Direction-Based Spatial Skylines

Xi Guo
Nagoya University
Nagoya 464-8601, Japan
guoxi@db.itc.nagoya-u.ac.jp

Yoshiharu Ishikawa
Nagoya University
Nagoya 464-8601, Japan
ishikawa@itc.nagoya-u.ac.jp

Yunjun Gao
Zhejiang University
Hangzhou 310027, P. R. China
gaoyj@zju.edu.cn

ABSTRACT

Traditional location-based services recommend nearest objects to the user by considering their spatial proximity. However, an object not only has its distance but also has its *direction* which originates from the user to it. In this paper, we study *direction-based spatial skyline queries* (*DSS* queries) which retrieve nearest objects around the user from different directions. The closer object is better than or *dominates* the further object if they are in the same direction. The objects that cannot be dominated by any other object are included in the *direction-based spatial skyline* (*DSS*). We propose algorithms to answer *snapshot queries* which find objects on the DSS according to the user's current position. We also develop algorithms to support *continuous queries* which retrieve objects on the DSS while the user is moving linearly. Extensive experiments verify the performance of our proposed algorithms using both real and synthetic datasets.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS; H.2.4 [**Systems**]: Query processing

General Terms

Algorithms

Keywords

Spatial databases, skyline queries, location-based services, directions

1. INTRODUCTION

In location-based services such as mobile recommendations and car navigations, a mobile user often receieves recommendations of interesting *POI* (point of interest) objects based on spatial closeness and the user's preference [15]. Generally, the nearest neighbor objects to the user's position are good choices. However, the simple nearest neighbor approach may not work well in some situations.

Example 1 Let us consider a motivating example shown in Fig. 1. A mobile user is on the way to his home, but he is wondering whether to go to a bar, a bookshop, or a fitness gym before going back to home. In addition, suppose that he would like to buy something at a convenience store. Which store is the best one for him? It depends on the *direction* he will take. If he decides to go to the bookshop from now, store D, which is the nearest neighbor one, is not a good choice because store C is on the way to the bookshop. To support his decision, it would be helpful that the recommendation system can present the best store for each direction. In this example, stores A to D should be recommended, but store E would be omitted because store B is better than store E in terms of the distance. ■

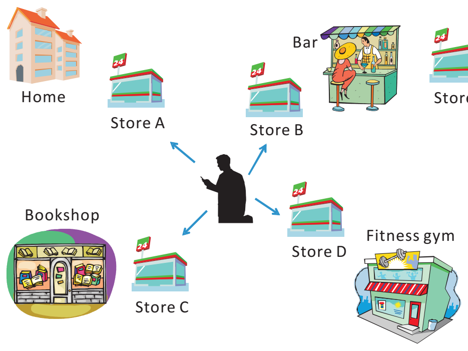
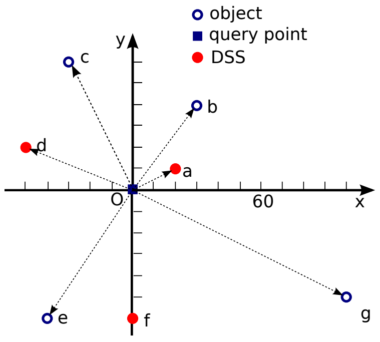


Figure 1: Motivating example

How can we recommend POIs to mobile users in such situations? In this paper, we propose a new type of spatial queries, namely *direction-based spatial skyline queries* (*DSS queries*), which find the best objects not only by comparing the distances but also by considering the directions. A DSS query retrieves nearest objects around the user from different directions. In the result set, each object is nearer than any other objects in the same direction with it. We explain the concept of a DSS query using an example. We define the problem in the two-dimensional Euclidean space and take the locations of the user and each POI as points. For simplicity, we do not take into account non-spatial attributes.

Example 2 In Fig. 2, there are seven target objects (POIs) around the query object (the user's location) $O = (0, 0)$. We have vectors $\vec{a}, \ldots, \vec{g}$ originating from O. In this example, let us consider that two vectors are in the *same direction* if their included angle is smaller than a specified threshold $\theta = \pi/3$. For instance, the objects in the same direction with a are b and g. Note that a is the closest object among $\{a, b, g\}$ from O. It means that a *dominates* other two objects in the same direction and becomes one of the objects to be recommended to the user. In a similar way, we can say that d dominates c and f dominates e, and thus a, d, and f are not dominated by other objects. Consequently a, d, and f constitute the *direction-based spatial skyline* (*DSS*) for O, i.e., the final result of the DSS query is $\{a, d, f\}$. ■



p_i	d_i	ω_i
a	22	27°
b	50	53°
d	54	158°
f	60	270°
c	67	117°
e	72	236°
g	112	333°

Figure 2: Example of a DSS query ($\theta = \pi/3$)

A DSS query is a new type of *skyline queries* [2], mostly focusing on the spatial context. We will give its formal definition in Section 2. The DSS query compares objects in terms of two spatial properties—*distances* and *directions*, rather than distances only considered in traditional spatial skyline queries [16]. In this paper, we study two types of DSS queries, viz. *snapshot queries* and *continuous queries*. An *snapshot DSS query* finds out the objects which are on the DSS according to the user's current position. We have shown its example above. The purpose of the query is to provide the current "best view" to the user. The user can identify the best POI for each direction.

We can extend the concept of a DSS query to the continous case. A *continous DSS query* retrieves the objects on the DSS while the user is moving linearly. Its typical use is to predict when and how the best view (i.e., DSS) changes while the movement. Consider the following example to explain the need for such a query.

Example 3 In Fig. 2, assume that the user linearly moved along the x-axis and he is currently at the position $O' = (60, 0)$. Now object g, which was dominated by a at the initial moment, is not dominated by a because they are in the different direction from the user. On the other hand, object d, which was on the DSS at the initial moment, is now dominated by a since it is further than a and they are in the same direction. In this case, the final result of the DSS query is $\{a, f, g\}$, which is different from $\{a, d, f\}$ at the initial moment. ■

The problem is how we can know when the DSS changes while the user's movement. A naïve solution is to issue a new query at every moment. However, it is quite costly. Our approach is to predict DSS changes based on precomputation. When the user arrives at the change point, which is predicted by the algorithm, we can update the former (i.e., old) DSS to a new one. The method can avoid the shortcoming of the naïve solution.

