World Wide Web (2013) 16:247–270
DOI 10.1007/s11280-012-0185-1

Efficient general spatial skyline computation

Qianlu Lin · Ying Zhang · Wenjie Zhang · Xuemin Lin

Received: 21 May 2012 / Revised: 13 August 2012 /
Accepted: 23 August 2012 / Published online: 8 September 2012

Abstract With the emergence of location-aware mobile device technologies, communication technologies and GPS systems, the location based queries have attracted great attentions in the database literature. In many user recommendation web services, the spatial preference query is used to suggest the objects based on their spatial proximity with the facilities. In this paper, we study the problem of *general spatial skyline* (*GSSKY*) which can provide the minimal candidate set of the optimal solutions for any monotonic distance based spatial preference query. Efficient progressive algorithm called *P-GSSKY* is proposed to significantly reduce the number of non-promising objects in the computation. Moreover, we also propose spatial join based algorithm, called *J-GSSKY*, which can compute *GSSKY* efficiently in terms of I/O cost. The paper conducts a comprehensive performance study of the proposed techniques based on both real and synthetic data.

Keywords general spatial skyline · spatial preference function · nearest neighbor search

Q. Lin · Y. Zhang (✉) · W. Zhang · X. Lin
School of Computer Science & Engineering,
University of New South Wales, Sydney, NSW 2052, Australia
e-mail: yingz@cse.unsw.edu.au

Q. Lin
e-mail: qlin@cse.unsw.edu.au

W. Zhang
e-mail: zhangw@cse.unsw.edu.au

X. Lin
e-mail: lxue@cse.unsw.edu.au

1 Introduction

With the development of mobile device technologies, communication technologies and GPS systems in recent years, there has been an increasing number of location based service systems specialized in providing interesting results through location-based queries which retrieve the desirable candidate objects for users based on the spatial proximity of the objects and facilities. For instance, as shown in Figure 1a, there are a set of apartments, bus stations and supermarkets in the map, and a user wants to rent an apartment which is close to both bus station and supermarket. In Figure 1b, each apartment is mapped to a point in a 2-dimensional space where the distances to the nearest bus station and supermarket are coordinate values of an apartment. As shown in Figure 1 the apartment a_4 derives the distance values by its closest bus station (b_1) and supermarket(s_1). Clearly, the smaller value is preferred. As there is no apartment with both shortest supermarket-distance and bus station-distance in the example, the user needs to make a trade-off. Suppose user has a preference function against the distances of an apartment regarding its closest bus station and supermarket, the system can return the apartment with best score regarding the preference function. If the preference function is in the form of $f(o) = 4 \times o.d_1 + o.d_2$ where $o.d_1$ and $o.d_2$ represent the closest distances of o to the supermarket and bus station respectively, then a_4 is the best choice. The answer becomes a_3 if we have $f(o) = o.d_1 + 4 \times o.d_2$. This is the distance based spatial preference query,[1] and the problem is studied in [13, 17, 23]. However, in many applications users cannot find a proper preference function. Therefore, it is desirable to provide a candidate set with small size for users such that they can make personal trade-off without missing any potential optimal solution.

Motivated by the above example application, in the paper we propose the *general spatial skyline* (*GSSKY*) operator. Given a set $\mathcal{O}$ of objects and a set $\mathcal{F}$ of facilities with m types, an object o can be mapped to a point $\tilde{o}$ in m-dimensional space, named distance space, where the coordinate value on i-th dimension is the distance of o to its nearest facility with type i. We say an object o_1 *spatially dominates* another object o_2 if $\tilde{o_1}$ *dominates* $\tilde{o_2}$ in the mapped distance space. Note that the *dominance* relationship in distance space is the same as the traditional skyline problem [1]; that is, we say $\tilde{o_1}$ *dominates* $\tilde{o_1}$ if $\tilde{o_2}$ is not larger than $\tilde{o_2}$ on any dimension i, where $1 \leq i \leq m$, and $\tilde{o_1}$ is smaller than $\tilde{o_2}$ on at least one dimension. Then the objects which are not *spatially dominated* by any other object are *general spatial skyline* objects. As shown in Section 2.2, the *general spatial skyline* objects can provide the minimal candidate of optimal solution for any monotonic distance based spatial preference query. Moreover, we show theoretically and experimentally that the number of *GSSKY* objects is usually much smaller than that of the objects.

Note that, there are some existing works [19, 20] which study the problem of *spatial skyline*, in which they only consider one single facility for each facility type. Please refer to Section 7 for detailed problem definition. The *spatial skyline* is useful when there is only one facility (e.g., the airport in a city) or only one single facility is chosen (e.g., the conference hotel) for each facility type. But their skyline model cannot provide the minimal candidate for the distance based spatial preference

[1] See Section 2.2 for the formal definition.

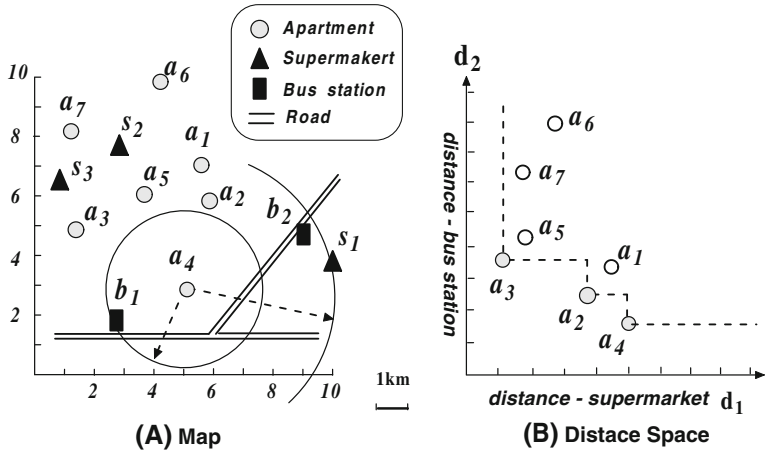


Figure 1 Motivating example.

queries discussed in the paper. For instance, we must choose one particular bus station and one particular supermarket in Figure 1 to define the spatial skyline objects under their skyline model. Consequently, such result is meaningless for the spatial preference queries which may consider multiple facilities for each type of facility. Moreover, the techniques developed in [19, 20] cannot be applied to the problem of *general spatial skyline* computation because multiple facilities are considered for each facility type in the paper.

Challenges A straightforward solution for the *GSSKY* query is to compute the distance values of all objects and then apply the traditional skyline algorithm. This is not efficient because the distance computation, i.e., calculating the distance of each object to its closest facility regarding a particular type, is expensive and we have to compute distance values for all objects. In the paper, we propose a novel *GSSKY* computation algorithm which aims to reduce the amount of distance computations by pruning non-promising objects. Our contributions can be summarized as follows.[2]

- It is the first work to introduce the *general spatial skyline* operator, which can provide a minimal candidate set for any monotonic distance based spatial preference query.
- Two efficient algorithms are proposed to compute the *general spatial skyline*.
- Comprehensive experiments demonstrate the efficiency of our techniques.

The remainder of the paper is organized as follows. We formally define the problem and related techniques in Section 2. Section 3 presents the all nearest neighbor based algorithm. Sections 4 and 5 propose the progressive *GSSKY* algorithm and spatial join based *GSSKY* algorithm respectively. Results of the comprehensive performance studies are presented in Section 6. Section 7 presents the related work. Finally, Section 8 concludes the paper.

[2]The article is the extension of the conference version [14].

Table 1 The summary of notations.

Notation	Definition
o ($\mathcal{O}$)	Object (a set of objects)
f ($\mathcal{F}$)	Facility (a set of facilities)
$\mathcal{F}_i$	All facilities in $\mathcal{F}$ with type i
$o.d_i$	The nearest neighbor distance of object o regarding $\mathcal{F}_i$
$o \lhd \mathcal{F}$	o is *fully hit* by $\mathcal{F}$
r_i	The maximal hit distance seen so far regarding facilities with type i
$o_1 \prec_{\mathcal{F}} o_2$	o_1 *spatially dominates* o_2 regarding $\mathcal{F}$
$GSSKY(\mathcal{O}, \mathcal{F})$	The general spatial skyline of $\mathcal{O}$ regarding the facilities $\mathcal{F}$
$nnd_{\min}(e, e_f)$ ($nnd_{\max}(e, e_f)$)	The minimal (maximal) nearest neighbor distance between the object entry e and the facility entry e_f
$MDBR$	The minimal nearest neighbor distance bounding rectangle
T	The nearest neighbor distance (NND) tuple
$T.e$	The object R-tree entry associated with the NND tuple T
$T.\mathcal{F}_i$	The facility R-tree entries with type i associated with the NND tuple T
$T.mdbr$	The minimal nearest neighbor distance bounding rectangle (MDBR) of the NND tuple T

2 Preliminary

In this section, we first formally define the problem of general spatial skyline computation in Section 2.1. In Section 2.2, we show that the *general spatial skyline* can provide a minimal candidate set for monotonic spatial preference functions. We introduce the incremental nearest neighbor algorithm in Section 2.4. Table 1 below summarizes mathematical notations frequently used.

