Information Sciences 305 (2015) 110–129

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Rotating MaxRS queries

Zitong Chen [a], Yubao Liu [a,*], Raymond Chi-Wing Wong [b], Jiamin Xiong [a], Xiuyuan Cheng [a], Peihuan Chen [a]

[a] Department of Computer Science, Sun Yat-Sen University, China
[b] Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China

ARTICLE INFO

Article history:
Received 29 March 2014
Received in revised form 7 November 2014
Accepted 1 February 2015
Available online 10 February 2015

Keywords:
Optimal location queries
MaxRS queries
Location-based service

ABSTRACT

Given a set of weighted objects in a data space, the MaxRS problem in spatial databases studied in a VLDB 2012 paper is to find a location for a rectangular region of a given size such that the weighted sum of all the objects covered by the rectangular region centered at the optimal location is maximized. This problem is useful in lots of location-based service applications, such as finding the location for a new fast food restaurant with a limited delivery range attracting the greatest number of customers. The existing MaxRS problem assumes that the rectangular region is always placed horizontally and is non-rotatable. However, under this assumption, the weighted sum of all the covered objects may not be the greatest when the rectangular region is rotatable. In this paper, we propose a generalized MaxRS problem called rotating MaxRS without this assumption. In rotating MaxRS, the rectangular region is rotatable and can be associated with an inclination angle. The goal of our problem is to find a location and an inclination angle such that the weighted sum of all the objects covered by the rectangular region of a given size centered at this location with this inclination angle is the greatest. We also present an efficient algorithm for the problem. Extensive experiments were conducted to verify the efficiency of our algorithms based on the real and synthetic datasets. The experimental results show that the weighted sum of all the objects in the rotating MaxRS queries can be increased with up to 300% on the synthetic datasets compared with existing non-rotating MaxRS queries, which shows the significance of the new rotating MaxRS queries.

1. Introduction

With the rapid development of location-based service applications, researchers in the database community have paid attention to the analysis issues of location-related data. Recently, several location analysis problems have been proposed. The *maximizing range sum* (MaxRS) problem [3] is one of these location analysis problems.

Given a set O of weighted objects in a two-dimensional data space, the MaxRS problem is to find a location for a rectangular region of a given size such that the weighted sum of all the objects covered by the rectangular region centered at this location is maximized. This optimal location query is very important and useful as a basic operation in a lot of real applications such as location planning, location-based service and profile-based marketing.

* Corresponding author at: No. 132, Wai Huan DongLu, Guangzhou Higher Education Mega Center, Guangzhou 510006, China. Tel.: +86 2039943315.
E-mail address: liuyubao@mail.sysu.edu.cn (Y. Liu).

http://dx.doi.org/10.1016/j.ins.2015.02.009
0020-0255/

Application Example 1

Assume that O denotes a set of residential estates in a city. We want to build a new fast food restaurant in the regions formed by a grid of streets. Consider the fast food delivery capability is often limited in practice. A key question is how to find a location for the new fast food restaurant such that the greatest number of the residential estates around the restaurant are attracted. We can assume the fast food delivery range corresponds to a rectangular region with a limited size. Then, a MaxRS query can answer the question and return an optimal location for the new fast food restaurant.

Application Example 2

There are some game applications such as the World of Tanks in which a variety of personalized game weapons are included. In such game applications, we may design a kind of personalized weapons which is provided with a limited attack area such as a fixed size of a rectangle range. Then, given a set of objects to be attacked in a two-dimensional space, a MaxRS query can be used to locate intelligently the position of the attack area such that the number of objects attacked by the weapon is the greatest.

The existing MaxRS problem assumes that the rectangular region is always placed horizontally and is non-rotatable.

However, for Application Example 1, the streets in a city are not always horizontal and the objects such as residential estates are often distributed on both sides of the streets. Thus, the rectangular regions of a limited size covering these objects are not always horizontal and is rotatable in practice. For example, a rotatable rectangular region may be better to represent the fast food delivery range in the application examples.

Moreover, under this assumption, for Application Example 2, the answer returned by the existing MaxRS queries may not be the greatest. For example, in Fig. 1, there are seven objects to be attacked, namely, o_1, o_2, ... and o_7. Each object is located at a point in a two-dimensional data space. Assume that each object has the same weight. Given the dimension of a rectangular region as the limited attack area, says $l \cdot w$ where l and w are the length and the width of the region, the existing MaxRS query returns the point p_1 as the optimal location as shown in Fig. 1(a). The rectangular region centered at p_1, whose size is $l \cdot w$, covers four objects o_1, o_2, o_3, and o_4 (i.e., the attacked objects). However, p_1 is not the best place if the rectangular region is *rotatable*. As shown in Fig. 1(b), there exists another optimal location p_2 such that the rectangular region centered at p_2 and *rotated* with the optimal inclination angle $\theta_1 = 80^\circ$ covers five objects, namely, o_2, o_3, o_4, o_5 and o_6, the greatest number of objects covered. Note that this rectangular region can have multiple optimal angles. For example, $\theta_2 = -80^\circ$ in Fig. 1(b) is another optimal angle.

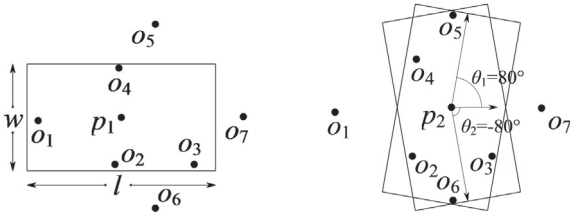
How to find such a rotatable rectangular region for the practical application has not been studied before.

Motivated by the above observations, in this paper, we propose a generalized MaxRS problem called *rotating MaxRS* in which the assumption mentioned in the existing problem is removed. The rectangular region is rotatable and can be associated with an inclination angle. Specifically, the purpose of our problem is to find a location and an inclination angle such that the weighted sum of all the objects covered by the rectangular region of a given size centered at this location and rotated with this inclination angle is the greatest.

In this paper, we simply call the rectangular region of a given size centered at a point in the data space the *given rectangle* of this point. For clarity, when we write "rectangle", we mean "rectangular region", and both terms are used interchangeably in the rest of the paper.

To the best of our knowledge, we are the first to focus on this generalized problem. The number of candidate points in a data space for the optimal location and the number of candidate inclination angles for the optimal angle are *infinite*. It is challenging to find the optimal location and the optimal angle. Since the inclination angle has not been considered in the literature, existing solutions are not suitable for our problem. We need to design a new algorithm for our problem.

As shown in Section 3, the optimal locations are contained in the *most overlapped region* of the rectangular regions centered at the given object points and rotated with the *same* optimal inclination angle. For example, Fig. 2 shows 5 rectangular regions centered at o_2, o_3, o_4, o_5 and o_6 rotated with an angle of 80°. The optimal location p_2 (described in Fig. 1(b)) is inside



(a) The MaxRS problem (b) Our problem

Fig. 1. The problem comparison.

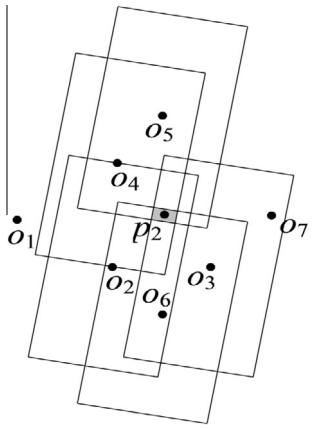


Fig. 2. An example for the problem property.

the shaded region which denotes to the most overlapped region of the rectangular regions centered at the given object points and rotated with the *same* optimal inclination angle.

Based on the above property, we propose an efficient algorithm which finds the optimal location and the optimal angle such that the weighted sum of all the objects covered is the greatest.

The contributions of this paper are as follows. (1) We proposed a generalized MaxRS problem called rotating MaxRS. Compared with the existing MaxRS query, a rotating MaxRS query returns both a location and an inclination angle for a rectangular region of a given size such that the weighted sum of all the objects covered by the rectangular region centered at this location and rotated with this inclination angle is the greatest. (2) We presented an efficient solution for our problem. This solution is based on partitioning the data space. By pruning lots of regions and only examining promising space regions, the proposed solution can quickly find the optimal location and the optimal angle. (3) We conducted extensive experiments to verify the efficiency of our algorithms. Based on real and synthetic datasets, the experimental results show the efficiency of our proposed solution. As shown in the experiments, in the synthetic datasets containing 2,500,000 objects, a rotating MaxRS query can be answered within 300 s in most cases. On the real dataset containing more than 120,000 objects, a rotating MaxRS query can be answered within 6 s. The experimental results also show that the weighted sum of all the covered objects in the rotating MaxRS queries can be increased with up to 300% on the synthetic datasets compared with the existing non-rotating MaxRS queries.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 gives the problem definition. Section 4 describes our proposed query algorithms. Section 5 evaluates the proposed algorithms. Section 6 concludes this paper with future directions.

2. Related work

The location analysis problem in the database community originally came from the *facility location problem* [2,8,15], which has been extensively studied in past years. The facility location problem is to locate preferred facilities with respect to a given set of clients, and is shown to be NP-hard. Different from the facility location problem where the number of the candidate locations for the facilities is usually limited, in the location analysis problem, the number of the candidate locations to be found can be infinite. It is challenging to find the optimal locations from the whole data space.

In the following, we classify the related studies into 3 parts. The first part (Section 2.1) is the nearest-neighbor based optimal location problem. In this problem, we want to find the optimal location by finding the nearest neighbor of each object first and then obtaining a coverage area of each object. The second part (Section 2.2) is the coverage based optimal location problem. In this problem, similarly, we want to find the optimal location by using the coverage area given by a user. The third part (Section 2.3) is the spatial preference query. In this problem, we want to find the optimal location by using the spatial preferences.

2.1. Nearest-neighbor based optimal location problem

Several location analysis issues have been studied [3,17,18,10,24,21,4,23,12,20,9,19]. The MaxBRNN problem [2] is to find a region with the greatest *influence* in the L_2-norm space. A solution with an exponential-time complexity was proposed in [2].

The MaxBRNN problem was also studied in [17] in which the first polynomial-time complexity algorithm, *MaxOverlap*, was proposed. Some variations, such as the extension of the *MaxOverlap* algorithm in a three-dimensional space and other L_p-norm metric spaces, were studied in [18], an extended version of [17]. Recently, the *MaxSegment* algorithm, an improved

algorithm for the MaxBRNN problem, was proposed in [10]. Both the running time and the storage cost of the *MaxSegment* algorithm are significantly smaller than the *MaxOverlap* algorithm. The improved algorithm *OptRegion* for the MaxBRNN problem in a two-dimensional space was proposed in [9]. An approximate method was recently proposed for the MaxBRNN problem in [21]. Furthermore, the generalized MaxBR*k*NN problem was studied in [24] where a client may have different probabilities to visit different servers and at the same time, a server is assumed to have different target sets of clients.

