The VLDB Journal manuscript No.
(will be inserted by the editor)

Efficiently Processing Snapshot and Continuous Reverse k Nearest Neighbors Queries

Muhammad Aamir Cheema · Wenjie Zhang · Xuemin Lin* · Ying Zhang

Received: date / Accepted: date

Abstract Given a set of objects and a query q, a point p is called the reverse k nearest neighbor (RkNN) of q if q is one of the k closest objects of p. In this paper, we introduce the concept of influence zone which is the area such that every point inside this area is the RkNN of q and every point outside this area is not the RkNN. The influence zone has several applications in location based services, marketing and decision support systems. It can also be used to efficiently process RkNN queries. First, we present efficient algorithm to compute the influence zone. Then, based on the influence zone, we present efficient algorithms to process RkNN queries that significantly outperform existing best known techniques for both the *snapshot* and *continuous* RkNN queries. We also present a detailed theoretical analysis to analyse the area of the influence zone and IO costs of our RkNN processing algorithms. Our experiments demonstrate the accuracy of our theoretical analysis. This paper is an extended version of our previous work [9]. We make the following new contributions in this extended version: 1) we conduct a rigorous complexity analysis and show that the complexity of one of our proposed algorithms in [9] can be reduced from $O(m^2)$ to $O(km)$ where $m > k$ is the number of objects used to compute the influence zone ; 2) we show that our techniques can be applied to dimensionality higher than two; and 3) we present efficient techniques to handle data updates.

Muhammad Aamir Cheema, Wenjie Zhang, Ying Zhang
School of Computer Science and Engineering,
The University of New South Wales, Australia
E-mail: {macheema,zhangw,yingz}@cse.unsw.edu.au

Xuemin Lin
Software College, East China Normal University, China
and
School of Computer Science and Engineering,
The University of New South Wales, Australia
E-mail: lxue@cse.unsw.edu.au
* corresponding author

1 Introduction

The reverse k nearest neighbors (RkNN) query [24, 45, 28, 33, 36, 41, 34, 23] has received significant research attention ever since it was introduced in [24]. A RkNN query finds every data point for which the query point q is one of its k nearest neighbors. Since q is close to such data points, q is said to have high influence on these points. Hence, the set of points that are the RkNNs of a query is called its influence set [24]. Consider the example of a gas station. The drivers for which this gas station is one of the k nearest gas stations are its potential customers. In this paper, the objects that provide a facility or service (e.g., gas stations) are called *facilities* and the objects (e.g., the drivers) that use the facility are called *users*. The influence set of a given facility q is then the set consisting of every user for which q is one of its k closest facilities.

In this paper, we first introduce a more generic concept called *influence zone* and then we show that the influence zone can be used to efficiently compute the influence set (i.e., RkNNs). Consider a set of facilities $F = \{f_1, f_2, \cdots, f_n\}$ where f_i represents a point in Euclidean space and denotes the location of the i^{th} facility. Given a query $q \in F$, the influence zone Z_k is the area such that for every point $p \in Z_k$, q is one of its k closest facilities and for every point $p' \notin Z_k$, q is not one of its k closest facilities.

The influence zone has various applications in location based services, marketing and decision support systems. Consider the example of a coffee shop. Its influence zone may be used for market analysis as well

as targeted marketing. For instance, the demographics of its influence zone may be used by the market researchers to analyse its business. The influence zone can also be used for marketing, e.g., advertising bill boards or posters may be placed in its influence zone because the people in this area are more likely to be influenced by the marketing. Similarly, the people in its influence zone may be sent SMS advertisements.

Note that the concept of the influence zone is more generic than the influence set, i.e., the RkNNs of q can be computed by finding the set of users that are located in its influence zone. In this paper, we show that our influence zone based RkNN algorithms significantly outperform existing best known algorithms for both the *snapshot* and *continuous* RkNN queries (formally defined in Section 2).

Existing RkNN processing techniques [33,36,41,10, 23] require a *verification* phase to answer the queries. Initially, the space is pruned by using the locations of the facility points. Then, the users that are located in the unpruned space are retrieved. These users are the possible RkNNs and are called candidates. Finally, in the verification phase, a range query is issued for every candidate to check if it is a RkNN or not.

In contrast to the existing approaches, our influence zone based algorithm does not require the verification phase. Initially, we use our algorithm to efficiently compute the influence zone. Then, every user that is located in the influence zone is reported as RkNN. This is because a user can be the RkNN if and only if it is located in the influence zone. Similarly, to continuously monitor RkNNs, initially the influence zone is computed. Then, to update the results, we only need to monitor the users that enter or leave the influence zone (i.e., the users that enter in the influence zone become the RkNNs and the users that leave the influence zone are no more the RkNNs). To further improve the performance, we present efficient methods to check whether a point lies in the influence zone or not.

It is important to note that the influence zone of a query is the same as the Voronoi cell of the query when $k = 1$ [34]. For arbitrary value of k, there does not exist an equivalent representation in literature (i.e., order k Voronoi cell is different from the influence zone). Nevertheless, we show that a precomputed order k Voronoi diagram can be used to compute the influence zone (see Section 5.1). However, using the precomputed Voronoi diagrams is not a good approach to process spatial queries as mentioned in [49]. For instance, the value of k is not known in advance and precomputing several Voronoi diagrams for different values of k is expensive and incurs high space requirement. In Section 5.1, we state several other limitations of this approach.

