relationship between the points.

We then presented point-wise and unit group-wise algorithms to compute the G-Skyline groups efficiently. A comprehensive experimental study is reported demonstrating the benefit of our algorithms. We also discussed two alternative dominance definitions, AG-Skyline and PG-Skyline, which are the superset and subset of G-Skyline, respectively.

Algorithm 3.3: The unit group-wise algorithm for computing G-Skyline groups.

input : a DSG and group size k.
output: G-Skyline(k) groups.

```
1  build 1-unit group as candidate groups following reverse order of point index;
2  for each candidate group G in 1-unit groups do
3      if |G^last|_p = k then
4          output G^last;
5          break;
6      else if |G^last|_p < k then
7          break;
8      i=2;
9      while the set of candidate group G', such that |G'|_u = i − 1, is not empty do
10         for each G' do
11             for each unit group u_i in G' do
12                 add u_i's parents to Parents Set PS;
13             for each unit group u_j in tail set of G' do
14                 delete u_j if u_j is in PS;
15             for each remaining unit group u in tail set of G' do
16                 add u to G' to form a new candidate group G'' that |G''|_u = i;
17         delete G';
18         output G'' if |G''|_p = k;
19         delete G'' if |G''|_p ⩾k;
20         i++;
```

Table 3.3: Results of case study.

$\{p_{11}, p_8\}$	$\{p_6, p_{11}\}$	$\{p_{11}, p_{10}\}$	$\{p_6, p_3\}$	$\{p_1, p_{11}\}$	$\{p_1, p_6\}$
10	14	12	10	8	7

Table 3.4: Sample of G-Skyline groups on the NBA dataset.

G1	Michael Jordan	Anthony Davis	Kyrie Irving	Allen Iverson	Jerry West	high PTS
G2	Magic Johnson	John Stockton	Isaiah Thomas	Chris Paul	Rajon Rondo	high AST
G3	Michael Jordan	Bill Russell	Magic Johnson	Lance Blanks	Hakeem Olajuwon	high PTS,REB,AST,STL,BLK
G4	Maurice Cheeks	Rich Barry	Slick Watts	Baron Davis	Brad Daugherty	very balanced
G5	Julius Erving	Elvin Hayes	Michael Jordan	Khris Middleton	Alvin Robertson	high STL,BLK

Chapter 4

Skyline Diagram

4.1 Introduction

Similarity queries is a foundational problem in many application domains which retrieves similar objects given a query object. One of the important similarity queries, kNN queries, has been extensively studied which retrieves the k nearest (or most similar) objects based on a predefined distance or similarity metric. For objects with multiple attributes, the similarity or distance on different attributes are typically aggregated with a predefined weight. In many scenarios, it may not be clear how to define the relative weights in order to aggregate the attributes. *Skyline*, also known as *Maxima* in computational geometry or *Pareto* in business management field, is important for multi-criteria decision making or multi-attribute similarity retrieval. Without assuming any relative weights of the attributes, the skyline of a set of multi-dimensional data points consists of the points for which no other point exists that is better (or more similar) in at least one dimension and at least as good (as similar) in every other dimension. In other words, skyline provides all pareto-similar objects to

the given query object that are not dominated by any other objects.

Running Example. There are many example applications that such skyline queries may be desired. For instance, a physician who is treating a heart disease patient may wish to retrieve similar patients based on their demographic attributes and diagnosis test results in order to enhance the treatment for the patient. A car dealer who wishes to price a used car competitively may attempt to retrieve all similar (competitor) cars on the market based on a set of attributes such as mileage and year. For simplicity, we use the running example below to illustrate the skyline definition as well as algorithm descriptions throughout the chapter.

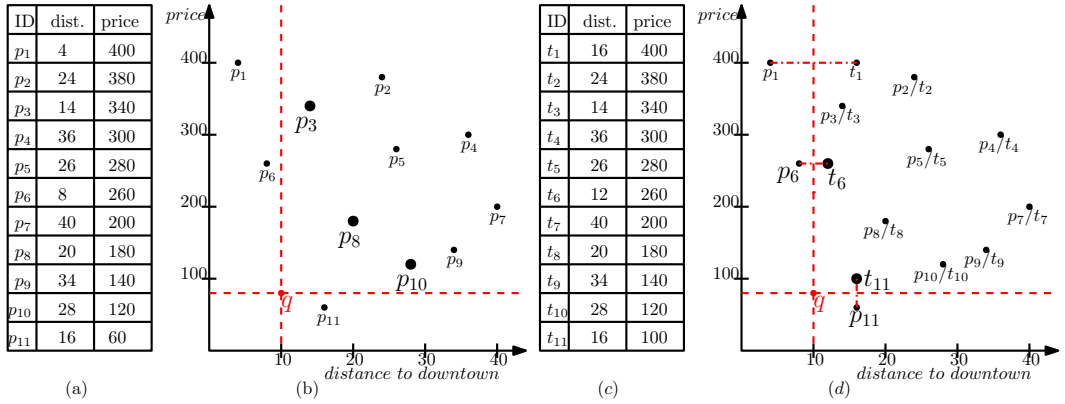


Figure 4.1: A skyline example of hotels.

Assuming a hotel manager wishes to retrieve all competitor hotels that are similar to the hotel with respect to price and distance to downtown. Figure 4.1(a) illustrates a dataset $P = \{p_1, p_2, ..., p_{11}\}$, each representing a hotel with two attributes: the distance to downtown and the price. Figure 4.1(b) shows the corresponding points in the two dimensional space where the x and y coordinates correspond to the two attributes respectively.

Given a query hotel $q = (10, 80)$, if we only consider the hotels with higher price and longer distance to downtown, i.e., the points in the first quadrant with

q as the origin, the skyline are p_3, p_8, p_{10} as shown in Figure 4.1(b) (we refer to this as *quadrant skyline*). If we consider all hotels, we can compute the skyline in each quadrant independently and take the union which is $p_3, p_8, p_{10}, p_6, p_{11}$ (we refer to this as *global skyline*). Alternatively, if only absolute difference for each dimension matters, we can map all data points to the first quadrant with q as the origin and the distance to q as the mapping function, and then compute the skyline from all the mapped points (we refer to this as *dynamic skyline*). The mapped points with $t_i[j] = |p_i[j] - q[j]| + q[j]$ on each dimension j are shown in Figure 4.1(c) and (d). It is easy to see that t_6 and t_{11} are skyline in the mapped space, which means p_6 and p_{11} are the dynamic skyline with respect to query q. We note that dynamic skyline result is always a subset of global skyline result since the mapped points may dominate some points that are otherwise global skyline.

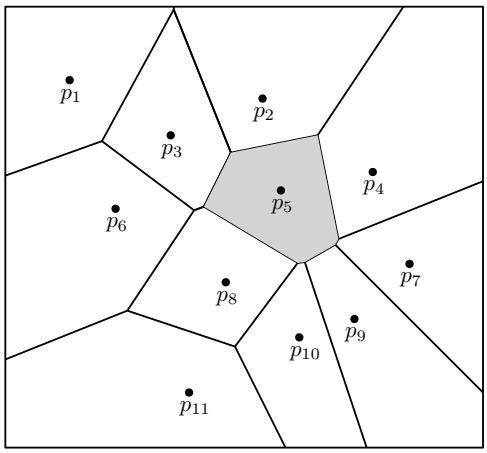


Figure 4.2: Voronoi diagram.

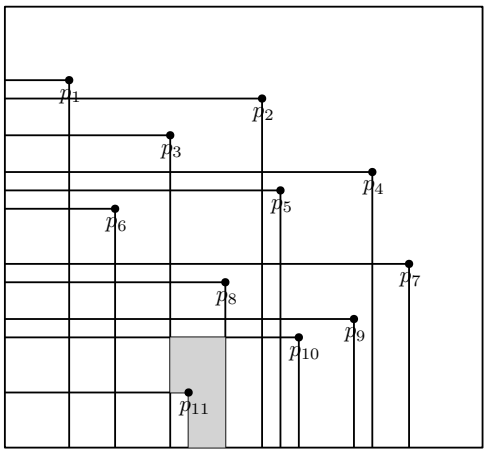


Figure 4.3: Skyline diagram of quadrant skyline queries.

Motivation. Given the importance of such skyline queries, it is desirable to precompute the skyline for a random query point to facilitate and expedite such queries in real time. Voronoi diagram [11] is commonly used to compute and facilitate kNN queries. Therefore, inspired by the Voronoi diagram which

captures the regions with same kNN query results, we propose a fundamental structure in this chapter, referred to as skyline diagram, to capture the query regions with the same skyline result and to facilitate skyline queries.

Given a set of points (or seeds), Voronoi diagram (as shown in Figure 4.2) partitions the plane into a set of polygons corresponding for each seed, each point in the region is closer to the seed than to any other seeds. These regions are called Voronoi cells. In other words, the points in the same Voronoi cell have the same nearest neighbor which is the seed in the cell. For example, the query points in the shaded region have p_5 as the nearest neighbor. kth-order Voronoi diagram can be built for kNN queries where the points in each Voronoi cell has the same kNN results (may not correspond to the seed in the cell as in the Voronoi diagram).

Analogously, given a set of points (or seeds), our proposed skyline diagram partitions the plane into a set of regions, which we call skyline polyominos, and the points in each skyline polyomino have the same skyline results. Figure 4.3 shows an example skyline diagram for quadrant skyline queries given the same seeds. The query points in the shaded region have the same skyline result of p_8, p_{10}. In addition to facilitating skyline queries, skyline diagram also has many other applications.

PIR based Skyline. Skyline diagram can be used to enable efficient Private Information Retrieval (PIR) based skyline queries, similar to using Voronoi diagram for PIR based kNN queries [53]. To protect user privacy or the query object (such as which patient the physician is treating), PIR technique based on cryptography [15] can be used to retrieve items from a server in possession of a database without revealing which item is retrieved. A straightforward PIR implementation for skyline queries requires a large number of computational-

ly expensive PIR retrievals to get potential candidate skyline points and then evaluate which ones are skyline, which becomes computationally prohibitive. Skyline diagram can be used to precompute the skyline for each skyline polyomino, such that PIR retrievals are only employed to retrieve the actual skyline points based on which skyline polyomino the query point belongs to.

