# Directional Skyline Queries

Eman El-Dawy, Hoda M.O. Mokhtar, and Ali El-Bastawissy

Faculty of Computers and Information, Cairo University, Cairo, Egypt
{emiomar2000,hmokhtar)@gmail.com, alibasta@hotmail.com

**Abstract.** Continuous monitoring of queries over moving objects has become an important topic as it supports a wide range of useful mobile applications. A continuous skyline query involves both static and dynamic dimensions. In the dynamic dimension, the data object not only has a distance from the query object, but it also has a direction with respect to the query object motion. In this paper, we propose a direction-oriented continuous skyline query algorithm to compute the skyline objects with respect to the current position of the user. The goal of the proposed algorithm is to help the user to retrieve the best objects that satisfy his/her constraints and fall either in any direction around the query object, or is aligned along the object's direction of motion. We also create a precomputed skyline data set that facilitates skyline update, and enhances query running time and performance. Finally, we present experimental results to demonstrate the performance and efficiency of our proposed algorithms.

**Keywords:** Skyline queries, continuous query processing, moving object databases, direction-based skyline queries.

## 1 Introduction

With the rapid advances in wireless communications and technologies for tracking the positions of continuously moving objects, algorithms for efficiently answering queries about large numbers of moving objects are increasingly needed [2, 6]. This meets the requirements for location based services (LBS) [2, 14]. A moving object is simply defined as an object whose location and/or geometry changes continuously over time [13]. Consequently, a moving object database is a database that is designed to efficiently handle the huge amount of location data, and also to efficiently manage, and query location information. The main difference between moving object databases and traditional databases is that traditional databases are best suited for static data while moving object databases were designed for dynamic data (i.e. data continuously changing over time). Furthermore, moving object databases are tailored for high frequency of updates that is a common consequence of the rapidly changing location information [13]. These new types of data (i.e. location data) generated new types of queries that query the spatial and/or temporal properties of moving objects. Skyline queries are an important class of queries that target applications involving multi-criteria decision making. In general, a skyline query is defined as: Given a set of multi-dimensional data points, a skyline query retrieves a set of data points that are not

Y. Xiang et al. (Eds.): IDCS 2012 and ICDKE 2012, LNCS 7646, pp. 302–315, 2012.

dominated by any other points in all dimensions. A skyline query finds a set of interesting objects, satisfying a set of possibly conflicting conditions.

In our proposed algorithms we evaluate the skyline objects based on both distance and direction with respect to the query point "QP". The basic idea is as follows: for a given query point QP, we propose two approaches, the first one is an *all directional continuous skyline query algorithm* to find the interesting objects around the user in all directions, the second one is a *my direction continuous skyline query* that finds the interesting objects to the user that **only** fall along his motion direction.

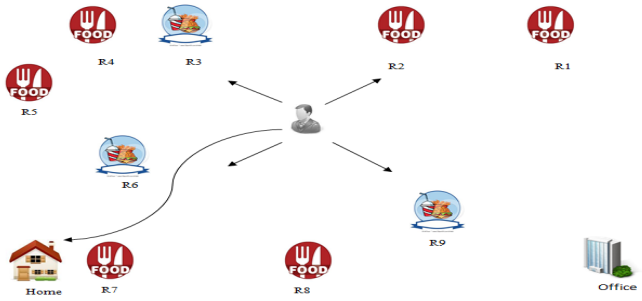


**Fig. 1.** Motivation example

Example 1: Consider the example illustrated in Fig. 1. Suppose a person wants to take lunch before going home, and he searches for the best restaurant in terms of price, rank, and distance. In this example there are 2 static attributes (price, rank), and a dynamic attribute (distance). And in his way he found that he forgot an important thing in his office and decided to go back to his office.

Considering first the static skyline as shown in Fig. 2, the person wants to take his lunch and he searches for the preference restaurants in terms of price and rank and he found that restaurants R2, R4 and R6 are interesting and can be retrieved by a skyline query. Thus, if we have a point "p" we can see if it dominates another point "q", then, "p" dominates "q" if it is better or as good as "q" on all considered directions.

Suppose now we also consider the dynamic attributes, in this example we have two types of dynamic attributes: *distance and motion direction* of the query object. Consider the example shown in Fig. 3, in this example we have 6 nearest neighbor objects (R7,R8,R9,R10,R11,R12) around the query object "QP" located at (0, 0) and sorted by the distance with respect to QP.

In this example we have vectors $\vec{R7}$ ,…, $\vec{R12}$ originating from QP.

We assume that two vectors are in the same direction if the included angle between the two objects is smaller than a predefined threshold $\theta_{threshold}$. In this paper we fix the value of $\theta_{threshold}$ to be $\pi/3$. Consequently, the object R7 is in the same direction as R8 as the included angle between the two vectors R7 and R8 is equal to $26^{o}$. In addition, R7 is closer to QP than R8, thus R7 dominates the other objects in the same direction (i.e. object R8) and will be one of the recommended objects to the user.