2.1 Problem definition

A point x referred in the paper, by default, is in d-dimensional numerical space. Let $\delta(x, y)$ denote the Euclidian distance between two points x and y.[3] In the paper, $\mathcal{F}$ represents a set of facilities and $\mathcal{F}_i$ denotes all facilities in $\mathcal{F}$ with type i. And a facility f is a point in the space with a particular facility type.

An object o is a point in d-dimensional numerical space. The distance of o regarding $\mathcal{F}_i$, denoted by $o.d_i$, is the *nearest neighbor distance* between o and $\mathcal{F}_i$, i.e., $o.d_i = \min(\delta(o, f) \text{ for any } f \in \mathcal{F}_i)$. As shown in Figure 1, given a set $\mathcal{F}$ of facilities with m types (categories), an object o can be mapped to a point in m dimensional space. Then we define the *spatial dominance* relationship as follows.

Definition 1 (Spatial dominance) Given two objects o_1, o_2 and a set $\mathcal{F}$ of facilities, We say object o_1 *spatially dominates* another object o_2 regarding $\mathcal{F}$, denoted by $o_1 \prec_{\mathcal{F}} o_2$, if and only if $o_1.d_j \leq o_2.d_j$ for **any** type j, and there is a facility type i such that $o_1.d_i < o_2.d_i$.

[3] We focus on Euclidian distance in the paper. Nevertheless, our techniques can be easily extended to other L_p norm distances.

Example 1 In Figure 1, we have $a_2 \prec_{\mathcal{F}} a_1$, $a_3 \prec_{\mathcal{F}} a_5$, and $a_4 \not\prec_{\mathcal{F}} a_5$.

Based on the *spatial dominance* relation, we come up with the definition of *general spatial skyline* as follows.

Definition 2 (General spatial skyline) Given a set $\mathcal{O}$ of objects and a set $\mathcal{F}$ of facilities, the general spatial skyline of $\mathcal{O}$ regarding $\mathcal{F}$, denoted by $GSSky(\mathcal{O}, \mathcal{F})$, are objects which are not *spatially dominated* by any other objects regarding $\mathcal{F}$.

Example 2 In Figure 1, we have $GSSky(\mathcal{O}, \mathcal{F}) = \{a_2, a_3, a_4\}$.

Problem statement In this paper we investigate the problem of efficiently computing general spatial skyline against a set of objects and facilities with multiple number of types.

2.2 Minimal candidate property

Given a set $\mathcal{O}$ of objects and a set $\mathcal{F}$ of facilities with m types, a distance based spatial preference function p is defined as follows, where the score of an object regarding $\mathcal{F}$, denoted by o_s, is derived based on the distance values to its closest facilities.

$$o_s = p(o.d_1, \ \ldots, o.d_m) \tag{1}$$

Recall that $o.d_i$ denotes the distance between o and its closest facility with type i. For presentation simplicity, we use "spatial preference function" to abbreviate "distance based spatial preference function" in the paper whenever there is no ambiguity. The following theorem indicates that the *GSSKY* provides the minimal candidate set for all increasing spatial preference functions.

Theorem 1 *Let $\mathcal{P}$ denote the family of all increasing spatial preference functions regarding $\mathcal{F}$, for any $p \in \mathcal{P}$ the object with the best score is in $GSSky(\mathcal{O}, \mathcal{F})$. For any object o in $GSSky(o, \mathcal{F})$, there exists a spatial preference function $p \in \mathcal{P}$ such that o has the best score regarding p.*

Proof For any object $o_2 \notin GSSky(\mathcal{O}, \mathcal{F})$, there is an object o_1 such that $o_1 \in GSSky(\mathcal{O}, \mathcal{F})$ and $o_1 \prec_{\mathcal{F}} o_2$ according to definition of *GSSKY*. We have $o_1.d_j \leq o_2.d_j$ for any $j \in [1, m]$ and there exists $i \in [1, m]$ such that $o_1.d_i < o_2.d_i$. According to the monotonic property of the functions, we have $p(o_1) < p(o_2)$ for any increasing spatial preference function p. With similar rationale, there is an increasing spatial preference function p for each object $o \in GSSky(\mathcal{O}, \mathcal{F})$ such that $p(o)$ has lowest score among all objects. Therefore, the theorem holds. □

2.3 *GSSKY* size estimation

Based on [6], we have the following theorem which estimates the size of *GSSKY* objects with independent assumption.

Theorem 2 *Suppose the locations of the facilities and objects are independent to each other, then the expected number of GSSKY object is* $O(\frac{(\ln(n))^{d-1}}{(d-1)!})$ *where n is the number of objects in* $\mathcal{O}$.

2.4 Incremental nearest neighbor technique

As our general spatial skyline algorithm proposed in Section 4 is based on the incremental nearest neighbors(INN) computation, we introduce INN technique [8] in this subsection. Unlike the k nearest neighbor query where k is known beforehand, the INN algorithm will incrementally output the next closest neighbor, i.e., the $(l+1)$-th nearest neighbor where l is the number of neighbors seen so far, on user's demand.

Suppose the objects are organized by an R-tree, a priority queue $\mathcal{Q}$ is used to maintain a set of R-tree entries (intermediate entries and data entries) where a key of an entry is its minimal distance to the query point. The root of the R-tree is pushed to $\mathcal{Q}$ at the beginning of the algorithm. For each incremental nearest neighbor request, the algorithm outputs the data entry in $\mathcal{Q}$ with smallest key value. Note that we say an object is in $\mathcal{Q}$ if its corresponding data entry or any of its parent entry is in $\mathcal{Q}$. Specifically, if the entry with smallest key value is a data entry which is associated with an object o, o is output and popped from $\mathcal{Q}$. Otherwise, the intermediate entry (i.e., index or leaf node) is popped and expanded, and all its child entries will be pushed into $\mathcal{Q}$. The procedure will be repeated until the entry on top of $\mathcal{Q}$ is a data entry. In this way, the system can incrementally output the next closest neighbor and [8] theoretically and experimentally shows the efficiency of their INN algorithm.

Algorithm 1 *ANN based GSSKY* $(\mathcal{O}, \mathcal{F})$

Input : $\mathcal{O}$: the objects,
$\mathcal{F}$: the facilities
Output : $\mathcal{S}$: **GSSKY** $(\mathcal{O}, \mathcal{F})$
1 **for** each facility type i in $[1..m]$ **do**
2 Compute the $o.d_i$ for each object $o \in \mathcal{O}$ by applying ANN [10] algorithm against $\mathcal{O}$ and $\mathcal{F}_i$;
3 $\mathcal{S} \leftarrow$ compute skyline on the distances of the objects;
4 **return** $\mathcal{S}$

3 All Neareast Neighbor (ANN) based GSSKY algorithm

Since the *GSSKY* problem is exactly same as the traditional skyline problem if all objects are mapped to the distance space $\mathcal{D}$, a straitforward solution for the *GSSKY* computation is to first compute the distances for all objects regarding $\mathcal{F}$, and then apply the existing skyline algorithm. As the computation of the distance values of the objects regarding facilities with type i can be achieved by all nearest neighbor (ANN) queries against $\mathcal{O}$ and $\mathcal{F}_i$, in this subsection, we apply the state-of-the-art ANN technique [3] to compute the *GSSKY GSSKY*. The Algorithm 1 outlines the ANN based general spatial skyline computation. Note that all existing non-index

skyline techniques can be applied in Line 3 once the distance values of all objects are available. As shown in our initial empirical study, the dominant cost of Algorithm 1 is the distance computation.

4 Efficient progressive *GSSKY* algorithm

In this section, we first introduce the motivation of the progressive *GSSKY* algorithm in Section 4.1, then Section 4.2 presents the details of the algorithm.

4.1 Motivation

As shown in the empirical study, the dominant cost of the *GSSKY* computation comes from the distance computation of the objects. Consequently, even if we apply the state-of-the-art technique to compute distances for all objects, the ANN based *GSSKY* algorithm is inefficient in terms of both I/O and CPU costs. Motivated by this, in this section we aim to reduce the number of distance computations during the *GSSKY* query processing.