Besides, the algorithm in [4] was proposed to find an optimal location instead of an optimal region for the L_1-norm space. [23] proposed the min-dist optimal location query that finds a location which minimizes the average distance from each client to its closest server when a new server is built at this location. [12,13] proposed to select a location from a given set of potential locations for a new server so that the average distance between a client and its nearest server is minimized. [19] proposed to find the top-t most influential sites from a given set of sites in a given spatial region. [20] firstly proposed an algorithm framework to find all optimal locations in the setting of road networks. [6] proposed the continuous min–max distance bounded query in road networks.

2.2. Coverage based optimal location problem

In the above studies, there are two kinds of objects involved, namely clients and servers. The optimal cost functions computed based on the clients and the servers are used to locate the optimal locations in the data space. However, there is only *one type* of objects in the existing MaxRS problem and thus the objects do *not* have their competitive or collaborative relationship [3]. The optimal cost function in the MaxRS problem is to maximize the weighted sum of all the objects covered by the rectangular region centered at the optimal location.

In fact, in the computational geometry community, the MaxRS problem has been extensively studied in past years, which is called the *max-enclosure rectangle problem* [7,11]. Recently, a scalable algorithm extending existing solutions was presented in [3]. The approximate MaxRS was studied in [16]. Different from these studies, we focus on a generalized problem which cannot be addressed by existing solutions. We also proposed a new solution for this problem.

To the best of our knowledge, we are the first to consider the rotating MaxRS problem and propose a new solution to find the optimal locations and the optimal angle for the given rectangular region.

2.3. The spatial preference query

The other related studies include the top-k spatial preference query [14,22]. Similar to our problem, these studies select the related spatial objects based on the cost functions from a given data space. The top-k spatial preference query returns the k objects in the spatial database with the highest *scores*. The score of an object is computed based on the ranking function which is defined with the quality of features of an object. Different from our problem, the number of the candidate points to be selected is often *limited* in these queries.

3. Problem definition

Given a set O of objects in a two-dimensional data space S, each object $o \in O$ is associated with a positive weight, denoted by $w(o)$, which denotes the importance of the client. For example, $w(o)$ corresponds to the number of people in the residential estate o in the application example. Each object is located at a point in the data space. The dimension of a rectangular region R is specified as $l \cdot w$, where $0 < w \leqslant l$. Here, l and w correspond to the *length* and the *width* of the region. The rectangular region of a given size centered at a point $p \in S$ is called the *given rectangle* of p, denoted by R_p. A given rectangle R_p rotated with an inclination angle θ can be denoted as $R_p(\theta)$. Consider the Cartesian coordinate system whose origin is the center point p. The *inclination angle* of R_p is formed by the horizontal axis pointing to the right and the line parallel to the long side of R_p. In general, an inclination angle θ ranges from -90° to 90°. For example, as shown in Fig. 1(b), the rectangular region centered at the optimal location p has an inclination angle $\theta_1 = 80^\circ$, which can be denoted as $R_p(80^\circ)$.

Our rotating MaxRS problem is defined as follows.

Definition 1 (*Problem*). Given a set O of weighted objects in a two-dimensional data space S, and a rectangular region of a given size, our problem is to find an optimal location $p \in S$ and an optimal angle θ such that $\sum_{o_i \in R_p(\theta) \wedge o_i \in O} w(o_i)$ is the greatest.

For example, in Fig. 1(b), p_2 is an optimal location, and the rectangular region centered at p_2 has the optimal angle $\theta = 80^\circ$. This means that the sum of the weights of all the objects covered by this rectangular region centered at this location and rotated with this angle is the greatest. If each object has the same weight, then our problem is to find the greatest number of objects covered by the given rectangle centered at the optimal location and rotated with the optimal angle. There may have multiple optimal locations which are in a consecutive region, we call this region the *optimal location region*.

In the existing MaxRS problem [3,7,11], an important property is used. Specifically, if the point $p \in S$ is covered by the given rectangle of $o_i \in O$, then o_i is also covered by the given rectangle of p. Based on this property, the optimal location region corresponds to the most overlapped region among the given rectangles of all the objects in O.

Fortunately, in our problem, we can have the similar property. That is, *if the point $p \in S$ is covered by the given rectangle of $o_i \in O$ rotated with the angle θ, then o_i is also covered by the given rectangle of p rotated with the same angle θ.*

Based on this property, we can derive that given an optimal angle, the optimal location region corresponds to the *most overlapped region* of the given rectangles of all the objects rotated with the same optimal angle. In particular, the most overlapped region may correspond a single point or a line segment.

As shown in Fig. 2, the optimal location p_2 in Fig. 1(b) is covered by the given rectangles of o_2, o_3, o_4, o_5 and o_6 all rotated with the same optimal angle $\theta = 80^\circ$. These objects are also covered by the given rectangle of p_2 rotated with the same optimal angle $\theta = 80^\circ$.

In the existing MaxRS problem, the rectangular region is always placed horizontally and that is equivalent that the inclination angle for the rectangular region is always set to 0°. Thus, the existing MaxRS problem can be viewed as a special case of our problem in which the inclination angle is restricted to 0°. However, in practice, the optimal angle for the rectangular region may not be equal to 0°. Thus, the existing solutions cannot find the best location for a rotatable rectangular region.

In our problem, the best location for the rectangular region is to be found. The challenges for our problem involves two aspects. One aspect is the *infinite* number of points in the data space which causes the difficulty in finding an optimal location. The other aspect is the *infinite* number of angles which causes the difficulty in finding an optimal angle.

Since the existing solutions cannot support the inclination angle not equal to 0°, they cannot be applicable to our problem. Thus, we have to design a new algorithm for our problem.

4. The proposed algorithms

In this section, we would like to present a baseline algorithm and then introduce the proposed algorithms for our problem. The basic concepts, the theoretical properties and the algorithm analysis are included in this section.

4.1. Baseline algorithm

Our proposed algorithm to be described later solves a general rotating MaxRS problem where both the optimal angle and the optimal location are to be found. The existing algorithm solves a specific non-rotating MaxRS problem where there is no need to find the optimal angle.

In general, we can design a baseline algorithm for our problem based on the existing algorithms for the MaxRS problem. Since the existing scalable algorithm in [3] is an I/O-optimal algorithm, we choose the existing memory based solutions [7,11] to design the baseline algorithm. The baseline algorithm works as follows.

Step 1: We are given an optimal angle at the first place. Since there are no solutions for finding such an optimal angle for a given dataset D, we can execute our proposed algorithm on D and obtain both the optimal location p and the optimal angle θ.
Step 2: We rotate the x-axis and the y-axis in D using the optimal angle θ found and obtain the rotated dataset D'.
Step 3: We execute the existing algorithm on this rotated dataset D' and obtain the optimal location p' for D'.

It is easy to know that the weighted sum of all the objects covered by the rectangular region of a given size centered at the optimal location p' with the optimal angle θ is also the greatest for D in our problem.

However, this baseline algorithm involves Step 1 which needs an optimal algorithm to find the optimal angle. In the following, we will present the proposed algorithm for our problem.

4.2. Basic concepts

In general, given any two angles θ_1 and θ_2, we define an *angle interval* in the form of $[\theta_1, \theta_2]$ as the set of all angles which are smaller than or equal to θ_2 and is larger than or equal to θ_1, where $-90^\circ \leqslant \theta_1 \leqslant \theta_2 \leqslant 90^\circ$.

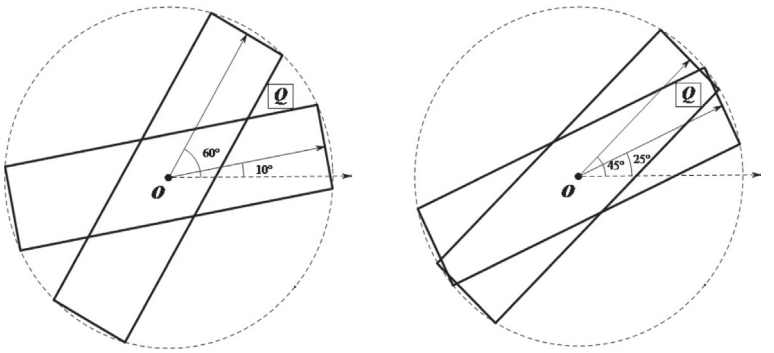
The basic idea for the proposed algorithm is to iteratively partition the whole data space into a number of small quadrants and search the quadrants which possibly contain a part of the optimal location region. The partitioning process is guided by the *upper* and *lower bounds* of a quadrant Q (denoted by $UPP(Q)$ and $LOW(Q)$, respectively).

In general, given a quadrant Q and a set O of weighted objects, we define the following concepts.

Definition 2 (*Intersection Angle Range*). The *intersection angle range* of a quadrant Q at $o \in O$, denoted by $IAR_o(Q)$, is defined as $IAR_o(Q) = \{\theta | R_o(\theta) \cap Q \neq \emptyset\}$. That is, it is equal to a set of angle values θ such that the given rectangle of o rotated with θ, $R_o(\theta)$, intersects the region Q.

Definition 3 (*Containment Angle Range*). The *containment angle range* of Q at $o \in O$, denoted by $CAR_o(Q)$, is defined as $CAR_o(Q) = \{\theta | Q \subseteq R_o(\theta)\}$. That is, it is equal to a set of angle values θ such that the given rectangle of o rotated with $\theta, R_o(\theta)$, contains the region Q.

Fig. 3 shows the examples for the intersection angle range and containment angle range, where $IAR_o(Q)$ is equal to $[10^\circ, 60^\circ]$ and $CAR_o(Q)$ is equal to $[25^\circ, 45^\circ]$.



(a) $IAR_o(Q)$ (b) $CAR_o(Q)$

Fig. 3. The examples for the intersection angle range and containment angle range.

$IAR_o(Q)/CAR_o(Q)$ is equal to $\emptyset$ if R_o does not intersect/contain Q for all possible inclination angles. It is obvious that $CAR_o(Q) \subseteq IAR_o(Q)$. In general, the union of $IAR_o(Q)/CAR_o(Q)$ for *all* $o \in O$ is represented as $IAR(Q)/CAR(Q)$. That is, $IAR(Q) = \cup_{o\in O} IAR_o(Q)$ and $CAR(Q) = \cup_{o\in O} CAR_o(Q)$.

Definition 4 (*Upper and Lower Angle Range*). The *upper angle range* of Q, denoted by $UAR(Q)$, is defined as $UAR(Q) = \{\theta | \arg\max_\theta \sum_{o\in O\wedge\theta\in IAR_o(Q)} w(o)\}$. The *lower angle range* of Q, denoted by $LAR(Q)$, is defined as $LAR(Q) = \{\theta | \arg max_\theta \sum_{o\in O\wedge\theta\in CAR_o(Q)} w(o)\}$.

Next, we take an example to illustrate the above concepts. Assume that Q intersects R_{o_1}, R_{o_2}, R_{o_3}, R_{o_4} and R_{o_5}. Their intersection angle ranges are represented as the angle intervals which are denoted as line segments in Fig. 4.