Similarly, we can say that R9 dominates R11, and R10 dominates R12, and thus R7, R9, R10 are not dominated by other objects. Hence, the result of the dynamic skyline part is {R7, R9, and R10} w.r.t. the query object QP, and it forms the result for the candidate objects located all around the query object (i.e. in all directions). However, when we consider only the objects that QP meets along its motion (as he moves towards the East direction), we can find that the point R7 is the only candidate for the dynamic skyline query (i.e. in one direction, the East). In the remaining of the paper we show the details of our proposed direction-based skyline evaluation algorithms.

| Restaurant | Price | Rank |
|---|---|---|
| 1 | 6 | 4 |
| 2 | 5 | 1 |
| 3 | 5 | 6 |
| 4 | 3 | 2 |
| 5 | 2 | 6 |
| 6 | 1 | 3 |

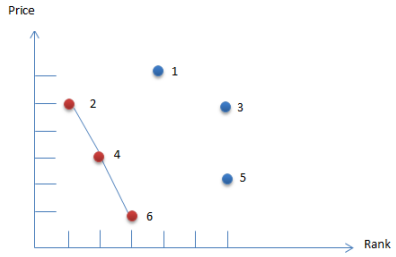


**Fig. 2.** Example of static skyline

| Restaurant® | D® | Dir® |
|---|---|---|
| R7 | 22 | 27° |
| R8 | 5 | 53° |
| R9 | 54 | 158° |
| R10 | 6 | 270° |
| R11 | 67 | 117° |
| R12 | 72 | 236° |

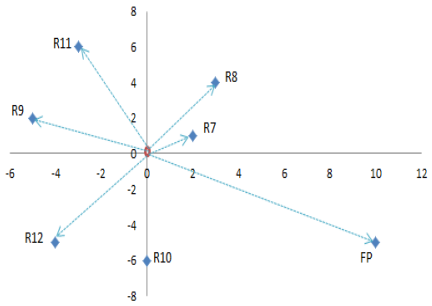


**Fig. 3.** Example of dynamic skyline

Generally, the crucial questions in skyline queries are: How to retrieve the best results to the user with respect to both the static attributes and the dynamic attributes in all directions, and/or in his direction? Also, it is interesting to know how to update the results of the query in the most efficient way. In summary, the contributions of this work can be enumerated as follows:

1. We propose two algorithms: the *all directional-based continuous skyline query* (All_Skyline), and *the my direction continuous skyline query* (My_Dir_Skyline) algorithms which not only retrieve the objects satisfying the user requirement criteria, but also consider the direction of motion of the user in order to return to the user the objects that are both satisfying his requirements and are along his route.

2. We enhanced the computation of the directional skyline technique proposed in [5] by using an additional filter based on the furthest point (FP) introduced in [4].
3. We efficiently update the query result by creating a pre-computed set of candidate objects to facilitate updating the result and consequently enhancing query running time and performance following a similar approach to the one proposed in [4].
4. We compare our work to the directional spatial skyline (DSS) algorithm presented in [5]. To have a fair comparison we changed the DSS algorithm to query the static attributes as well rather than considering the dynamic attributes only.

The rest of this paper is organized as follows: In section 2, we describe the related work. In section 3, we propose our solutions for directional skyline queries. The experimental results are presented in section 4. Finally, section 5 concludes and proposes directions for possible future work.

## 2 Related Work

For their importance, skyline queries were targeted in many works as an essential query type for many applications. The authors in [1] propose two algorithms for the skyline problem namely, a straightforward non-progressive Block-Nested-Loop (BNL), and a Divide-and-Conquer (DC) algorithm. In the BNL approach they recursively compares each data point with the current set of candidate skyline points, which might be dominated later. The DC approach divides the search space and evaluates the skyline points from its sub-regions, respectively, followed by merge operations to evaluate the final skyline points. Both algorithms may incur several iterations, and are inadequate for on-line processing. In [10], two progressive algorithms are presented: the bitmap approach and the index method. Bitmap encodes dimensional values of data points into bit strings to speed up the dominance comparisons. The index method classifies a set of d-dimensional points into d-lists, which are sorted in increasing order of the minimum coordinate. The index scans the lists synchronously from the first entry to the last one. With the pruning strategies, the search space is reduced. In [11], the authors develop BBS (branch-and-bound skyline), a progressive algorithm also based on nearest neighbor search, which is I/O optimal. The authors in [9] introduce the concept of Spatial Skyline Queries (SSQ). Where, given a set of data points P and a set of query points Q, SSQ retrieves those points of P which are not dominated by any other point in P considering their derived spatial attributes with respect to query points in Q. The authors in [12] expand the skyline query concept and propose continuous skyline query which involves not only static dimensions but also the dynamic ones. In this paper the authors use spatio-temporal coherence to refer to those spatial properties that do not change abruptly between continuous temporal scenes. The positions and velocities of moving points do not change by leaps between continuous temporal scenes, which enable them to maintain the changing skyline incrementally. In [7], the authors propose the ESC algorithm, an Efficient update approach for Skyline Computations, where objects with dynamic dimensions move in an unrestricted manner, which creates a pre-computed second skyline set that facilitates an efficient and incremental skyline update strategy and results in a quicker