Reverse Skyline. Skyline diagram can be used to facilitate the computation of reverse skyline queries [17, 23, 43, 52], similar to using Voronoi diagram for reverse $k$nearest neighbor (RkNN) queries [48]. Given a dataset P, a Reverse Skyline Query given a query point q retrieves all points $p_i \in P$ where q is the skyline of p_i. Given the skyline diagram, each skyline polyomino R has the same skyline results QR. All points in R are the reverse skyline of the points in QR. For example, in Figure 4.3, any point in the shaded polyomino have skyline $\{p_8, p_{10}\}$, and they will be the reverse skyline of p_8 and p_{10}.

Authenticating Skyline. Skyline diagram can be used to authenticate skyline results from outsourced computation, similar to using Voronoi diagram for authenticating kNN queries [54]. Before outsourcing the dataset to the cloud server, the data owner needs to build an authenticated data structure of the dataset, which is an index structure with its root signed by the data owner. After receiving the query from client, the cloud server sends query results, root signature, and verification object to the client. The client can authenticate the query results by the root signature, verification object, and the public key of data owner [32]. Using the skyline diagram will enable efficient construction of the authenticated data structure.

Challenges. While there are many applications of skyline diagram, it is non-trivial to compute the diagram. For quadrant or global skyline queries, a straightforward approach is to draw vertical and horizontal grid lines crossing

each point, which divides the plane into $O(n^2)$ cells. We can easily show that each of these cells has the same skyline since there are no points within the cell that would change the dominance relationship of the points. Thus, we can compute the skyline for each cell, each requiring $O(n \log n)$ time. The time complexity of such a baseline algorithm is $O(n^3 \log n)$ which is not efficient.

For computing the skyline diagram for dynamic skyline, the time complexity can be significantly higher. Because of the mapping function, a straightforward approach is to draw horizontal and vertical bisector lines of each pair of points on each dimension, in addition to the grid lines crossing each point. These resulting subcells are guaranteed to have the same dynamic skyline since there are no points or mapped points in each subcell that would change the dominance relationship of the points. Since the plane is divided into $O(\binom{n}{2}^2)$ subcells, such a baseline algorithm requires $O(n^5 \log n)$ complexity which is prohibitively high.

Contributions. In this chapter, we formally define a novel structure, skyline diagram, which enables precomputation of skyline queries as well as other applications. We study the skyline diagram with respect to three different skyline query definitions, quadrant, global, and dynamic skyline, and propose efficient algorithms. For conciseness and due to the limited space, we only focus on two dimensional space in this chapter, but we note that all our proposed algorithms are directly adaptable to high dimensional space. We briefly summarize our contributions as follows.

- For the first time, we define a novel structure, skyline diagram, to enable precomputation of skyline queries. The skyline diagram consists of skyline regions, referred to as skyline polyominos, each of them corresponding to the same set of skyline result. Similar to Voronoi diagram for kNN

queries, skyline diagram has many applications including computation of skyline queries, reverse skyline queries, PIR based skyline queries, and authentication of outsourced skyline queries.

- To compute the skyline diagram for quadrant and global skyline, we present a baseline algorithm with $O(n^3)$ time complexity and define an important notion of skyline cell. Furthermore, based on the observation of some interesting properties of the skyline results of neighboring cells, we propose an improved $O(n^3)$ algorithm utilizing the directed skyline graph, which performs much better than the baseline algorithm in practice. Finally, we quantify the exact relationship between the skyline results of neighboring cells, and present two scanning algorithms which further improves the performance.

- To compute the skyline diagram for dynamic skyline, we first present a baseline algorithm with $O(n^5)$ time complexity and define an important notion of skyline subcell. Furthermore, based on the observation that dynamic skyline query result is a subset of global skyline, we present an improved subset algorithm utilizing the skyline diagram of global skyline, which requires $O(n^5)$ but is better in practice. Finally, based on the relationship of the skyline results of neighboring subcells, we present a scanning algorithm which achieves $O(n^4 \log n)$ time.

- We conduct comprehensive experiments on real and synthetic datasets. The experimental results show our proposed algorithms are efficient and scalable.

Organization. The rest of the chapter is organized as follows. Section 4.2 presents the related work. Section 4.3 introduces some background knowledge

and formally define skyline diagram. The algorithms for computing the skyline diagram for quadrant/global skyline and dynamic skyline are presented in Section 4.4 and 4.5 respectively. We report the experimental results and findings for performance evaluation in Section 4.6. Section 4.7 concludes the chapter.

4.2 Related Work

The problem of computing skyline (Maxima) is a fundamental problem in computational geometry field because the skyline is an interesting characterization of the boundary of a set of points. The skyline computation problem was firstly studied in computational geometry [26] which focused on worst-case time complexity. [24,35] proposed output-sensitive algorithms achieving $O(n \log v)$ in the worst-case where v is the number of skyline points which is far less than n in general. Several works [3,4,7,14] in both computational geometry and database fields focused on how to achieve the best average-case time complexity. For a detailed survey both for worst-case and average-case, please see [19].

Since the introduction of the skyline operator by Börzsönyi et al. [7], skyline has been extensively studied in the database field. Many algorithms are proposed in the context of relational query engine and external memory model, for example, [19, 49]. Based on the traditional skyline definition, [1, 25] studied the parallel algorithms for skyline.

Many works also studied extensions or variants of the classical skyline definitions. Papadias et al. [42] studied *group-by skyline* which groups the objects based on their values in one dimension and then computes the skyline for each group, and *k-skyband* which computes objects dominated by at most k objects (the case $k = 0$ corresponds to the conventional skyline) based on individual

dominance relationship. Skyline in subspace, i.e., a subset of the dimensions or points, was studied in [8, 45, 46, 51]. [17, 52] discussed the reverse skyline problem which is similar to the reverse k-nearest neighbor problem. [16] presented skyline-based statistical descriptors for capturing the distributions over pairs of dimensions. Some works defined and studied the skyline on different data types/domains. For example, [47] and [13] studied the spatial skyline and a more general metric skyline, respectively. [21] proposed the skyline for moving objects. [18, 29, 37, 44, 56] studied the skyline problem for uncertain data.

The most related works to our skyline diagram are the "safe zone" for location-based skyline queries [12, 21, 27, 31]. Huang et al. [21] presents the first work on continuous skyline query processing. Given a set of n data points $< x_i, y_i; v_{xi}, v_{yi}; p_{i1}, ..., p_{im} > (i = 1, ..., n)$, where x_i and y_i are positional coordinates in two dimensional space, v_{xi} and v_{yi} are the velocity in the X and Y dimensions, while $p_{ij}(j = 1, ..., m)$ are the m static nonspatial attributes, which will not change with time. For a query point q starting from (x_q, y_q) moving with (v_{qx}, v_{qy}), q poses continuous skyline query while moving, and the queries involve both distance and all other static dimensions. Such queries are dynamic due to the change in spatial variables. In their solution, they compute the skyline for x_q, y_q at the start time 0. Subsequently, continuous query processing is conducted for each user by updating the skyline instead of computing from scratch. Lee et al. [27] studies a similar problem to [21]. Both of them rely on the assumption that the velocities of the moving points are known. Generally speaking, they compute the skyline for query points moving on a line segment. Lin et al. [31] studies a problem of computing the skyline for a range. They employed the similar idea for authenticating skyline queries in [30] [32]. Cheema et al. [12] proposes a safe zone for a query point q. A safe

zone is the area such that the results of a query q remain unchanged as long as the query lies inside the area. Both [31] and [12] study the location-based skyline problem with m static attributes and one dynamic attribute, which is the distance to the query point.

The main difference between the above work and our work is that they only consider one dynamic attribute, while in our case all attributes can be dynamic. The skyline polyomino can be considered as a generalization of the safe zone in two or multidimensional space. Furthermore, it is non-trivial to extend these query techniques from one dynamic attribute to two or multi-dimensional case, as fundamentally these algorithms convert the problem to nearest neighbor queries for the single dynamic attribute and utilize Voronoi diagram.

4.3 Preliminaries and Problem Definitions

In this section, we introduce our skyline diagram definition and related concepts as well as their properties which will be used in our algorithm design. For reference, a summary of notation is given in Table 4.1.

Table 4.1: The summary of notations.

Notation	Definition
P	dataset of n points
$p_i[j]$	the j^{th} attribute of p_i
q	query point
n	number of points in P
$C_{i,j}$	Cell with bottom left corner coordinate (i, j)
$Sky(C_{i,j})$	the skyline of Cell $C_{i,j}$
$SC_{i,j}$	Subcell with bottom left corner coordinate (i, j)
$Sky(SC_{i,j})$	the skyline of Subcell $C_{i,j}$

Definition 4.1. (**Skyline**). Given a dataset P of n points in d-dimensional space. Let p and p' be two different points in P, we say p dominates p', denoted

by $p \prec p'$, if for all i, $p[i] \leqslant p'[i]$, and for at least one i, $p[i] < p'[i]$, where $p[i]$ is the i^{th} dimension of p and $1 \leqslant i \leqslant d$. The skyline points are those points that are not dominated by any other point in P.

Definition 4.2. (**Dynamic Skyline Query** [17]). Given a dataset P of n points and a query point q in d-dimensional space. Let p and p' be two different points in P, we say p dominates p' with regard to the query point q, denoted by $p \prec p'$, if for all i, $|p[i] - q[i]| \leqslant |p'[i] - q[i]|$, and for at least one i, $|p[i] - q[i]| < |p[i] - q[i]|$, where $p[i]$ is the i^{th} dimension of p and $1 \leqslant i \leqslant d$. The skyline points are those points that are not dominated by any other point in P.

The traditional skyline computation is a special case of dynamic skyline query where the query point is the origin. On the other hand, computing dynamic skyline given a query point q is equivalent to computing the traditional skyline after transforming all points into a new space where q is the origin and the absolute distances to q are used as mapping functions. Take Figure 4.1 as an example, given a query point $q = (10, 80)$, p_6 dominates p_1 because p_6's corresponding point t_6 in the mapped space dominates p_1's corresponding point t_1. Because no other points can dominate t_6 and t_{11}, the result of dynamic skyline query given q is $\{p_6, p_{11}\}$.