Below, we summarize our contributions.

- We present an efficient algorithm to compute the influence zone. Based on the influence zone computation algorithm, we present efficient algorithms that outperform best known techniques for both *snapshot* and *continuous* RkNN queries.
- Our main algorithm uses an algorithm similar to the one proposed in [41]. It was shown that the complexity of that algorithm is $O(m^2)$ [41] where m is the number of facilities used to prune the search space. In this extended version, we conduct a rigorous complexity analysis and show that the complexity of the algorithm can be reduced to $O(km)$ when k is smaller than m.
- We demonstrate that the influence zone computation technique can be extended for dimensionality higher than two. We also present techniques to efficiently update the influence zone when the underlying data sets issue updates.
- We provide a detailed theoretical analysis to analyse the IO costs of our influence zone and RkNN computation algorithms, the area of the influence zone and the number of RkNNs. The analysis is applicable to arbitrary dimensionality and the experiment results demonstrate the accuracy of our theoretical analysis.
- Our extensive experiments on real and synthetic data demonstrate that our proposed algorithms are several times faster than the existing best known algorithms for two dimensional snapshot and continuous RkNN queries.

This paper is an extended version of our previous work [9]. In this extended version, we conduct a rigorous complexity analysis and show that the complexity of one important algorithm (Algorithm 2) can be reduced from $O(m^2)$ to $O(km)$ when k is smaller than m (see Section 5.4). In this version, we also demonstrate that our influence zone computation technique can be extended for the dimensionality higher than two (see Section 3.4). We also present a theoretical analysis that is applicable to arbitrary dimensionality and its accuracy is verified by experimental results. In Section 6, we extend our techniques to efficiently update the influence zone when the underlying data set is updated by insertions or deletions.

The rest of the paper is organized as follows. In Section 2, we define the problem and describe the related work. Section 3 presents our technique to efficiently compute the influence zone. In Section 4, we present efficient techniques to answer RkNN queries by using the influence zone. A detailed theoretical analysis is presented in Section 5. The techniques to handle data

updates are presented in Section 6 followed by the experiment results in Section 7. Section 8 concludes the paper.

2 Preliminaries

2.1 Problem Definition

First, we define a few terms and notations. Consider a set of facilities $F = \{f_1, f_2, \cdots f_n\}$ and a query $q \in F$ in a Euclidean space[1]. Given a point p, C_p denotes a circle centered at p with radius equal to $dist(p, q)$ where $dist(p, q)$ is the distance between p and q. $|C_p|$ denotes the number of facilities that lie within the circle C_p (i.e., the count of facilities such that for each facility f, $dist(p, f) < dist(p, q)$). Please note that the query q can be one of the k closest facilities of a point p iff $|C_p| < k$. Now, we define influence zone and RkNN queries.

Influence zone Z_k. Given a set of facilities F and a query $q \in F$, the influence zone Z_k is the area such that for every point $p \in Z_k$, $|C_p| < k$ and for every point $p' \notin Z_k$, $|C_{p'}| \geq k$.

Now, we define the reverse k nearest neighbor (RkNN) queries. RkNN queries are classified [24] into *bichromatic* and *monochromatic* RkNN queries. Below, we define both.

Bichromatic RkNN queries. Given a set of facilities F, a set of users U and a query $q \in F$, a bichromatic RkNN query is to retrieve every user $u \in U$ for which $|C_u| < k$.

Consider that the supermarkets and the houses in a city correspond to the set of facilities and users, respectively. A bichromatic RkNN query may be used to find every house for which a given supermarket is one of the k closest supermarkets.

Monochromatic RkNN queries. Given a set of facilities F and a query $q \in F$, a monochromatic RkNN query is to retrieve every facility $f \in F$ for which $|C_f| < k + 1$.

Please note that for every f, C_f contains the facility f. Hence we have condition $|C_f| < k + 1$ instead of $|C_f| < k$. Consider a set of police stations. For a given police station q, its monochromatic RkNNs are the police stations for which q is one of the k nearest police stations. Such police stations may seek assistance (e.g., extra policemen) from q in case of an emergency event.

Snapshot vs continuous RkNN queries. In a snapshot query, the results of the query are computed only once. In contrast, in a continuous query, the results are to be continuously updated as the objects in the underlying data sets change their locations. In this paper, we focus on a special case of continuous RkNN queries where only the users change their locations.

Given a set of facilities F, a query $q \in F$ and a set of users U, a continuous RkNN query is to continuously update the bichromatic RkNNs of q when one or more users change their locations.

A gas station may want to continuously monitor the vehicles for which it is one of the k closest gas stations. It may issue a continuous RkNN query to do so.

Throughout this paper, we use RNN query to refer to the RkNN query for which $k = 1$. Table 1 defines other notations used throughout this paper.

Table 1 Notations

Notation	Definition
q	the query point
C_p	a circle centered at p with radius $dist(p, q)$
$\|C_p\|$	the number of facilities located inside C_p
$B_{x:q}$	a perpendicular bisector between point x and q
$H_{x:q}$	a half-plane defined by $B_{x:q}$ containing point x
$H_{q:x}$	a half-plane defined by $B_{x:q}$ containing point q

2.2 Related work

2.2.1 Snapshot RkNN Queries

Korn *et al.* [24] were first to study RNN queries. They answer the RNN query by pre-calculating a circle for each data object p such that the nearest neighbor of p lies on the perimeter of the circle. RNN of a query q is every point that contains q in its circle. Techniques to improve their work were proposed in [45,28].