response time. In [3], the authors present a framework for processing predictive skyline queries for moving objects. The authors present two schemes, namely RBBS (BBS with Rescanning and Repacking) and TPBBS (time-parameterized R-tree with Non- Spatial dimensions), to answer predictive skyline queries. In [5], the authors propose a direction-based spatial skyline(DSS) queries to retrieve nearest objects around the user from different directions and they don't take into account non-spatial attributes. In [14] they extend the work in paper [5] by augmenting road networks. Finally, in [4] the authors present a Multi-level Continuous Skyline Query algorithm(MCSQ), which basically creates a pre-computed skyline data set that facilitates skyline update, and enhances query running time and performance.

Inspired by the importance of skyline queries, in this work with follow the same update approach presented in [4] and augment it with an additional selection criteria that is based on the use of direction of motion of the query object as in [5].

# 3 Directional Skyline Query

## 3.1 Problem Definition

In Location Based Services (LBS) most queries are continuous queries. Unlike snapshot queries (instantaneous queries) that are evaluated only once, continuous queries require continuous evaluation as the query results vary with the change of location and/or time. In the same sense, continuous skyline query processing has to recompute the skyline when the query object and/or the databases objects change their locations. However, updating the skyline of the previous moment is more efficient than conducting a snapshot query at each moment.

In this paper we consider 2-dimensional data (both the data points which are static data points, and the query point that is a moving point), and the direction of the dynamic points in skyline query w.r.t. the query point QP, as we need to get the nearest objects around QP from different directions, as well as the objects in the direction of QP. We focus on moving query points, where the query point changes its location over time. We assume we are given a threshold angle $\theta_{threshold}$ (that represents the accepted deviation angle between the query point and the data point) specified at query time. We treat time as a third dimension for the moving point. In table 1, we summarize the symbols used in this paper.

**Table 1.** Symbols and description

| Symbol | Description |
|---|---|
| $\vec{p}_i$ | Vector between QP and $p_i$ |
| $\theta_{threshold}$ | Acceptable angle of deviation around QP |
| $D(p_i)$ | Distance between QP and $p_i$ |
| FP | Furthest point in the static skyline |

**Table 1.** (*continued*)

| | |
|---|---|
| $Dir_{pi}$ | Direction of $\vec{p}_i$ :the angle between $\vec{p}_i$ and(1,0) |
| $\lambda_{ij}(\lambda_{pipj})$ | Included angle between $\vec{p}_i$ and $\vec{p}_j$ |
| $\varphi_{ij}(\varphi_{pipj})$ | Partition angle between $\vec{p}_i$ and $\vec{p}_j$ |
| $S_{static}$ | Static part in the skyline query result |
| $S_{dynamic}$ | Dynamic part in the skyline query result |

The included angle between $\vec{p}_i$ and $\vec{p}_j$ is denoted by $\lambda_{pipj}$ ($\lambda_{ij}$ for short). $\lambda_{ij}$ should be in the range $0 \leq \lambda_{ij} \leq \pi$, and is defined by the following equation as proposed in [5]:

$$\lambda_{ij} = \arccos \frac{\vec{p}_i \cdot \vec{p}_j}{|\vec{p}_i| . |\vec{p}_j|} \tag{1}$$

We use Equation (1) to define objects alignment (in same direction) as follows.

**Definition 1 (Same Direction).** Given two data objects $p_i$ and $p_j$, and a moving query point QP. $p_i$ and $p_j$ are in the same direction w.r.t QP if $0 \leq \lambda_{ij} \leq \theta_{threshold}$.

Thus, in Fig. 3, if $\theta_{threshold}$ is given as $\pi/3$, object R8 is in the same direction as object R7 w.r.t QP since the included angle $\lambda_{R7R8}$ between their vectors is less than $\theta_{threshold}$ ($\lambda_{R7R8} = 27^o$). On the other hand object R7 is on a different direction than object R9 w.r.t QP as $\lambda_{R7R9} > \theta_{threshold}$.

Since in this work we deal with moving query points, we consider the distance function to be the traditional *Euclidean* distance for its simplicity and applicability to our work. Thus the distance between any 2 points $p_1(x_1, y_1), p_2(x_2, y_2)$ at time instant $t_i$ is given by:

$$D(p_1, p_2) = \sqrt{(x_1(t_i) - x_2(t_i))^2 + (y_1(t_i) - y_2(t_i))^2} \tag{2}$$

**Definition 2 (Dominate).** Given a query point QP, a threshold angle $\theta_{threshold}$, and two data points $p_1, p_2$. If the distance $D(Qp, p_1) \leq D(Qp, p_2)$ and $p_1, p_2$ are in the same direction w.r.t QP, this implies that $p_1$ *dominates* $p_2$.