Assume that each object has the same weight 1. $UAR(Q)$ and $UPP(Q)$ can be computed as follows. Suppose that $IAR(Q)$ is known in advance (we will describe how to compute it later). We can scan all the angle intervals in $IAR(Q)$ in the ascending order of angles and find out these angles shared by the most objects. In this example, it is easy to derive that $UAR(Q)$ corresponds to the interval $[70°, 80°]$ which is shared by four objects o_2, o_3, o_4 and o_5. Similarly, if $CAR(Q)$ is known in advance, then we can compute $LAR(Q)$.

Intuitively, if each object has the same weight then $UAR(Q)/LAR(Q)$ corresponds to the common angle ranges in $IAR(Q)/CAR(Q)$ shared by the most objects.

Definition 5 (*Upper and Lower Bounds*). We define the upper and lower bounds on the maximum weighted sum of objects in O covered by a given region of a point in Q, denoted by $UPP(Q)$ and $LOW(Q)$, respectively. $UPP(Q) = \max_\theta \sum_{o\in O\wedge\theta\in IAR_o(Q)} w(o)$ and $LOW(Q) = \max_\theta \sum_{o\in O\wedge\theta\in CAR_o(Q)} w(o)$.

In our example in Fig. 4, since there exists an angle θ (e.g., 70°) such that $R_{o_2}(\theta)$ intersects Q, $UPP(Q)$ is equal to 4. Similar to $UPP(Q)$, we can compute $LOW(Q)$.

For each point p in Q, we denote $WS(p)$ to be the weighted sum of all the objects covered by $R_p(\theta)$ which is the largest among all possible angles θ. That is, $WS(p) = \max_\theta \sum_{o\in O\wedge p\in R_o(\theta)} w(o)$.

Lemma 1. *For each* $p \in Q, LOW(Q) \leqslant WS(p) \leqslant UPP(Q)$.

Proof. Since $p \in Q$, for any θ, we can derive $p \in R_o(\theta)$ from $\theta \in CAR_o(Q)$ and we can derive $\theta \in IAR_o(Q)$ from $p \in R_o(\theta)$. So, for any θ, we can derive $\sum_{o\in O\wedge\theta\in IAR_o(Q)} w(o) \leqslant \sum_{o\in O\wedge p\in R_o(\theta)} w(o) \leqslant \sum_{o\in O\wedge\theta\in IAR_o(Q)} w(o)$.

Next, we can derive $\max_\theta \sum_{o\in O\wedge\theta\in IAR_o(Q)} w(o) \leqslant \max_\theta \sum_{o\in O\wedge p\in R_o(\theta)} w(o) \leqslant \max_\theta \sum_{o\in O\wedge\theta\in IAR_o(Q)} w(o)$.

That is, $LOW(Q) \leqslant WS(p) \leqslant UPP(Q)$. This lemma holds. □

Lemma 1 shows the correctness of the upper and lower bounds for Q (i.e., $UPP(Q)$ and $LOW(Q)$).

Next, we would like to introduce how to compute $IAR(Q)$ and $CAR(Q)$. Before that, we give some notations to be used for computing $IAR(Q)$ and $CAR(Q)$. In general, a partitioning quadrant Q corresponds to a rectangular region in the data space which is formed by four vertices and four edges. Each vertex can be any point in the data space and each edge is formed by two vertices. Without loss of generality, assume that Q consists of the vertices v_1, v_2, v_3 and v_4 and the edges $e_1 = (v_1, v_2)$, $e_2 = (v_2, v_3)$, $e_3 = (v_3, v_4)$ and $e_4 = (v_4, v_1)$. Note that the location for each vertex can be known when we partition the data space.

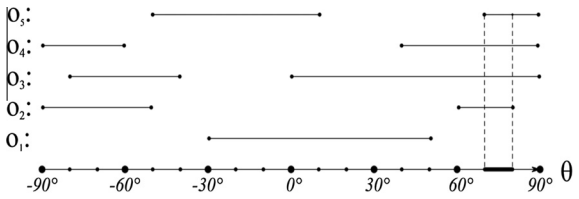


Fig. 4. An example for the upper bound.

Definition 6 (*Vertex Range*). For any vertex v of Q, the *vertex range* of v with $o \in O$ is defined as $VertexR(v, o) = \{\theta | v \in R_o(\theta)\}$.

Definition 7 (*Edge Range*). For any edge $e = (v_l, v_r)$ of Q, the *edge range* of e with $o \in O$ is defined as $EdgeR(e, o) = \{\theta | R_o(\theta)$ intersects $e\}$.

Fig. 5 shows an example for *VertexR* and *EdgeR*, where Q consists of the vertices v_1, v_2, v_3 and v_4. Assume that there is only one object o on the edge (v_1, v_2). The given rectangle of $R_o(0^\circ)$ with dashed lines is given in the figure. Since the given rectangle R_o does not cover any vertex of Q no matter what angle R_o is rotated with, we have VertexRs for all vertices equal to $\emptyset$. Since R_o intersects the edge (v_1, v_2) when R_o is rotated with any angle in the angle interval $[-90^\circ, 90^\circ]$, we have $EdgeR((v_1, v_2), o) = [-90^\circ, 90^\circ]$. It is easy to know *EdgeR*s for the other edges are equal to $\emptyset$.

How to compute $VertexR(\cdot, \cdot)$ and $EdgeR(\cdot, \cdot)$ will be described in Section 4.4.

Now, we are ready to describe how to compute $IAR(Q)$ and $CAR(Q)$ with the following lemmas.

Lemma 2. *$IAR(Q) = \cup_{i=1}^{4} EdgeR(e_i, o_j)$ where $o_j \in O$ and $o_j \notin Q$.*

Proof. It is easy to derive that, if Q intersects a given rectangle, then this given rectangle at least intersects an edge of Q. According to the definitions of $IAR(Q)$ and $EdgeR(e, o)$, this lemma holds. □

The above lemma suggests that we can compute $IAR(A)$ by performing a union operation among $EdgeR(e_i, o_j)$ where $i \in [1, 4], o_j \in O$ and $o_j \notin Q$.

Note that if $o_j \in Q$, then the given rectangle of o_j, $R(o_j)$, will be inside Q or intersect the edges of Q no matter what angle $R(o_j)$ is rotated with. We can easily derive that $IAR(Q)$ corresponds to the angle interval $[-90^\circ, 90^\circ]$. That is we need not compute $\cup_{i=1}^{4} EdgeR(e_i, o_j)$.

Lemma 3. *$CAR(Q) = \cap_{i=1}^{4} VertexR(v_i, o_j)$ where $o_j \in O$.*

Proof. It is easy to derive that, Q is contained by a given rectangle iff each vertex of Q is covered by the given rectangle. According to the definitions of $CAR(Q)$ and $VertexR(v, o)$, this lemma holds. □

The above lemma suggests that we can compute $CAR(Q)$ by performing a union operation among $VertexR(v_i, o_j)$ where $i \in [1, 4], o_j \in O$.

4.3. Algorithm phases and theoretical properties

In general, the proposed algorithm works as follows. Initially, the whole data space is set to be the quadrant Q. Next, we compute the upper and lower bounds for the quadrant Q, namely $UPP(Q)$ and $LOW(Q)$. After that, we examine if the algorithm stopping conditions are satisfied. If the stopping conditions are satisfied, the algorithm finds the optimal location and the optimal angle, and returns them as the answers. Otherwise, we need to partition the quadrant Q into four equal-size

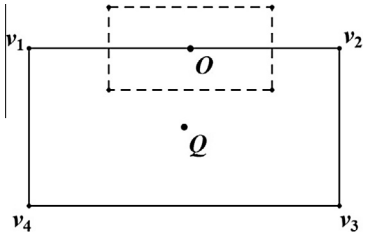


Fig. 5. An example for VertexR and EdgeR.

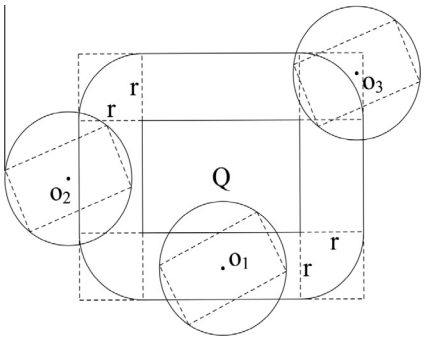


Fig. 6. Computing the *ROS*(*Q*).

small quadrants at the center of *Q*. The promising small quadrants are chosen to be handled further. There are three key phases included in the proposed algorithm.

- The first phase is to compute the upper and lower bounds for the quadrants (Section 4.3.1).
- The second phase is to identify the optimal location region (Section 4.3.2).
- The third phase is to find the optimal location and the optimal angle (Section 4.3.3).

Next, we will introduce the key algorithm phases in detail.

4.3.1. Phase 1: compute the upper and lower bounds

Given a quadrant *Q*, in order to compute the upper and lower bounds for *Q*, namely $UPP(Q)$ and $LOW(Q)$, we need to know the objects whose given rectangles intersect *Q*. We define the *related object set* of *Q*, denoted by $ROS(Q)$, as the set of all the objects whose given rectangles intersect *Q*. Next, we will introduce how to compute $ROS(Q)$ with an example. As shown in Fig. 6, there is a *rounded rectangular region* of *Q* which is centered at the center of *Q* with $r = \sqrt{(l^2 + w^2)}/2$. There is also a rectangular region of *Q* marked in the dashed line and centered at the center of *Q*. Three rotating circles C_1, C_2 and C_3 are also given for o_1, o_2 and o_3, respectively. Based on Fig. 6, we can have Observation 1.

Observation 1. *o* is inside the rounded rectangular region of *Q*, if and only if $o \in ROS(Q)$.

For example, o_1 is in $ROS(Q)$ since it is inside the rounded rectangular region, and o_2 and o_3 are not in $ROS(Q)$ since they are outside the rounded rectangular region.

In general, we can build an R-tree [5] for all objects. When computing $ROS(Q)$, we can issue a range query to determine if an object is inside the rounded rectangular region of *Q*. However, it is not convenient to issue a range query based on the rounded rectangular region. In our algorithm implementation, we can use the *loose* version of $ROS(Q)$, denoted as $LROS(Q)$, by the dashed rectangular region of *Q* instead of the rounded rectangular region of *Q* to issue the range query. For example, in Fig. 6, we can easily know that o_1 and o_3 are inside the dashed rectangular region of *Q* by a range query. Then, o_1 and o_3 are in $LROS(Q)$. Similarly, o_2 is not in $LROS(Q)$ since it is outside the dashed rectangular region of *Q*.

In general, as shown in Fig. 6, the size of the dashed rectangular region of *Q* is not too much larger than that of the rounded rectangular region of *Q*. Thus, there are not too many objects not in the $ROS(Q)$ that are included in the $LROS(Q)$.

Next, based on $LROS(Q)$, we can compute the upper and lower bounds of *Q*, namely $UPP(Q)$ and $LOW(Q)$. Since all the objects that intersect *Q* are included in $LROS(Q)$, we only need to examine the objects in $LROS(Q)$ and all other objects can be ignored.

In detail, the computation steps for $UPP(Q)$ are as follows.