Generally, we may compute the nearest neighbor distance values of the objects in two ways:

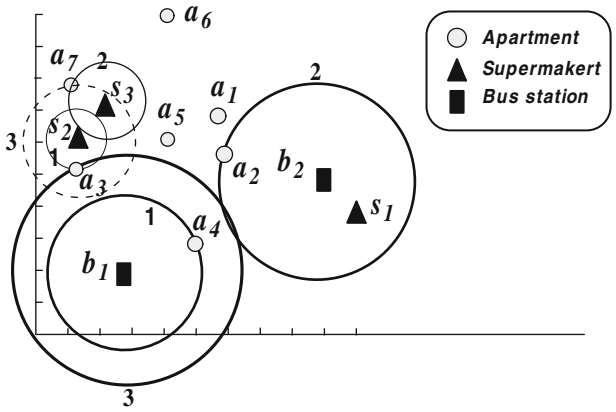
Object oriented search For each object o, we compute the $o.d_i$ by applying the nearest neighbor(NN) algorithm [18] where o is the query point. For instance, as shown in Figure 1 $o_4.d_1$ and $o_4.d_2$ can be derived by issuing two NN queries against $\mathcal{F}_1 = \{s_1, s_2, s_3\}$ and $\mathcal{F}_2 = \{b_1, b_2\}$ respectively, where o_4 is the query point. Particularly, the all nearest neighbor (ANN) algorithm [3] can also be considered as an object oriented method in which the object distances are computed in a batch fashion.

The advantage of the object oriented search is that, for a given object or a set of objects, we can directly derive the distances of the objects. However, as there is no priori knowledge about the distance values of the *unvisited* objects, like Algorithm 1 in Section 3, we have to compute distance values for all objects to ensure the correctness of *GSSKY* computation.

Facility oriented search Instead of computing distance values for each individual object, we can derive them by applying incremental nearest neighbor(INN) algorithm against facilities simultaneously where the query point is a facility. As shown in Figure 2, for each facility $f \in \mathcal{F}$, we maintain a radius f_r and we say an object o has been *hit* by f if $\delta(f, o) \leq f_r$. The distance between o and f is called the hit distance of o regarding f. Similarly, we say an object o is *fully hit* by $\mathcal{F}$, denoted by $o \lhd \mathcal{F}$, if o has been hit by **all types** of facilities; that is, for any $\mathcal{F}_i$, there exists a facility $f \in \mathcal{F}_i$ such that o is *hit* by f. For each facility type i, we maintain a global radius r_i which is the maximal hit distance seen so far regarding facilities with type i.

At each iteration, for each type i we find a facility f in $\mathcal{F}_i$ to invoke a new hit by expanding f_r such that the increase of r_i is minimized. Clearly, the global radius r_i is non-decreasing in the search. Due to the monotonic property of r_i, we can safely set $o.d_i$ to the hit distance when it is *hit* first time by a facility with type i. Recall that an

Figure 2 Running example.



object may be *hit* multiple times by the facilities with the same type. Therefore, we say a *hit* is a *redundant hit* if the object has been *hit* by another facility with the same type.

Example 3 Figure 2 illustrates a snapshot of facility oriented search in which we use a circle to record each hit of the objects. Specifically, circles with thin(bold) line represent the hits from bus stations (supermarkets) and the number of a circle indicates the accessing order. Moreover, the circle with solid (dashed) line represents a *non-redundant hit* (*redundant hit*). In Figure 2, $a_3.d_1$ and $a_4.d_2$ are derived in the first iteration. In the third iteration, the hit of a_7 regarding s_3 is a *redundant hit* because a_7 has been *hit* by s_2 in the second iteration.

Without loss of generality, we assume the hit distance is distinct for each facility type in the paper. Note that the duplication can be easily handled by visiting all objects with the same hit distance. Because of the monotonic property of the hit distance (i.e., r_i), the following lemma is immediate, which enables us to obtain the lower bound of the distance values for the *unvisited* objects.

Lemma 1 *In the facility oriented search, we have $o.d_i > r_i$ if an object o has not been hit by any facility with type i so far.*

Based on Lemma 1, the following theorem implies that we can safely prune some objects from the $GSSKY(\mathcal{O}, \mathcal{F})$ without distance computation.

Theorem 3 *In the facility oriented search, suppose there exists an object o_1 which has been hit by* ***all types*** *of facilities, an object o_2 can be pruned from $GSSKY(\mathcal{O}, \mathcal{F})$ if o_2 has not been hit by* ***any*** *facility.*

Proof We have $o_1.d_i \leq r_i$ since o_1 has been *hit* by all types of facilities. On the other hand, we have $o_2.d_i > r_i$ for any object o_2 which has not been *hit* by any facility. It is immediate that $o_1 \prec_{\mathcal{F}} o_2$ and hence o_2 can be pruned from $GSSKY(\mathcal{O}, \mathcal{F})$. Therefore, the theorem holds. □

Example 4 In Figure 2, objects $\{a_1, a_5, a_6\}$ can be pruned from $GSSKY(\mathcal{O}, \mathcal{F})$ without any distance computation because none of them has been *hit* by any facility when a_3 is *fully hit*.

Another advantage of facility oriented search is that, as shown in [8], the amortized cost for each hit distance computation in INN query is cheaper than that of a NN query because the INN algorithm can share the computation by continuously maintain the priority queue. This implies that if the proportion of *redundant hit* is not significant, the facility oriented method is more efficient. Intuitively, the proportion of the *redundant hit* will increase against the global radius r_i regarding $\mathcal{F}_i$ because the larger the radius, the higher chance an object is *hit* by multiple facilities in $\mathcal{F}_i$. Another disadvantage of the the facility oriented search is that we need to maintain a priority queue for each facility and it is not space efficient when the number of facilities is very large.

Motivated by the advantages and disadvantages of the object oriented search and facility oriented search, we propose an efficient progressive *GSSKY* algorithm which combines both methods in an effective way. The algorithm consists of three phases. In the first phase, we apply the facility oriented search to compute object distances and prune objects (i.e., removing non-skyline objects) based on Lemma 1 and Theorem 3. This is feasible because the number of facilities is usually much smaller than that of objects in real applications. When we find that the computation of facility oriented search becomes less efficient due to large amount of *redundant hits*, the algorithm goes to phase two, in which we compute the distances of the remaining objects based on the object oriented search (i.e., NN query). Finally, in phase three we apply the existing skyline algorithm to finalize the *GSSKY* computation.

4.2 Progressive *GSSKY* algorithm

In the paper, we assume a set $\mathcal{O}$ of objects are organized by R-Tree, denoted by $R_{\mathcal{O}}$, and all facilities with type i are also organized by R-Tree $R_{\mathcal{F}_i}$. The Algorithm 2 illustrates the details of the efficient *GSSKY* algorithm.

In Lines 2–9, we apply the facility oriented search to compute the distances of the objects until there exits an object which has been *fully hit*. Particularly, a *local priority queue* is employed by each facility f for INN query, i.e., retrieving the next closest neighbor of f. For each facility type $i \in [1, m]$, we use a *global priority queue* $\mathcal{Q}_i$ to maintain the current closest neighbors (i.e., objects) of the facilities in $\mathcal{F}_i$ where the distances are keys. In Line 4, the object o on the top of $\mathcal{Q}_i$ is popped and an INN query is issued by its associated facility to retrieve the next closest neighbors o_2. Then o_2 is pushed into $\mathcal{Q}_i$. Line 6 sets $o.d_i$ to the current hit distance (recorded by r_i) if it is a *non-redundant hit*. When the loop is terminated (Line 9), o is a *GSSKY* object and kept in $\mathcal{S}$, and objects which have been *hit* at least once are kept in the candidate set $\mathcal{C}$. According to Theorem 3, all remaining objects can be pruned. For I/O efficiency, we keep the page ids of the nodes (i.e., intermediate entries) of the $R_{\mathcal{O}}$ visited so far. In the following facility oriented search (Lines 10–21), we do not access a node of $R_{\mathcal{O}}$ if its page id is recorded (i.e., all of its descendant data entries correspond to the pruned objects) and hence the I/O cost can be saved.