The dynamic skyline query consider the dominance among all points. Given a query point, if we consider each quadrant divided by the query point independently, i.e., only consider dominance among points within the same quadrant, we can define global skyline query below.

Definition 4.3. (**Global Skyline Query** [17]). Given a dataset P of n points and a query point q in d-dimensional space. The query point q divides the d-dimensional space into 2^d quadrants. Let p and p' be two different points *in*

the same quadrant of P, we say p dominates p' with regard to the query point q, denoted by $p \prec p'$, if for all i, $|p[i]-q[i]| \leqslant |p'[i]-q[i]|$, and for at least one i, $|p[i]-q[i]| < |p'[i]-q[i]|$, where $p[i]$ is the i^{th} dimension of p and $1 \leqslant i \leqslant d$. The skyline points are those points that are not dominated by any other point in P.

Given a query point, we refer to the global skyline from a single quadrant as ***Quadrant Skyline Query***. In other words, the global skyline is the union of the quadrant skyline from all quadrants. Back to Figure 4.1, given the query point q, the quadrant skyline is $\{p_3, p_8, p_{10}\}$ in the first quadrant, $\{p_6\}$ in the second quadrant, $\varnothing$ in the third quadrant, and $\{p_{11}\}$ in the fourth quadrant. The global skyline is the entire set of $\{p_3, p_6, p_8, p_{10}, p_{11}\}$. It is easy to see that the dynamic skyline is a subset of the global skyline. This property will be used to design algorithms for skyline diagram of dynamic skyline.

Similar to the definition of Voronoi cell and kth-oder Voronoi diagram for kNN query, we define the skyline polyomino and skyline diagram for skyline query as follows.

Definition 4.4. (**Skyline Polyomino**). A polyomino SP_i is a skyline polyomino (short for skymino), if given any two query points q_a and q_b in SP_i, q_a's skyline result $Sky(q_a)$ equals to q_b's skyline result $Sky(q_b)$, meanwhile for any query point q_c outside SP_i, the skyline result $Sky(q_c)$ of q_c does not equal to $Sky(q_a)$.

Definition 4.5. (**Skyline Diagram**). Given a dataset P of n points $p_1, ..., p_n$. We define the Skyline Diagram of P as the subdivision of the plane into a set of polyominos with the property that any points in the same polyomino have the same skyline query result.

Problem Statement. Given n points, in this chapter, our goal is to compute the skyline diagram for quadrant/global skyline queries and dynamic skyline queries efficiently.

4.4 Skyline Diagram of Quadrant and Global Skyline

In this section, we present algorithms for computing skyline diagram of quadrant and global skyline. We first show a baseline algorithm and define an important notion of skyline cell, which will be used by all our proposed algorithms. We then present an improved algorithm using directed skyline graph. Both algorithms have $O(n^3)$ time complexity but directed skyline graph algorithm is much faster than the baseline in practice. We then present an important property of the skyline of neighboring cells and present two faster algorithms based on the property, scanning algorithm and aggressive scanning algorithm.

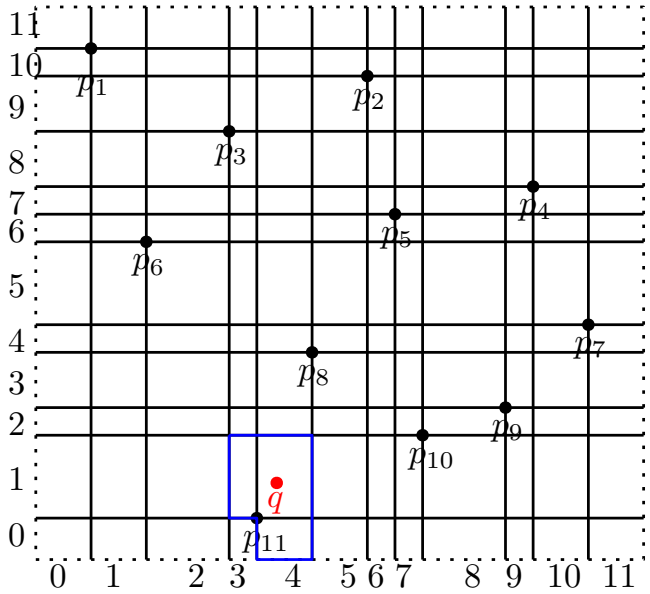


Figure 4.4: Quadrant skyline query.

4.4.1 Baseline Algorithm

We first show a baseline algorithm for computing skyline diagram and introduce an important notion, skyline cell. The key for computing skyline diagram is to find regions such that any query points in the same region have the same skyline result. Intuitively, we can find small regions that are guaranteed to have the same result and then merge them to form bigger regions.

Skyline Cell. If we draw one horizontal and one vertical line over each point, these $O(n)$ grid lines divide the plane into $O(n^2)$ cells. For example, in Figure 4.4, the horizontal and vertical lines over each of the 11 points divide the plane into 144 cells. It is clear that any query points inside each cell are guaranteed to have the same quadrant and global skyline because there are no points in the cell that would change the dominance relationship of the points with respect to the query point. We name the cell as Skyline Cell.

Definition 4.6. (**Skyline Cell**). The horizontal and vertical lines over each point divide the plane into skyline cells. Any query points in the same skyline cell have the same skyline results for quadrant and global skyline.

Finding skyline for each skyline cell. Since we know that query points in each skyline cell have the same skyline results, we can employ any skyline algorithm to compute the skyline for each cell. Given a cell $C_{i,j}$, we denote $Sky(C_{i,j})$ as its skyline result (the notation is also shown in Table 1). We can then merge the skyline cells with the same results to form skyline polyominos. Since the skyline computation of n points for each cell takes $O(n \log n)$ time and there are $O(n^2)$ skyline cells, the total time complexity is $O(n^3 \log n)$. If the n points are sorted on x-axis, we can compute the skyline for one cell in $O(n)$ time. Therefore, the total time can be reduced to $O(n^3)$. This baseline

algorithm is shown in Algorithm 4.1. After the points are sorted (Line 1), the steps for computing skyline in $O(n)$ based on ordered points are shown in Lines 5-12 where $g_{i,j}$ is the left lower intersection of skyline cell $C_{i,j}$.

Algorithm 4.1: The baseline algorithm for skyline diagram of quadrant skyline queries.

```
   input  : a set of n points and skyline cells C_{i,j}.
   output: skyline of each skyline cell Sky(C_{i,j}).
 1 sort the points in ascending order on x-axis;
 2 for i=0 to n do
 3     for j=0 to n do
 4         for k=1 to n do
 5             if p_k[x] > g_{i,j}[x] && p_k[y] > g_{i,j}[y] then
 6                 add p_k to the candidate list;
 7         choose the first element p_first as the first skyline;
 8         p_temp = p_first;
 9         for l=2 to |candidate list| do
10             if p_l[y] < p_temp[y] then
11                 add p_l to skyline pool;
12                 p_temp[y] = p_l[y];
13         return skyline pool as Sky(C_{i,j});
```

Merging skyline cells into skyline polyominos. Once we have the skyline results for each cell, we can merge the cells with same results to form skyline polyominos. For each skyline cell, we search its upper and right cells and combine those cells if they share the same skyline. The entire merging requires $O(n^2)$ time.

Example 4.1. In Figure 4.4, the skyline cells $C_{4,0}$, $C_{4,1}$, and $C_{3,1}$ share the same skyline result $\{p_8, p_{10}\}$, and hence are combined to form a skyline polyomino.

Complexity. Finding skyline phase requires $O(n^3)$ time, and merging phase requires $O(n^2)$ time. Therefore, the total time for the baseline algorithm (Algorithm 4.1) is $O(n^3)$.

4.4.2 Directed Skyline Graph Algorithm

In the baseline algorithm, we need to compute skyline for each skyline cell from scratch which is costly. In this subsection, based on the observation of some interesting relationships of the skyline results of neighboring cells, we propose an incremental algorithm utilizing the directed skyline graph for computing skyline for neighboring cells. Note that the merging step of the skyline cells remains the same as the baseline.

Our algorithm is based on the key observation that when moving from one cell to its neighboring upper or right cell, the only point that will cause the skyline result to change is the point on the crossed grid line. For example, in Figure 4.4, given cell $C_{0,0}$, the skyline is $\{p_1, p_6, p_{11}\}$. When moving to its right cell $C_{1,0}$ across the p_1 grid line, the new result is the skyline of the remaining points after removing p_1, that is $\{p_6, p_{11}\}$. Similarly, when moving from $C_{0,0}$ to its upper cell $C_{0,1}$ across the p_{11} grid line, the new result is the skyline of the points after removing p_{11}, that is $\{p_1, p_6, p_{10}\}$. Based on this observation, we propose to use a data structure called the directed skyline graph to facilitate the incremental computation of the skyline from one cell to its neighboring cell.

We first briefly describe the directed skyline graph (DSG) adapted from [34] and explain how it can be used to facilitate the incremental skyline computation and then present our algorithm utilizing the graph for computing the skyline for all skyline cells.

Given n points, we first compute its skyline layers by employing the skyline layer algorithm from [34]. The skyline layers of our running example is shown in Figure 4.5. The first skyline layer consists of all skyline points in the original dataset. The second skyline layer consists of all skyline points of the remaining points after removing the points from the first skyline layer. And similarly for

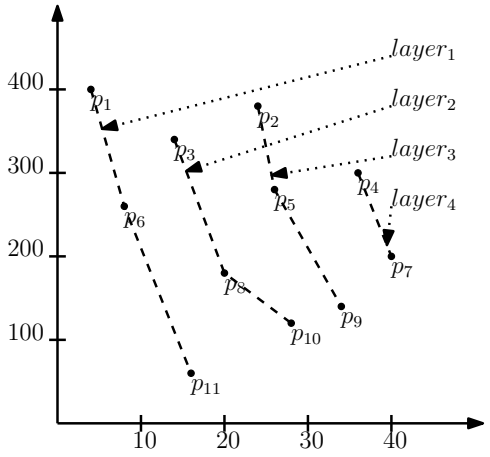


Figure 4.5: Skyline layers.