- We can compute $EdgeR(e, o)$ for each edge *e* of *Q* and each object $o \in LROS(Q)$. Then, according to Lemma 2, we can compute $IAR(Q)$.
- By scanning all the angles in $IAR(Q)$, we can compute $UAR(Q)$ and $UPP(Q)$ as shown in Fig. 4.

Similarly, after computing $VertexR(v, o)$, we can obtain $CAR(Q)$ according to Lemma 3. Then, we can compute $LAR(Q)$ and $LOW(Q)$ based on $CAR(Q)$.

Note that $EdgeR(e, o)$ and $VertexR(v, o)$ are equal to $\emptyset$ if $o \in LROS(Q)$ and $o \notin ROS(Q)$. This is because the given rectangle R_o does not intersect *Q* no matter what angle R_o is rotated with. For example, in Fig. 6, $o_3 \in LROS(Q)$ and $o_3 \notin ROS(Q)$. $EdgeR(e, o_3)$ and $VertexR(v, o_3)$ are equal to $\emptyset$.

Note that if the size of a quadrant is larger than the size of a given rectangle (i.e., $l \cdot w$), then this quadrant *Q* cannot be contained by the given rectangle. That means we need not compute the lower bounds for each of these quadrants.

4.3.2. Phase 2: identify the optimal location region

The proposed algorithm will stop with an optimal location region. The optimal location region corresponds to either a rectangle or a non-rectangle which indicate two stopping conditions. Here, a non-rectangle means a single point or a line segment. Then, it is a key phase to identify the optimal location region. In this subsection, we will firstly introduce the properties about the optimal location region. Then, we will introduce how to determine the optimal location region with stopping conditions.

As shown in Section 3, an optimal location region corresponds to the intersection among the given rectangles of some objects each rotated with the same optimal angle. The intersection can be a rectangle or a non-rectangle. The result of a rectangle is straightforward. Next, we elaborate more the result of a non-rectangle with the example in Fig. 7. Assume that the dimension of the given rectangle is set to $l = 8$ and $w = 6$. For simplicity, the given rectangles for the objects are not shown in Fig. 7.

In Fig. 7(a), there are five objects o_1, o_2, o_3, o_4 and o_5. The two numbers in the bracket near to each object is the location information (i.e., the coordinate values). The optimal region corresponds to the single point p and the optimal angle is equal to 0°. That is, $\cap_{i=1}^{5} R_{o_i}(0^\circ) = \{p\}$. In Fig. 7(b), there are four objects o_1, o_2, o_3 and o_4. The optimal region corresponds to the line segment (p_1, p_2) and the optimal angle is also equal to 0°. That is, $\cap_{j=1}^{4} R_{o_j}(0^\circ) = \{p'\}$, where p' is any point on the line segment (p_1, p_2).

Next, we will introduce the properties about the optimal location region which corresponds to a non-rectangle (i.e., a single point or a line segment). We are given n objects o_1, $o_2, \ldots$, and o_n and their given rectangles R_{o_1}, $R_{o_2}, \ldots$, and R_{o_n}, respectively. Assume the optimal angle is equal to θ. Firstly, we introduce the property about a single point.

Lemma 4. *If an optimal location region corresponds to a single point p, then there exists at least one object along each side of the given rectangle* $R_p(\theta)$.

Proof. Assume that this lemma does not hold. That is, for the given rectangle $R_p(\theta)$, there exists a side/edge along which there are no objects. Next, we give an example to show this assumption cannot hold.

We can take Fig. 8 as an example, where there are five objects o_1, o_2, o_3, o_4 and o_5, and $\cap_{i=1}^{5} R_{o_i}(\theta) = \{p\}$. The given rectangle $R_p(\theta)$ can cover all objects. Assume that there are no objects on the edge (A, B) of the given rectangle.

Next, we can find another optimal location p'. The given rectangle $R_{p'}(\theta)$ can also cover all objects. Note that, o_5 is the object nearest to the edge (A, B) and locates on the edge (A', B') of $R_{p'}(\theta)$. That is, there are no objects which are inside the region $ABB'A'$.

That means, $\cap_{i=1}^{n} R_{o_i}(\theta) = \{p, p'\}$. This contradicts that p is the single optimal location. Thus, this lemma holds. □

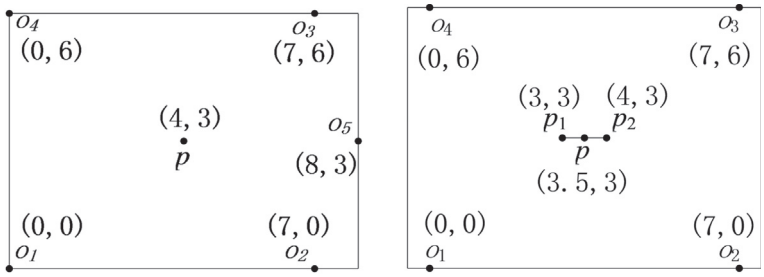


Fig. 7. An example for the optimal location region.

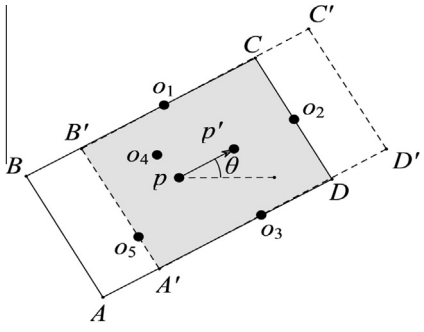


Fig. 8. An example for Lemma 4.

Next, we introduce the property about the optimal location region which corresponds to a line segment. Assume that (p_1, p_2) is a line segment in data space S, where p_1 and p_2 are any two points in S, and $S_{(p_1,p_2)}$ denotes the set of all points on the line segment. Given a set of *general* rectangles $RS = \{R_1, R_2, R_3, \ldots, R_n\}$ which can have any size and rotate with any angle in data space S. In general, we have the following lemmas.

Lemma 5. *If the intersection among the rectangles in RS corresponds to a line segment* (p_1, p_2)*, namely* $\cap_{i=1}^{n} R_i = \{p | p \in S_{(p_1,p_2)}\}$*, then there exist two rectangles whose sides cover the line segment* (p_1, p_2).

Proof.

Step 1. We want to prove that each point on (p_1, p_2) is along the sides of the two rectangles. Otherwise, there exist a point p' that is on (p_1, p_2) and is not along any side of the rectangles. Then, p' must be inside each rectangle since $p' \in \cap_{i=1}^{n} R_i$. For each rectangle R_i $(1 \leqslant i \leqslant n)$, we can find such a $\epsilon_i > 0$ that the circle $N(p', \epsilon_i)$ centered at p' with the radius ϵ_i is inside R_i. That is, $N(p', \epsilon_i) \subseteq R_i$. Let $\varepsilon = min_{1 \leqslant i \leqslant n} \epsilon_i > 0$. Then, we can find a circle $N(p', \varepsilon)$ centered at p' with the radius ε. Then, $N(p', \varepsilon) \subseteq N(p', \epsilon_i) \subseteq R_i$. Thus, $N(p', \varepsilon) \subseteq \cap_{i=1}^{n} R_i$. That means, the intersection of the rectangles to a circle $N(p', \varepsilon)$ instead of the line segment (p_1, p_2). Then, there is a contradiction.

Step 2. We can pick any $4n + 1$ different points from the line segment (p_1, p_2). The total number of the edges for the n rectangles is equal to $4n$. According to the Step 1, each point on (p_1, p_2) is along the side of a certain rectangle. According to the drawer principle, there exist two points p'_1 and p'_2 locating on the same side of this rectangle. Without loss of generality, assume this rectangle is R_1. The whole line segment will also be along this side of R_1 since it is contained by each rectangle.

Step 3. We want to prove each point on (p_1, p_2) also is along the side of another rectangle except R_1. Otherwise, we can assume that there exists a point p'' that is on (p_1, p_2) and is not along any side of the other rectangles. It is easy to know $p'' \in \cap_{i=2}^{n} R_i$. Then, p'' is inside R_j $(2 \leqslant j \leqslant n)$.
We can find such a $\epsilon_j > 0$ that the circle $N(p'', \epsilon_j)$ centered at p'' with the radius ϵ_j is inside R_j. That is, $N(p'', \epsilon_j) \subseteq R_j$. We can let $\varepsilon' = min_{2 \leqslant j \leqslant n} \epsilon_j > 0$. Then, we can find a circle $N(p'', \varepsilon')$ centered at p'' with the radius ε'. Then, $N(p'', \varepsilon') \subseteq N(p'', \epsilon_j) \subseteq R_j$. Thus, $N(p'', \varepsilon') \subseteq \cap_{j=2}^{n} R_j$. Since p'' is on the side of R_1, $\cap_{i=1}^{n} R_i$ cannot correspond to a line segment (p_1, p_2). This is a contradiction.

Step 4. Similar to the Step 2, we can also pick any $4(n-1) + 1$ different points from the line segment (p_1, p_2). According to the drawer principle, there exist two points that are on the same side of a certain rectangle except R_1. Without loss of generality, assume this rectangle is R_2. Similar to the Step 2, the whole line segment (p_1, p_2) will also be along the side of R_2. Thus, this lemma holds. □

Moreover, we can have the following lemma for the rectangles whose sides containing/covering the end-points of the line segment (p_1, p_2).

Lemma 6. *If the intersection among the rectangles in RS corresponds to a line segment, then there exist two rectangles covering the two different end-points of the line segment.*

Proof.

1. Assume that $n = 2$, namely, $\cap_{i=1}^{2} R_i = \{p | p \in S_{(p_1,p_2)}\}$. Then, it is easy to confirm the lemma holds. That is, if the intersection of two rectangles corresponds to a line segment, then the two rectangles locate at the different sides of this line segment.
2. Without loss of generality, we assume this lemma holds as $n = k$. Next, we want to prove that this lemma also holds as $n = k + 1$. According to Lemma 5, among the $k + 1$ given rectangles, $R_1, R_2, R_3, R_4, \ldots$, and R_{k+1}, there exist two rectangles on which edges the line segment locates.
   If the two rectangles locate at the different sides of the line segment, then this lemma holds as $n = k + 1$. Otherwise, assume that the two rectangles are R_1 and R_2, which locate at the same side of the line segment. Let $R_{12} = R_1 \cap R_2$. Since the line segment locates on the edges of R_1 and R_2, we can know that R_{12} should be a rectangle.
   Then, we can have k given rectangles, R_{12}, R_3, $R_4, \ldots$, and R_{k+1}. The intersection of those rectangles corresponds to the line segment (p_1, p_2).
   Since this lemma holds as $n = k$, there exist two rectangles, such as, R_i and R_j, among the k rectangles such that (p_1, p_2) locates on their edges. Moreover, both R_i and R_j locate at the different sides of the line segment. If R_i and R_j are not R_{12}, then this lemma holds as $n = k + 1$. Otherwise, R_i or R_j is R_{12}. Without loss of generality, assume that R_i is R_{12}. That is, $R_i = R_1 \cap R_2$. Then, R_1 and R_j locate at the different sides of the line segment. Thus, this lemma holds. □

Next, we can apply the above both lemmas into our problem. Let p_{best} denote the optimal single point or any point on the optimal line segment. Let the optimal angle be θ_{best}. Given n objects, it is easy to know that the given rectangle $R_{p_{best}}(\theta_{best})$ can cover all the objects whose given rectangles' intersection correspond to the optimal location region.