Algorithm 2 Efficient Progressive *GSSKY* Algorithm ($\mathcal{O}$, $\mathcal{F}$)

```
   Input  : O : the objects,
            F : the facilities with m types
   Output : S : GSSKY( O, F)
 1 S := ∅; C := ∅; r_i := 0 for each F_i;
 2 while true do
 3     for each facility type i in [1..m] do
 4         o ← next object in facility oriented search regarding F_i;
 5         if the hit of o is a non-redundant hit then
 6             o_{d_i} := r_i; C := C ∪ o;
 7         if o is fully hit by F then
 8             S := o; C := C − o;
 9             Terminate the while loop;
10 while true do
11     for each facility type i in [1..m] do
12         o ← next object in facility oriented search regarding F_i;
13         if o is a candidate object and the hit of o is a non-redundant hit then
14             o_{d_i} := r_i;
15             if SkylineTest(S, o) then
16                 if o is fully hit by F then
17                     C := C − o; S := S + o;
18             else
19                 C := C − o;
20         if #redundant hit is larger than #non-redundant hit then
21             Terminate the while loop;
22 for each object o ∈ C do
23     calculate o_{d_i} by NN query if o has not been hit regarding F_i ;
24 for each object o ∈ C accessed in non-decreasing order based on Σ_{i=1}^{m} o_{d_i} do
25     if SkylineTest(S, o) then
26         S := S ∪ o;
27 return S
```

In Lines 10–21, we continue the facility oriented search and try to identify the *GSSKY* objects and prune the non-promising ones. Particularly, if the object o output in Line 12 is a candidate object (i.e., $o \in \mathcal{C}$) and the hit is a *non-redundant hit*, Line 15 checks if there exits an object $s \in \mathcal{S}$ (i.e., *GSSKY* objects seen so far) such that $s \prec_{\mathcal{F}} o$. Note that if o has not been *hit* by any facility with type i, o_{d_i} is temporarily set to r_i in the test. We say an object o passes the skyline test (**SkylineTest**) if it is not *spatially dominated* by any object in $\mathcal{S}$. In the case o passes the test (Lines 16–17), it is a *GSSKY* object **if** o has been *fully hit*. Otherwise, we cannot claim o is a *GSSKY* object at this moment because the lower bound of the distance is employed in the skyline test. Line 19 eliminates the object from $\mathcal{C}$ if o fails the test. As discussed in Section 4.1, we should stop the facility oriented search when r_i becomes large because most of the hit might be *redundant hit*. However, it is unlikely to find the optimal stop time since the distributions of the following *redundant hits* and *non-redundant hits* are unknown. In the paper, we employ a simple but effective criteria. The number of *non-redundant hits* and *redundant hits* are counted in the algorithm, and Line 21 terminates the facility oriented search if there are more *redundant hits*.

Lines 22–23 calculate the missing distances for the objects in the candidate set, where the object oriented search is employed. Recall that the missing of o_{d_i} value implies that the object o has not been *hit* by any facility with type i so far. The remaining part of the algorithm is similar to the SFS Algorithm [4]. Particularly, we sort all the candidate objects based on the sum of their distance values (i.e., $\sum_{i=1}^{m} o_{d_i}$) in *non-decreasing* order (Line 22), and Line 25 checks if an object is *spatially dominated* by the *GSSKY* objects ($\mathcal{S}$) seen so far. The objects passed the test are *GSSKY* objects (Line 26).

Correctness For the correctness of the Algorithm 2, we need the following properties: (i) any object pruned at Lines 15 and 25 are not *GSSKY* object, (ii) all objects unaccessed in Algorithm 2 are not *GSSKY* objects, and (iii) the object in $\mathcal{S}$ cannot be dominated by any other object in $\mathcal{O}$. Below is a formal proof.

Proof The correctness of the property (i) is immediate based on the definition of *GSSKY* if o has been *fully hit* at Lines 15 and 25. If o_{d_i} is replaced by r_i at Line 15 (i.e., o has not been *hit* by any facility with type i) and o is dominated by an object $s \in \mathcal{S}$, we can claim that s *spatially dominates* o regarding $\mathcal{F}$ due to the monotonic property of r_i (Lemma 1).

The correctness of property (ii) is immediate base on Theorem 3.

We prove the correctness of property (iii) by the contradiction. Suppose the object s is in $\mathcal{S}$ but s is *spatially dominated* by another object o. We can assume o is a *GSSKY* object because of the *dominance transitivity* property of *spatial dominance*, i.e., $o_1 \prec_{\mathcal{F}} o_2$ and $o_2 \prec_{\mathcal{F}} o_3$ implies $o_1 \prec_{\mathcal{F}} o_3$. If s is put in $\mathcal{S}$ at Line 8 or Line 17, o should be included in $\mathcal{S}$ before s. This is because $o \prec_{\mathcal{F}} s$ implies s is *fully hit* after o due to the monotonic property in the facility oriented search. This is against the facts that s is not *spatially dominated* by any objects in $\mathcal{S}$ or s is the first object being *fully hit*. We can come up with similar contradiction if s is put in $\mathcal{S}$ at Line 26 because we access objects based the sum of their distance values. □

Performance analysis Upon each hit in Algorithm 2 (Lines 2–21), an INN query is issued to retrieve the next closest neighbor of a facility $f \in \mathcal{F}_i$ where $i \in [1, m]$ and the global priority queue $\mathcal{Q}_i$ is updated. The cost is $C_{inn} + O(\log(n_f))$ where C_{inn} and n_f denote the average cost of an INN query and the average number of facilities for each type respectively. If it is a *non-redundant hit* , the skyline test is invoked which costs $|\mathcal{S}|$ in the worst case where $|\mathcal{S}|$ is the size of $\mathcal{S}$. In Lines 22–23, the cost is $(m-1) \times |\mathcal{C}| \times C_{nn}$ in the worst case where C_{nn} is the average cost for NN query and $|\mathcal{C}|$ dentoes the candidate set size. Recall that a candidate object will be *hit* at least once. The sorting cost in Line 24 is $O(|\mathcal{C}| \times \log(|\mathcal{C}|))$ and the cost of skyline computation in Lines 24–26 is $|\mathcal{S}|^2$ in the worse case. In summary, let n_r and n_t denote the number of *redundant hits* and *non-redundant hits*, the time complexity of Algorithm 2 is $O((n_r + n_s) \times (C_{inn} + \log(n_f)) + n_s \times |\mathcal{S}| + (m-1) \times |\mathcal{C}| \times C_{nn} + |\mathcal{C}| \times \log(|\mathcal{C}|) + |\mathcal{S}|^2)$. Note that, in practice $|\mathcal{S}|$ and $|\mathcal{C}|$ are much smaller than the total number of objects, and hence the algorithm is quite efficient.

Following theorem estimates the number of objects accessed, i.e., objects have been *hit* at least once, in Algorithm 2 based on the uniform and independence assumption.

Theorem 4 *Suppose that (i) objects and facilities are uniformly distributed in the space* $[0, 1]^2$*, (ii) there are* n_f *facilities for each type, and (iii) the locations are independent regarding different types of facilities. The expected number of objects accessed in Algorithm 2 is* $n(1 - (1 - \pi \bar{X}^2)^m)$ *where* n *and* n_f *are the number of objects and facilities for each type. Particularly, we have* $\bar{X}$ *equals* $\int_{r=0}^{c}(1 - F(r))' r \, d(r)$ *where* $c = \frac{1}{\sqrt{2}n_f}$*, and* $F(r) = (1 - (n_f \pi r^2)^m)^n$.

Proof According to the uniform assumption and there are same number of facilities for each type, r_i is same in each iteration where $1 \leq i \leq m$, which is denoted by r. The probability that none of the objects is fully *hit* for given r, denoted by $F(r)$, is $(1 - (n_f \pi r^2)^m)^n$ due to the uniform and independence assumption. Let X denote the distance r when the first object is fully *hit*, then its expected value $\bar{X}$ equals $\int_{r=0}^{c}(1 - F(r))' \times r \, d(r)$ where $c = \frac{1}{\sqrt{2}n_f}$. Consequently, the expected number of objects accessed is $n(1 - (1 - \pi \bar{X}^2)^m)$. □

5 Efficient spatial join based *GSSKY* algorithm

In this section, we present the spatial join based algorithm which focuses on improving the I/O efficiency of the *GSSKY* computation. Particularly, the motivation and details of the algorithm are introduced in Sections 5.1 and 5.2 respectively.

5.1 Motivation

Although the progressive *GSSKY* Algorithm introduced in Section 4 is efficient in terms of CPU cost in our empirical study, it has two drawbacks which may inherently limit its overall performance in some scenarios. Firstly, in the facility oriented searching phase of Algorithm 2 we need conduct the incremental search on each individual facility, which implies that the Algorithm does not scale well towards the number of facilities. Secondly, to guarantee the monotonic property during the nearest neighbor distance computations, the facilities and objects are accessed based on their corresponding nearest neighbor distances. This causes the problem of I/O inefficiency because an object may be accessed multiple times. This is confirmed in the empirical study, which reports that the I/O performance of the progressive *GSSKY* Algorithm is poor when the buffer size is small.

Motivated by this, in this section we propose a new algorithm following the *synchronized R*-tree traversal paradigm used in [2, 10, 11, 16] to address the above two issues. More specifically, instead of applying the incremental search on individual facility or object, the proposed algorithm will process the objects and facilities in a level by level fashion such that the nearest neighbor distance computations and skyline dominance check can be conducted on a group of objects or facilities. Moreover, to avoid computing nearest neighbor distances for all objects, we naturally integrate the skyline dominance check into the nearest neighbor distance computation so that some non-promising objects in the same groups can be pruned without calculating individual nearest neighbor distance. Following is a motivating example to illustrate the underline principles of the spatial join based *GSSKY* computation algorithm.