In this paper we consider the evaluation of continuous skyline queries while moving, and hence the queries involve distance, direction, along with other static dimensions. Consequently, such queries are dynamic due to the change in the spatial variables. In our solution, we only compute the initial skyline at the start time $t_0$. Subsequently, continuous query processing is conducted for each user by updating the skyline instead of computing a new one from scratch each time. Without loss of generality, we restrict our discussion to what follows the MIN skyline annotation, in which smaller values of distance are preferred in our comparison to determine the dominance between two

points, and we assume that our data set points are static and the query object is the only moving object.

### 3.2 Directional Multi-level Continuous Skyline Query (DCSQ)

In this section we discuss our proposed directional continuous skyline update algorithms. Our algorithms distinguish the data points that are permanently in the skyline and use them to derive a search bound as we compute the static skyline points ($S_{static}$). To proceed in our algorithm we first need to distinguish the data point which is the furthest point (FP) in the $S_{static}$ points, we use the same approach presented in [4]. In general, the FP point is the static data point in the skyline which has the maximum distance to the query point; assume we have a point "P", its predecessor is "$P^-$", and it's successor is"$P^+$"; so to enter the point P in a dynamic skyline ($S_{dynamic}$) it must have an advantage over FP in their distance function value. We used $\lambda$ as the included angle between two points and $\theta_{threshold}$ as an acceptable angle; if the angle between the predecessor of the point P and the angle between the successors of P is greater than $\theta_{threshold}$ then P will be entered into the $S_{dynamic}$ part of the skyline.

Fig. 12 shows how we can compute All_Skyline in all directions with respect to QP, and Fig. 13 shows how we can compute My_Dir_Skyline only in the direction of QP. This characteristic of the FP point and the acceptable angle $\theta_{threshold}$ enables us to use them as a filter for $S_{dynamic}$ points before entering to the continuous skyline query(i.e. before considering the full skyline = $S_{static}$+ $S_{dynamic}$).

**Definition 3 (*Directional Multi-level Continuous Skyline Query (DCSQ)*).** A DCSQ *with M levels* is a continuous skyline query that creates a pre-defined data set for the dynamic part in the skyline such that the number of the pre-defined data sets depends on the value of M, and then uses these data sets to update the pervious skyline set and consequently updating the first skyline.

In brief, our algorithm proceeds as follows: Initially, we classify skyline into static skyline ($S_{static}$), and dynamic skyline ($S_{dynamic}$). Then, we compute static skyline only once at the beginning, and at every time instance we basically update the distance function "D" and the direction "$dir_R$" with respect to the new location of the query point and the given deviation tolerance $\theta_{threshold}$. Next, after we compute the static skyline ($S_{static}$) we create a view for the data set with the data except the pre-considered static points in the skyline, so the dynamic skyline part will not rescan static skyline points and consequently returns the new result in faster response time as justified in [4]. Finally, the DCSQ computation depends on pre-computation of the set of dynamic data points which will be used to update the previous result set and yields faster response time.

In the remaining of this section we technically present our proposed algorithm in more details. We basically propose two algorithms:

**First: All_Skyline Algorithm**

- First we compute the dominating points with respect to the non-spatial attributes. Then, we use the resulting set ($S_{static}$) to compute the furthest point (FP).
- Using the FP as a filter for the data points, we compare the distance between QP and each data point, if less than the distance to FP, then this data point might be in

the 1st level of the dynamic skyline denoted by "$S1_{dynamic}$". Thus, at any time if we find that there exist a data point whose distance to the query object is greater than the distance of the furthest point (i.e. D (FP) < D (p)), we can just terminate checking the remaining points as we keep them sorted by distance.

- We keep checked points in a list denoted by "eval_obj", then, we compute for each point (p) in "eval_obj" the angle between it and its predecessor ($p^-$) point and the angle with its successor point ($p^+$) as points are put in a circular list. If the angle between p- and p+ (denoted by the partition angle φ) is greater than the given acceptable deviation angle $\theta_{threshold}$, then we add the point 'p' to $S1_{dynamic}$.
- We save the difference between the points in eval_obj(is a list of visited points) and its successor point in a list of angles Φ and if we find that all the angels in Φ are less than $2\theta_{threshold}$ then we terminate checking the remaining point as in [5].
- Next, we merge $S1_{dynamic}$ along with $S_{static}$ to obtain the result for the first skyline.
- To update the results of the skyline query we use the same approach in [4].

Fig. 12 shows how we can compute the All_Skyline algorithm.