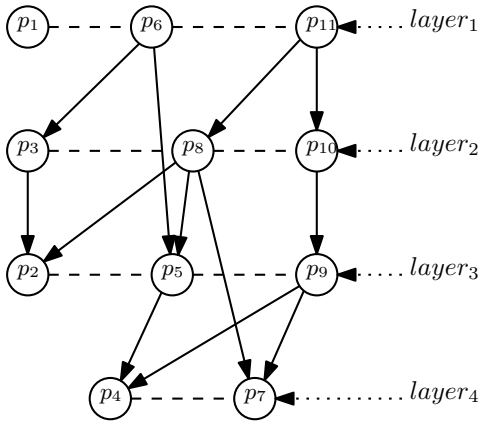


Figure 4.6: Directed skyline graph.

the remaining skyline layers. There are several properties for skyline layers: 1) the points on the same layer cannot dominate each other, 2) the points on a lower layer may dominate the points on a higher layer, and 3) the points on a higher layer cannot dominate the points on a lower layer. Based on these skyline layers, we obtain the directed skyline graph which captures all the direct dominance relationships between the points as shown in Figure 4.6. For example, p_6 directly dominates p_3 and p_5. We note that the directed skyline graph algorithm from [34] includes both direct and indirect dominance relationships (e.g. p_6 dominates p_4 indirectly). We adapted it such that we only include the direct links which is needed to solve our problem.

We now show how we can incrementally compute the skyline from one cell to its neighboring cell utilizing the skyline graph. When moving from one cell to its right neighboring cell across the grid line over p, there are two changes in the skyline result caused by the point p: 1) p is no longer a skyline, 2) new skyline points may appear since they are not dominated by p anymore with respect to the query point in the new cell. So all we need to do is to remove p as well as its dominance links from the skyline graph, any of the children

Algorithm 4.2: The directed skyline graph algorithm for skyline diagram of quadrant skyline queries.

```
   input  : a set of n points and skyline cells C_{i,j}.
   output: skyline of each skyline cell Sky(C_{i,j}).
 1 compute the directed skyline graph DSG;
 2 Sky(C_{0,0}) = Sky(P);
 3 for i=1 to n do
 4     delete the point p_i between C_{i-1,0} and C_{i,0} from DSG;
 5     delete the link between p_i and its directed children;
 6     Sky(C_{i,0}) = Sky(C_{i-1,0}) - p_i + the children of p_i without any
       remaining parent;
 7 for i=0 to n do
 8     for j=1 to n do
 9         delete the point p_j between C_{i,j-1} and C_{i,j} from DSG;
10         delete the link between p_j and its directed children;
11         Sky(C_{i,j}) = Sky(C_{i,j-1}) - p_j + the children of p_j without any
           remaining parent;
```

points of p without remaining parents will be a new skyline (since it is no longer dominated by any points).

Example 4.2. Given $C_{0,0}$, its skyline is the set of points on the first skyline layer, $\{p_1, p_6, p_{11}\}$. When moving from $C_{0,0}$ to its right neighboring cell $C_{1,0}$ across the p_1 grid line, to compute the new skyline, all we need to do is to remove p_1 (p_1 does not have any direct dominance links), hence the skyline for $C_{1,0}$ is simply $\{p_6, p_{11}\}$ after removing p_1 from the skyline set. When we move further to $C_{1,0}$'s right neighboring cell $C_{2,0}$ across the p_6 grid line, we just need to remove p_6 and remove the dominance links from p_6 to p_3 and p_5. Since p_3 is no longer dominated by any points after p_6 is removed, it becomes a new skyline. Hence the skyline for $C_{2,0}$ consists of the remaining skyline p_{11} and the new skyline p_3, i.e. $\{p_3, p_{11}\}$.

Given any cell, we can also compute its upper neighboring cell in a similar way. Hence our algorithm starts from the origin cell $C_{0,0}$, and incrementally

computes the first row of cells from left to right. Then it incrementally computes all the rows from bottom to up. The algorithm is shown in Algorithm 2.2. The directed skyline graph is computed in Line 1 and the skyline for $C_{0,0}$ is computed in Line 2. The skyline for the first row is computed in Lines 3-6. Lines 7-11 compute the skyline of the remaining rows.

Complexity. As we iterate through all the cells in one row, we are removing dominance links from the skyline graph. Each link costs one update and the total number of links is $O(n^2)$ as shown in Figure 4.6. Therefore, it requires $O(n^2)$ time to compute the skyline of cells in one row. Since there are n rows that need to be computed, the time complexity for the directed skyline graph algorithm (Algorithm 2.2) is $O(n^3)$. We note that in practice, the number of links is much smaller than n^2. Hence the algorithm is much faster than the baseline algorithm in practice.

4.4.3 Scanning Algorithm

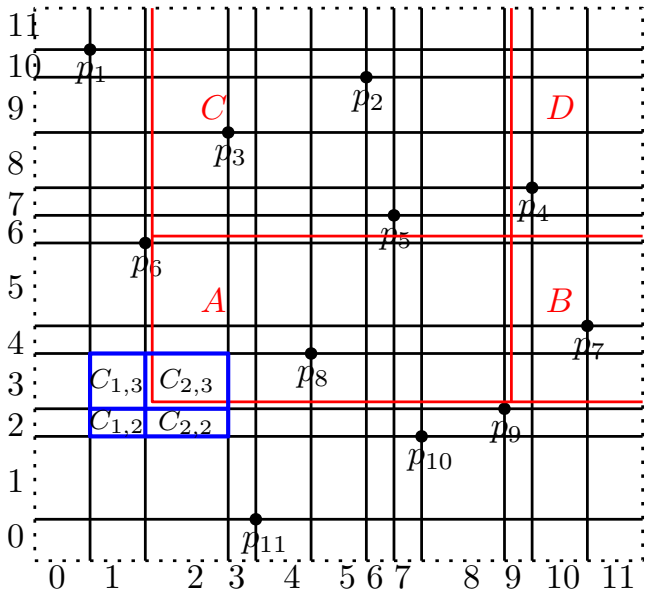


Figure 4.7: Scanning algorithm.

The previous algorithm still involves computation of skyline, ideally we would like to avoid the computation as much as possible. We observed earlier that the skyline results for neighboring cells are different only due to the point on the shared grid line. For example, in Figure 4.7, $Sky(C_{1,2})$ and $Sky(C_{2,2})$ are different due to p_6, same for $Sky(C_{1,3})$ and $Sky(C_{2,3})$. Similarly, $Sky(C_{1,2})$ and $Sky(C_{1,3})$ are different due to p_9, same for $Sky(C_{2,2})$ and $Sky(C_{2,3})$. In this subsection, we observe an interesting property of the exact relationship between the skyline results of neighboring cells, and present a new $O(n^2)$ time algorithm utilizing this property for computing skyline for all cells. Again, the merging of cells into skyline polyominos stays the same as the baseline.

Theorem 4.1. Given any skyline cell $C_{i,j}$ (except the ones that have a point as its upper right corner), and its right cell $C_{i+1,j}$, upper cell $C_{i,j+1}$, and upper right cell $C_{i+1,j+1}$, their skyline results have a relationship as follows:

$$Sky(C_{i,j}) = Sky(C_{i+1,j}) + Sky(C_{i,j+1}) - Sky(C_{i+1,j+1})^1$$

Proof. Given a cell $C_{i,j}$, we define the following. p_R (p_C) denotes the point that lies on the upper (right) grid line of $C_{i,j}$. Range A is the rectangle formed by the grid lines crossing p_R and p_C (excluding the two points). Range B is the right rectangle of A. Range C is the upper rectangle of A. And Range D is the upper right rectangle of A. An example is shown in Figure 7.

Consider $C_{i,j}$'s upper right cell $C_{i+1,j+1}$, we denote $SkyP(A)$ as the set of points in range A contributed to $Sky(C_{i+1,j+1})$. And similarly for $SkyP(B)$, $SkyP(C)$, and $SkyP(D)$. Note that $SkyP(D)$ will be empty if $SkyP(A)$ is not empty which will dominate all points in D.

[1] multiset operation.

We can compute the skyline results of the four cells as follows:

$$Sky(C_{i,j}) = \{p_R\} \cup \{p_C\} \cup SkyP(A)$$

$$Sky(C_{i+1,j}) = \{p_R\} \cup SkyP(A) \cup SkyP(C)$$

$$Sky(C_{i,j+1}) = \{p_C\} \cup SkyP(A) \cup SkyP(B)$$

$$Sky(C_{i+1,j+1}) = SkyP(A) \cup SkyP(B) \cup SkyP(C) \cup SkyP(D)$$

Then we have:

$$Sky(C_{i+1,j}) + Sky(C_{i,j+1}) - Sky(C_{i+1,j+1})$$

$$= (\{p_R\} \cup SkyP(A) \cup SkyP(C)) + (\{p_C\} \cup SkyP(A) \cup SkyP(B))$$

$$-(SkyP(A) \cup SkyP(B) \cup SkyP(C) \cup SkyP(D))$$

$$= \{p_R\} \cup \{p_C\} \cup SkyP(A) = Sky(C_{i,j})$$

□

Example 4.3. In Figure 4.7, given cell $C_{1,2}$, p_R on its upper grid line is p_9 and p_C on its right grid line is p_6. Consider the skyline result of its upper right cell $C_{2,3}$, we have $SkyP(A) = \{p_8\}$, $SkyP(B) = \varnothing$ as p_7 is dominated by p_8, $SkyP(C) = \{p_3\}$ as p_2, p_5 are dominated by p_8, and $SkyP(D) = \varnothing$ as p_4 is dominated by p_8. We have skyline result for the upper right cell $Sky(C_{2,3}) = \{p_3, p_8\}$, the upper cell $Sky(C_{1,3}) = \{p_6, p_8\}$, and the right cell $Sky(C_{2,2}) = \{p_3, p_8, p_9\}$. It is easy to see that the skyline for the given cell is $Sky(C_{1,2}) = Sky(C_{2,2}) + Sky(C_{1,3}) - Sky(C_{2,3}) = \{p_6, p_8, p_9\}$.