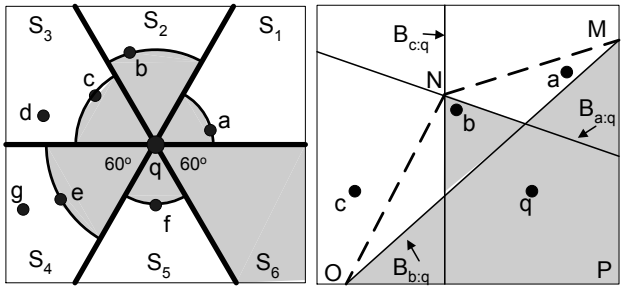
Now, we briefly describe the existing techniques that do not require preprocessing. These techniques have three phases namely *pruning*, *containment* and *verification*. In the pruning phase, the space that cannot contain any RkNN is pruned by using the set of facilities. In the containment phase, the users that lie within the unpruned space are retrieved. These are the possible RkNNs and are called the candidates. In the verification phase, a range query is issued for each candidate object to check if q is one of its k nearest facility or not.

First technique that does not need any preprocessing was proposed by Stanoi *et al.* [33]. They solve RNN queries by partitioning the whole space centred at the query q into six equal regions of 60° each (S_1 to S_6 in Fig. 1(a)). It can be proved that the nearest facility to q in each region defines the area that can be pruned. In other words, assume that f is the nearest facility to q in a region S_i. Then any user that lies in S_i and lies at a distance greater than $dist(q, f)$ from q cannot be the RNN of q. Fig. 1(a) shows nearest neighbors of q

[1] Although, like existing techniques [41,10], the focus of this paper is two dimensional location data, in Section 3.4, we show that the techniques can be extended to higher dimensionality.

in each region and the white area can be pruned. Only the users that lie in the shaded area can be the RNNs. The RkNN queries can be solved in a similar way, i.e., in each region, the k-th nearest facility of q defines the pruned area.

Tao *et al.* [36] proposed TPL that uses the property of perpendicular bisectors to prune the search space. Consider the example of Fig. 1(b), where a bisector between q and a is shown as $B_{a:q}$ which divides the space into two half-spaces. The half-space that contains a is denoted as $H_{a:q}$ and the half-space that contains q is denoted as $H_{q:a}$. Any point that lies in the half-space $H_{a:q}$ is always closer to a than to q and cannot be the RNN for this reason. Similarly, any point p that lies in k such half-spaces cannot be the RkNN. TPL algorithm prunes the space by the bisectors drawn between q and its neighbors in the unpruned area. Fig. 1(b) shows the example where the bisectors between q and a, b and c are drawn ($B_{a:q}$, $B_{b:q}$ and $B_{c:q}$, respectively). If $k = 2$, the white area can be pruned because every point in it lies in at least two half-spaces.



(a) Six-regions pruning (b) TPL and FINCH

Fig. 1 Related techniques

In the containment phase, TPL retrieves the users that lie in the unpruned area by traversing an R-tree that indexes the locations of the users. Let m be the number of facility points for which the bisectors are considered. An area that is the intersection of any combination of k half-spaces can be pruned. The total pruned area corresponds to the union of pruned regions by all such possible combinations of k bisectors (a total of $m!/k!(m-k)!$ combinations). Since the number of combinations is too large, TPL uses an alternative approach which has less pruning power but is cheaper. First, TPL sorts the m facility points by their Hilbert values. Then, only the combination of k consecutive facility points are considered to prune the space (total m combinations).

Achtert *et al.* [1] and Emrich *et al.* [13] propose pruning techniques that can be applied on the rectangles. They use these pruning techniques to prune the intermediate entries of the R-tree that indexes the facilities. It was demonstrated that the proposed techniques reduce the number of accessed pages. Moreover, pruning techniques proposed in [13] are more effective than the pruning techniques of [1].

Wu *et al.* [41] propose an algorithm called FINCH. Instead of using bisectors to prune the objects, they use a convex polygon that approximates the unpruned area. Any object that lies outside the polygon can be pruned. Fig. 1(b) shows an example where the shaded area is the unpruned area. FINCH approximates the unpruned area by a polygon $MNOP$. The algorithm can prune the intermediate nodes of the R-tree and the objects that lie outside this polygon. Clearly, the containment checking is cheaper than TPL (e.g., point containment can be done in logarithmic time for convex polygons). FINCH was shown to be superior to TPL [36].

It is worth mentioning that some of the existing work focus on computing Voronoi cell (or order k Voronoi cell) on the fly. More specifically, Stanoi *et al.* [34] compute Voronoi cell to answer RNN queries. On fly computation of order k Voronoi cell was presented in [49, 18] to monitor kNN queries. Yiu *et al.* [46] study the problem of common influence join and propose techniques for computing order k Voronoi cell on the fly. Unfortunately, none of the above mentioned approaches is applicable for RkNN queries. A straight forward extension is to compute several order k Voronoi cells and join them to construct the influence zone. However, this is computationally expensive because it requires constructing every order k Voronoi cell that contains q (see Section 5.1). The pre-processing based approach is also not suitable as discussed later in Section 5.1.