Example 2: Continuing with Example 1, assume we want to compute the All_Skyline algorithm. First, if we consider the non-spatial attributes to compute the dominating points we get $S_{static}$={R2,R4,R6} as the preferred restaurant in the non-spatial dimensions. Next, we compute the furthest point (FP) using the $S_{static}$ set and we find that R6 is the FP point.

Then, for the points in the 6-NN in Fig. 2. If the distance of any point to QP is greater than the distance of the furthest point (i.e. D (FP) < D (p)), we can just terminate checking the remaining points in the 6-NN as we already have them sorted by distance as shown in Fig. 3 where the D (FP) =112.

Thus, as shown in Fig. 3 we can see that D(R7)<D(FP) so we save R7 in "eval_obj", and also in $S1_{dynamic}$. Assume we checked R7,R8,R9,R10 as shown in Fig. 3, the vectors of those objects partition the 2π angle into four partition angles $\varphi_{R7R8}$ =26°, $\varphi_{R8R9}$=105°,$\varphi_{R9R10}$ =112°, and $\varphi_{R10R7}$=117°, and also the distance of the checked objects are smaller than FP so we can terminate our process and $S1_{dynamic}$ ={R7,R9,R10}. Finally, we merge $S1_{dynamic}$ along with $S_{static}$ to obtain the result for the first skyline = {R7, R9, R10, R2, R4, R6} sorted by distance to QP.

**Second: My_Dir_Skyline Algorithm**

- As the previous algorithm we first compute $S_{static}$ and the FP point.
- If a point 'p' is closer than FP and the angle between the QP and p is smaller than $\theta_{threshold}$, then p is in the direction of the QP and is entered into $S1_{dynamic.}$
- Next, we merge $S1_{dynamic}$ along with $S_{static}$ to obtain the result for the first skyline.
- To update the results of the skyline query we use the same approach in [4].

Fig. 13 shows how we can compute the My_Dir_Skyline algorithm.

We also make some changes in the directional spatial skyline (DSS) to make them consider both spatial and static attributes as we describe in Fig. 14.

Example 3: Assume we want to compute the My_Dir_Skyline for the example presented in Example 1 and 2. If the point p in our 6-NN example has a smaller distance

than FP and the angle between the QP and p is smaller than $\theta_{threshold}$, then p is in the direction of the QP and entered to $S1_{dynamic.}$ As shown in Fig. 3 we can find that R7 is the dynamic skyline point as it is in the direction of QP (East); so $S1_{dynamic}$ ={R7}. Next, we merge $S1_{dynamic}$ along with $S_{static}$ to obtain the result for the first skyline= {R7, R2, R4, R6}.

## 4 Experimental Evaluations

In this section we present our experiments for evaluating the proposed algorithms. We evaluated the performance by comparing it with the MCSQ algorithm [4] and DSS algorithm [5]. For the accuracy, our proposed algorithm retrieves that same skyline result as the result obtained by both algorithms proposed in [4, 5]. For our moving query point we use the Brinkhoff generator for moving object trajectories [8]. As for the data points, we use a real data set about residential building from Aswan town in Egypt. We assumed that our data set points are static and the query object is a dynamic object which changes its location with respect to time. Since our data set is reasonable in size, we do not use index structure in our experiments and all the data points are stored in memory. We used 1000 record and $\theta_{threshold}$=60° as a default value on most of our experiments. We used SQL server 2005 and visual studio 2008 (C#) in our experiment. Experiments are implemented on a 3GHz Pentium 4 PC with 1.0GB memory, running Windows XP service pack 3.

In the first experiment we used two dimensional non-spatial attributes and we vary the size of the data set (200, 400, 600, 800, and 1000) and observe the effect of increasing the number of records on the CPU time and on update time. Fig. 4 shows the effects of varying the number of records on both the All_Skyline and the My_Dir_Skyline algorithms, and compares it to the MCSQ and DSS algorithms. We can observe that the proposed algorithms enhance the CPU time, however the MCSQ algorithm has a smaller response time as it does not need to compute the direction. Fig. 5 shows that our algorithms and the MCSQ algorithm are better than the DSS algorithm in terms of the time needed to update the results of skyline query result as both approaches use a candidate set for updating the skyline results. So our algorithms improve the update time, however, the CPU time is increased due to the use of the additional directional filter that acts as an additional dimension in our computation.