In general, we can have the following theorem.

Theorem 1. *There exist two objects along the opposite sides of the given rectangle* $R_{p_{best}}(\theta_{best})$.

Proof.

1. Assume that the optimal location region corresponds to a line segment. According to Lemma 6, there exist two objects, such as, o_1 and o_2, and the line segment is along the sides of $R_{o_1}(\theta_{best})$ and $R_{o_2}(\theta_{best})$. Moreover, $R_{o_1}(\theta_{best})$ and $R_{o_2}(\theta_{best})$ locate at the different sides of the line segment. In other words, o_1 and o_2 locate at the different sides of the line segment. For any point p_{best} on the line segment, we can know that o_1 and o_2 are along the opposite sides of the given rectangle $R_{p_{best}}(\theta_{best})$.
2. Assume that the optimal location region corresponds to a single point. Then, p_{best} denotes this single point. According to Lemma 4, there is at leat one object on each side of the given rectangle $R_{p_{best}}(\theta_{best})$. That is, there exist two objects along the opposite sides of $R_{p_{best}}(\theta_{best})$.

Thus this theorem holds. □

Next, we will describe how to determine the two stopping conditions. The optimal region with a rectangle can be determined as follows. The proposed algorithm is based on the iterative partitioning of data space. Specifically, the algorithm begins to partition the whole data space into four equal-sized small quadrants. Then, the small quadrants with the greatest upper bound are examined. A quadrant being examined is typically partitioned into four equal-sized quadrants at its center.

During the splits of the quadrants, we can know that the upper bound of the quadrant will decrease monotonically, and at the same time, the lower bound of the quadrant will increase monotonically. If the optimal location region corresponds to a rectangle, then we will find a quadrant Q whose upper bound and lower bound will reach the same value (i.e., $UPP(Q) = LOW(Q)$), after a limited number of splits. Then, we need not split the quadrant Q further. This is because the quadrant Q is a part of the optimal location region.

On the other hand, the optimal region with a non-rectangle can be determined when we meet the partitioned quadrants all intersecting the same set of given rectangles of the objects and having the same upper bound. In order to examine if the intersecting given rectangles of the objects meet a single point or a line segment, we adopt a threshold called *MaxCount* to control the number of times a quadrant is allowed to be partitioned with the same set of intersecting given rectangles of the objects and the same upper bound. The threshold *MaxCount* can be specified in advance. When the threshold is exceeded, the algorithm will check if the intersecting given rectangles meet at a single point or a line segment.

In general, we can summarize the algorithm stopping conditions as follows.

- Stopping condition 1: During the splits of the quadrants, there exists a quadrant Q such that $UPP(Q) = LOW(Q)$.
- Stopping condition 2: During the splits of the quadrants, there exists a quadrant Q such that the given rectangles intersecting with Q meet at a single point or a line segment.

Lemma 7. *Our proposed algorithm can be stopped with the stopping conditions.*

Proof. For our problem,we can know that there must be a solution. That means we can always find an optimal location region in which the weight sum of the covered objects is the greatest. The basic idea of our proposed algorithm is to iteratively partition the data space into a number of small quadrants and then handle the promising quadrants efficiently to find the optimal location region. In general, an optimal location region corresponds to a rectangle or a non-rectangle (i.e., a single point or a line segment).

Case 1: The final solution is a rectangle.
Since the final region is a rectangle, after a limited number of iteratively partitioning the data space, there must be a quadrant Q such that $UPP(Q) = LOW(Q)$. This means that the quadrant Q is a part of the optimal location region. Since any location in Q is an optimal location, our algorithm can be stopped with an optimal location in this case.
Case 2: The final solution is a non-rectangle.
Since the final solution is a non-rectangle (i.e., a single point or a line segment), after a limited number of iteratively partitioning the data space, there must be a quadrant Q such that $UPP(Q)$ is unchanged and Q intersects the same set of given rectangles of the objects. Besides, since the final solution is a single point or a line segment, the most overlapped region of the given rectangles of the objects related to Q with the same inclination angle corresponds to a single point or a line segment. This means that the intersecting given rectangles of the objects related to Q must meet at a single point or a line segment. Our algorithm can examine all possible given rectangles of the objects related to Q and then find the optimal location and the optimal angle in this case (i.e., the *candidate generation-and-test* approach which will be introduced later). Then, our algorithm can also be stopped with an optimal location in this case. □

4.3.3. Phase 3: find the optimal location and the optimal angle

Once the proposed algorithm stops with an optimal location region. Then, we need to find the optimal location and the optimal angle and return them as the answers.

Firstly, we will introduce how to handle the case that an optimal location region is a rectangle. In this case, there exists such a quadrant Q whose upper bound and lower bound are the same value. Moreover, the quadrant Q is a part of the optimal location region. Then, we can return any point in Q as an optimal location. We can also return any angle in $LAR(Q)$ as an optimal angle.

Next, we will introduce how to handle the case that an optimal location region is a non-rectangle (i.e., a single point or a line segment).

Firstly, we will introduce how to compute θ_{best} and then discuss how to find p_{best}. According to Theorem 1, there are two objects along the opposite sides of $R_{p_{best}}(\theta_{best})$. Consider two cases. The first case is that the two objects, say o_1 and o_2, are along the two long sides of $R_{p_{best}}(\theta_{best})$. The computation for θ_{best} is as follows.

- As shown in Fig. 9, we can construct a circle $C(o_1, w)$ centered at o_1 with the radius w which is the width of the given rectangle.
- The side containing o_2 has a unique intersection (i.e., T_1 or T_2) with the circle $C(o_1, w)$. It is easy to verify that $\angle o_1T_2o_2 = 90^\circ$ or $\angle o_1T_1o_2 = 90^\circ$.
- It is easy to compute the angle $\angle o_1o_2T_2$ or $\angle o_1o_2T_1$. Then, we can determine the directions for the vectors $\overrightarrow{o_2T_2}$ and $\overrightarrow{o_2T_1}$. Then, the θ_{best} can be easily computed (which will be described next in detail).

The second case is that the two objects o_1 and o_2 are along the two short sides. Then, we can construct a circle $C(o_1, l)$ centered at o_1 with the radius l which is the length of the given rectangle. We can also know the unique intersection point between this circle and the edge containing o_2. Then, we can compute θ_{best} similar to the case for the long sides.

However, we do not know which pair of objects in which opposite sides in advance. Next, we propose a *candidate generation-and-test* approach for computing θ_{best} and p_{best}. This approach includes four steps.

- The approach finds the *possible* objects which can be along the sides of $R_{p_{best}}(\theta_{best})$. Assume that the algorithm stops with a quadrant Q which is a part of the optimal location region. Then, we can compute the convex set of $ROS(Q)$, denoted by $CVX(ROS(Q))$. Since $R_{p_{best}}(\theta_{best})$ covers all the objects in $ROS(Q)$, only the objects in $CVX(ROS(Q))$ possibly locate on the sides of $R_{p_{best}}(\theta_{best})$. The other objects in $ROS(Q)$ cannot be along the sides.
- For any pair of possible objects in the convex set, we firstly assume that they are along the long sides. Then, we can compute the candidate angles as shown in Fig. 9. Next, we assume that they are along the short sides. Similarly, we can compute the candidate angles for the case on the short sides. Then, we can compute the candidate angles for this pair of possible objects. After examining all pairs of possible objects, we can compute a set of candidate angles.
- Given any candidate angle and any given rectangle of the object in $ROS(Q)$, we need to test if the intersection of the given rectangles of objects in $ROS(Q)$ rotated with this candidate angle is not equal to $\emptyset$. If so, then this candidate angle is an optimal angle θ_{best}.
- After θ_{best} is obtained, p_{best} corresponds to the intersection of the given rectangles for all objects in $ROS(Q)$ rotated with the angle θ_{best}.

4.4. The computation for VertexR and EdgeR

Firstly, we would like to introduce how to compute $VertexR(v, o)$.

Definition 8 (*Rotating Circle*). The *rotating circle* of $o \in O$ is centered at o with the radius $r = \sqrt{(l^2 + w^2)}/2$, denoted by C_o.

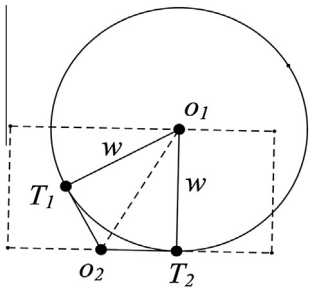


Fig. 9. An example for the optimal angle computation.

Intuitively, the rotating circle of o can be obtained by rotating the given rectangle of o around the center o.

Assume that $R_o(\theta)$ and C_o are given as shown in Fig. 10 in which line 1 and line 2 are parallel to the long side and the short side of R_o, respectively.

It is obvious that v cannot be covered by R_o if v is outside C_o. Suppose that v is inside C_o including the boundary of C_o. Let $k \neq 0$ be the slope of the line 1. Since line 2 is perpendicular to line 1, the slope of line 2 is equal to $-1/k$. As shown in Fig. 10, if v is covered by R_o, then the distance h_1 from v to the line 1 satisfies $h_1 \leqslant w/2$ and the distance h_2 from v to the line 2 satisfies $h_2 \leqslant l/2$. Since the locations (i.e., the coordinates) for v and o are already known, it is easy to compute the slope of the line 1 (i.e., k) which corresponds to the inclination angle range for R_o. Then, we can easily compute $VertexR(v, o)$. It is easy to verify that the computation for $VertexR(v, o)$ takes $O(1)$ time.

Next, we discuss the computation of $EdgeR(e, o)$ where $e = (v_l, v_r)$. It is obvious that R_o will not intersect e if e is outside C_o. Suppose that e or a part of e is inside C_o including the boundary of C_o.

As shown in Fig. 11, there are only *three* cases for the edge e intersecting R_o.

- One of a vertex of e is inside R_o. For example, the vertex v_{r_1} of the edge (v_l, v_{r_1}) is inside R_o.
- Both vertices of e are not inside R_o. There are two cases.
  - The edge e crosses both long/short sides of R_o. For example, the edge (v_l, v_{r_2}) corresponds to this case and crosses the length edges of R_o.
  - The edge e crosses one long side and one short side of R_o. For example, the edge (v_l, v_{r_3}) corresponds to this case.

We can connect the vertex v_l/v_r and the object o by a line segment $(v_l, o)/(v_r, o)$ and extend this line segment from v_l/v_r. Then, the extended line segment will intersect the boundary of C_o. The intersection point is called the *extension point* of v_l/v_r. For example, in Fig. 11, v'_l, v'_{r_2} and v'_{r_3} are the extension points of v_l, v_{r_2} and v_{r_3}, respectively. Note that the extension point of a vertex may be this vertex itself if the vertex locates on the boundary of C_o.