We note that the above property holds for all skyline cells except the ones

that have a point as its upper right corner. For these cells, their skyline is the upper right point because this point dominates all the upper right region. For example, in Figure 4.7, $Sky(C_{4,3}) = \{p_8\}$, and $Sky(C_{6,6}) = \{p_5\}$.

Based on these properties, we present a scanning algorithm as shown in Algorithm 4.3. The basic idea is to start from the top and rightmost cell, and scan the cells from top down and right to left, then utilizing the property in Theorem 1 to compute the skyline for each cell. We first initialize the skyline results for the skyline cells on the top row and rightmost column to $\varnothing$ (Lines 1-3). Then for each cell $C_{i,j}$, if there is a point p on its upper right corner, we set $Sky(C_{i,j}) = \{p\}$ (Line 7). Otherwise, we use $Sky(C_{i,j}) = Sky(C_{i+1,j}) + Sky(C_{i,j+1}) - Sky(C_{i+1,j+1})$ to compute the skyline of $C_{i,j}$ (Line 9).

Algorithm 4.3: The scanning algorithm for skyline diagram of quadrant skyline queries.

```
   input  : a set of n points and skyline cells C_{i,j}.
   output: skyline of each skyline cell Sky(C_{i,j}).
1  for i=0 to n do
2      Sky(C_{i,n}) = ∅;
3      Sky(C_{n,i}) = ∅;
4  for i=n-1 to 0 do
5      for j=n-1 to 0 do
6          if there is a point p on the upper right corner of C_{i,j} then
7              Sky(C_{i,j})={p};
8          else
9              Sky(C_{i,j}) = Sky(C_{i+1,j}) + Sky(C_{i,j+1}) − Sky(C_{i+1,j+1});
```

Complexity. There are $O(n^2)$ cells, each cell requires $O(n)$ time for multiset computation. Therefore, Algorithm 4.3 requires $O(n^3)$ time in total.

4.4.4 Aggressive Scanning Algorithm

All previous algorithms involve computing skyline for each skyline cell (divided by the grid lines) and then merging them into skyline polyominos. Ideally, if we can find the skyline polyominos directly rather than combining the skyline cells, we can save the cost of computing skyline for each skyline cell. In this subsection, we show an aggressive scanning algorithm that achieves this goal.

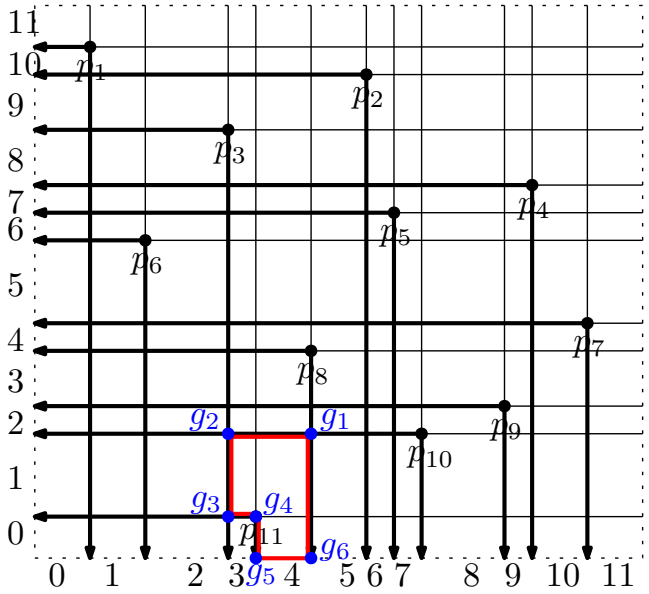


Figure 4.8: Aggressive scanning algorithm.

We observed previously that when we move from one cell to its right cell, the only change in the skyline result is caused by the point on the crossed grid line. In fact, we can further observe that if the point on the crossed grid line lies below the cell, then the skyline result does not change at all. This is because we are only considering the points in the cell's upper right quadrant. For example, $C_{3,1}$ has skyline result $\{p_8, p_{10}\}$. When we move from $C_{3,1}$ to $C_{4,1}$ crossing point p_{11}, the skyline remains the same because p_{11} is below the cells and does not affect the result. Similarly, when we move from one cell to its upper cell, if the point on the crossed grid line is to the left of the cells, the

skyline result does not change either. In other words, each point only affects the skyline result of its lower and left cells, not its upper or right cells. Motivated by this observation, instead of drawing grid lines over each point to divide the plane into skyline cells, we can draw two half-open grid lines starting from each point, one downward and another leftward. These $O(2n)$ grid line segments divide the plane into a set of polyominos, each containing one or more cells. Since we know that each point will not affect the skyline result of its upper and right cells, we can show that any query points in such formed polyominos have the same skyline results. We have a theorem as follows.

Theorem 4.2. Given a set of points, if we draw two half-open grid lines starting from each point, one downward and another leftward, each polyomino formed by these $O(2n)$ lines is a skyline polyomino and any query points inside have the same first quadrant skyline query results.

Proof. Given a skyline polyomino formed by these half-open grid lines, if we consider the upper right corner query point for each of the skyline cells in the polyomino, they have the same set of points in their upper right quadrant, thus they have the same skyline results. We have showed earlier all points in the same skyline cell have the same quadrant skyline results, hence all query points in the same polyomino have the same first quadrant skyline results. □

Example 4.4. In Figure 4.8, the red polyomino contains three cells, $C_{3,1}$, $C_{4,0}$, and $C_{4,1}$. $Sky(C_{3,1}) = Sky(C_{4,0}) = Sky(C_{4,1}) = \{p_8, p_{10}\}$.

While it is straightforward to visually see the skyline polyominos from the figure (e.g., Figure 4.8), we need to represent the skyline polyominos computationally by its vertices, which are the intersection points of the half-open grid

Algorithm 4.4: The aggressive scanning algorithm for skyline diagram of quadrant skyline queries.

```
   input  : a set of n points.
   output: skyline polyominos.
1  /*compute all the intersection points and link them by left and right
   neighbors in Lines 2-8*/;
2  sort the points in descending order on y-axis, p_1 (p_n) is the point with
   highest (lowest) y-coordinate;
3  p_1.left = (0, p_1[y]);
4  for i=2 to n do
5      insert p_i into sorted queue X by x-axis and its new index is j;
6      p_i.left = (p_{j-1}[x], p_i[y]);
7      (p_{j-1}[x], p_i[y]).right = p_i;
8      for j=i to 1 of sorted queue X do
9          (p_{j-1}[x], p_i[y]).left = (p_{j-2}[x], p_i[y]);
10         (p_{j-2}[x], p_i[y]).right = (p_{j-1}[x], p_i[y]);
11 /*similarly, we can compute the lower neighbor of each intersection
   point which is omitted due to space limit*/;
12 for each intersection point g_0 do
13     skymino_g = {g_0}; g= g_0;
14     skymino_g.append(g.left); g = g.left;
15     while g[x]! = g_0[x] do
16         skymino_g.append(g.lower); g = g.lower;
           skymino_g.append(g.right); g = g.right;
17     return skymino_g;
```

lines including the points themselves. We now show how to compute the coordinates of these vertices and then how to find the vertices for each polyomino.

We observe that for each point p, its horizontal grid line only intersects with the vertical grid lines from its upper points, i.e. with larger y coordinates. Hence, given a point $p(x, y)$, we can compute all the intersection points on its horizontal grid line as $g(x_j, y)$ where x_j is the x coordinate from those points with larger y coordinates than p. For each intersection point, we record its left and right neighbor, so we can retrieve the vertices for each polyomino. Similarly, for each point, we compute the intersection points on its vertical

grid line, and record the lower and upper neighbor for each intersection point. The detailed algorithm is shown in Algorithm 4.4.

Example 4.5. For p_4, its horizontal line intersects with the vertical lines of p_2, p_3, p_1, hence the intersection points on its horizontal line are $(p_2[x], p_4[y])$, $(p_3[x], p_4[y]), (p_1[x], p_4[y])$, and $(0, p_4[y])$. For each point, it has a left/right and upper/lower neighbor, e.g. $(p_3[x], p_4[y]).right = p_4$.

Once all the intersection points are computed and linked by their left/right and lower/upper neighbors, we can retrieve the sequence of vertices for each polyomino. We can see that each intersection point has a uniquely corresponding polyomino with the point as its upper right corner. Therefore, for each intersection point g, we find the sequence of vertices forming its corresponding polyomino. The polyominos are either rectangles or half-rectangles with lower left side shaped like steps. Hence we first retrieve g's left neighbor. We then repeatedly find the next lower neighbor and right neighbor until the right neighbor reaches the same y coordinate as the original intersection point g.

Example 4.6. For the intersection point $g_1(p_8[x], p_{10}[y])$, we first find its left vertex $g_2(p_3[x], p_{10}[y])$. We then find the lower vertex $g_3(p_3[x], p_{11}[y])$, and the right vertex $g_4(p_{11}[x], p_{11}[y])$ in the first iteration. Because g_4 is not meeting the grid line at g_1 yet, it continues to find the next lower vertex $g_5(p_{11}[x], 0)$ and the right vertex $g_6(p_8[x], 0)$. Now the algorithm stops as g_6 reaches the y grid line of g_1. The sequence of vertices for the skymino corresponding to g_1 is hence $g_1, g_2, g_3, g_4, g_5, g_6$.

Complexity. The computation of intersection points requires $O(n^2)$ time. Because each grid line segment between two neighboring intersection points

will be used at most twice for constructing skyminos, the skymino constructing step requires $O(n^2)$ time. Therefore, Algorithm 4.4 requires $O(n^2)$ time.

4.4.5 Skyline Diagram of Global Skyline

To compute the skyline diagram of global skyline, we just need to compute the quadrant skyline for the four quadrants independently and then take the union of the results. In this subsection, we show an important result on the lower bound for the number of skyminos in a skyline diagram of global skyline.