2.2.2 Continuous RNN Queries

Computation-efficient monitoring of continuous range queries [14,25,7], nearest neighbor queries [29,48,44, 21,37] and reverse nearest neighbor queries [2,42,23, 40] has received significant attention. Below, we briefly describe the algorithms that monitor continuous RNN queries.

Benetis *et al.* [2] presented the first continuous RNN monitoring algorithm. However, they assume that velocities of the objects are known. First work that does not assume any knowledge of objects' motion patterns was presented by Xia *et al.* [42]. Their proposed solution is based on the six 60^o regions based approach described earlier in this section. Kang *et al.* [23] proposed a continuous monitoring RNN algorithm based on the bisector based (TPL) pruning approach. Both of these algorithms continuously monitor RNN queries by monitoring the unpruned area.

Wu *et al.* [40] propose the first technique to monitor RkNNs. Their technique is based on the six-regions based RNN monitoring presented in [42]. More specif-

ically, they issue k nearest neighbor (kNN) queries in each region instead of the single nearest neighbor queries. The users that are closer than the k-th NN in each region are the candidate objects and they are verified if q is one of their k closest facilities. To monitor the results, for each candidate object, they continuously monitor the circle around it that contains k nearest facilities.

Cheema *et al.* [10] propose Lazy Updates that is the best known algorithm to continuously monitor RkNN queries. Emrich *et al.* [12] independently proposed an approach similar to Lazy Updates [10]. Lazy Updates not only reduces the computation cost but also significantly reduces the communication cost. The existing approaches call the expensive pruning phase whenever the query or a candidate object changes the location. Lazy Updates saves the computation time by reducing the number of calls to the expensive pruning phase. They assign each moving object a safe region and propose the pruning techniques to prune the space based on the safe regions. The pruning phase is not needed to be called as long as the related objects remain inside their safe regions.

It is worth mentioning that all of the existing techniques solve the general problem where every data point including the query point is moving. In this paper, we solve a special case of the problem where the facilities do not move and the users are moving.

2.2.3 RNN queries under other settings

In this section, we provide an overview of the RNN queries studied in other popular problem settings.

RNN queries in road networks. Yiu *at al.* [47] are the first to study RNN queries in large graphs. They present an interesting observation that is used to prune the search space while traversing the graph in search of RNN. Safar *et al.* [32] use Network Voronoi Diagram (NVD) [30] to efficiently process the RNN queries in spatial networks. In a following work [39], they extend their technique to answer snapshot RkNN queries and reverse k furthest neighbor queries in spatial network.

Sun *et al.* [35] study the continuous monitoring of RNN queries in spatial networks. The main idea is that for each query a multi-way tree is created that helps in defining the monitoring region. Only the updates in the monitoring region affect the results.

Li *et al.* [26] present a technique to continuously monitor RkNN queries in spatial networks. They propose a novel data structure called dual layer multiway tree (DLM tree) which is used to represent the monitoring region of RkNN queries. They present several observations to reduce the size of the region that is to be monitored for a RkNN query.

Cheema *et al.* [11] propose Lazy Updates that answers continuous RkNN queries in Euclidean space as well as in spatial networks. Each object and query is assigned a safe region and the expensive pruning phase is not required as long as the query and relevant objects remain in their respective safe regions. The proposed technique reduces the computation cost as well as the communication cost.

RNN queries on uncertain data. Probabilistic RNN queries [8,27,3,4] has also received significant attention from the research community. The basic idea behind these techniques is as follows. Each uncertain object and query is approximated by a rectangular [8,4] or a circular [27] region. Pruning techniques are developed to prune the space based on these regions. Then, each object that cannot be pruned is treated as a candidate object and its probability of being the RNN is computed.

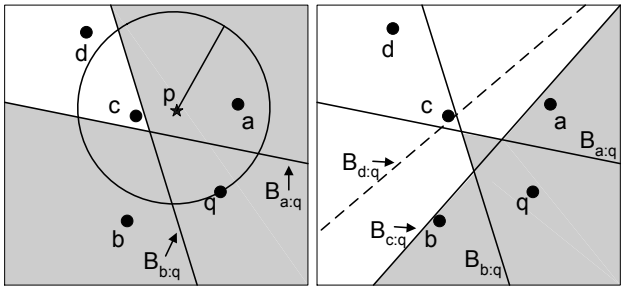
3 Computing Influence Zone

3.1 Problem Characteristics

Given two facility points a and q, a perpendicular bisector $B_{a:q}$ between these two points divides the space into two halves as shown in Fig 2(a). The half plane that contains a is denoted as $H_{a:q}$ and the half plane that contains q is denoted as $H_{q:a}$. The perpendicular bisector has the property that any point p (depicted by a star in Fig. 2(a)) that lies in $H_{a:q}$ is closer to a than q (i.e., $dist(p,a) \leq dist(p,q)$) and any point y that lies in $H_{q:a}$ is closer to q than a (i.e., $dist(y,q) \leq dist(y,a)$). Hence, q cannot be the closest facility of any point p that lies in $H_{a:q}$, i.e., C_p contains at least one facility a. We say that the point p is pruned by the bisector $B_{a:q}$ if p lies in $H_{a:q}$. Alternatively, we say that the point a prunes the point p. In general, if a point p is pruned by at least k bisectors then C_p contains at least k facilities (i.e., $|C_p| \geq k$).