In the following, the arc $\overset{\frown}{v'_l, v'_r}$ on the boundary of C_o means the *minor arc* that is the smaller of the two arcs formed when the boundary of C_o is divided into two unequal parts by the extension points v'_l and v'_r.

Definition 9 (*Arc Range*). For each arc a along the boundary of C_o, the *arc range* of a is defined as $ArcR(a, o) = \{\theta | a \text{ intersects } R_o(\theta)\}$.

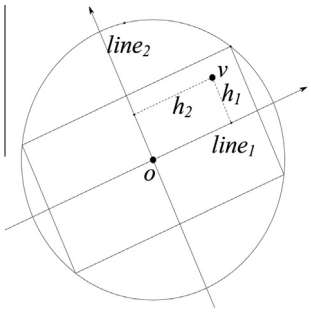


Fig. 10. Computing the VertexR.

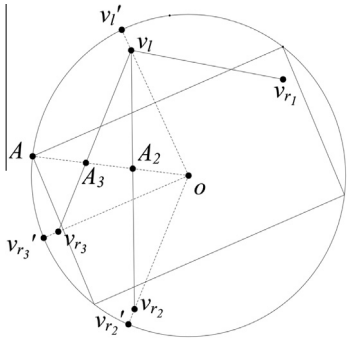


Fig. 11. Computing the EdgeR.

Lemma 8. *$EdgeR(e,o) = VertexR(v_l,o) \cup VertexR(v_r,o) \cup ArcR(\widehat{v'_l v'_r},o)$, where v'_l/v'_r is the extension point of v_l/v_r and $\widehat{v'_l v'_r}$ is the arc on the boundary of C_o.*

Proof. In general, there are three cases for the edge e. As shown in Fig. 11, the edge e can correspond to the edges (v_l, v_{r_1}), (v_l, v_{r_2}) and (v_l, v_{r_3}), respectively. For any angle $\theta \in EdgeR(e,o)$, we can have $\theta \in VextexR(v_l,o), \theta \in VextexR(v_r,o)$ or $\theta \in ArcR(\widehat{v'_l v'_r},o)$. Thus, $EdgeR(e,o) \subseteq VertexR(v_l,o) \cup VertexR(v_r,o) \cup ArcR(\widehat{v'_l v'_r},o)$.

On the other hand, for any angle $\theta' \in VextexR(v_l,o) \cup VextexR(v_l,o)$, it is easy to know $\theta' \in EdgeR(e,o)$ where $e = (v_l, v_r)$. For any angle $\theta'' \in ArcR(\widehat{v'_l v'_r},o)$, as shown in Fig. 11, we can know A is the intersection point between C_o and R_o, which is on the boundary of C_o. Then, we can connect A and o by a line segment (A,o) which intersects $(v_l, v_{r_2})/(v_l, v_{r_3})$ at the intersection point A_2/A_3. Then, $\theta'' \in EdgeR(e,o)$. Then, $VertexR(v_l,o) \cup VertexR(v_r,o) \cup ArcR(\widehat{v'_l v'_r},o) \subseteq EdgeR(e,o)$. Thus, this lemma holds. □

It is easy to verify that the computation for $EdgeR(e,o)$ takes $O(1)$ time.

Next, we would like to introduce how to compute the $ArcR(a,o)$ by an example as shown in Fig. 12. Assume that the given arc a correspond to the minor arc $\widehat{AB}$ marked in bold arc on the boundary of C_o. Among the three sub-figures, the arc $\widehat{AB}$ is the shortest and the longest in Fig. 12(a) and (b), respectively. It is easy to know that R_o will intersect the arc $\widehat{AB}$ if one of the vertices of R_o is on the arc $\widehat{AB}$. Otherwise, R_o will not intersect the arc $\widehat{AB}$. As shown in Fig. 12(a), R_o intersects the arc $\widehat{AB}$ on the intersection points A and B when R_o is rotated with the angles θ_1 and θ_2, respectively. R_o does not intersect $\widehat{AB}$ as it is rotated with the angle θ_3. Note that the vectors $\overrightarrow{op_1}$, $\overrightarrow{op_2}$ and $\overrightarrow{op_3}$ are parallel to the long side of R_o rotated with θ_1, θ_2 and θ_3, respectively.

The $ArcR(\widehat{AB},o)$ can be computed as follows. We can rotate R_o around the center o in an anti-clockwise direction from the angle θ_1 to the angle θ_2. Then, R_o will firstly intersect the arc $\widehat{AB}$ on the intersection point A. We continue to rotate R_o until the intersection point B is passed. Then, R_o will not intersect this arc. In particular, when R_o locates at the angle θ_3 in Fig. 12(a), R_o does not intersect the arc $\widehat{AB}$.

In general, based on Fig. 12, we can derive the following rules.

- For the case $\angle AoB < 2\arctan(w/l), ArcR(a,o) = [\theta_1, \theta_1 + \angle AoB] \cup [\theta_2 - \angle AoB, \theta_2]$. This case corresponds to Fig. 12(a).
- For the case $\angle AoB + 2\arctan(w/l) \geqslant \pi, ArcR(a,o) = [-90^\circ, 90^\circ]$. This case corresponds to Fig. 12(b).
- Otherwise, $ArcR(a,o) = [\theta_1, \theta_2]$. This case corresponds to Fig. 12(c).

It is easy to know that θ_3 is not included in $ArcR(a,o)$ in Fig. 12(a).

The computation for $ArcR(a,o)$ takes $O(1)$ time.

Before ending this subsection, we want to *standardize* the angle values for each kind of angle range, namely, *VertexR*, *EdgeR* and *ArcR*. In detail, we want to make sure that the angle value is smaller than or equal to 90° and is larger than or equal to -90°. For example, the angle interval $[60^\circ, 100^\circ]$ can be standardized as two equivalent intervals $[60^\circ, 90^\circ]$ and $[-90^\circ, -80^\circ]$.

4.5. Algorithm description and analysis

A detailed algorithm description is given in Algorithm 1. The following four steps are included.

The first step is for initialization. We begin to take the whole data space as a quadrant Q. Initially, the upper bound of Q, $UPP(Q)$, is set to the weighted sum of all objects, namely $\sum_{i=1}^{|O|} w(o_i)$. The heap H is used to store the partitioned quadrants and is ordered by the upper bounds of the quadrants to prioritize the quadrants. Initially, H is set to Q that is the whole data

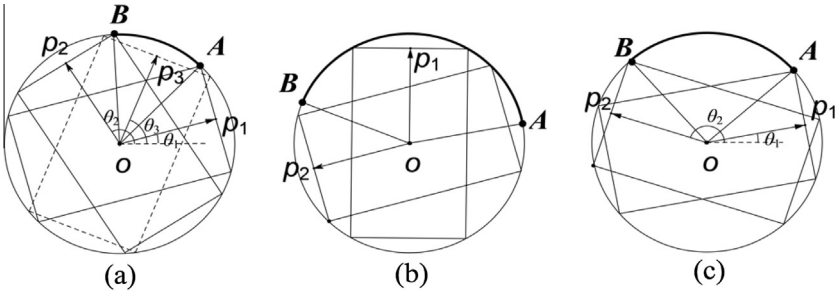


Fig. 12. Computing the ArcR.

space. The variable *MaxLow* is used to keep track of the greatest lower bound seen so far. Initially, *MaxLow* is set to zero. It is used to prune the quadrants whose upper bounds are smaller than the lower bound of a quadrant. The variable *MaxCount* is used to determine the optimal location region of a non-rectangle, which value is specified in advance. The variable *count* is a temporary counter variable and is set to zero initially. The variable *LastUpp* is used to keep track of the greatest upper bound seen last time. Initially, *LastUpp* is set to the weighted sum of all objects. Next, the R-tree is built for all objects.

The second step is to compute $LOW(Q)$ (i.e., Line 10) and $UPP(Q)$ (i.e., Line 31) for the quadrant Q which is the top entry of the heap. Initially, the quadrant Q is the whole data space.

The third step is to examine if the stopping conditions are satisfied. In general, the algorithm can stop with an optimal location region which corresponds to a rectangle or a non-rectangle. If the former is satisfied, the algorithm can return any point in Q and any angle in $LAR(Q)$. Otherwise, the latter (i.e., a non-rectangle) is satisfied. The latter can be identified by examining if the counter reaches the largest number (i.e., *MaxCount*). If so, the algorithm can compute the optimal location and the optimal angle by the *candidate generation-and-test* approach.

The fourth step is executed if the stopping conditions are not satisfied. The algorithm can partition the quadrant Q into four equal-sized small quadrants at the center of Q. The *promising* quadrants, whose upper bounds are at least the greatest lower bound seen so far, are inserted into the heap.

Algorithm 1. Rotating MaxRS query algorithm

```
 1: Q ← the whole data space;
 2: UPP(Q) ← ∑_{i=1}^{|O|} w(o_i);
 3: insert Q into H;
 4: MaxLow ← 0;//the current largest lower bound
 5: LastUpp ← UPP(Q);//the last upper bound
 6: count ← 0;//the control counter
 7: build the R-tree index for all objects;
 8: while H is not empty do
 9:    Q ← remove top entry from H;
10:    compute LOW(Q) and LAR(Q);
11:    if LOW(Q) == UPP(Q) then
12:       return any point in Q and any angle in LAR(Q);
13:    end if
14:    if UPP(Q) == LastUpp then
15:       count ← count + 1;
16:       if count ⩾ MaxCount then
17:          if a single point or a line segment is found by the candidate generation-and-test approach then
18:             return the optimal location and the optimal angle;
19:          else
20:             count ← 0;
21:          end if
22:       end if
23:    else
24:       LastUpp ← UPP(Q);
25:       count ← 0;
26:    end if
27:    if LOW(Q) > MaxLow then
28:       MaxLow ← LOW(Q);
29:    end if
30:    partition Q into four equal-sized small quadrants at the center of Q;
31:    compute the upper bounds for each small quadrant;
32:    insert the promising quadrants whose upper bounds are at least MaxLow into H;
33: end while
```

Next, we will prove the correctness of the proposed algorithm.

Theorem 2. Algorithm 1 *is correct.*

Proof. The correctness of Algorithm 1 is embodied in two aspects. That is, the algorithm can find the optimal location and the optimal angle correctly.

There are two cases. One case is that the algorithm stops with an optimal rectangle. As shown in the algorithm description, the data space is iteratively partitioned into different quadrants. During the splits of the quadrants, the upper bounds of the quadrants are decreased and the lower bounds of the quadrants are increased. Once we can find the quadrant Q whose upper bound and lower bound are equal. That means, we have found an optimal location region overlapping the quadrant Q. In other words, any point in the region of Q is the optimal location. As shown in Section 4.2, the optimal location region corresponds to the intersecting given rectangles which share the most common angles in $LAR(Q)$. That means, any angle in LAR is the optimal angle.