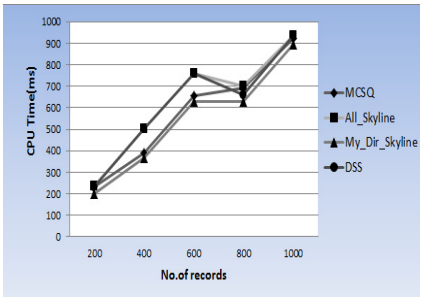


**Fig. 4.** CPU time for different no. of records

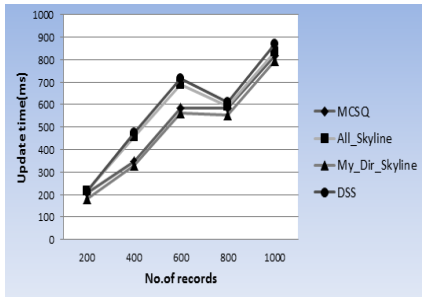


**Fig. 5.** Update time for different no. of records

In the second set of experiments we study the effect of varying the number of levels (2, 3, 4, and 5) on the CPU time and update time. In the first experiment we compare the All_Skyline and the My_Dir_Skyline algorithms with the MCSQ algorithm. As shown in Fig. 6, My_Dir_Skyline algorithm requires less CPU time compared to All_Skyline algorithm as it needs fewer computations to compute the points in the same direction of the user. We also observe that MCSQ algorithm requires less CPU time than All_Skyline algorithm. The reason behind this observation is that both All_Skyline and the My_Dir_Skyline algorithms require more computations to compute the direction of the objects. In Fig. 7 we show the update time of both the All_Skyline and the My_Dir_Skyline algorithms and the MCSQ algorithm. As shown as number of levels increases, the update time decrease as we need less time to update our skyline query result using the previous candidate sets of the skyline.

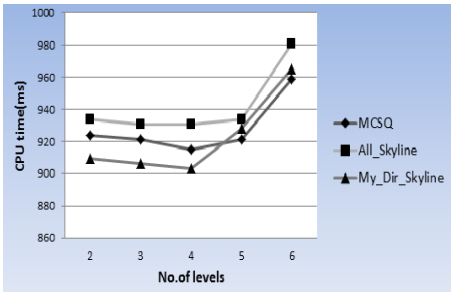


**Fig. 6.** CPU time for different no. of levels

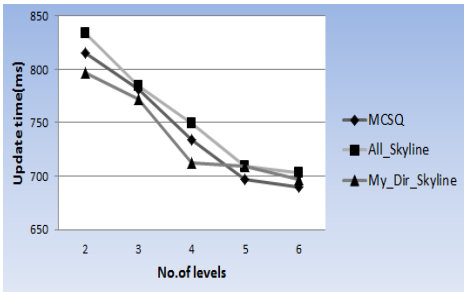


**Fig. 7.** Update time for different no. of levels

The third experiment examines the effect of varying the value of the angle $\theta_{threshold}$ on both the number of examined objects to get our skyline points, and on the output result size (skyline points). In these experiments we vary the angle in the range [15, 85] and study the effect on both CPU time and update time. Fig. 8 and Fig. 9 show that CPU and update time decrease when the value of $\theta_{threshold}$ increases. The reason behind this observation is that more objects can dominate larger angle ranges given a larger value of θ.

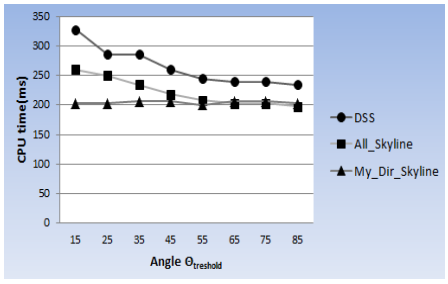


**Fig. 8.** CPU time for different $\theta_{threshold}$

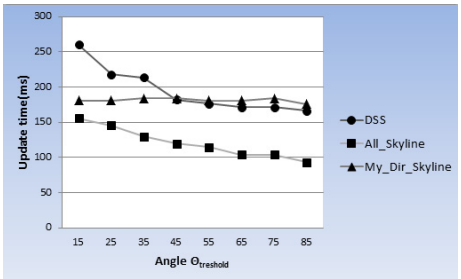


**Fig. 9.** Update time for different $\theta_{threshold}$

In the fourth experiment we study the effect of varying the value of $\theta_{threshold}$ which varies in the range of [15, 85] on the number of checked nearest neighbor objects until we get the skyline points and on the objects in the skyline result. Fig. 10 shows that

both the number of checked nearest neighbor objects and the objects in the skyline query decreases by increasing the value of $\theta_{threshold}$ as an object can dominate larger angle ranges given a larger value of $\theta_{threshold}$.In the last experiment we examine the effect of varying the value of $\theta_{threshold}$ on the number of objects in the skyline. In fig. 11 we can see that when we increase the value of $\theta_{threshold}$ the number of points in the skyline query results decreases in the All_Skyline algorithm. The reason behind this observation is that more objects can dominate larger angle ranges given a larger value of $\theta_{threshold}$, however in the My_Dir_Skyline algorithm the number of points in the skyline increases when the value of $\theta_{threshold}$ increases because when the value of θ increases the range of directions around the query object increases so the number of candidates increases and consequently it has more points in the skyline results.