Existing work [36,41,10] use this observation to prune the space that cannot contain any RkNN of q. More specifically, an area can be pruned if at least k bisectors prune it. In Fig. 2, five facility points (q, a, b, c and d) are shown. In Fig. 2(a) the bisectors between q and two facility points a and b are drawn (see $B_{a:q}$ and $B_{b:q}$). If k is 2, then the white area can be pruned because it lies in two half-planes ($H_{a:q}$ and $H_{b:q}$) and $|C_{p'}| \geq 2$ for any point p' in it. The area that is not pruned is called unpruned area and is shown shaded.

Although it can be guaranteed that for every point p' in the pruned area $|C_{p'}| \geq k$, it cannot be guaranteed that for every point p in the unpruned area $|C_p| < k$ if we only consider a subset of the bisectors instead of all



(a) Unpruned area is not influence zone (b) Unpruned area is influence zone

Fig. 2 Computing influence zone Z_k $(k = 2)$

bisectors. In other words, the unpruned area is not the influence zone. For example, in Fig. 2(a), the point p lies in the unpruned area but $|C_p| = 2$ (i.e., C_p contains a and c). Hence, the shaded area of Fig. 2(a) is not the influence zone.

One straight forward approach to compute the influence zone is to consider the bisectors of q with every facility point f. If the bisectors of q and all facilities are considered, then the unpruned area is the area that is pruned by less than k bisectors. Fig. 2(b) shows the unpruned area (the shaded polygon) after the bisectors $B_{c:q}$ and $B_{d:q}$ are also considered. It can be verified that the shaded area is the influence zone (i.e., for every p in the shaded area $|C_p| < 2$ and for every p' outside it $|C_{p'}| \geq 2$).

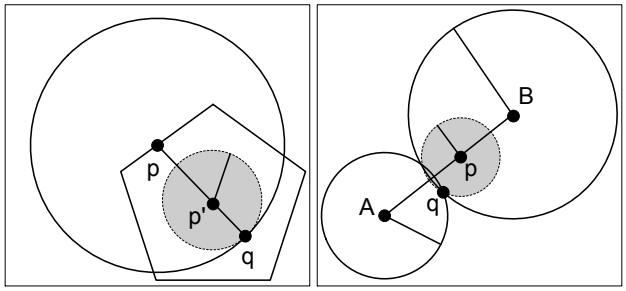
However, this straight forward approach is too expensive because it requires computing the bisectors between q and all facility points. We note that for some facilities, we do not need to consider their bisectors. In Fig. 2(b), it can be seen that the bisector $B_{d:q}$ (shown in broken line) does not affect the unpruned area (shown shaded). In other words, if the bisectors of a, b and c are considered then the bisector $B_{d:q}$ does not prune more area. Hence, even if $B_{d:q}$ is ignored, the influence zone can be computed.

Next, we present some lemmas that help us in identifying the facilities that can be ignored. Without loss of generality, we assume that the data universe is bounded by a square. Since we use bisectors to prune the space, the unpruned area is always a polygon and is interchangeably called unpruned polygon hereafter. Below we present several lemmas that not only guide us to the final lemma but also help us in few other proofs in the paper.

Lemma 1 *A facility f can be ignored if, for* every *point p of the unpruned polygon, the facility f lies outside C_p.*

Proof As described earlier, a point p can be pruned by the bisector $B_{f:q}$ iff $dist(p, f) < dist(p, q)$. In other words, the point p can be pruned iff C_p contains f. Hence, if f lies outside C_p, it cannot prune p. If f lies outside C_p for *every* point p, it cannot prune *any* point of the unpruned polygon and can be ignored for this reason. □

Checking containment of f in C_p for every point p is not feasible. In next few lemmas, we simplify the procedure to check if a facility point can be ignored.



(a) Lemma 2 and 3 (b) Lemma 4

Fig. 3

Lemma 2 *Let pq be a line segment between two points q and p. Let p' be a point on pq. The circle $C_{p'}$ is contained by the circle C_p.*

Fig. 3(a) shows an example where the circle $C_{p'}$ (the shaded circle) is contained by C_p (the large circle). The proof is straight forward and is omitted. Based on this lemma, we present our next lemma.

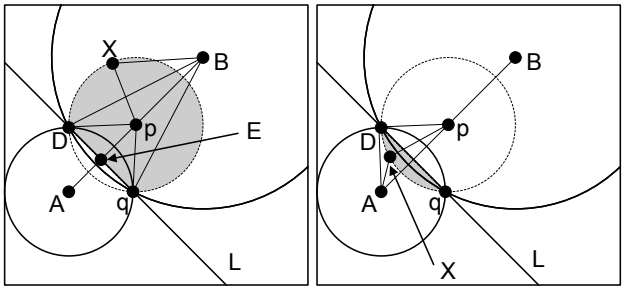
Lemma 3 *A facility f can be ignored if, for* every *point p on the* boundary *of the unpruned polygon, f lies outside C_p.*

Proof We prove the lemma by showing that we do not need to check containment of f in $C_{p'}$ for any point p' that lies within the polygon. Let p' be a point that lies within the polygon. We draw a line that passes through q and p' and cuts the polygon at a point p (see Fig. 3(a)). From Lemma 2, we know that C_p contains $C_{p'}$. Hence, if f lies outside C_p, then it also lies outside $C_{p'}$. Hence, it suffices to check the containment of f in C_p for every point p on the boundary of the polygon. □

The next two lemmas show that we can check if a facility f can be ignored or not by only checking the containment of f in C_v for every vertex v of the unpruned polygon.