The other case is that the algorithm stops with single point or a line segment. As shown in Section 4.3.2, we can also identify the quadrant Q which is overlapped with the optimal location region. As shown in Section 4.3.3, the approach of candidate generation-and-test can test all combinations for the possible given rectangles for the objects in $ROS(Q)$ and the possible angles for the given rectangles. Thus, the approach of candidate generation-and-test can find an optimal location and an optimal angle. Thus, in this case, we can also find the optimal location and the optimal angle correctly. □

Algorithm Time and Space Analysis. The algorithm execution time mainly comes from Lines 7, 9, 10, 17, 31, and 32. The time for Line 7 is $O(n \log n)$ [5], where n is the total number of the objects. In our algorithm, we adopt the binary heap structure to store the quadrants. Thus, Line 9 needs $O(\log n)$ time. Line 10 includes three parts. The first part is to issue a range query based on the R-tree to determine the objects in the related object set of the quadrant Q, namely $ROS(Q)$. This part needs $O(m + n^{1/2})$ time in the worst case [1], where m denotes the greatest number of the objects returned among all range queries for the quadrants. The second part is to compute the containment angle range (CAR) for each object in $ROS(Q)$. This part needs $O(m)$ time. The third part is to compute the lower angle range (LAR) for the quadrant by scanning the angles in all CARs. This part needs $O(m \log m)$ time. Thus, Line 10 needs $O(m \log m + \log n)$ time in total.

Line 17 includes two parts. The first part is to compute the convex set for $ROS(Q)$, namely, $CVX(ROS(Q))$. The second part is to test the given rectangle for the objects in $CVX(ROS(Q))$. Assume that the size of $ROS(Q)$ is equal to γ. In the worst case, each object in $ROS(Q)$ is included in $CVX(ROS(Q))$ and each candidate angle is tested. Then, Line 17 needs $O(\gamma^3)$ time. In practice, $\gamma \ll n$. For example, in the experiments, with the default setting, on the NE real dataset, γ is at most 493 and n is 80,000.

Line 31 is similar to Line 10 and also needs $O(m \log m + n^{1/2})$ time. Line 32 is a heap insertion operation whose cost is small.

Assume that α denotes the number for the iterative partitioning in the proposed algorithm. Thus, the whole while-statement part of the algorithm needs $O(\alpha(m \log m + n^{1/2}))$ time. In the worst case, we can have $m = n$, and this part needs $O(\alpha(m \log m + n^{1/2}))$ time. Thus, the proposed algorithm needs $O(\alpha(n \log n + n^{1/2}))$ time in total.

The space cost for the proposed algorithm comes from the storage for all objects, the storage for the R-tree and the storage for the heap. The storage for all objects needs $O(n)$ space. The storage for the binary heap needs $O(n)$ space. Thus, the proposed algorithm needs $O(n)$ space in total.

5. The experimental results

In this section, we report the experimental results for our proposed algorithm. We can compare our proposed algorithm with the baseline algorithm (described in Section 4.1) in terms of algorithm performance including the running time and the storage cost. All algorithms are implemented in C++. All the experiments were performed on a Linux machine with an Intel 3 GHz CPU and 4 GB memory.

Similar to [3], two synthetic datasets and one real dataset are used in the experiments. The object distributions in the two synthetic datasets are *uniform* (UN) and *Gaussian* (GA). The cardinalities of synthetic datasets (i.e., $|O|$) is set from 500,000 to 2,500,000 (by default, 1,500,000). The real North East (NE) dataset is downloaded from the R-tree Portal.[1] The cardinalities of real datasets is set to be from 40,000 to 120,000 (by default, 80,000). For all datasets, we normalize the range of coordinates to $[0, 1]$. The weight of each object is set to 1 under default settings. The variable $MaxCount$ is initially set to 1000.

For each kind of dataset, the effects of the cardinalities of datasets and the size of a given rectangle on the algorithm performance were reported. The size of a given rectangle is set to be $0.0005 * 0.0005$, $0.001 * 0.001$, $0.002 * 0.002$, $0.004 * 0.004$, $0.006 * 0.006$ and $0.008 * 0.008$, respectively (by default, $0.002 * 0.002$).

The first set of experiments is to study the algorithm performance with the effect of cardinalities of the synthetic datasets. As shown in Fig. 13(a) and (b), the running time and the storage of the proposed algorithm increase with the number of the objects. With the increased number of objects, the time for building the R-tree and the time for examining the given rectangles increase. In fact, the main storage for the proposed algorithm comes from the cost for the object storage and the R-tree. With the increased number of the objects, the proposed algorithm also needs more space to store the objects and the R-tree.

The second set of experiments is to study the algorithm performance with the effect of the sizes of the given rectangles based on the synthetic datasets with the default cardinality. As shown in Fig. 14(a), the running time of the proposed algorithm increases with the increased sizes of the given rectangles. This is because the examining time become larger since the size of the given rectangle is larger.

[1] http://www.rtreeportal.org.

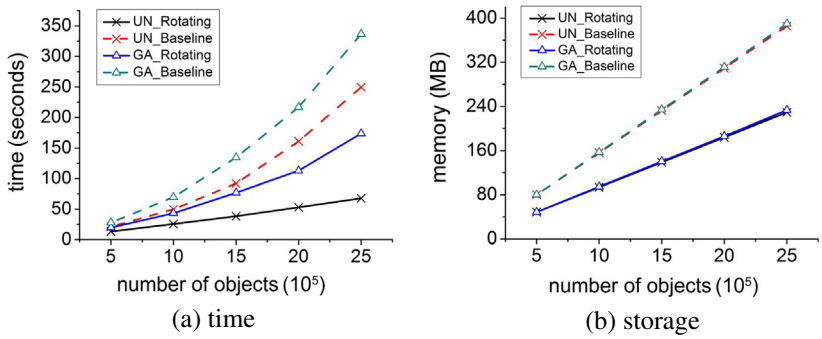


Fig. 13. Results with the cardinalities of synthetic datasets.

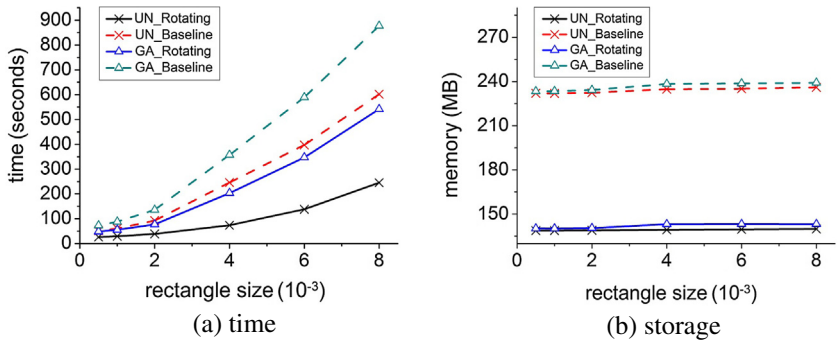


Fig. 14. Results with the sizes of given rectangles on the synthetic datasets.

As shown in Fig. 14(b), the storage of the proposed algorithm is almost the same with the increased sizes of the given rectangles for the uniform distribution. The storage of the proposed algorithm does not have large changes with different sizes of the given rectangles. Since the storage for the proposed algorithm mainly depends on the object storage and the R-tree storage, the space cost does not have large changes with the default number of the objects.

The third set of experiments is to study the algorithm performance with the effect of the cardinalities of the real datasets. In this set of experiments, the object weight distributions are uniform (UN) and Gaussian (GA), respectively. As shown in Fig. 15(a) and (b), the running time and the storage of the proposed algorithm increase with the number of the objects.

The fourth set of experiments is to study the algorithm performance with the effect of sizes of the given rectangles based on the real datasets with the default cardinality. In this set of experiments, the object weight distributions are also uniform and Gaussian, respectively. As shown in Fig. 16(a), the running time decreases with the increased sizes of the given rectangles. As shown in Fig. 16(b), the storage of the proposed algorithm does not have large changes with the decreased sizes of the given rectangles.

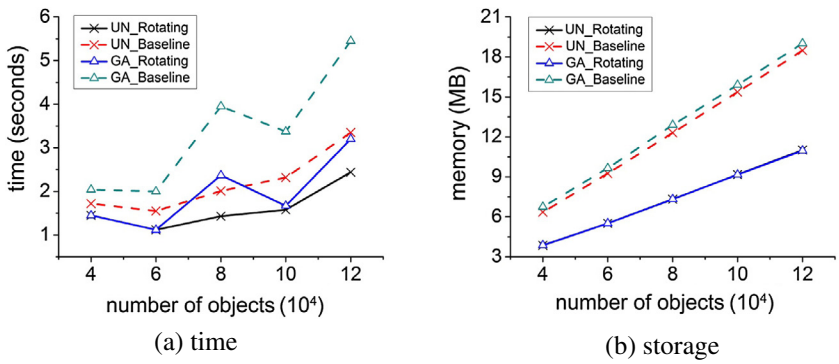


Fig. 15. Results with the cardinalities of real dataset.

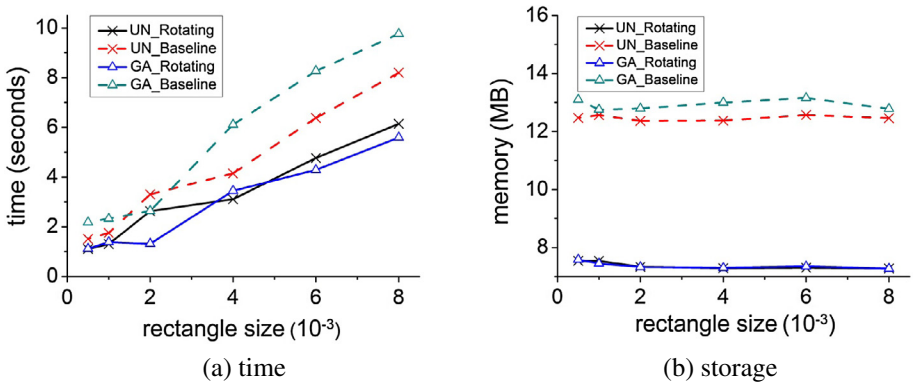


Fig. 16. Results with the sizes of given rectangles on the real datasets.

As shown in the above experiments, our proposed algorithm outperforms the baseline algorithm in terms of running time and storage. This is because the baseline algorithm has to take additional costs to obtain the optimal angle by executing our proposed algorithm.

Next, we describe the fifth set of experiments. We know that the baseline algorithm described in Section 4.1 requires a priori knowledge of the optimal angle. In this set of experiments, we consider another baseline algorithm which does not have this priori knowledge. This baseline algorithm is also based on the existing non-rotating MaxRS query algorithm. Specifically, assume that we are given a fixed (rotation) angle. The baseline algorithm with a fixed angle works as follows.

Step 1: We rotate the x-axis and the y-axis in the given dataset with this fixed angle and obtain the rotated dataset.
Step 2: We execute the existing algorithm on the rotated dataset and obtain the corresponding optimal location.
Step 3: We can know the weighted sum of all the objects covered by the rectangular region centered at the corresponding optimal location.