Theorem 4.3. (**Lower Bound**). Given a dataset P of n points in the plane, there are at least $(n+1)^2/9$ skyminos if there is no two points sharing the same x and y coordinates.

Proof. We first prove that at most three adjacent skyline cells in the same column can belong to the same skymino. Consider any two skyline cells C_1 and C_2 in the same column, if they belong to the same skymino, i.e., the skyline results for C_1 and C_2 are the same, we will show: 1) there is at most one cell between C_1 and C_2, i.e., there are at most two points (two grid lines) between C_1 and C_2, and 2) there are at most one point on the left of this column and at most one on the right. We prove this by contradiction.

Assume there are two points p_a and p_b on the left of this column at the same time. W.l.o.g., we assume C_2 is lower than C_1, and $p_a[y] > p_b[y]$, for the x coordinate, there are three cases: $p_a[x] < p_b[x]$, $p_a[x] > p_b[x]$, and $p_a[x] = p_b[x]$. For the first case $p_a[x] < p_b[x]$, the skyline in third quadrant for C_1 contains p_a, p_b. However, the skyline in second quadrant for C_2 contains p_b, and p_a cannot be a skyline for C_2 because it is dominated by p_b. The similar analysis applies to the second case. For the third case $p_a[x] = p_b[x]$, the skyline

in third quadrant for C_1 contains p_a. However, the skyline in second quadrant for C_2 contains p_b. Therefore, C_1 and C_2 have different skyline results and do not belong to the same skymino.

By contradiction, we conclude at most three adjacent skyline cells in the same column can belong to the same skymino. Similar analysis applies to the row case. Therefore, at most 3×3 skyline cells can belong to the same skymino. Hence there are at least $(n+1)^2/9$ skyminos. □

4.5 Skyline Diagram of Dynamic Skyline

In this section, we study the algorithms for skyline diagram of dynamic skyline. We first present a baseline algorithm and define an important notion of skyline subcell. Then based on the observation that dynamic skyline query result is a subset of global skyline, we present an improved subset algorithm utilizing the skyline diagram of global skyline. Finally, based on the relationship of the skyline results of neighboring subcells, we present a scanning algorithm with improved complexity.

4.5.1 Baseline Algorithm

Similar to the skyline diagram of quadrant and global skyline, we can first find small regions that are guaranteed to have the same dynamic skyline, and them merge them to form skyline polyominos.

Skyline Subcell. In skyline diagram of quadrant and global skyline, each point contributes a horizontal and vertical grid line to divide the plane into skyline cells which are guaranteed to have the same result for quadrant skyline queries. For dynamic skyline, all points will be mapped to the first quadrant

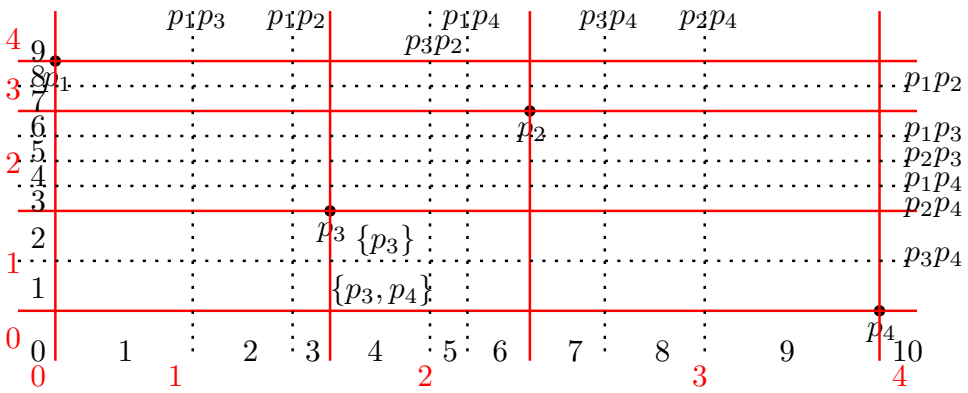


Figure 4.9: Skyline subcells for dynamic skyline (solid grid lines for cells and dotted lines for subcells).

with respect to the query point and may dominate the points who are otherwise global skyline points. Hence the points in the skyline cell are not guaranteed to have the same dynamic skyline. Therefore, to account for mapped points, in addition to the grid lines over each point, we draw a vertical and horizontal bisector line between each pair of points. In total, we have $O\binom{n}{2}$ horizontal lines and $O\binom{n}{2}$ vertical lines which leads to $O(\binom{n}{2}^2)$ regions. Figure 4.9 shows an example with 4 points. The $\binom{4}{2}$ bisector lines between each pair of points and the 4 grid lines over each point divide the plane into 121 regions. We can see that these regions are guaranteed to have the same dynamic skyline, since there are no points or mapped points in each of these regions that would change the dominance relationship of the points. To distinguish with skyline cell for quadrant and global skyline, we name these regions skyline subcells for dynamic skyline. The algorithm for computing skyline subcells is very straightforward as shown in Algorithm 4.5.

Definition 4.7. (**Skyline Subcell**). The vertical and horizontal bisectors of each pair of points divide the plane into skyline subcells. Any query points in the same skyline subcell have the same dynamic skyline.

Algorithm 4.5: The algorithm for computing skyline subcells.

```
   input  : a set of n points.
   output: skyline subcell Sky(SC_{i,j}).
1  for i=0 to n-2 do
2      for j=i+1 to n-1 do
3          p_ip_j[x] = (p_i[x]+p_j[x])/2;
4          p_ip_j[y] = (p_i[y]+p_j[y])/2;
```

Finding skyline for each skyline subcell. Once we have the skyline subcells, we can compute the skyline for each subcell. A baseline algorithm is trivial and similar to the skyline computation for skyline cells as shown in Algorithm 4.6. For each subcell $SC_{i,j}$, it first maps all the points to the first quadrant with respect to the subcell (Line 4-5). It then computes the skyline of the mapped points. Since skyline can be computed in $O(n)$ time if the points are sorted on one dimension, and there are $O(n^4)$ subcells, the entire algorithm (Algorithm 4.6) can be finished in $O(n^5)$.

Algorithm 4.6: The baseline algorithm for skyline diagram of dynamic skyline.

```
   input  : skyline subcells SC_{i,j}.
   output: skyline of each skyline subcell Sky(SC_{i,j}).
1  for i=0 to mx do
2      for j=0 to my do
3          for k=1 to n do
4              p_k[x]' = |p_k[x] − SC_{i,j}[x]|;
5              p_k[y]' = |p_k[y] − SC_{i,j}[y]|;
6          employ skyline algorithm on p'_k for k = 1, ..., n to compute the
           skyline as the output of SC_{i,j};
```

4.5.2 Subset Algorithm

As we discussed earlier, the mapped points may dominate additional points that would have been global skyline points. As a result, the dynamic skyline of each subcell $SC_{i,j}$ is a subset of the global skyline of the skyline cell it belongs to. For example, in Figure 4.9, $Sky(SC_{3,1})$ is a subset of $Sky(C_{1,1})$. Therefore, we can first use the algorithms in previous section to compute the skyline of the skyline cells, and then compute the exact skyline of each subcell from this set rather than the entire n points. The detailed algorithm is shown in Algorithm 4.7 which is very similar to the baseline algorithm. The only difference is we just need to consider the output of skyline results of each skyline cell rather than the entire n points. Although the worst case time complexity is the same as the baseline algorithm $O(n^5)$, on average, the number of skyline for n points is only $O(\log n)$. Therefore, the amortized time complexity for the subset algorithm is reduced to $O(n^4 \log n)$. We will show that the subset algorithm is indeed significantly faster than the baseline algorithm in practice in Section 4.6.

Algorithm 4.7: The subset algorithm for skyline diagram of dynamic skyline.

input : global skyline result of each skyline cell $Sky(C_{i,j})$.
output: dynamic skyline result of each skyline subcell $Sky(SC_{i,j})$.

```
1 for k=0 to mx do
2     for l=0 to my do
3         find C_{i,j} such that SC_{k,l} ∈ C_{i,j};
4         Sky(SC_{k,l}) = dynamic skyline of the points in Sky(C_{i,j})
```

4.5.3 Scanning Algorithm

The baseline and subset algorithms compute the skyline for each subcell from scratch. To further improve the efficiency, in this subsection, we propose an in-

cremental scanning algorithm based on the relationship of the dynamic skyline results of neighboring subcells. This is due to the observation that as we move from one subcell to its neighboring subcell on the right, the only difference of the skyline result is caused by the two points that contributed to the bisector line between the two subcells. We just need to consider these two points in addition to the skyline result of the previous subcell. So similar to the scanning algorithm for quadrant skyline queries, we first compute $Sky(SC_{0,0})$ for the lower left subcell. We then scan the subcells from left to right on the first row and compute the skyline incrementally. We then compute each of the remaining rows from bottom up. The detailed algorithm is shown in Algorithm 4.8.

Example 4.7. In Figure 4.9, $Sky(SC_{4,2}) = \{p_3\}$, for $SC_{4,1}$, we only need to check $\{p_3\} \cup \{p_3, p_4\} = \{p_3, p_4\}$. Because p_3, p_4 cannot dominate each other, therefore, $Sky(SC_{4,1}) = \{p_3, p_4\}$.