In the following, we formalize the notion of a DSS query in snapshot and continuous setting respectively, and present the corresponding query processing algorithms. The rest of this paper is organized as follows. In Section 2, a DSS query is formally defined. Section 3 describes the query processing algorithm for the snapshot case. In Section 4, we extend the algorithm to the continuous case. Section 5 reports experimental results and our findings. In Section 6, we overview the related work. Finally, Section 7 concludes the paper.

2. PRELIMINARIES

In this section, we formalize DSS queries. Table 1 summarizes the symbols used in this paper.

2.1 Direction

Consider a database contains the target object set $P = \{p_1, \ldots, p_n\}$ in the two-dimensional Euclidean space. We assume that the user's location is given as a query object q. The vector going from q to p_i is abbreviated as $\overrightarrow{p_i}$. We denote the Euclidean distance between p_i and q as $d_{p_i} = |\overrightarrow{p_i}|$ and denote the angle between $\overrightarrow{p_i}$ and the unit vector $(1, 0)$ as ω_{p_i} $(0 \le \omega_{p_i} < 2\pi)$. We refer to d_{p_i} and ω_{p_i} as p_i's *distance* and *direction*, respectively, and use the abbreviations d_i and ω_i if the context is clear. For example, the vector $\overrightarrow{a}$ in Fig. 2 has the distance $d_a = |\overrightarrow{a}| = 22$ and the direction $\omega_a = 27°$.

Table 1: Symbols and descriptions

Symbol	Description
$\overrightarrow{p_i}$	Vector from q to p_i
θ	Threshold for an acceptable angle
d_i (d_{p_i})	Distance between q and p_i $(d_{p_i} = \lvert\overrightarrow{p_i}\rvert)$
ω_i (ω_{p_i})	Direction of $\overrightarrow{p_i}$: the angle between $\overrightarrow{p_i}$ and (1, 0)
λ_{ij} $(\lambda_{p_ip_j})$	Included angle between $\overrightarrow{p_i}$ and $\overrightarrow{p_j}$
φ_{ij} $(\varphi_{p_ip_j})$	Partition angle between p_i and p_j

We consider the relationship of two target objects in terms of the query object. Intuitively, two objects p_i and p_j are in the *same direction* if their directions happen to be the same with each other $(\omega_i = \omega_j)$. This definition, however, is too strict in practice. Therefore, we argue that two objects are in the same direction if their directions are approximately equal $(\omega_i \approx \omega_j)$. For this purpose, we assume that the threshold θ for an *acceptable angle* is specified by the user, and θ should satisfy the condition $0 \le \theta < \frac{\pi}{2}$. We limit that θ is less than $\pi/2$. This is reasonable because the two vectors turn orthogonal when θ reaches $\pi/2$. We denote the *included angle* between $\overrightarrow{p_i}$ and $\overrightarrow{p_j}$ by $\lambda_{p_ip_j}$ (λ_{ij} for short). λ_{ij} should be in the range $0 \le \lambda_{ij} \le \pi$. It is defined as follows:

$$\lambda_{ij} = \arccos \frac{\overrightarrow{p_i} \cdot \overrightarrow{p_j}}{|\overrightarrow{p_i}| \cdot |\overrightarrow{p_j}|}. \tag{1}$$

Using these assumptions, we define the concept of the same direction.

Definition 1 (Same Direction) For the given target objects p_i and p_j, we say that p_i and p_j are in the *same direction* from q if the condition $0 \le \lambda_{ij} \le \theta$ holds. □

In Fig. 2, if θ is given as $\pi/3$, object b is in the same direction with a from q since the included angle λ_{ab} between their vectors is smaller than θ. On the other hand, object d is in a different direction with a due to $\lambda_{ad} > \theta$.

2.2 Dominance Relationships and DSS Query

The dominance relationship between two objects in terms of their directions and distances is straightfowardly defined as follows.

Definition 2 (Dominance Relationship) If two objects p_i and p_j are in the same direction and p_i is closer than p_j (i.e., $d_i < d_j$) from q, we say that p_i *dominates* p_j, and denote the dominance relationship as $p_i \prec p_j$. □

Note that a dominance relationship is not defined for two objects in different directions.

Based on dominance relationships, we can define direction-based spatial skylines.

Definition 3 (Direction-Based Spatial Skyline) For the target objects $\{p_1, \ldots, p_n\} \in P$, the objects which cannot be dominated by any other object are included in the *direction-based spatial skyline* (*DSS*). □

Definition 4 (Direction-Based Spatial Skyline Query) A *direction-based spatial skyline query* (*DSS query*) finds the objects on the DSS. It is represented by the pair (q, θ), where q is a specified query object and θ is a threshold angle.

Before going to the next section, we would like to mention two minor issues and their solutions.

1. In the following discussion, we consider the case of $0 < \theta < \pi/2$ and omit the case of $\theta = 0$. In the latter case, most of the target objects are on the DSS as p_i can dominate p_j only if p_i and p_j are on the same radial line and p_i is closer than p_j. Although the term "same direction" well matches this situation, the result is meaningless.

2. When the query object q is exactly located on a target object p_i, we can not define the skyline appropriately because vector $\overrightarrow{p}$ is a zero vector. One reasonable solution is that we extend the definition of a direction-based spatial skyline: if a target object p_i is located within the distance ε from q, where ε is a small positive constant, we treat p_i as the *unique skyline object*. It dominates the remaining target objects in all the directions. To simplify the presentation, we do not consider this exceptional case in the following discussion, but the extension of the approach is fairly easy.

3. SNAPSHOT DSS QUERIES

3.1 Definition and Observations

As mentioned in Section 1, a snapshot DSS query finds the "best view" objects for the user's current position. Its definition is presented as follows.