On the given dataset, the weighted sum of all the covered objects computed by the baseline algorithm is denoted as WS_{non}. Similarly, we can execute our proposed rotating MaxRS query algorithm on the given dataset and compute the weighted sum of all the covered objects, which is denoted as WS_r. In the experiments, we compare the weighted sum of all the objects covered of the rotating MaxRS queries and the existing non-rotating MaxRS queries. In detail, we will study the increase rate (IR) of weighted sum, which is defined as $IR = (WS_r - WS_{non})/WS_{non}$. Note that the default value of the fixed (rotation) angle is set to 0° in the experiments. This means that the rotated dataset is the same as the given dataset by default.

On the real datasets, the results on IR with different sizes of datasets are reported in Fig. 17(a) where $|O|$ is varied from 40k to 120k with an interval 20k, and the fixed angle and the size of given rectangle are set under default settings. The results on IR with different fixed angles are reported in Fig. 17(b) where the fixed angles are varied from 5° to 80°, the number of objects and the size of a given rectangle are set under default settings. The results on IR with different sizes of rectangles are reported in Fig. 17(c) where the number of objects and the fixed angle are set under default settings. In the figures, the IR value fluctuates. This trend can be explained as follows.

Note that the optimal angle in a dataset is a particular value. We know that if the fixed angle is exactly equal to the optimal angle found, then the solution returned by the existing algorithm is very good and thus the IR value is 0. However, in most of the cases, the fixed angle is not equal to the optimal angle found. Sometimes, the solution returned by the existing

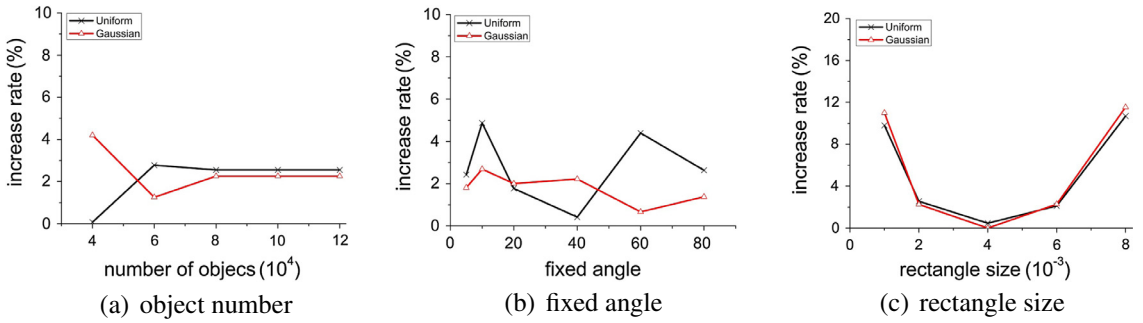


Fig. 17. Results on the increase rate with the real datasets.

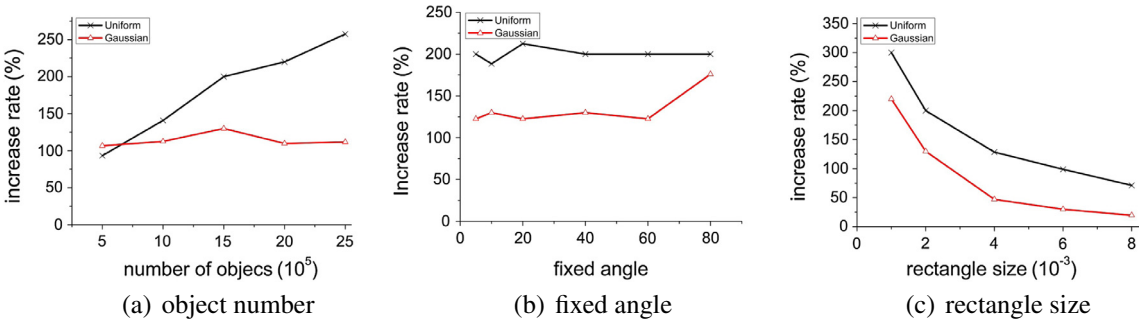


Fig. 18. Results on the increase rate with the synthetic datasets.

algorithm is not good. Besides, in some other cases, the solution returned by the existing algorithm is good. Thus, the *IR* value fluctuates when the number of objects, the fixed angle and the size of given rectangle change. In addition, in these figures, in most of the cases, when the fixed angle is close to the optimal angle found, the *IR* value is small. Note that we may have multiple optimal angles in the dataset.

On the synthetic datasets, the object weight is set to 1 under default settings, and the object distributions are uniform and Gaussian, respectively. Similarly, the results on *IR* with different sizes of datasets, different fixed angles, and different sizes of rectangles are reported in Fig. 18(a), (b), and (c), respectively. In these figures, we often have a larger *IR* for the uniform distribution of objects compared with the Gaussian distribution of objects. This is because the optimal angle found is often farther from the fixed angle used for rotation when the object distribution is uniform on the synthetic dataset. As shown in the results, the value of *IR* can be increased with up to 300% on the synthetic datasets.

Based on the results in Figs. 17 and 18, we can conclude that the value of *IR* mainly depends on the optimal angle found in the dataset. In general, if the fixed (rotation) angle is closer to the optimal angle, WS_{non} is closer to WS_r, then we have a smaller *IR*. Otherwise, we have a larger *IR*.

Summary

The extensive experimental results verify the efficiency of our proposed algorithms in terms of the running time and the storage. A rotating MaxRS query can be answered by our proposed algorithm within 300 s in most cases on the synthetic datasets and even within 6 s on the real datasets. The storage of our proposed algorithm requires less than 240 MB and 12 MB in most cases for the synthetic datasets and the real datasets, respectively. The optimal angles found in the experiments are not equal to 0° in most cases.

6. Conclusions

In this paper, we proposed a new problem called rotating MaxRS. The purpose of our problem is to find an optimal location and an optimal angle such that the weighted sum of all the objects covered by the rectangular region of a given size centered at the optimal location and rotated with the optimal angle is the greatest. We also proposed an efficient algorithm for our problem. The algorithm iteratively partitions the whole data space into small quadrants and only examines the quadrants possibly containing the optimal location. The upper and lower bounds of a quadrant were proposed to guide the partitioning process. Extensive experiments were conducted to verify the efficiency of our algorithm based on the real and synthetic datasets. In our future work, we will continue to study the rotating MaxRS problem including (a) online query technique for the problem, (b) extending problem such as rotating Max*k*RS or rotating MinRS problem, (c) any I/O-optimal algorithm for the problem, and (d) the problem in other metric systems such as L_1 and L_{max}.

Acknowledgements

We are very thankful to the anonymous reviewers for the very useful comments. The research of Yubao Liu, Zitong Chen, Jiamin Xiong, Xiuyuan Cheng and Peihuan Chen is supported by Grant NSFC 61070005. The research of Raymond Chi-Wing Wong is supported by Grant FSGRF14EG34.

References

[1] L. Arge, M. de Berg, H.J. Haverkort, K. Yi, The priority r-tree: a practically efficient and worst-case optimal r-tree, in: SIGMOD 2004, 2004, pp. 347–358.
[2] S. Cabello, J.M. Diaz-Banez, S. Langerman, C. Seara, I. Ventura, Reverse facility location problems, in: CCCG 2005, 2005, pp. 68–71.
[3] D.-W. Choi, C.-W. Chung, Y. Tao, A scalable algorithm for maximizing range sum in spatial databases, PVLDB 5 (11) (2012) 1088–1099.
[4] Y. Du, D. Zhang, T. Xia, The optimal-location query, in: SSTD 2005, 2005, pp. 163–180.
[5] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: SIGMOD 1984, 1984, pp. 47–57.

[6] Y.-K. Huang, L.-F. Lin, Efficient processing of continuous min-max distance bounded query with updates in road networks, Inform. Sci. 278 (2014) 187–205.
[7] H. Imai, T. Asano, Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane, J. Algorithms 4 (3) (1983) 310–323.
[8] J. Krarup, P.M. Pruzan, The simple plant location problem: survey and synthesis, Eur. J. Oper. Res. 12 (1) (1983) 36–57.
[9] H. Lin, F. Chen, Y. Gao, D. Lu, Optregion: finding optimal region for bichromatic reverse nearest neighbors, in: DASFAA 2013, 2013, pp. 146–160.
[10] Y. Liu, R.C.-W. Wong, K. Wang, Z. Li, C. Chen, Z. Chen, A new approach for maximizing bichromatic reverse nearest neighbor search, Knowl. Inf. Syst. 36 (1) (2013) 23–58.
[11] S.C. Nandy, B.B. Bhattacharya, A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids, Comput. Math. Appl. 29 (8) (1995) 45–61.
[12] J. Qi, R. Zhang, L. Kulik, D. Lin, Y. Xue, The min-dist location selection query, in: ICDE 2012, 2012, pp. 366–377.
[13] J. Qi, R. Zhang, Y. Wang, A.Y. Xue, G. Yu, L. Kulik, The min-dist location selection and facility replacement queries, World Wide Web 17 (6) (2014) 1261–1293.
[14] J.B. Rocha-Junior, A. Vlachou, C. Doulkeridis, K. Nørvåg, Efficient processing of top-k spatial preference queries, PVLDB 3 (1) (2010) 93–104.
[15] B.C. Tansel, R.L. Francis, T.J. Lowe, Location on networks: a survey, Manage. Sci. 29 (4) (1983) 498–511.
[16] Y. Tao, X. Hu, D.-W. Choi, C.-W. Chung, Approximate maxrs in spatial databases, PVLDB 6 (13) (2013) 1546–1557.
[17] R.C.-W. Wong, M.T. Ozsu, A.W.-C. Fu, P.S. Yu, L. Liu, Efficient method for maximizing bichromatic reverse nearest neighbor, PVLDB 2 (1) (2009) 1126–1137.
[18] R.C.-W. Wong, M.T. Ozsu, A.W.-C. Fu, P.S. Yu, L. Liu, Y. Liu, Maximizing bichromatic reverse nearest neighbor for lp-norm in two- and three-dimensional spaces, VLDB J. 20 (6) (2011) 893–919.
[19] T. Xia, D. Zhang, E. Kanoulas, Y. Du, On computing top-t most influential spatial sites, in: VLDB 2005, 2005, pp. 946–957.
[20] X. Xiao, B. Yao, F. Li, Optimal location queries in road network databases, in: ICDE 2011, 2011, pp. 804–815.
[21] D. Yan, R.C.-W. Wong, W. Ng, Efficient methods for finding influential locations with adaptive grids, in: CIKM 2011, 2011, pp. 1475–1484.
[22] M.L. Yiu, X. Dai, N. Mamoulis, M. Vaitis, Top-k spatial preference queries, in: ICDE 2007, 2007, pp. 1076–1085.
[23] D. Zhang, Y. Du, T. Xia, Y. Tao, Progressive computation of the min-dist optimal-location query, in: VLDB 2006, 2006, pp. 643–654.
[24] Z. Zhou, W. Wu, X. Li, M.L. Lee, W. Hsu, Maxfirst for maxbrknn, in: ICDE 2011, 2011, pp. 828–839.