In Figure 3, a set of apartments (i.e., objects) $\{a_1, a_2, \ldots, a_7\}$ are organized by an R-tree R_A. Particularly, a_1, a_2 and a_3 are allocated to R-tree node A_1, the R-tree nodes A_2 and A_3 keep $\{a_4, a_5\}$ and $\{a_6, a_7\}$ respectively. The root of the R_A is A_0, and the organization of R_A is shown on the right side of Figure 3. Similarly, Figure 3 shows that bus stations (1st type of facilities) and supermarkets (2nd type of facilities) are organized by R_B and R_S respectively.

For an entry e (intermediate entry or data entry) in the object R-tree R_A, we use a tuple T, namely nearest neighbor distance(NND) tuple, to keep the information used for nearest neighbor distance computation of e, which consists of three elements: $T.e$, $\{T.\mathcal{F}_i\}$ with $1 \leq i \leq m$, and $T.mbr$ where m is the number of facility types in the query. Particularly, $T.e$ refers to the object R-tree entry e; $T.\mathcal{F}_i$ denotes a set of i-th type ($1 \leq i \leq m$) facility R-tree entries which contribute to the nearest neighbor distance computation of $T.e$; $T.mbr$ is the minimal nearest neighbor distance bounding rectangle (MDBR) of the entry $T.e$, and the i-th value of its lower (upper) corner, denoted by $T.mbr\_low[i]$ ($T.mbr\_high[i]$), is the minimal (maximal) nearest neighbor distance which is derived from facility entries in $T.\mathcal{F}_i$. Note that the $T.mbr$ is a rectangle in the distance space.

Suppose we have $T.e$ referring to A_1 in Figure 3, $T.\mathcal{F}_1 = \{M_1, M_2\}$ and $T.\mathcal{F}_2 = \{B_1, B_2\}$. By applying the nearest neighbor distance computation technique in [3, 22], we can derive the minimal(maximal) nearest neighbor distance regarding two R-tree entries (e.g., A_1 and B_1); That is, given an object R-tree entry A_1 and a facility R-tree entry B_1, we can come up with the minimal (maximal) nearest neighbor distances $nnd_{\min}(A_1, B_1)$ ($nnd_{\max}(A_1, B_1)$) such that for any objects in A_1, its nearest neighbor distance regarding facilities in B_1, denoted by $\mathrm{NND}(o, B_1)$, is bounded by $nnd_{\min}(A_1, B_1)$ and $nnd_{\max}(A_1, B_1)$. Then we can derive $T.mbr$ based on minimal (maximal) nearest neighbor distance between A_1 and facility entries in $\{T.\mathcal{F}_i\}$, which is formally defined as follows.

$$T.mbr\_low[i] = \min\{nnd_{\min}(T.e, e_f) | e_f \in T.\mathcal{F}_i\} \tag{2}$$

and

$$T.mbr\_high[i] = \min\{nnd_{\max}(T.e, e_f) | e_f \in T.\mathcal{F}_i\} \tag{3}$$

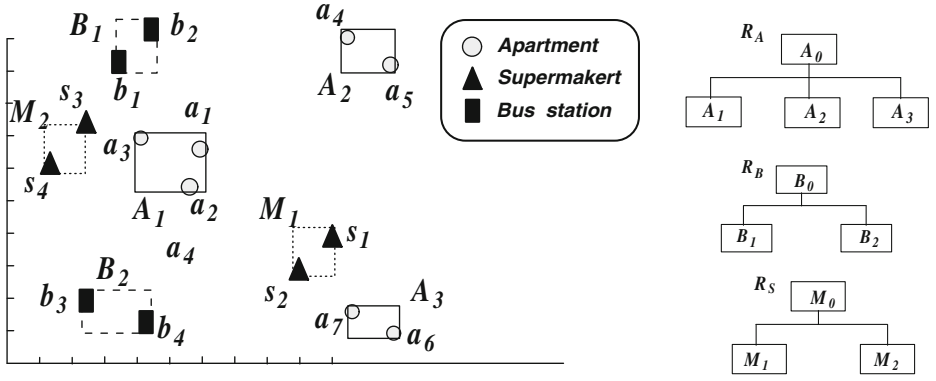


Figure 3 *R*-trees for objects and facilities.

In Figure 4, the shaded rectangles ($T_1.mdbr$, $T_2.mdbr$ and $T_3.mdbr$) in the distance space correspond to the MDBRs of the R-tree nodes A_1, A_2 and A_3 in Figure 3, which are obtained based on (2) and (3) above. Particularly, the point p_1 (p_2) in Figure 4 is the lower (upper) corner of $T_1.mdbr$, i.e., $T_1.mdbr\_low$ ($T_1.mdbr\_high$).

Based on the $T.mdbr$, we can prune the entries in facility R-trees and object R-tree from further computation. Clearly, we do not need to explore the facilities in a facility R-tree node e' in $T.\mathcal{F}_i$ for the nearest neighbor distance computation of the objects with in $T.e$ if $nnd_{\min}(T.e, e') > T.mdbr\_high[i]$ for $1 \le i \le m$. Recall that m is the number of facility types which equals 2 in the example. In Figure 3, we have $nnd_{\max}(A_1, M_2) < nnd_{\min}(A_1, M_1)$ then M_1 can be removed from $T.\mathcal{F}_2$ since none of the facilities in M_1 contributes to the nearest neighbor distance computation of A_1. This is called `distance based` pruning.

Besides the facility R-tree entries, we can also conduct the `dominance based` pruning on object R-tree entries. As shown in Figure 4, the upper corner of $T_1.mdbr$ (p_1) dominates the lower corner of $T_2.mdbr$ (p_3) in the distance space. This implies that none of the objects in A_2 (a_4 and a_5) can be $GSSKY$ object because all objects in A_2 are dominated by any object in A_1 (a_1, a_2 or a_3) according to the definition of MDBR. Consequently, we can exclude A_2 (i.e., all objects in A_2) from the $GSSKY$ computation.

Based on above observations, in Section 5.2 we will show how to effectively apply the `distance based` and `dominance based` pruning techniques in a level by level fashion.

5.2 Spatial join based *GSSKY* algorithm

We first introduce how to update the MBDR of an `NND` tuple based on a facility entry e_f with type i in Algorithm 3. For a given facility entry e_f, Line 2 calculates the minimal and maximal nearest neighbor distances between the object entry $T.e$ and the facility entry e_f based on techniques in [3, 22]. Then Line 3 updates the $MDBR$ of T ($T.mdbr$) according to (2) and (3). Moreover, a facility entry can be pruned (`distance based` pruning) in Line 6 if it cannot contribute to the nearest neighbor distance computation of $T.e$.

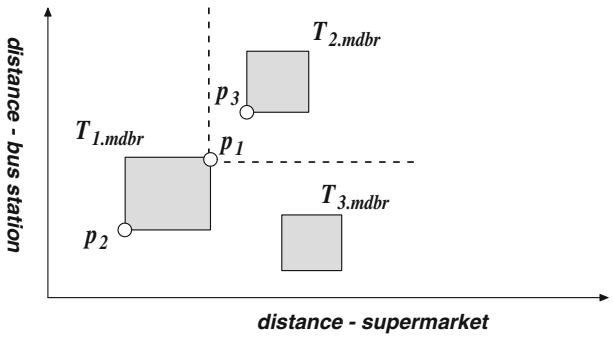


Figure 4 Example for minimal nearest neighbor distance bounding rectangle (MDBR).