Definition 5 (Snapshot DSS Query) Given a DSS query (q, θ), an *snapshot DSS query* finds all the objects on the DSS. □

In this section, we explain how to answer a snapshot DSS query efficiently. A naïve solution to this problem is to compare every object with all the other objects. If the object cannot be dominated by any other objects, it is on the DSS. This approach uses the definition of a DSS directly and has $O(n^2)$ time complexity. Obviously, we can get rid of the further comparisons for an object if it turns out that it is dominated by another object. However, the cost is still quite large. Therefore, we propose an alternative method to tackle a snapshot DSS query efficiently. The method is based on two observations when we check the objects based on the *increasing distance order.*

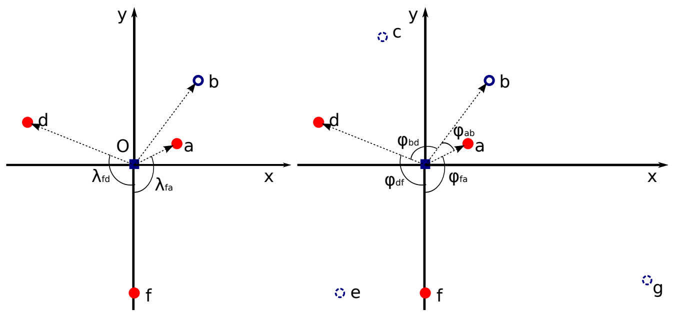
Observation 1: Limiting the scope to adjacent objects

The first observation is that when we dertermine whether an object is on the DSS, we do not need to check its dominance relationships with all the other objects. To explain the idea, we introduce the notion of *adjacent objects.*

Definition 6 (Adjacent Object) The objects are *adjacent* to each other if they are adjacent in the circular list sorted by the order of directions. □

For each object, there are two adjacent objects: the *predecessor* and the *successor*. When we check an object, we only need to consider the dominance relationships between its adjacent objects.

Example 4 Let us consider the example in Fig. 2 again and we already checked a, b, and d. Next we need to check object f. The situation is shown in Fig. 3(a). Since $\omega_a = 27°$, $\omega_b = 53°$, $\omega_d = 158°$, and $\omega_f = 270°$ as shown in Fig. 2, the circular list is $[a, b, d, f]$ (note that the successor of f is a). When we check f, we only need to consider adjacent objects d and a because they *only* have chances to dominate f. Object d (a) dominates f when the included angle λ_{df} (λ_{fa}) is equal to or smaller than θ. ■



(a) Observation 1 (b) Observation 2

Figure 3: Two observations

Based on this observation, we can derive the following property.

Property 1 Let the current target object be p_i and p_i's adjacent objects among the checked objects be p_j and p_k. If both of the included angles λ_{ij} and λ_{ik} satisfy the properties $\lambda_{ij} \geq \theta$ and $\lambda_{ik} \geq \theta$, p_i is a DSS object. Otherwise, p_i is not a DSS object. □

Observation 2: Early termination

The second observation is that we can finish the process by only checking a subset of the whole object set. We first introduce the notion of a *partition angle.*

Definition 7 (Partition Angle) Suppose that objects p_i's successor object is p_j. We define the *partition angle* between p_i and p_j as

$$\varphi_{ij} = (\omega_j - \omega_i) \mod 360°. \tag{2}$$

□

Example 5 Let us continue our example. Assume that we have checked the objects a, b, d, and f. As shown in Fig. 3(b), the vectors of these objects partition the 2π angle into four partition angles $\varphi_{ab} = 26°$, $\varphi_{bd} = 105°$, $\varphi_{df} = 112°$, and $\varphi_{fa} = 117°$. All these angles are smaller than $2\theta = 120°$. It means that any other object, which is coming in the future checks, will be dominated by the current checked objects because at least one of the included angles between its adjacent objects is smaller than θ. Consequently, we can terminate the process at this stage and the DSS is $\{a, d, f\}$. ■

Based on this observation, we can derive the Property 2 below.

Property 2 If all the partition angles for the checked objects are smaller than 2θ, we can terminate the process and we can say that all the DSS objects are obtained. □

The algorithm for a snapshot DSS query is based on the above two observations.

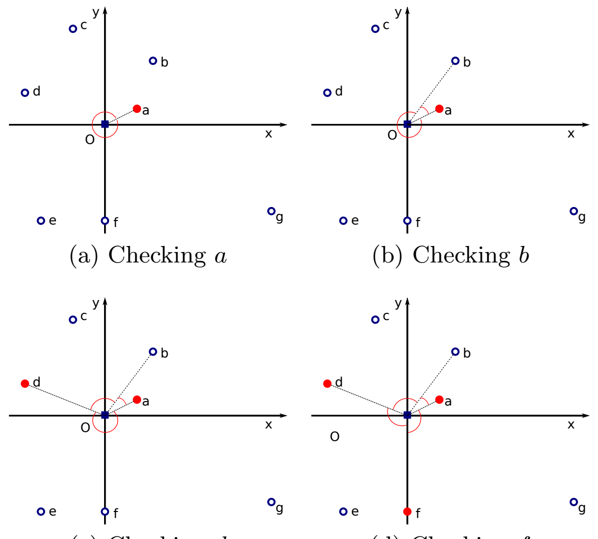
3.2 Query Processing Algorithm

The query processing algorithm is shown in Algorithm 1. First we retrieve the nearest neighbor object using a spatial index (lines 2 and 3). It is on the DSS since it is superior in the distance attribute even though other unchecked objects may in the same direction with it. After initializing variables, we check the target objects according to the increasing distance order (lines 7 to 15). In the algorithm, we maintain the list of checked object l, which is sorted by the directions. At line 9, we insert the object p into the direction list l. The function insert() returns adjacent objects of p (p^- and p^+) in l. At lines 10 to 12, we determine whether p is on the DSS by testing the included angles λ_{p^-p} and λ_{pp^+}. At line 15, we check whether all the angles in Φ is smaller than 2θ. If yes, we can terminate the procedure.

Algorithm 1 Snapshot DSS Query

```
 1: procedure SnapshotDSSQuery($q, \theta$)
 2:     init_NN_query($q$);                    ▷ Initialize the NN query
 3:     $p \leftarrow$ get_next();                  ▷ Get the first NN object
 4:     $S \leftarrow \{p\}$;                        ▷ Set of objects on the DSS
 5:     $l \leftarrow [p]$;                        ▷ Initialize the direction list
 6:     $\Phi \leftarrow \{\varphi_{pp}\}$;             ▷ Initialize the partition angle set
 7:     repeat
 8:         $p \leftarrow$ get_next();              ▷ Get the next NN object
 9:         $\langle p^-, p^+ \rangle \leftarrow l$.insert($p$);
                          ▷ Insert $p$ to $l$ and get its adjacent objects
10:         if $\lambda_{pp^-} \geq \theta \wedge \lambda_{pp^+} \geq \theta$ then
11:             $S \leftarrow S \cup \{p\}$;                  ▷ $p$ is on the DSS
12:         end if
13:         $\Phi \leftarrow (\Phi - \{\varphi_{p^-p^+}\}) \cup \{\varphi_{p^-p}, \varphi_{pp^+}\}$;
14:                                   ▷ Update the partition angle set
15:     until all angles in $\Phi$ are smaller than $2\theta$
16:     output $S$;
17: end procedure
```