Lemma 4 *Given a line segment AB and a point p on AB. The circle C_p is contained by $C_A \cup C_B$, i.e., every point in the circle C_p is either contained by C_A or by C_B (see Fig. 3(b)).*

Proof Fig. 4 shows the line segment AB and the point p. It suffices to show that the boundary of C_p is contained by $C_A \cup C_B$. If q lies on AB, the lemma can be proved by Lemma 2. Otherwise, we identify a point D such that AB is a segment of the perpendicular bisector between D and q. Then, we draw a line L that passes through points D and q. First, we show that the part of the circle C_p that lies on the right side of L (i.e., the shaded part in Fig. 4(a)) is contained by C_B. Then, we show that the part of the circle C_p that lies on the left side of L (i.e., the shaded part in Fig. 4(b)) is contained by C_A.



(a) For right side of L (b) For left side of L

Fig. 4 Illustration of Lemma 4

We can find the length of qB (denoted as $\overline{qB}$) by using the triangle $\triangle qpB$ and applying the law of cosines (see Fig. 4(a)).

$$\overline{qB} = \sqrt{(\overline{pB})^2 + (\overline{pq})^2 - 2 \cdot \overline{pB} \cdot \overline{pq}(Cos\measuredangle Bpq)} \quad (1)$$

For any point X that lies on the boundary of C_p and is on the right side of L (i.e., the boundary of the shaded circle in Fig. 4(a)), consider the triangle $\triangle\ pXB$. The length of BX can be computed using the law of cosines.

$$\overline{BX} = \sqrt{(\overline{pB})^2 + (\overline{pX})^2 - 2 \cdot \overline{pB} \cdot \overline{pX}(Cos\measuredangle BpX)} \quad (2)$$

Please note that the triangles $\triangle qpB$ and $\triangle DpB$ are similar because $\overline{Dp} = \overline{qp}$ and $\overline{DB} = \overline{qB}$ (any point on a perpendicular bisector $B_{u:v}$ is equi-distant from u and v). Due to similarity of triangles $\triangle qpB$ and $\triangle DpB$, $\measuredangle Bpq = \measuredangle BpD$.

It can be shown that $\overline{BX} \leq \overline{qB}$ by comparing Eq. (1) and Eq. (2). This is because $\overline{pX} = \overline{pq}$ and $\measuredangle BpX \leq (\measuredangle Bpq = \measuredangle BpD)$. Since cosine monotonically decreases as the angle increases from 0° to 180°, $\overline{BX} \leq \overline{qB}$. This means the point X lies within the circle C_B.

Similarly, for any X that lies on the part of circle C_p that is on left side of the line L (see Fig. 4(b)) it can be shown that $\overline{AX} \leq (\overline{AD} = \overline{Aq})$. This can be achieved by considering the triangles $\triangle pXA$ and $\triangle pDA$ and using law of cosines to obtain $\overline{AX}$ and $\overline{AD}$ (the key observation is that $\measuredangle XpA \leq \measuredangle DpA$). □

Lemma 5 *A facility f can be ignored if, for every vertex v of the unpruned polygon, the facility f lies outside C_v.*

Proof Let AB be an edge of the polygon. From Lemma 4, we know that if a facility f lies outside C_A and C_B, then it lies outside C_p for every point p on the edge AB. This implies that if f lies outside C_v for every vertex v of the polygon then it lies outside C_p for every point p that lies on the boundary of the polygon. Such facility f can be ignored as stated in Lemma 3. □

Next lemma shows that we only need to check this condition for *convex* vertices. First, we define the convex vertices.

Definition 1 Consider a polygon P where V is the set of its vertices. Let H_{con} be the convex hull of V. The vertices of H_{con} are called convex vertices of the polygon P and the set of the convex vertices is denoted as V_{con}.

Fig. 5 shows an example where a polygon with vertices A to J is shown in broken lines. Its convex hull is shown in solid lines which contains the vertices A, C, E, G and I and these vertices are the convex vertices. Note that $V_{con} \subseteq V$.

Lemma 6 *A facility f can be ignored if it lies outside C_v for every* convex vertex *v of the unpruned polygon P.*

Proof By definition of a convex hull, the convex hull H_{con} contains the polygon P. If a facility point f does not prune any point of the convex polygon H_{con}, it cannot prune any point of the polygon P because $P \subseteq H_{con}$. Hence, it suffices to check if f prunes any point of H_{con} or not. From Lemma 5, we know that f does not prune any point of H_{con} if it lies outside C_v for every vertex v of H_{con}. Hence, f can be ignored if it lies outside every C_v where v is a vertex of the convex polygon (i.e., v is a convex vertex). □

The above lemma identifies a condition for a facility f to be ignored. Next lemma shows that any facility that does not satisfy this condition prunes at least one point of the unpruned area. In other words, next lemma shows that the above condition is tight.