Algorithm 4.8: The scanning algorithm for skyline diagram of dynamic skyline.

input : a set of n points and skyline subcells $SC_{i,j}$.
output: skyline of each skyline subcell $Sky(SC_{i,j})$.

```
1 employ skyline algorithm to compute the skyline of Subcell SC_{0,0};
2 for i=1 to mx do
3 |   Sky(SC_{i,0}) = Sky(SC_{i-1,0}) ⋃ the points contributing to the bisectors
  |   between SC_{i-1,0} and SC_{i,0};
4 for i=0 to mx do
5 |   for j=1 to my do
6 |   |   Sky(SC_{i,j}) = skyline from Sky(SC_{i,j-1}) ⋃ the points
  |   |   contributing to the bisectors between SC_{i,j-1} and SC_{i,j};
```

The key step in the above algorithm is to compute the updated skyline given the skyline result of the previous cell and the new points contributing

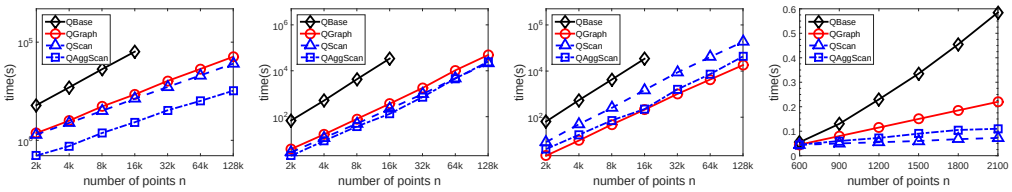
to the bisectors (line 3 and line 8). When adding a new point, there are two cases: 1) the new point becomes a skyline point which may dominate some existing skyline points, or 2) the new point is dominated by existing skyline points. To determine if the new point is dominated by existing skyline points, we can do a binary search to find the skyline point p_i such that $p_i[x] \leqslant p[x]$ and $p[x] \leqslant p_{i+1}[x]$. If $p_i[y] \geqslant p[y]$, the new point is a skyline point, otherwise, the new point is dominated by p_i. This can be finished in $O(\log n)$ time. If the new point is a skyline point, we need to remove those points dominated by the new point. If we sort the skyline points in ascending order on x-axis and descending order on y-axis, we can delete those points in $O(\log n)$ time.

Complexity..

Since the computation of updated skyline for each subcell only costs $O(\log n)$ time, and there are $O(n^4)$ subcells, the overall worst case time complexity for the scanning algorithm (Algorithm 8) is $O(n^4 \log n)$.

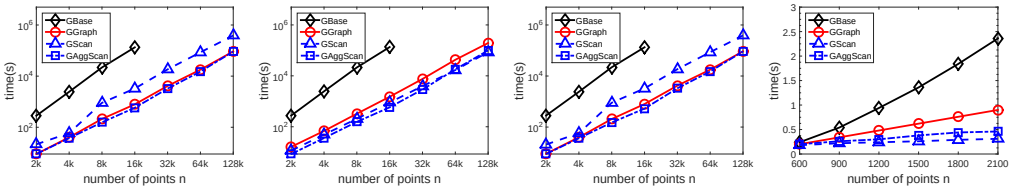
4.6 Experiments

In this section, we present experimental studies evaluating our proposed algorithms .



(a) time cost of CORR (b) time cost of INDE (c) time cost of ANTI (d) time cost of NBA

Figure 4.10: The impact of $n$on skyline diagram of quadrant skyline queries.



(a) time cost of CORR (b) time cost of INDE (c) time cost of ANTI (d) time cost of NBA

Figure 4.11: The impact of $n$on skyline diagram of global skyline queries.

4.6.1 Experiment Setup

We first evaluate the algorithms for computing skyline diagram of quadrant and global skyline, and then the algorithms for dynamic skyline. Since this is the first work for skyline diagram with the new definition, our performance evaluation was conducted against the baseline algorithms. We implemented all algorithms in Python. We ran experiments on 1) a computation server with Intel Xeon E5-4627 v3 $*$ 4 CPUs with 1024GB RAM running Ubuntu 14.04 for parallel implementations, and 2) a desktop with Intel Core i7 running Ubuntu 14.04 with 64GB RAM for serial implementations.

We compare the following four algorithms for skyline diagram of quadrant and global skyline, respectively.

- QBase/GBase: Baseline algorithm
- QGraph/GGraph: Skyline graph algorithm
- QScan/GScan: Scanning algorithm
- QAggScan/GAggScan: Aggressive scanning algorithm

We compare the following three algorithms for skyline diagram of dynamic skyline.

- DBase: Baseline algorithm

- DSubset: Subset algorithm
- DScan: Scanning algorithm

We used both synthetic datasets and a real NBA dataset in our experiments. To study the scalability of our methods, we generated independent (INDE), correlated (CORR), and anti-correlated (ANTI) datasets following the seminal work [7]. To study the main features of the algorithms on different dataset distributions, for the synthetic datasets, we make sure no two points lie on the same x-axis or y-axis. We also built a dataset[2] that contains 2384 NBA players who are league leaders of playoffs. Each player has two attributes (Points and Rebounds) that measure the player's performance. To study the effect of dataset domain sizes, we generated datasets with small domain size (10^4) and large domain size 10^8.

4.6.2 Skyline Diagram of Quadrant Skyline

In this subsection, we report the results for computing skyline diagram of quadrant skyline queries.

Figures 4.10(a)(b)(c) present the time cost of QBase, QGraph, QScan, and QAggScan with varying number of points n for the three synthetic datasets. The results of QBase algorithm on CORR, INDE, and ANTI dataset are almost the same which means the data distribution has no impact on baseline algorithm. We did not report the result of the baseline algorithm in some figures due to the high cost when n is big. All the proposed algorithms scale well with the increasing number of points.

We first examine each algorithm and compare its performance on difference

[2]Extracted from http://stats.nba.com/leaders/alltime/?ls=iref:nba:gnav on 04/15/2015.

datasets. For the QGraph algorithm, the time on INDE dataset is higher than CORR and ANTI datasets. This is because the number of links between parent and children nodes in the directed skyline graph is larger for INDE dataset. For the QScan algorithm, the time on ANTI dataset is much higher than INDE dataset which is much higher than CORR dataset. This is because the number of skyline in each cell in ANTI dataset is much more than INDE and CORR datasets. Therefore, it requires more time to do the multiset operation on ANTI dataset. For the QAggScan algorithm, it is much faster than QGraph and QScan on CORR dataset because there are much fewer intersections thus fewer polyominos on CORR dataset. However, the performance of AggScan is not so good on ANTI dataset due to the huge number of intersections on ANTI dataset.

Comparing different algorithms, QGraph, QScan, and QAggScan significantly outperform QBase, which validates the effectiveness of our algorithms. QAggScan outperforms QScan on all datasets thanks to its combined steps of finding skyline polyominos directly (but we will see an opposite result on real NBA dataset later). For CORR and INDE datasets, QAggScan is the most efficient out of all algorithms, while for ANTI dataset, QGraph has the best performance due to the reason we explained earlier.

Figure 4.10(d) reports the time cost of QBase, QGraph, QScan, and QAggScan with varying number of points n for the real NBA dataset. The difference between synthetic datasets and NBA dataset is that there are some points sharing the same x-axis or y-axis on NBA dataset which leads to fewer cells. Herein, the time cost of 2100 points on NBA is significantly smaller than that of 2000 points on synthetic datasets. Comparing different algorithms, the performances of QScan and QAggScan are similar and QScan is a little better

than QAggScan which is opposite to the performances on synthetic datasets. The reason is that on NBA dataset, the number of cells is much smaller but the number of intersections is similar. But both QScan and QAggScan outperform QGraph.

4.6.3 Skyline Diagram of Global Skyline

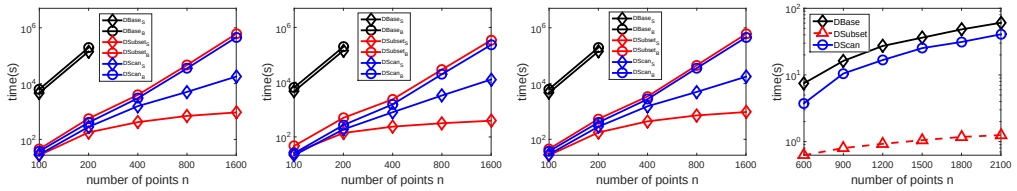
In this subsection, we report the experimental results for computing skyline diagram of global skyline queries.

Figures 4.11(a)(b)(c)(d) present the time cost of GBase, GGraph, GScan, and GAggScan with varying number of points n for the three synthetic datasets and the NBA dataset. The performances of all algorithms on CORR and ANTI dataset are similar. The reason is that for global skyline queries, we compute the quadrant skyline from four quadrants. CORR (ANTI) dataset is correlated (anti-correlated) on the first and third quadrant while ANTI (CORR) dataset is correlated (anti-correlated) on the second and fourth quadrant. The total time cost of global skyline queries is about four times of quadrant skyline queries on INDE dataset. For CORR and ANTI datasets, the total time of global skyline queries is about twice of the sum of quadrant skyline on CORR and ANTI datasets.

Comparing different algorithms, on CORR and ANTI datasets, the performance of GAggScan is similar to GGraph, both are better than GScan. For INDE dataset, the performances of GScan and GAggScan are similar and both are better than GGraph. In summary, GAggScan is the best for all datasets. For NBA dataset, the time cost of global skyline is about 5 times of the quadrant skyline which suggests that the real NBA dataset is somewhat correlated.

4.6.4 Skyline Diagram of Dynamic Skyline

In this subsection, we report the experimental results for computing skyline diagram of dynamic skyline. To study the impact of dataset domain, we run our dynamic skyline experiments on two datasets with domain size 10^4 and 10^8, respectively. A bigger domain means a smaller probability of bisectors of any two points overlapping with each other, hence a larger number of skyline subcells.



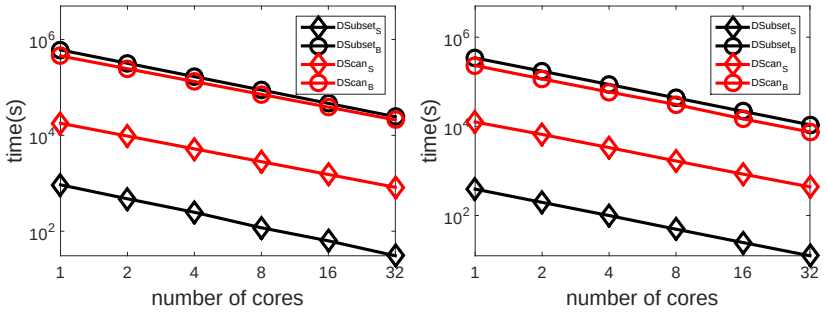
(a) time cost of CORR (b) time cost of INDE (c) time cost of ANTI (d) time cost of NBA

Figure 4.12: The impact of $n$on skyline diagram of dynamic skyline.