Example 6 We start from the nearest object a (Fig. 4(a)). We set the partition angle to $\varphi_{aa} = 2\pi$. Next we check the second nearest object b (Fig. 4(b)). The object b is not on the DSS as $\lambda_{ab} < \theta$. The partition angles are $\varphi_{ab} < 2\theta$ and $\varphi_{ba} > 2\theta$. Since the termination conditon is not satisfied, the procedure continues. Then we examine the third nearest object d (Fig. 4(c)). It is on the DSS due to $\lambda_{db} > \theta$ and $\lambda_{da} > \theta$. As the partition angles are $\varphi_{ab} < 2\theta$, $\varphi_{bd} < 2\theta$, and $\varphi_{da} > 2\theta$, the procedure proceeds. Next we check the fourth nearest object f (Fig. 4(d)). It is on the DSS because $\lambda_{df} > \theta$ and $\lambda_{fa} > \theta$. The partition angles are $\varphi_{ab} < 2\theta$, $\varphi_{bd} < 2\theta$, $\varphi_{df} < 2\theta$ and $\varphi_{fa} < 2\theta$. Thus, the procedure terminates and we have found out all the DSS objects $\{a, d, f\}$. ■



(a) Checking a (b) Checking b (c) Checking d (d) Checking f

Figure 4: Processing snapshot DSS query ($\theta = \pi/3$)

4. CONTINUOUS DSS QUERIES

4.1 Problem Formulation

In this section, we discuss continuous DSS queries. As pointed out in Section 1, a continuous DSS query presents up-to-date DSS objects while the user is continuously moving. A naïve solution is to issue a new query at every moment. Nevertheless, it is quite costly. Our alternative approach is based on *prediction*—we predict when and how the DSS changes in the near future.

For the problem formulation, we introduce some assumptions and notations. Let $t = 0$ be the *current time*. We assume that the user's location $\vec{q} = (x_q, y_q)^t$ is given as follows using time parameter $t \geq 0$:

$$\vec{q} = \begin{pmatrix} x_q \\ y_q \end{pmatrix} = \begin{pmatrix} x_v \\ y_v \end{pmatrix} t + \begin{pmatrix} \bar{x}_q \\ \bar{y}_q \end{pmatrix}, \quad (3)$$

where the user moves from $(\bar{x}_q, \bar{y}_q)^t$ with a constant velocity $(x_v, y_v)^t$.

A continuous DSS query is defined as follows.

Definition 8 (Continuous DSS Query) Given that the user moves linearly as depicted in Eq. (3). For the given parameter τ ($\tau > 0$), a *continuous DSS query* answers when and how DSS changes during the time interval $[0, \tau]$. □

We can easily extend the definition to an *interval-based DSS query*: given a time interval $[t_s, t_e]$, it reports DSS changes while the interval. The extention is straightforward.

Example 7 Assume that we need to predict the change of DSS until $t = \tau$ ($\tau > 0$). As an example, given $\tau = 100$, the continuous DSS query procedure output the result such as

$$DSS = \begin{cases} \{a, d, f\} & t \in (0, 4) \\ \{a, d, f, g\} & t \in (4, 23) \\ \{a, f, g\} & t \in (23, 59) \\ \{a, g\} & t \in (59, 100). \end{cases} \quad (4)$$

It indicates that the DSS is $\{a, d, f\}$ at the start time $t = 0$ and it changes to be $\{a, d, f, g\}$ at $t = 4$, turns $\{a, f, g\}$ at $t = 23$ and then turns $\{a, g\}$ at $t = 59$. We call $t = 4$, $t = 23$ and $t = 59$ *change moments*. Using this query result, we can achieve efficient DSS updates: when $t = 4$, $t = 23$, and $t = 59$ the former DSS is replaced with the next DSS. When $t = \tau = 100$, we issue a new continuous DSS query and predict new DSSs. ■

Our assumption was constant movement of the user holds for short time period τ, but it is not necessary true. In practice, a constant velocity is not a restriction. If we can predict when DSS changes, we can also predict *where* DSS changes. Therefore, even if the speed of the user changes, we can perform DSS update correctly. In contrast, the change of the movement direction makes the prediction invalid. When we detect a change of the direction, we need to restart the query process.

4.2 Basic Idea and Outline of Algorithm

For continuous queries, we also employ the approach for answering snapshot DSS queries. The baseic idea of the snapshot algorithm is summarized as follows:

1. Check the candidates in turn with the order of the increasing distances.

2. For each candidate, determine whether it is on the DSS by checking the included angles with its two adjacent objects.

3. When all partition angles are smaller than 2θ, finish the procedure.

Example 8 Let us extend our example to a continuous DSS query case. Fig. 5 illustrates that the user is moving from position $(\bar{x}_q, \bar{y}_q)^t = (0, 0)^t$ with a constant speed $(x_v, y_v)^t = (1, 0)^t$. We consider a continuous DSS query for the time interval $(0, 100)$ with $\theta = \pi/3$.

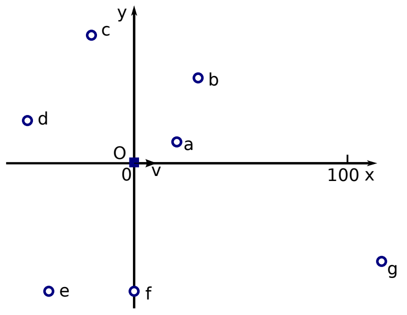


Figure 5: Example of a continuous DSS query

Similar to the case of a snapshot DSS query, we check the objects from the nearest one in turn with the distance order. Unlike the snapshot case, the checking order of the objects changes when the user moves because the distances of the objects from the query point vary while the query point moves.

When $t = 0$, the nearest object is a, but when $t = 75$ it changes from a to g. Let us focus on the time interval $(0, 75)$ where the nearest object is a. Of course, object a is on the DSS. Next we consider the second nearest objects in this interval. The second nearest objects are b while $(0, 71)$ and g while $(71, 75)$.

For the time interval $(0, 71)$, we process object b in three steps like what we did for the snapshot case. First, we find the order of the directions of the checked objects a and b during $(0, 71)$. Second, we dertermine whether b is on the DSS by caculating its included angles (λ) with its adjacent objects. Third, we check whether we can terminate the process by comparing the partition angles (φ) with 2θ. In this case, we can not terminate the process[1] and we proceed to consider the third nearest objects in $(0, 71)$. They are d while $(0, 7)$, f while $(7, 45)$, and g while $(45, 71)$. In a similar manner, we process the object d for $(0, 7)$ in three steps. The process continues until the termination.