Lemma 7 *If a facility f lies in* any *C_v for any convex vertex v of the unpruned polygon P then there exists at least one point p in the polygon P that is pruned by f.*

Proof If f lies in C_v for any $v \in V_{con}$, it means that $dist(f, v) < dist(f, q)$. Hence, f prunes the vertex v. Since $V_{con} \subseteq V$, the vertex v is a point in the polygon P. □

3.2 Algorithm

Based on the problem characteristics we described earlier in this section, we propose an algorithm to efficiently compute the influence zone. We assume that the facilities are indexed by an R-tree [15]. The main idea is that the facilities are iteratively retrieved and the space is iteratively pruned by considering their bisectors with q. The facilities that are close to the query q are expected to prune larger area and are given priority.

Algorithm 1 presents the details. Initially, the whole data space is considered as the influence zone and the root of the R-tree is inserted in a min-heap h. The entries are iteratively de-heaped from the heap. The entries in the heap may be rectangles (e.g., intermediate nodes) or points. If a de-heaped entry e completely lies outside C_v of all convex vertices of the current influence zone (e.g., the current unpruned area), it can be ignored. Otherwise, it is considered valid (lines 5 to 7). If the entry is valid and is an intermediate node or a leaf node, its children are inserted in the heap (lines 8 to 10). Otherwise, if the entry e is valid and is a data object (e.g., a facility point), it is used to prune the space. The current influence zone is also updated accordingly (line 12). The algorithm stops when the heap becomes empty.

Algorithm 1 Compute Influence Zone

Input: a set of objects O, a query $q \in O$, k
Output: Influence Zone Z_k

```
1: initialize Z_k to the boundary of data universe
2: insert root of R-tree in a min-heap h
3: while h is not empty do
4:    deheap an entry e
5:    for each convex vertex v of Z_k do
6:       if mindist(v, e) < dist(v, q) then
7:          mark e as valid; break
8:    if e is valid then
9:       if e is an intermediate node or leaf then
10:         insert every child c in h with key mindist(q, c)
11:      else if e is an object then
12:         update the influence zone Z_k using e (Algorithm 2)
```

The proof of correctness follows from the lemmas presented in the previous section because only the objects that do not affect the unpruned area are ignored. It is also important to note that the entries of R-tree are accessed in ascending order of their minimum distances to the query. The nearby facility points are accessed and the unpruned area keeps shrinking which results in a greater number of upcoming entries being pruned. Hence, the entries that are far from the query are never accessed.

3.2.1 Updating unpruned polygon

Now, we briefly describe how to update the unpruned polygon (or current influence zone) when a new facility point f is considered (line 12 of Algorithm 1). The main idea is similar to [41]. The intersection points between all the bisectors are maintained. Each intersection point is assigned a counter that denotes the number of bisectors that prune it. Fig. 6 shows an example ($k = 2$) where three bisectors $B_{a:q}$, $B_{b:q}$ and $B_{c:q}$ have been considered. The counter of intersection point v_{11} is 2 because it is pruned by $B_{b:q}$ and $B_{c:q}$. The counter of v_8 is 1 because it is pruned only by $B_{c:q}$. It can be immediately verified that the unpruned area can be defined by only the intersection points with counters less than k [41] (see the shaded area of Fig. 6). Hence, we can discard the intersection points with counters at least equal to k.

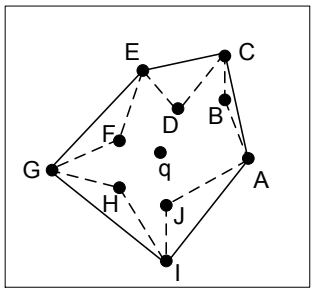


Fig. 5 Convex Polygon

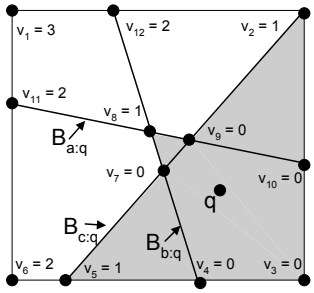


Fig. 6 Computing counters

Algorithm 2 shows the details of updating the influence zone when a new facility f is considered. Firstly, the algorithm computes the new intersection points between $B_{f:q}$ and the existing bisectors. The counters of these new intersection points are also computed (line 1). Then, the algorithm updates the counters of all existing intersection points (line 2). More specifically, the counter of an existing intersection point p is incremented by one if $B_{f:q}$ prunes p. Otherwise, the counter remains unchanged. The algorithm discards the intersection points with counters at least equal to k (line 3). Then, the algorithm determines the convex vertices and computes the current unpruned polygon (lines 4 and 5). Recall that determining the convex vertices is important in order to apply Lemma 6.

Algorithm 2 update influence zone

Input: current influence zone Z_k, a new facility f
Output: updated influence zone Z_k

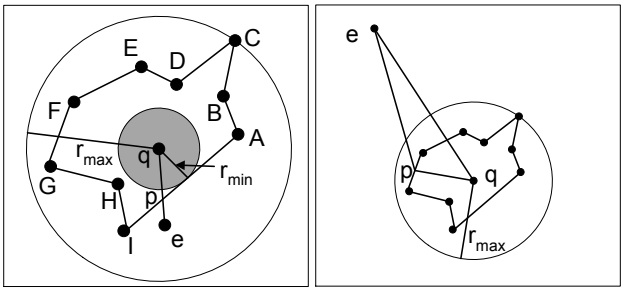
```
1: compute new intersection points and their counters
2: update the counters of existing intersection points
3: discard intersection points with counters at least equal to k
4: find the convex vertices
5: compute the unpruned polygon
```

We remark that the first three lines of Algorithm 2 are the same as used in the technique proposed by Wu

et. al [41]. They showed that the complexity of these lines is $O(m^2)$ where m is the number of existing bisectors contributing to the unpruned polygon. Later in Section 5.4, we conduct a more rigorous complexity analysis and show that the overall complexity of Algorithm 2 can be reduced to $O(km)$ when k is smaller than m.