Figures 4.12(a)(b)(c)(d) present the time cost of DBase, DSubset, and DScan with varying number of points n for the three synthetic datasets and the NBA dataset. Similar to the global skyline, all algorithms have the same performances on CORR and ANTI datasets. Again, DSubset and DScan significantly outperform DBase, which verifies the effectiveness of our algorithms. For the dataset with smaller domain size, DSubset significantly outperforms DScan. For the dataset with bigger domain, DScan is better than DSubset due to the larger number of subcells. The time cost of all algorithms on NBA dataset is much smaller than synthetic datasets because there are a large number of points on the same x-coordinate and y-coordinate which leads to a small number of subcells.

4.6.5 Performance Improvements through Parallel Implementation

In order to reduce the skyline query processing time, we demonstrate that our algorithms can be parallelized by dividing and assigning tasks to each core. For subset algorithm, we distribute the workload of computing the global skyline for each cell evenly. Similarly, for the scanning algorithm, after determining the points contributing to the bisectors, we assign the computation of different cells to different cores. We run the experiments on the computation server with 40 cores and the results are shown in Figures 4.13(a)(b). The figures show the parallel algorithms are much more efficient and provide almost linear speedups.



(a) time cost of CORR/ANTI. (b) time cost of INDE.

Figure 4.13: The impact of parallel.

4.7 Conclusions

In this chapter, for the first time, we proposed a novel concept called skyline diagram. Given a set of points, it partitions the plane into a set of skyline polyominos where query points in each polyomino have the same skyline query results. We studied skyline diagram for three kinds of skyline queries, namely, quadrant, global, and dynamic skyline. We then presented several efficient

algorithms to compute the skyline diagram motivated by interesting properties of skyline. Each algorithm illustrates its corresponding advantages and disadvantages based on different distributions of input datasets. The experimental results on the real NBA dataset and the synthetic datasets showed our algorithms are efficient and scalable.

As future work, we are interested in implementing the skyline diagram based solutions for the various applications, including PIR based skyline queries, reverse skyline queries, and authentication of skyline queries.

Chapter 5

Conclusion and Future Directions

In this dissertation, We illustrated a state-of-the-art output-sensitive skyline computation algorithm and a precomputation structure, skyline diagram, to efficiently compute skyline. For finding Pareto optimal groups, we adapted the original skyline definition to the group level which captures the quintessence of original skyline definition.

For future work, we list several potential directions as follows.

- k-nearest neighbors queries can be used for classification, and kNN classification is one of the most popular classification algorithm due to its simplicity. Similar to kNN classification, can we adapt a "skyline classification"? And what are the advantages and disadvantages of skyline classification?

- Differential privacy has recently become a de facto standard for private computation. How to design a differential private skyline queries algorithm is significant.

Bibliography

[1] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman. Parallel skyline queries. In *ICDT*, pages 274–284, 2012.

[2] J. L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.

[3] J. L. Bentley, K. L. Clarkson, and D. B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. In *SODA*, pages 179–187, 1990.

[4] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. *J. ACM*, 25(4):536–543, 1978.

[5] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[6] H. Blunck and J. Vahrenhold. In-place algorithms for computing (layers of) maxima. In *Algorithm Theory - SWAT 2006*, pages 363–374, 2006.

[7] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[8] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *SIGMOD*, pages 503–514, 2006.

[9] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16(4):361–368, 1996.

[10] T. M. Chan and P. Lee. On constant factors in comparison-based geometric algorithms and data structures. In *Symposium on Computational Geometry*, page 40, 2014.

[11] B. Chazelle and H. Edelsbrunner. An improved algorithm for constructing k th-order voronoi diagrams. *IEEE Trans. Computers*, 36(11):1349–1354, 1987.

[12] M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang. A safe zone based approach for monitoring moving skyline queries. In *EDBT*, pages 275–286, 2013.

[13] L. Chen and X. Lian. Dynamic skyline queries in metric spaces. In *EDBT*, pages 333–343, 2008.

[14] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–719, 2003.

[15] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 41–50, 1995.

[16] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Summarizing two-dimensional data with skyline-based statistical descriptors. In *SSDBM*, pages 42–60, 2008.

[17] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, pages 291–302, 2007.

[18] X. Ding, X. Lian, L. Chen, and H. Jin. Continuous monitoring of skylines over uncertain data streams. *Inf. Sci.*, 184(1):196–214, 2012.

[19] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB J.*, 16(1):5–28, 2007.

[20] X. Hu, C. Sheng, Y. Tao, Y. Yang, and S. Zhou. Output-sensitive skyline algorithms in external memory. In *SODA*, pages 887–900, 2013.

[21] Z. Huang, H. Lu, B. C. Ooi, and A. K. H. Tung. Continuous skyline queries for moving objects. *IEEE Trans. Knowl. Data Eng.*, 18(12):1645–1658, 2006.

[22] H. Im and S. Park. Group skyline computation. *Inf. Sci.*, 188:151–169, 2012.

[23] M. S. Islam, R. Zhou, and C. Liu. On answering why-not questions in reverse skyline queries. In *ICDE*, pages 973–984, 2013.

[24] D. G. Kirkpatrick and R. Seidel. Output-size sensitive algorithms for finding maximal vectors. In *Symposium on Computational Geometry*, pages 89–96, 1985.

[25] H. Köhler, J. Yang, and X. Zhou. Efficient parallel skyline processing using hyperplane projections. In *SIGMOD*, pages 85–96, 2011.

[26] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[27] M. Lee and S. Hwang. Continuous skylining on volatile moving data. In *ICDE*, pages 1568–1575, 2009.

[28] C. Li, N. Zhang, N. Hassan, S. Rajasekaran, and G. Das. On skyline groups. In *CIKM*, pages 2119–2123, 2012.

[29] X. Lian and L. Chen. Reverse skyline search in uncertain databases. *ACM Trans. Database Syst.*, 35(1), 2010.

[30] X. Lin, J. Xu, and H. Hu. Authentication of location-based skyline queries. In *CIKM*, pages 1583–1588, 2011.

[31] X. Lin, J. Xu, and H. Hu. Range-based skyline queries in mobile environments. *IEEE Trans. Knowl. Data Eng.*, 25(4):835–849, 2013.

[32] X. Lin, J. Xu, H. Hu, and W. Lee. Authenticating location-based skyline queries in arbitrary subspaces. *IEEE Trans. Knowl. Data Eng.*, 26(6):1479–1493, 2014.

[33] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *ICDE*, pages 86–95, 2007.

[34] J. Liu, L. Xiong, J. Pei, J. Luo, and H. Zhang. Finding pareto optimal groups: Group-based skyline. *PVLDB*, 8(13):2086–2097, 2015.

[35] J. Liu, L. Xiong, and X. Xu. Faster output-sensitive skyline computation algorithm. *Inf. Process. Lett.*, 2014.

[36] J. Liu, J. Yang, L. Xiong, and J. Pei. Secure skyline queries on cloud platform. *ICDE*, 2017.

[37] J. Liu, H. Zhang, L. Xiong, H. Li, and J. Luo. Finding probabilistic k-skyline sets on uncertain data. In *CIKM*, pages 1511–1520, 2015.

[38] H. Lu, C. S. Jensen, and Z. Zhang. Flexible and efficient resolution of skyline query size constraints. *IEEE Trans. Knowl. Data Eng.*, 23(7):991–1005, 2011.

[39] F. Luccio and F. Preparata. On finding the maxima of a set of vectors. *Instituto di Scienze dell'Informazione, Università di Pisa, Corso Italia*, 40:56100, 1973.

[40] M. Magnani and I. Assent. From stars to galaxies: skyline queries on aggregate data. In *EDBT*, pages 477–488, 2013.

[41] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

[42] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[43] Y. Park, J. Min, and K. Shim. Parallel computation of skyline and reverse skyline queries using mapreduce. *PVLDB*, 6(14):2002–2013, 2013.

[44] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *VLDB*, pages 15–26, 2007.

[45] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *VLDB*, pages 253–264, 2005.

[46] J. Pei, Y. Yuan, X. Lin, W. Jin, M. Ester, Q. Liu, W. Wang, Y. Tao, J. X. Yu, and Q. Zhang. Towards multidimensional subspace skyline analysis. *ACM Trans. Database Syst.*, 31(4):1335–1381, 2006.

[47] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *VLDB*, pages 751–762, 2006.

[48] M. Sharifzadeh and C. Shahabi. Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries. *PVLDB*, 3(1):1231–1242, 2010.

[49] C. Sheng and Y. Tao. On finding skylines in external memory. In *PODS*, pages 107–116, 2011.

[50] Y. Tao, L. Ding, X. Lin, and J. Pei. Distance-based representative skyline. In *ICDE*, pages 892–903, 2009.

[51] Y. Tao, X. Xiao, and J. Pei. Efficient skyline and top-k retrieval in subspaces. *IEEE Trans. Knowl. Data Eng.*, 19(8):1072–1088, 2007.

[52] G. Wang, J. Xin, L. Chen, and Y. Liu. Energy-efficient reverse skyline query processing over wireless sensor networks. *IEEE Trans. Knowl. Data Eng.*, 24(7):1259–1275, 2012.

[53] L. Wang, X. Meng, H. Hu, and J. Xu. Bichromatic reverse nearest neighbor query without information leakage. In *Database Systems for Advanced Applications - 20th International Conference, DASFAA 2015, Hanoi, Vietnam, April 20-23, 2015, Proceedings, Part I*, pages 609–624, 2015.

[54] M. L. Yiu, E. Lo, and D. Yung. Authentication of moving knn queries. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 565–576, 2011.

[55] N. Zhang, C. Li, N. Hassan, S. Rajasekaran, and G. Das. On skyline groups. *IEEE Trans. Knowl. Data Eng.*, 26(4):942–956, 2014.

[56] W. Zhang, X. Lin, Y. Zhang, W. Wang, and J. X. Yu. Probabilistic skyline operator over sliding windows. In *ICDE*, pages 1060–1071, 2009.