As shown in Fig. 5, we can represent the query process by a tree structure. We expand the tree nodes in a depth-first manner and every branch of the tree means a different checking order. Our example discussed above follows the path *root* $\rightarrow a \rightarrow b \rightarrow d \rightarrow f \rightarrow \cdots$. ■

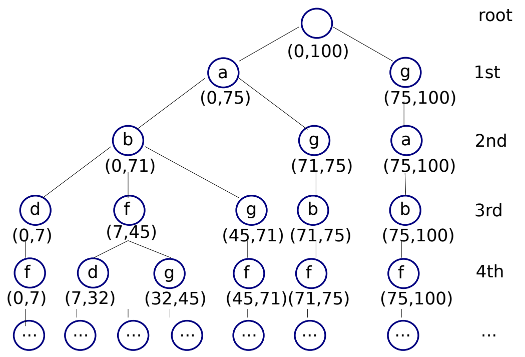


Figure 6: Process tree for continual DSS query

Algorithm 2 presents the outline of the continuous query algorithm. The algorithm calls the function FINDDSS recursively as the tree expansion process shown above. At line 7, function CNNQUERY retrieves the k-nearest objects for the moving query point $\vec{q}$ within the time interval $\mathcal{I}$. The result is a set of pairs with the form $\langle p, \mathcal{I}_p \rangle$, where p is the k-nearest object while the time interval $\mathcal{I}_p$. For instance, in the example of Fig. 5, CNNQUERY$(\vec{q}, (0, 100), 1)$ is $\{\langle a, (0, 75)\rangle, \langle g, (75, 100)\rangle\}$. In the long version of the paper [8], we explain how to implement the function by extending the existing algorithm proposed by Tao, Papadias, and Shen [17].

[1]Since $\theta < \pi/2$, at least three DSS objects are required to cover 2π angles.

Algorithm 2 Continuous DSS Query

```
 1: procedure CONTINUOUSDSSQUERY($\vec{q}$, $\theta$, $\mathcal{I}$)
                        ▷ $\mathcal{I}$ is the target time interval: $\mathcal{I} = [0, \tau]$
 2:     $r \leftarrow$ create_root_node();          ▷ Create a root node
 3:     FINDDSS($\vec{q}, \mathcal{I}, 1, r$);
 4:     output $S$;
 5: end procedure

 6: procedure FINDDSS($\vec{q}$, $\mathcal{I}$, $k$, $n$)
 7:     foreach $\langle p, \mathcal{I}_p \rangle \in$ CNNQUERY($\vec{q}, \mathcal{I}, k$) do
                              ▷ Find $k$-NN object(s) while $\mathcal{I}$
 8:         $\mathcal{A}$ = FINDADJACENTOBJ($p, \mathcal{I}_p$);
                          ▷ Find $p$'s adjacent objects while $\mathcal{I}_p$
 9:         foreach $\langle p^-, p^+, \mathcal{I} \rangle \in \mathcal{A}$ do
10:             foreach $\mathcal{I}' \in$ DOMCHECK($p, \langle p^-, p^+, \mathcal{I}\rangle$) do
                                              ▷ Check dominance
11:                 $n$.create_child_node($\langle p, \mathcal{I} \rangle$);
12:             end for
13:         end for
14:         if NOTTERMINATE() then
                        ▷ Termination condition is not satisfied
15:             FINDDSS($\vec{q}, \mathcal{I}, k+1, S$);
                                  ▷ Expand the child nodes
16:         end if
17:     end for
18: end procedure
```

At line 8, FINDADJACENTOBJ() finds the adjacent objects of object p within the time interval $\mathcal{I}_p$. Note that p may have different adjacent objects while different sub-intervals. In the algorithm, j represents the adjacent objects for a certain interval. In Section 4.3, we discuss the function in detail. At line 10, DOMCHECK() checks whether p is on the DSS and updates the DSS results S. The algorithm for the function is explained in Section 4.4. At line 14, NOTTERMINATE() determines whether we can terminate the process by checking all patition angles. In Section 4.5, its detail is described.

4.3 Finding Adjacent Objects

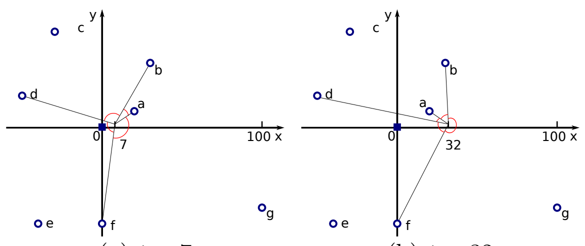
As described in Section 3, we consider two adjacent objects for the current target object. In contrast to the snapshot case, such neighborhood relationship is only valid for a certain time interval because the directions of objects depend on the moving query point q.

Example 9 Let us consider our example in Fig. 6 again. Assume that we take the branch $a \rightarrow b \rightarrow f$ and arrive at the node $\langle d, (7, 32)\rangle$. When $t = 7$, the adjacent objects of d are b and f as shown in Fig. 7(a). However, when $t = 32$, the adjacent objects are a and f as illustrated in Fig. 7(b). The change happens at $t = 18$ when a and b are *co-linear* with q. In other words, b and f are adjacent to d during $(7, 18)$, and a and f are adjacent to d during $(18, 32)$. Hence for the node $\langle d, (7, 32)\rangle$ in Fig. 6, we maintain two direction order lists: $\langle a, b, d, f\rangle_{(7,18)}$ and $\langle b, a, d, f\rangle_{(18,32)}$. ■

The observation above can be summarized as follows.

Property 3 The direction order changes when two objects are co-linear. The new direction order is obtained from the former one by swapping two co-linear objects. □

This means that we need to maintain a group of direction order lists for every tree node in order to preserve different neighborhoods for different time intervals.