Algorithm 3 *AddFacility* (T, i, e_f)

Input : T : an NND tuple,
i : the facility type,
e_f : a facility R-tree entry with type i.
Output : T : the updated NND tuple

1 $T.\mathcal{F}_i := T.\mathcal{F}_i \cup e_f$;
2 Compute $nnd_{\min}(T.e, e_f)$ and $nnd_{\max}(T.e, e_f)$;
3 Update $T.mdbr\_low[i]$ and $T.mdbr\_high[i]$;
4 **for** each facility entry e'_f in $T.\mathcal{F}_i$ **do**
5 **if** $nnd_{\min}(T.e, e'_f) > T.mdbr\_high[i]$ **then**
6 $T.\mathcal{F}_i := T.\mathcal{F}_i - e'_f$;
7 **return** T

Algorithm 4 *Spatial Join based Algorithm* $(\mathcal{O}, \mathcal{F})$

Input : $\mathcal{O}$: the objects,
$\mathcal{F}$: the facilities
Output : $\mathcal{S}$: **GSSKY**($\mathcal{O}$, $\mathcal{F}$)

1 $\mathcal{S} := \emptyset$; $H := \emptyset$;
2 Let $R_{\mathcal{O}}$ denotes the R-tree on $\mathcal{O}$ and $R_{\mathcal{F}_i}$ denotes the R-tree for facilities with type i ;
3 generate a new NND tuple T;
4 $T.e \leftarrow$ the root of $R_{\mathcal{O}}$;
5 **for** each facility type i in $[1..m]$ **do**
6 **AddFacility**(T, i, the root of $R_{\mathcal{F}_i}$) ;
7 push T into min heap H ;
8 **while** H is not empty **do**
9 $T \leftarrow H$.top();
10 Pop the top tuple of H;
11 **if** $T.e$ is data entry and all facilities in $T.\mathcal{F}_i$ are data entries **then**
12 $o \leftarrow$ the object associated with $T.e$;
13 $o.d_i = T.mdbr\_low[i]$ with $1 \le i \le m$;
14 **if SkylineTest**($\mathcal{S}$, o) **then**
15 $\mathcal{S} := \mathcal{S} \cup o$;
16 **else**
17 **for** each child entry e' of $T.e$ **do**
18 generate a new tuple t for e' where $t.e := e'$;
19 **for** each facility type i in $[1..m]$ **do**
20 **for** each child facility entry e_f in $T.\mathcal{F}_i$ **do**
21 **AddFacility**(t, i, e_f);
22 **if SkylineTest**($\mathcal{S}$, $t.mdbr\_low$) **then**
23 Push t into H ;
24 **if** $t.e$ is an intermediate entry **then**
25 put point $t.mdbr\_high$ into $\mathcal{S}$ as a dummy object;
26 **return** $\mathcal{S}$

Algorithm 4 illustrates the details of the spatial join based *GSSKY* algorithm. A min heap H is used to maintain NND tuples in the Algorithm, and the key of a tuple is the sum of the coordinate values of the lower corner of its $MDBR$, i.e., $\sum_{i=1}^{m} T.mdbr\_low[i]$. H is set empty in Line 1. An NND tuple is generated in Lines 4–6

based on the roots of object R-tree ($R_{\mathcal{O}}$) and facility R-trees ($\{R_{\mathcal{F}_i}\}$) and pushed into H in Line 7. For each NND tuple on the top of H, if the object entry and facility entries are data entries (Line 11), i.e., the nearest neighbor distances of the object are calculated (Line 13), Line 14 checks if the object (i.e., the object associated with $T.e$) is *spatially dominated* by the *GSSKY* objects seen so far, which are kept in $\mathcal{S}$. Line 15 puts the object in $\mathcal{S}$ if the object survives the skyline dominance test. On the other hand, if the object entry or one of the facility entries is an intermediate entry, Line 18 generates an NND tuple t for each child entry e' of e.[4] Then we update the MDBR of the tuple t by applying Algorithm 3 against each child entry of the facility entries in $\{T.\mathcal{F}_i\}$ with $1 \leq i \leq m$. Thereafter, an NND tuple can be safely pruned from *GSSKY* computation if it is *spatially dominated* by any object in $\mathcal{S}$. Otherwise, Line 23 pushes tuple t into H for further computation. To enable pruning object entries based on the intermediate entries, Line 25 inserts S a dummy object when an intermediate entry survives the skyline test, which corresponds to the upper corner of the entry. The algorithm terminates when H is empty and all *GSSKY* objects are kept in $\mathcal{S}$.

Correctness Similar to Algorithm 2, for the correctness of the Algorithm 4, we need the following properties: (i) if an object R-tree entry is pruned, none of the objects in the entry is *GSSKY* object, and (ii) the object in $\mathcal{S}$ cannot be dominated by any other object in $\mathcal{O}$. Below is a formal proof.

Proof Property (i): If an object R-tree entry is pruned in Algorithm 4, this implies that the lower corner of the MDBR of the entry is *spatially dominated* by another object or the upper corner of the MDBR of an object R-tree entry. It is immediate that all objects in the entry cannot be *GSSKY* object.

Property (ii): since the object R-tree entries are accessed in non-decreasing order of the sum of the distance values of the lower corner of their MDBRs, an object can only be dominated by another entry with less key values. Therefore, it is safe to claim an object is *GSSKY* object in the algorithm if it is not *spatially dominated* by any object seen so far. □

Performance analysis In the worst case, all facilities are involved in the minimal and maximal nearest neighbor distance computation for each object R-tree entry. Therefore the time cost is $O(n \times m \times n_f)$ in the worst case where n is the number of objects, m represents the number of facility types and n_f denotes the average number of facilities for each type of facility. Nevertheless, the empirical study shows that the two pruning rules (i.e., `distance based` and `dominance based` pruning rules) are quite effective. Moreover, since the R-tree entries for object R-tree and facility R-trees are accessed in a level by level fashion and an object R-tree entry will be accessed at most once in Algorithm 4, the empirical study shows that the I/O performance of Algorithm 4 significantly outperforms the competitors when the buffer size is small.

[4] In the case that $T.e$ is data entry but $T.\mathcal{F}_i$ contains an intermediate facility entry, we simply set $e' = e$ at Line 17. Similar strategy goes to facility entries in Line 20.

6 Performance evaluation

We present results of a comprehensive performance study to evaluate the efficiency and scalability of the proposed techniques in the paper. Following algorithms are evaluated.

ANN	The all nearest neighbor based technique presented in Section 3. The SFS algorithm [4] is used in Algorithm 1 for skyline computation.
P-GSSKY	The progressive *GSSKY* algorithm proposed in Section 4.
J-GSSKY	The join based *GSSKY* algorithm proposed in Section 5.

All algorithms in this paper are implemented in standard C++ with STL library support and compiled with GNU GCC. Experiments are run on a PC with Intel Xeon 2.40 GHz dual CPU and 4 G memory running Debian Linux. The disk page size is fixed to 4096 bytes. We used an LRU memory buffer whose size varies from 1 % to 100 % of the *R*-tree size of the objects.

Real datasets Two real spatial datasets, *CA* and *US* , are employed in the experiment.[5] *CA* consists of 104,217 locations of 44 different categories (e.g., church, lake and school). Each category corresponds to a facility type. The objects in *CA* are constructed as follows. We first normalize the space to $[0, 1]^2$ and then for each facility we randomly create 5 objects within distance 0.005. Consequently the number of objects in *CA* dataset is 521,085. Similarly, *US* dataset is obtained from the U.S. Geological Survey (USGS) and consists of 406,709 locations with 40 types, and the number of objects in *US* is 2,033,545. The *CA* data is the default object dataset and facility dataset in the experiments.

Synthetic datasets To study the scalability of the algorithms, we also create synthetic dataset, denoted by *SYN*, in the experiment. The objects and facilities are randomly generated in 2-dimensional space $[0, 1]^2$, i.e., both are uniformly distributed in the space. Specifically, the number of objects varies from 500K to 5M with default value 1M. There are 40 types of facilities and the number of facilities for each type varies from 500 to 10,000 with default value 2,000.

Work load The work load of each experiment consists of 200 *GSSKY* queries and *m* facility types are randomly chosen in each query where *m* varies from 2 to 5 with default value 3.

In the paper, the average *CPU* time as well as the average response time, which includes the *CPU* time and I/O latency, are used to measure the efficiency of the algorithms. We also record the average *GSSKY* size and the average number of *R*-tree nodes accessed in the algorithms.

Table 2 lists parameters which may have an impact on our performance study. In our experiments, all parameters use default values unless otherwise specified.

[5] *CA* and *US* are public available from websites http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm and http://www.geonames.usgs.gov/ respectively.

Table 2 System parameters.

Notation	Definition (default values)
m	The number of facility types (3)
n	The number of objects (1M)
n_f	The number of facilities for each type (2000)

6.1 *GSSKY* size

In this subsection, we investigate the size of the *GSSKY*. Figure 5 illustrates the *GSSKY* size on *SYN*, *CA* and *US* datasets where m varies from 2 to 5 and facilities are chosen from the *SYN*, *CA* and *US* respectively. For given m, three datasets have the similar number of *GSSKY* objects. As expected, the number of *GSSKY* objects increases quickly towards the number of types (m). Particularly, for $m = 2$ the *GSSKY* size is 13, 15 and 15 for *SYN*, *CA* and *US* respectively. When m goes to 5, it becomes 2,666, 5,508 and 5,911 respectively. For $m = 5$, the number of *GSSKY* objects occupies only around 5 %, 3 % and 6 % of the objects for *CA*, *US* and *SYN* datasets respectively. In practice, the number of facility types is usually small. Therefore, we do not further investigate the settings with $m > 5$.