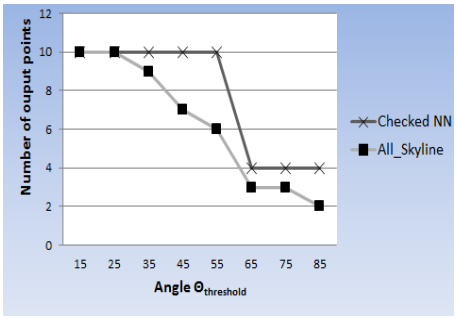


**Fig. 10.** Effects of varying $\theta_{threshold}$ on the No. of output results

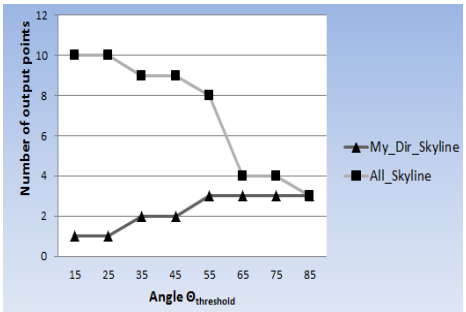


**Fig. 11.** Effects of varying value of $\theta_{threshold}$ on All_Skyline & My_Dir_Skyline

**Algorithm 1** All directional continuous skyline (All_Skyline)

1. Input: data set points $p = (p_1, p_2, ..., p_m)$ ,
   Query point: QP, $\theta_{threshold}$, Query point trajectory time instances $\tau_{QP}[t]$
2. Output: All_Skyline // continuous skyline query result
3. Compute $S_{static}$ at time $\tau_{QP}[1]$
4. FP = Find(FP) // compute FP
5. IF (D(p) <D( FP)) THEN {
6. $S_{dynamic}$={p} ELSE terminate // result set
7. Φ={ $\varphi_{pp}$} //initialize the partition angle set
8. REPEAT
9. p = get_next_ nn
10. p-,p+ // pred and succ of P
11. eval_obj=eval_obj ∪{p}
12. IF (D(P)<D(FP)) THEN {
13. IF ($\lambda_{pp-} \geq \theta_{threshold} \wedge \lambda_{pp+} \geq \theta_{threshold}$ ) THEN
14. $S_{dynamic}$ = $S_{dynamic}$ ∪{p} // p in dynamic skyline

**Fig. 12.** All_Skyline Procedure

```
15.                                  }
16.       ELSE terminate
17.   Φ = (Φ − {φ_{p-p+}})∪{φ_{p-p}, φ_{pp+}}; // update angle
18. UNTIL FALSE or all the objects are processed
19. Get S_dynamic                    }
20. All_Skyline= S_satatic+S_dynamic; //update All_Skyline.
21. Compute FP at time τ_QP[2] and Compute S_Mdynamic at
    time τ_QP[2]
22.FOR (2≤ t ≤ |τ_QP[t]| ) DO FOR (every point p in S_stat-
    ic)
23.       Update D (p, QP)
24.    END FOR
25.    S_dynamic= S_Mdynamic ;        // update S_dynamic
26.    All_Skyline= S_satatic + S_dynamic
27.    Compute FP at time τ_QP[t+1] and Compute S_Mdynamic
    at time τ_QP[t+1]
28.END FOR
29.END
```

**Fig. 12.** (*continued*)

**Algorithm2** My directional continuous skyline (My_Dir_Skyline)

1. Input:
   Data set points $p = (p_1, p_2, .., p_m)$,
   Query point: QP, $\theta_{threshold}$, Query point trajectory time instances $\tau_{QP}[t]$
2. Output: All_Skyline // continuous skyline query result
3. Compute $S_{static}$ at time $\tau_{QP}[1]$
4. FP = Find FP // compute FP
5. $S_{dynamic}=\emptyset$ ;
6. Init_NN_query(q) ; // initialize the NN query
7. IF(D(p)<D(FP)) THEN
8. $S_{dynamic}$ ={p}; ELSE terminate // result set
9. REPEAT
10. p = get_next_ NN
11. IF(D(P)<D(FP) ∧ $dir_{p} < \theta_{threshold}$) THEN
12. $S_{dynamic} = S_{dynamic} \cup \{p\}$; // p in dynamic skyline
13. ELSE terminate
14. UNTIL all the objects are processed
15. get $S_{dynamic}$ at time $\tau_{QP}[1]$
16. My_Dir_Skyline= $S_{satatic} + S_{dynamic}$ ; // update My_Dir_Skyline

**Fig. 13.** My_Dir_Skyline Procedure

```
17.    Compute FP at time τQP[2] and  Compute SMdynamic at
   time τQP[2]
18.    FOR (2≤t≤|τQP[t]| ) DO
19.         FOR( every point p in Sstatic)
20.            Update D (p, QP)
21.         END FOR
22.    Sdynamic=  SMdynamic;                 // update Sdynamic
23.    All_Skyline= Ssatatic +  Sdynamic
24.    Compute FP at time τQP[t+1]  and Compute   SMdynam-
   ic at time τQP[t+1]
25.    END FOR
26.    END
```