(a) $t = 7$ (b) $t = 32$

Figure 7: Change of direction order

Example 9 (Continued) Fig. 8 shows the process of creating direction order lists while traversing through $a \to b \to f \to d$. At the beginning, we traverse the first tree node $\langle a, (0, 75)\rangle$ and set its direction order list to $\langle a\rangle$. Then we reach the next tree node $\langle b, (0, 71)\rangle$. We initialize its direction order list by using its parent's lists and insert b into the list. However, the objects b and a are co-linear when $t = 18$ during this interval, and thus we split the interval into two sub-intervals $(0, 18)$ and $(18, 71)$. For $(0, 18)$, we keep the original direction order, namely, $\langle a, b\rangle$. But for $(18, 71)$, we create a new list $\langle b, a\rangle$ by swapping a and b. Next we reach the tree node $\langle f, (7, 45)\rangle$. We initialize its lists by using its parent's lists and insert f into two lists respectively. As object f has no co-linear objects in both intervals, we do not need to split the interval and swap objects. Then we create the lists for the tree node $\langle d, (7, 32)\rangle$ likewise. ■

Tree Node	Time Interval	List	Operation
$\langle a, (0, 75)\rangle$	$(0, 75)$	$\langle \boldsymbol{a}\rangle$	insert a
$\langle b, (0, 71)\rangle$	$(0, 18)$	$\langle a, \boldsymbol{b}\rangle$	insert b
	$(18, 71)$	$\langle \boldsymbol{b}, \boldsymbol{a}\rangle$	insert b; swap(a, b)
$\langle f, (7, 45)\rangle$	$(7, 18)$	$\langle a, b, \boldsymbol{f}\rangle$	insert f
	$(18, 45)$	$\langle b, a, \boldsymbol{f}\rangle$	insert f
$\langle d, (7, 32)\rangle$	$(7, 18)$	$\langle a, b, \boldsymbol{d}, f\rangle$	insert d
	$(18, 32)$	$\langle b, a, \boldsymbol{d}, f\rangle$	insert d

Figure 8: Incremental maintenance of direction order lists

The algorithm to find adjacent objects is presented in Algorithm 2 in the long version of our paper [8]. It is the implementation of FINDADJACENTOBJ() in Algorithm 2.

4.4 Checking Dominance

We describe the algorithm for function DOMCHECK() in Algorithm 2. In the case of an instance DSS query, we check the included angles of one object and its two adjacent objects to determine whether it is on the DSS. The object is dominated if at least one of the angles is smaller than θ. In the continuous case, however, included angles change when the user moves.

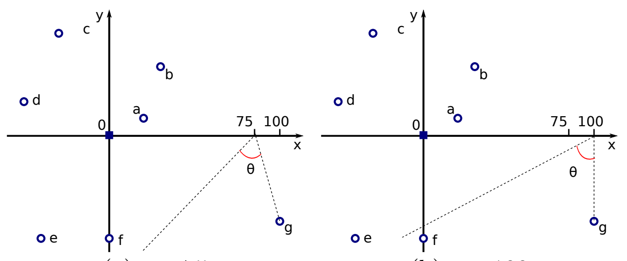
Example 10 Take Fig. 6 as an example again. Assume that we take the branch $g \to a \to b$ and arrive at the node $\langle f, (75, 100)\rangle$, and we want to determine whether it is on the DSS. Object f has adjacent objects a and g during $(75, 100)$. Let us consider the included angle λ_{fg}. When $t = 75$, $\lambda_{fg} > \theta = \pi/3$ (Fig. 9(a)), but when $t = 100$, $\lambda_{fg} < \theta$ (Fig. 9(b)). The change happens at the moment $t = 95$ when $\lambda_{fg} = \theta$. In other words, f is not dominated by g during $(0, 95)$, but it is dominated by g during $(95, 100)$. The dominance relationship of two adjacent objects changes when their included angle equals to θ. ■

Object p_i is not dominated by its adjacent object p_j during the time interval when their included angle $\lambda_{ij} \geq \theta$. The condition can be represented as follows:

$$\lambda_{ij} = \arccos \frac{\vec{p_i} \cdot \vec{p_j}}{|\vec{p_i}||\vec{p_j}|} \geq \theta \quad (0 \leq \lambda_{ij} \leq \pi). \qquad (5)$$

Note that $\vec{p_i}$ and $\vec{p_j}$ are time-parameterized vectors that change with parameter t. To obtain the time intervals for which the formula holds, we need to solve a quartic inequality. It is not a difficult task. We first solve the corresponding quartic equation numerically using GNU Scientific Library [7], then we can derive the valid intervals that make the above condition true [2].

For object p, we need to consider two adjacent objects p^- and p^+. We calculate the time intervals $\mathcal{I}^-$ and $\mathcal{I}^+$ when $\lambda_{pp^-} \geq \theta$ and $\lambda_{pp^+} \geq \theta$, respectively. Then we take their intersection to obtain the time interval while p is on the DSS. The algorithm to check dominance relationship is straightforward.



(a) $t = 75$ (b) $t = 100$

Figure 9: Change of dominance relationship

4.5 Checking Termination Condition

In the snapshot DSS query, the checking procedure terminates when all partition angles are smaller than 2θ. The partition angles depends on the direction order list. Note that, in the continous DSS query, one tree node has several direction order lists.

Example 11 Consider the sitiuation of Example 9. The tree node $\langle d, (7, 32)\rangle$ has two direction lists $\langle a, b, d, f\rangle_{(7,18)}$ and $\langle b, a, d, f\rangle_{(18,32)}$ as shown in Fig. 7. Their partition angle sets are $\Phi_{(7,18)} = \{\varphi_{ab}, \varphi_{bd}, \varphi_{df}, \varphi_{fa}\}$ and $\Phi_{(18,32)} = \{\varphi_{ba}, \varphi_{ad}, \varphi_{df}, \varphi_{fb}\}$. The procedure can terminate when all angles in $\Phi_{(7,18)}$ and $\Phi_{(18,32)}$ are smaller than 2θ during $(7, 18)$ and $(18, 32)$, respectively. ■

Therefore, we need to check partition angles for every list in order to determine whether we have found out all DSS objects. For checking, we check whether all φ's in every direction order list are less than 2θ while the time interval attached to the tree node. The outline of the checking procedure is presented in the long version of the paper [8].

5. EXPERIMENTS

This section experimentally evaluates the performance of the proposed algorithms. We implemented all algorithms in GNU C++ and conducted the experiments on an Intel Core2 Duo 2.40GHz PC (2.0GB RAM) with a Ubuntu Linux 2.6.31.