Figures 6 and 7 investigate the impact of the object size (n) and facility size (n_f) respectively where the objects and facilities are from *SYN*. Since locations of the facilities with different types are independent, the Theorem 2 can be applied to estimate the *GSSKY* size, and its accuracy is verified in both Figures. Moreover, as expected, the *GSSKY* size increases slowly on the number of objects (See Figure 6) and is independent to the number of facilities (See Figure 7).

6.2 Efficiency evaluation

In this section, we evaluate the efficiency of the algorithms proposed in the paper against various factor which may potentially affect the performance of the algorithms. More specifically, we will investigate the impact of the buffer size, data distribution, the number of facility types(m), the number of objects (n), the average number of facilities for each type of facilities (n_f).

Impact of buffer size We first evaluate the effect of the buffer on the performance of the *ANN*, *P-GSSKY* and *J-GSSKY* algorithms where *CA* dataset is employed. In the experiments, we record the number of pages accessed by the algorithms as well as the query response time. Figure 8 shows the response time and the number of I/O accesses as a function of buffer size (%). As expected, the performance of

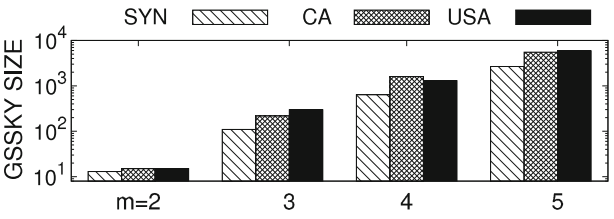


Figure 5 Different datasets and m.

Figure 6 Different n.

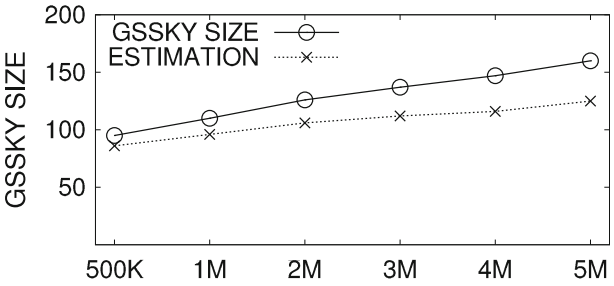
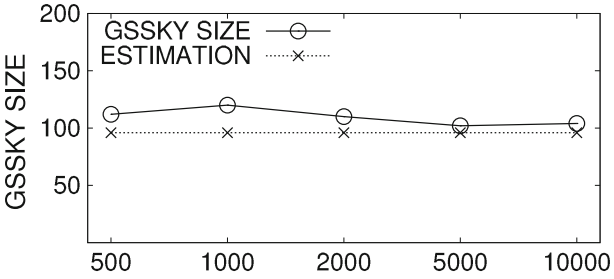


Figure 7 Different n_f.



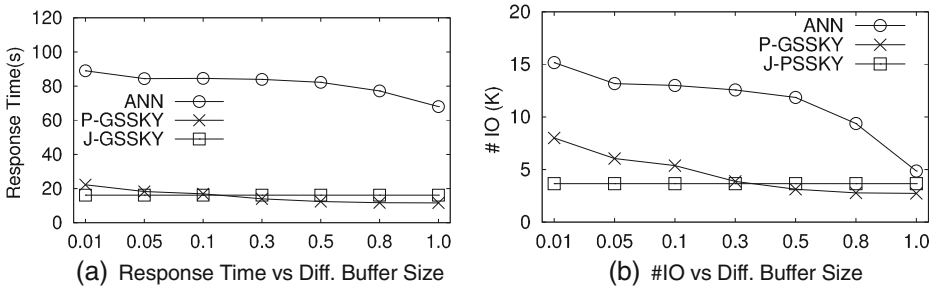


Figure 8 Impact of the buffer size.

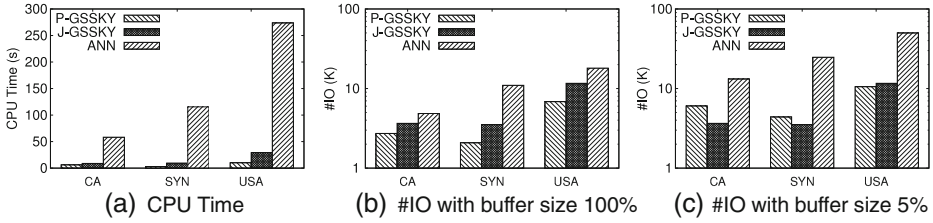


Figure 9 Impact of data distribution.

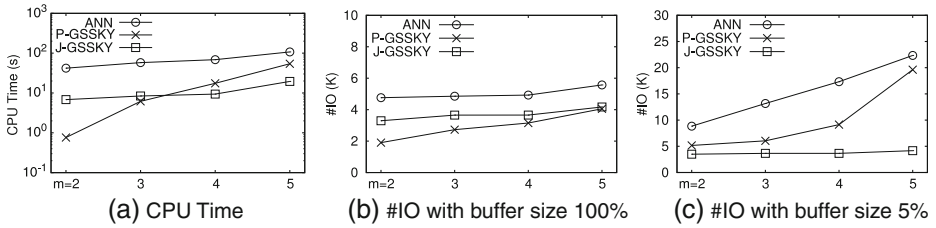


Figure 10 Impact of the number of facility types (m).

J-GSSKY scale well on the buffer size since the *J-GSSKY* carefully consider the I/O efficiency while the performance of *P-GSSKY* degrades sinigificantly when the buffer size is small. Moreover, as shown in Figure 8a, *P-GSSKY* and *J-GSSKY* algorithms significantly outperform the *ANN* algorithm especially when the buffer size becomes small.

Impact of data distribution In the second set of experiments, we evaluate the performance of the algorithms on *SYN*, *CA* and *US* datasets respectively. Figure 9a shows that *P-GSSKY* has the best performance in terms of *CPU* cost, and *J-GSSKY* is slightly slower than *P-GSSKY* but still significantly outperforms the *ANN* algorithm. Figure 9b and c show the number of I/O accesses of the algorithms when the buffer size equals 100 % and 5 % respectively. Observe that the I/O efficiency of the *P-GSSKY* is quit sensitive to the buffer size while the *J-GSSKY* scale well to the buffer size. For instance, in *CA* dataset, *P-GSSKY* invokes 6044 and 2727 disk accesses when buffer size equals 5 % and 100 % respectively, while the number of I/Os accessed by *J-GSSKY* remains the same (3654).

Impact of the number of facility types (m) Figure 10 investigates the performance of *ANN*, *P-GSSKY* and *J-GSSKY* algorithms as a function of the number of facility types (m). It is shown that performance of all algorithms degrades against m due to larger size of *GSSKY* objects and more distance computations.

As discussed in Section 5.1, the cost of *P-GSSKY* is more sensitive to the number of facilities since we need to conduct incremental nearest neighbor search for each facility. It is observed that the performance of *P-GSSKY* does not scale well towards

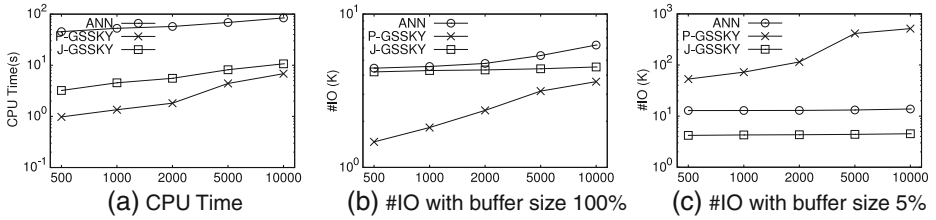


Figure 11 Impact of the number of facility types (n_f).

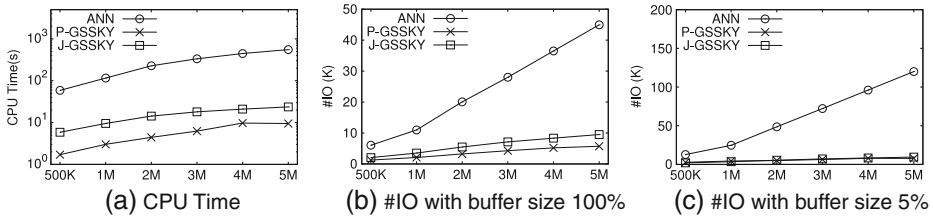


Figure 12 Impact of the number objects (n).

the number of facility types, even when the buffer size is large. On the other hand, the performance of *J-GSSKY* degrades much slower than that of *ANN* and *P-GSSKY*.