**Fig. 13.** (*continued*)

```
Algorithm 3 Directional continuous skyline (DSS) with
static attributes
1. Input: data set points p = (p1, p2, ..., pm)',
       Query point: QP and acceptable angle θthreshold

       Query point trajectory time instances τQP[t]
2. Output: DSS     // continuous skyline query result
3. Compute Sstatic at time τQP[1]
4. Sdynamic=Ø  ;
5. Init_NN_query(q) ;     // initilize the NN query
6. eval_obj=p ;           // get the first NN object
7. S={p};                 // result set
8. Φ={ φpp};    //initialize the partition angle set
9. REPEAT
10.       p = get_next_ nn
11.       p-,p+;                  // pred and succ of P
12.       eval_obj=eval_objU{p}
13.       IF(λpp- ≥ θthreshold ∧  λpp+ ≥ θthreshold ) THEN
14.           S=SU{p};   // p in dynamic skyline
15.           Φ = (Φ − {φp-p+ }) U{φp-p, φpp+ };
// update the part_angle set
16. UNTIL FLASE or all the objects are processed
17. get Sdynamic;  DSS= Sstatic + Sdynamic ;
18. FOR (2≤t≤| τQP[t]| ) DO
19.    FOR( every point p in Sstatic)
20.          UPDATE D (p, QP)
21.    END FOR
22.       Compute Sdynamic time τQP[t] //  update Sdynamic
23.       DSS= Ssatatic +  DSdynamic
24. END FOR
25. END
```

**Fig. 14.** DSS Procedure

## 5 Conclusions and Future Work

In this paper, we propose 2 algorithms which efficiently update Skyline query results through limiting the search space when updating skyline results, and by considering only the objects around the user from different directions and also the points in the same direction as the user. Experimental studies are conducted using a real data. The experiments show that the proposed methods are robust and efficient. For future work we aim to increase our data set size and use an index structure to study the effect of an index structure on the performance of our algorithm. We also plan to investigate the performance of our proposed algorithms in the case of moving data objects.

## References


1. Borzsony, S., et al.: The Skyline Operator. In: Proceedings of the 17th Intl. Conf. on Data Engineering, pp. 421–430 (2001)
2. Benetis, R., et al.: Nearest and Reverse Nearest Neighbor Queries for Moving Objects. The VLDB Journal 3(15), 229–249 (2006)
3. Chen, N., Shou, L., Chen, G., Gao, Y., Dong, J.: Predictive Skyline Queries for Moving Objects. In: Zhou, X., Yokota, H., Deng, K., Liu, Q. (eds.) DASFAA 2009. LNCS, vol. 5463, pp. 278–282. Springer, Heidelberg (2009)
4. El-Dawy, E., Mokhtar, H.M.O., El-Bastawissy, A.: Multi-level Continuous Skyline Queries (MCSQ). In: Zhang, J., Livraga, G. (eds.) Intl. Conf. on Data and Knowledge Engineering (ICDKE), pp. 36–40. IEEE, Milan (2011)
5. Guo, X., Ishikawa, Y., Gao, Y.: Direction-based spatial skylines. In: Proceedings of the Ninth ACM Intl. Workshop on Data Engineering for Wireless and Mobile Access, pp. 73–80. ACM, Indianapolis (2010)
6. Güting, R.H., Schneider, M.: Moving objects databases, 1st edn. Diane D. Cerra, San Francisco (2005)
7. Hsueh, Y.-L., Zimmermann, R., Ku, W.-S.: Efficient Updates for Continuous Skyline Computations. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) DEXA 2008. LNCS, vol. 5181, pp. 419–433. Springer, Heidelberg (2008)
8. Library, S.I., `http://www.research.att.com/~marioh/spatialindex/index.html`
9. Sharifzadeh, M., Shahabi, C.: The spatial skyline queries. In: Proceedings of the 32nd Intl. Conf. on Very Large Data Bases, Seoul, Korea, pp. 751–762 (2006)
10. Sistla, A.P., et al.: Modeling and Querying Moving Objects. In: Proceedings of the 3rd Intl. Conf. on Data Engineering, pp. 422–432 (1997)
11. Tan, K.-L., Eng, P.-K., Ooi, B.C.: Efficient Progressive Skyline Computation. In: Proceedings of the 27th Intl. Conf. on Very Large Data Bases, pp. 301–310 (2001)
12. Tung, A.K.H.: Continuous Skyline Queries for Moving Objects. IEEE Trans. on Knowledge and Data Eng., 1645–1658 (2006)
13. Xiong, X., et al.: Scalable Spatio-temporal Continuous Query Processing for Location-aware Services. In: Proceedings of the 16th Intl. Conf. on Scientific and Statistical Database Management, p. 317 (2004)
14. Guo, X., et al.: Direction-based surrounder queries for mobile recommendations. The VLDB Journal, 743–766 (2011)