5.1 Experimental Setup

We implemented the experiments by deploying both real and synthetic datasets. Some additional experiments are covered in the long version of our paper [8]. The real datasets came from line segment data of Long Beach from the TIGER database [18]. We made this point set by extracting the midpoint for each road line segment. The set consists of 50,747 points normalized in $[0, 1000] \times [0, 1000]$ space. There are three synthetic datasets s_1, s_2 and s_3 with different densities normalized in $[0, 1000] \times [0, 1000]$ space (Table 2). Here *density* means how many points fall into one square unit in average. The points in each synthetic dataset are distributed randomly. We indexed all datasets by using an R*-tree library [1]. with the block size as 8,192 bytes.

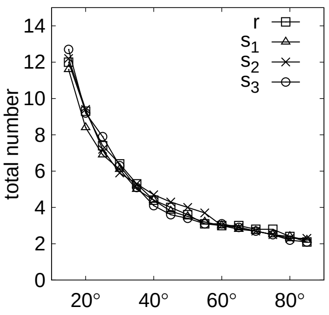
[2]The details are shown in the long version of the paper [8]

Table 2: Datasets

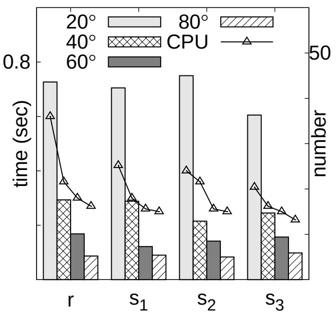
Dataset	Objects Number	Density (ρ)
r	$50,747$	–
s_1	$80,000$	0.08
s_2	$50,000$	0.05
s_3	$20,000$	0.02

5.2 Results on Snapshot DSS Queries

The first experiment studies the numbers of DSS objects under different θ's and different densities. Fig. 10(a) plots that the total numbers of DSS objects from datasets r, s_1, s_2 and s_3 when θ varies in $[15°, 85°]$. The total number decreases while θ increases. The reason is that one object can dominate more objects if θ becomes larger. However, difference of densities[3] does not affect the total number of DSS objects.



(a) DSS number



(b) Cost and checked NNs

Figure 10: Performance of snapshot queries

The second experiment explores the performance of the algorithm under different θ's and different densities. CPU times and the number of checked nearest objects (NNs) are measured. As shown in Fig. 10(b), the number of the checked objects is small (less than 0.14% of the whole objects set). And the number of checked NNs is impacted by the setting of θ. This is because if θ grows it is easier to reach the termination condition and we only need to check fewer NNs. The query cost depends on the number of checked NNs and decreases when θ increases. However, the query cost is independent of dataset densities.

5.3 Results on Continuous DSS Queries

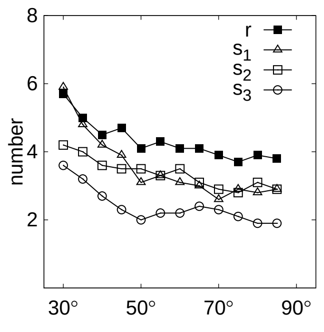
For the real dataset, we set the scenario as the user moves with the speed of 0.06 unit distance per unit time[4] along the positive x-axis during time intervals $[0, 10]$, $[0, 20]$ and $[0, 30]$.

The first experiment investigates the number of varying moments under different θ's and different time intervals. As demonstrated in Fig. 11(a), the number of varying moments decreases while θ increases because a DSS becomes stable when θ is getting larger. And the number of varying moments is fewer when the density is lower. Consequently, both the setting of θ and the density of dataset affect the number of varying moments. As shown in Fig. 11(b), the number of varying moments decreases while the time interval becomes shorter.

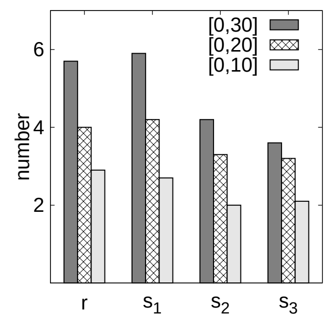
The second experiment studies the query costs under different θ's and different time intervals. In Fig. 12(a), the tree size decreases when θ grows. The reason is that we can terminate the query procedure ealier if θ is larger. And the query cost depends on the tree size. Thus, the query cost also decreases when θ grows. Moreover, the tree size decreases when objects density decreases. This is because we constructed the processing tree based on the distance order of objects, and the order changes less frequently when the dataset density is lower. Fig. 12(b) shows that the tree size and the query cost decrease when the time interval becomes shorter.

[3]Note that we did not know the density of the r dataset but we still aligned this set for comparisons.

[4]In the space of our datasets, 1 unit distance equals to 1 kilometer approximately. And we regard 1 unit time as 1 minute. Then we simulate the user's moving speed as human's average walking speed $1m/s$.

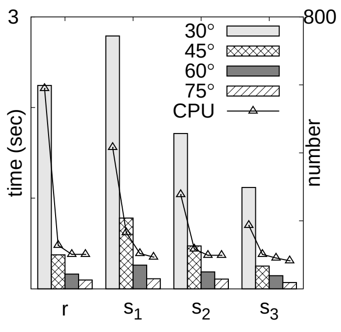


(a) $0 \leq t \leq 30$

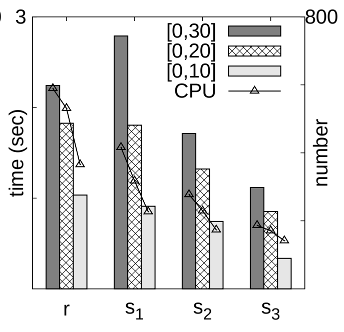


(b) $\theta = 30°$

Figure 11: Number of varying moments



(a) $0 \leq t \leq 30$



(b) $\theta = 30°$

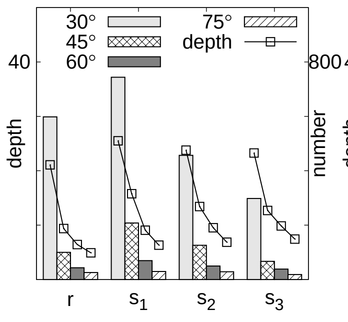
Figure 12: Costs and tree sizes of continuous queries

We also investigate the change of the properties of processing trees for continous queries under different θ's and different time intervals. Fig. 13(a) demonstrates that the tree depth decreases when θ grows, because we can terminate the query procedure earlier when θ is larger. However, the tree depth is not affected by densities. In other words, we can derive that the growth of the tree size is caused by the increase of the branches. Fig. 13(b) shows that the tree depth is not influenced by the length of the time interval. Therefore, these results confirm that our termination strategy is stable for different objects densities and different time interval lengths.