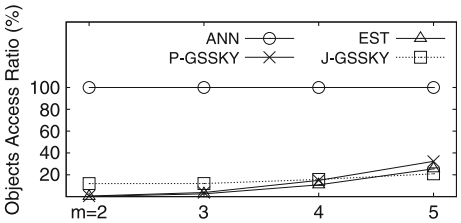
Impact of the number of facilities for each type (n_f) Figure 11 evaluates the performance of *ANN*, *P-GSSKY* and *J-GSSKY* algorithms against the number of facilities for each type (n_f) where the facilities are from *SYN* dataset whose size ranges from 200 to 10000. Similar trends in Figure 10 are observed in Figure 11 where *P-GSSKY* does not scale well towards n_f, i.e., the total number of facilities. Moreover, it is noticed in Figure 11c that the I/O efficiency of *P-GSSKY* is poor when the buffer size is small.

Impact of the number of objects (n) Figure 12 evaluates the performance of *ANN*, *P-GSSKY* and *J-GSSKY* algorithms as a function of the number of objects in the *SYN* dataset where n ranges from 500K to 5M. It is shown that both *P-GSSKY* and *J-GSSKY* algorithms are scalable to the number of objects, while the *CPU* and I/O costs of *ANN* grow significantly.

We evaluate the number of objects accessed in *ANN*, *P-GSSKY* and *J-GSSKY* algorithms where both objects and facilities are from *SYN* dataset, and the number of facility types m grows from 2 to 5. Moreover, we depict the estimated number of objects accessed by *P-GSSKY*, denoted by *EST*, based on Theorem 4. Figure 13 reports that all objects contribute to the distance calculation in *ANN* algorithm, while a large number of objects are pruned in *P-GSSKY* and *J-GSSKY* algorithms and hence the computational costs can be significantly reduced. It also demonstrates the accuracy of the Theorem 4.

In the last set of experiments, Figure 14 reports the maximal heap size of three algorithms during the *GSSKY* computation to evaluate the memory usage of three algorithms. As expected, *P-GSSKY* takes much more memory space than *ANN* and *J-GSSKY* because a heap is maintained for each facility during the computation.

Figure 13 Objects accessed vs m.



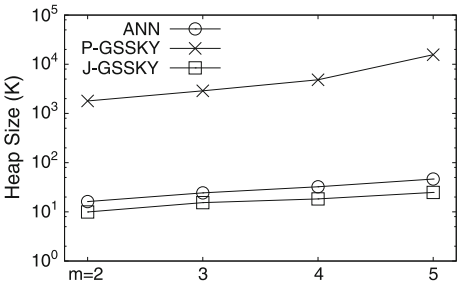


Figure 14 Heap size vs m.

6.3 Summary

As a short summary, our comprehensive performance study clearly demonstrates the effectiveness of *GSSKY* since we only need to maintain a small fraction of objects (*GSSKY* objects) to guarantee that all optimal solutions for arbitrary monotonic distance based spatial preference functions are kept. Moreover, the extensive performance evaluation shows that two proposed algorithms, *P-GSSKY* and *J-GSSKY*, are both effective and efficient. Particularly, when the number of facilities is not large and there is sufficient main memory available, which is not uncommon in many applications, the *P-GSSKY* algorithms has the best performance among three algorithms. Although the performance of *J-GSSKY* is slightly less efficient than *GSSKY* regarding the settings favoring the *GSSKY*, our extensive experiments shows that *J-GSSKY* is the most I/O efficient algorithm and scale well against the number of facilities.

7 Related work

Studies on skyline computation have a long history. Börzsönyi et al. [1] first investigate the skyline computation problem in the context of databases and propose an SQL syntax for the skyline query. They also develop the skyline computation techniques based on *block-nested-loop* and *divide-conquer* paradigms, respectively. Chomicki et al. [4] propose another block-nested-loop based computation technique, SFS (*sort-filter-skyline*), to take the advantages of a pre-sorting. Papadias et al. [15] propose a branch and bound search technique (BBS) to progressively output skyline points on datasets indexed by R-tree.

The problem of *spatial skyline* is first proposed in [19]. Given a set $\mathcal{O}$ of objects and a set $\mathcal{Q}$ of query points, each object has $|\mathcal{Q}|$ derived spatial attributes each of which is the distance of the object to a query point in $\mathcal{Q}$, and hence can be mapped to a point in $|\mathcal{Q}|$-dimensional space where $|\mathcal{Q}|$ is the number of query points in $\mathcal{Q}$. Then the spatial skyline regarding $\mathcal{O}$ and $\mathcal{Q}$ is the traditional skyline on $|\mathcal{Q}|$-dimensional space. Efficient algorithms are developed in [19] to compute spatial skylines by utilizing the R-Tree, convex hull, and voronoi diagram techniques. Son et al. further improve the spatial skyline computation techniques in [20]. Moreover, in [21] they investigate the problem based on the manhattan distance. In [5], Ke et al. investigate the problem in the road network. Besides the spatial skyline, there are also some related work in which skyline is computed based on the derived spatial attributes. In [9],

Huang et al. studies the problem of in-route skyline to find locations which are not dominated by other candidate locations regarding the network distance to a query location q and the corresponding detour distance. In [7, 12] spatial distance regarding a query point q is considered during the skyline computation, in which other dimensions of an objects are non-spatial attributes.

In many applications, the query points may come from the same categories (e.g., bus stations, supermarkets). For an object o and a particular category (i.e., facility type like bus station), users are only interested in the distance between o and its closest query point (i.e., facility) in that category. Consequently, the spatial skyline does not make sense in these applications because it does not considers the distances of o regarding a set of facilities (i.e., more than one facility) in the same category, and hence cannot provide minimal candidate set for distance based spatial preference queries studied in [13, 17, 23]. Moreover, the techniques in [19, 21] cannot be applied to the *GSSKY* computation because their pruning techniques assume there is only one facility for each type of facilities.

8 Conclusion and future work

In this paper, we introduce the *general spatial skyline* which can provide minimal candidate set of the optimal solution regarding any monotonic distance based spatial preference query. Efficient algorithms are proposed in the paper and comprehensive experiments are conducted to demonstrate the effectiveness and efficiency of the algorithms. As a possible future work, we will investigate the problem on the road network in which the network distance is considered.

References

1. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: ICDE (2001)
2. Brinkhoff, T., Kriegel, H.P., Seeger, B.: Efficient processing of spatial joins using r-trees. In: SIGMOD Conference, pp. 237–246 (1993)
3. Chen, Y., Patel, J.M.: Efficient evaluation of all-nearest-neighbor queries. In: ICDE (2007)
4. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: ICDE (2003)
5. Deng, K., Zhou, X., Shen, H.T.: Multi-source skyline query processing in road networks. In: ICDE (2007)
6. Godfrey, P.: Skyline cardinality for relational processing. In: FoIKS (2004)
7. Guo, X., Ishikawa, Y., Gao, Y.: Direction-based spatial skylines. In: MobiDE (2010)
8. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. TODS **24**(2), 265–318 (1999)
9. Huang, X., Jensen, C.S.: In-route skyline querying for location-based services. In: W2GIS (2004)
10. Huang, Y.W., Jing, N., Rundensteiner, E.A.: Spatial joins using R-trees: breadth-first traversal with global optimizations. In: VLDB (1997)
11. Huang, J., Wen, Z., Qi, J., Zhang, R., Chen, J., He, Z.: Top-k most influential locations selection. In: CIKM, pp. 2377–2380 (2011)
12. Kodama, K., Iijima, Y., Guo, X., Ishikawa, Y.: Skyline queries based on user locations and preferences for making location-based recommendations. In: GIS-LBSN (2009)
13. Lin, Q., Xiao, C., Cheema, M.A., Wang, W.: Finding the sites with best accessibilities to amenities. In: DASFAA (2011)
14. Lin, Q., Zhang, Y., Zhang, W., Li, A.: General spatial skyline operator. In: DASFAA, pp. 494–508 (2012)
15. Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: SIGMOD (2003)

16. Qi, J., Zhang, R., Kulik, L., Lin, D., Xue, Y.: The min-dist location selection query. In: ICDE, pp. 366–377 (2012)
17. Rocha-Junior, J.B., Vlachou, A., Doulkeridis, C., Nørvåg, K.: Efficient processing of top-k spatial preference queries. PVLDB **4**(2), 93–104 (2010)
18. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: SIGMOD Conference (1995)
19. Sharifzadeh, M., Shahabi, C.: The spatial skyline queries. In: VLDB (2006)
20. Son, W., Lee, M.W., Ahn, H.K., won Hwang, S.: Spatial skyline queries: an efficient geometric algorithm. In: SSTD (2009)
21. Son, W., won Hwang, S., Ahn, H.K.: Mssq: Manhattan spatial skyline queries. In: SSTD (2011)
22. Xia, T., Zhang, D., Kanoulas, E., Du, Y.: On computing top-t most influential spatial sites. In: VLDB (2005)
23. Yiu, M.L., Dai, X., Mamoulis, N., Vaitis, M.: Top-k spatial preference queries. In: ICDE (2007)