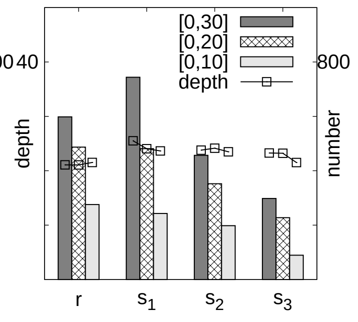
6. RELATED WORK

Skyline queries have received considerable attention from the database community since 2001 when the pioneering paper [2] considering skyline queries in relational databases appeared. Afterwards, many subsequent algorithms proposed to improve the approach. The well-known algorithms include branch-and-bound skyline algorithm (BBS) [11], sort-filter-skyline (SFS) [4], and linear elimination sort for skyline (LESS) [14], etc.

Skyline queries in spatial databases become a hot topic with the development of mobile technologies. Spatial skyline queries often consider the dynamic spatial attribute—*distance*. The distance attribute is different from other static attributes (e.g., price) because it depends on the query point (e.g., mobile user) which moves continuously in most of spa-



(a) $0 \leq t \leq 30$



(b) $\theta = 30°$

Figure 13: Tree depth of continuous queries

tial database applications. Some approaches [9, 19] address on skyline queries including the distance attribute for only one query point in the continuous case. Huang et al. [9] proposed efficient algorithms to solve the problem by exploiting the spatiaotemporal coherences. They observed that some skyline points are permanent and derived a search bound. Then they predicted the skyline changes. Zhang et al. [19] proposed another algorithm by predicting a valid scope for query results . Both of them can avoid unnecessary updates of skylines. On the other hand, the spatial skyline query in [16] focuses on the context where are several query points. In summary, these works studied skyline queries with non-spatial attributes and one dynamic spatial attribute—distance.

There are also several approaches not only consider the distance attribute but also consider other spatial attributes. Patroumpas et al. [12] proposed the notion of an *orientation-based query* which finds objects moving towards the query point. For example, it finds trucks moving towards the port from the west at a distance less than 2km. They used a *polar tree* to index the moving objects by their directions and retrieved the objects within the required direction and distance ranges. Lee et al. [10] studied the *nearest surrounder query* which retrieves nearest neighbors from the query point at different angles. For instance, it finds closer sights for the tourist to provide a good picture of his surroundings. The query regards the objects as *rectangles* in contrast to our algorithms, and takes the distance attribute as an *static* attribute (i.e., the query object is not moving). The DSS query also determines nearest surrounding objects but the decision is based on distances and the specified threshold angle θ. Our proposed algorithms cover not only the snapshot case but also the continuous case. Chen et al. [3] identified the *path nearest neighbor query* which retrieves the nearest neighbor along the user's moving path. For example, there are several gas stations along the user's moving path and it finds the potential nearest one for the user. These proposals consider not only the distance attribute but also other spatial attributes. However, to the best of our knowledge, the direction attribute is not used in the most of literatures.

We use continuous nearest neighbor queries to construct the processing tree in our continuous DSS query. There are many proposals on continuous nearest neighbor queries. Tao et al. [17] proposed an efficient algorithm to solve this problem (See the long version of our paper [8].). We also used the basic idea of Katerina et al. [13] which make use of the intersections of distance functions to find changes of nearest neighbors. There are also several variations of conventional continuous nearest neighbor queries such as *continous visible nearest neighbor search* [6] and *continuous obstructed nearest neighbor search* [5].

7. CONCLUSION

In this paper, we study the problem of *direction-based spatial skylines.* We develop efficient algorithms for efficiently processing snapshot DSS queries and continuous DSS queries, respectively. Extensive experiments with both real and synthetic datasets evaluate the performance of our proposed algorithms. The experiment results demonstrate that the proposed algorithms work well for both the snapshot case and the continuous case.

Acknowledgments

This research was partly supported by the Grant-in-Aid for Scientific Research (#21013023, #22300034) from the Japan Society for the Promotion of Science (JSPS).

8. REFERENCES

[1] R*-tree library. http://research.nii.ac.jp/˜katayama/homepage/research/srtree/English.html.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. *Proc. Int'l Conf. on Data Engineering (ICDE)*, 2001.

[3] Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu. Monitoring path nearest neighbor in road networks. *Proc. Int'l Conf. ACM SIGMOD*, 2009.

[4] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. *Proc. Int'l Conf. on Data Engineering (ICDE)*, 2003.

[5] Y. Gao and B. Zheng. Continuous obstructed nearest neighbor queries in spatial database. *Proc. Int'l Conf. ACM SIGMOD*, 2009.

[6] Y. Gao, B. Zheng, G. Chen, W. C. Lee, and G. Chen. Continuous visible reverse nearest neighbor queries. *Proc. Int'l Conf. on Extending Database Technology (EDBT)*, 2009.

[7] Gnu scientific library. http://www.gnu.org/software/gsl/.

[8] X. Guo, Y. Ishikawa, and Y. Gao. Direction-based spatial skylines (long version). http://www.db.itc.nagoya-u.ac.jp/˜guoxi/tmp/mobide-long.pdf, 2010.

[9] Z. Huang, H. Lu, B. C. Ooi, and A. K. H. Tung. Continuous skyline queries for moving objects. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(12):1645–1658, 2006.

[10] K. C. K. Lee, W.-C. Lee, and H. V. Leong. Nearest surrounder queries. *Proc. Int'l Conf. on Data Engineering (ICDE)*, 2006.

[11] D. Padadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. *Proc. Int'l Conf. ACM SIGMOD*, 2003.

[12] K. Patroumpas and T. Sellis. Monitoring orientation of moving objects around focal points. *Proc. Int'l Symp. on Spatial and Temporal Databases (SSTD)*, 2009.

[13] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos. Fast nearest-neighbor query processing in moving-object databases. *GeoInformatica*, 7(2):113–137, 2003.

[14] P. G. Ryan, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, 2005.

[15] J. Schiller and A. Voisard. *Location-Based Services.* Morgan Kaufmann, 2004.

[16] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, 2006.

[17] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, 2002.

[18] Tiger. http://tiger.census.gov/.

[19] B. Zhang, K. C. K. Lee, and W.-C. Lee. Location-dependent skyline query. *Proc. Int'l Conf. on Mobile Data Management (MDM)*, 2008.