3.2.2 Optimizations

In this section, we present few optimizations to improve the efficiency of Algorithm 1. It can be shown that the number of convex vertices is $O(m)$ where m is the number of bisectors considered so far [41] (i.e., m is the number of facilities used to update the current influence zone at line 12 of Algorithm 1). Hence, checking whether an entry of the R-tree is valid or not requires $O(m)$ distance computations (see lines 5 to 7 of Algorithm 1). Next, we present few observations and show that we can determine the validity of some entries by a single distance computation.



(a) Lemma 8 (b) Lemma 9

Fig. 7 Optimizations

Lemma 8 *Let r_{min} be the minimum distance of q to the boundary of the unpruned polygon. Then, an entry e is a valid entry if $mindist(q, e) < 2r_{min}$ (Fig. 7(a) shows r_{min}).*

Proof To prove that e is a valid entry, we show that there exists at least one point p in the unpruned polygon such that C_p contains e. If e lies inside the unpruned polygon then e is a valid entry because C_e contains e and e is a point in the unpruned polygon. Now, we prove the lemma for the case when e lies outside the unpruned polygon. Fig. 7(a) shows an entry e for which $dist(q, e) < 2r_{min}$. We draw a line that passes through e and q and intersects the boundary of the unpruned polygon at a point p. Clearly, $dist(p, e) = dist(q, e) - dist(p, q)$. We know that $dist(q, e) < 2r_{min}$ and $dist(p, q) \geq r_{min}$. Hence, $dist(p, e) \leq r_{min}$ which implies that $dist(p, e) \leq dist(p, q)$. Hence, e lies in C_p. □

Lemma 9 *Let r_{max} be the distance of q to the furthest vertex of the unpruned polygon. Then, an entry e of the R-tree is an invalid entry if $mindist(e, q) > 2r_{max}$.*

Proof Fig. 7(b) shows r_{max} and a point e such that $dist(e, q) > 2r_{max}$. Consider a point p on the boundary of the unpruned polygon. By the definition of r_{max}, $dist(p, q) \leq r_{max}$. Clearly, $dist(p, q)+dist(p, e) \geq dist(q, e)$ (this covers both the cases when p lies on the line qe and when $\triangle qpe$ is a triangle). Since, $dist(p, q) \leq r_{max}$ and $dist(e, q) > 2r_{max}$, $dist(p, e)$ must be greater than r_{max}. Hence, $dist(p, e) > dist(p, q)$ which means e lies outside C_p. This holds true for every point p on the boundary of the unpruned polygon. Hence, e can be ignored (i.e., e is invalid). □

If an entry of the R-tree satisfies one of the above two lemmas, we can determine its validity without computing its distances from the convex vertices. Note that r_{max} and r_{min} can be computed in linear time to the number of edges of the unpruned polygon and are only computed when the influence zone is updated at line 12 of Algorithm 1.

3.3 Checking containment in the influence zone

The applications that use influence zone may require to frequently check if a point or a shape lies within the influence zone or not. Although the suitability of a method to check the containment depends on the nature of the application, we briefly describe few approaches.

One simple approach is to record all the objects that were accessed during the construction of the influence zone (the objects for which the bisectors were considered). If a shape is pruned by less than k of these bisectors then the shape lies inside the influence zone otherwise it lies outside the influence zone. This approach takes linear time in number of the accessed objects. Moreover, checking whether a point is pruned by a bisector $B_{f:q}$ is easy (e.g., if $dist(p, f) < dist(p, q)$ then the point p is pruned otherwise not). Hence, a point containment check requires $O(m)$ distance computations where m is the number of the accessed objects.

Before we show that the point containment can be done in logarithmic time, we define a *star-shaped polygon* [31]. A polygon is a star-shaped polygon if there exists a point z in it such that for each point p in the polygon the segment zp lies entirely in the polygon. The point z is called a kernel point. The polygon shown in Fig. 7(a) is a star-shaped polygon and q is its kernel point. Fig. 8(a) shows a polygon that is not star-shaped (the segment qp does not lie entirely in the polygon).

Let n be the number of vertices of a star-shaped polygon. After a linear time pre-processing, every point containment check can be done in $O(log\ n)$ if a kernel point of the polygon is known [31]. Please see [31] for more details.

Lemma 10 *The influence zone is always a star-shaped polygon and q is its kernel point.*

Proof We prove this by contradiction. Assume that there is a point p in the influence zone such that the segment pq does not lie completely within the influence zone. Fig. 8(a) shows an example, where a point p' lies on the segment pq but does not lie within the influence zone. From Lemma 2, we know that C_p contains $C_{p'}$. Since p' is a point outside the influence zone, $|C_{p'}| \geq k$. As $C_{p'}$ is contained by C_p, $|C_p| \geq k$. Hence, p cannot be a point inside the influence zone which contradicts the assumption. □

Since the maximum number of vertices of the influence zone is $O(m^2)$, the point containment check can be done in $O(log\ m)$. Next, we present two simple checks to reduce the cost of containment check in certain cases by using r_{max} and r_{min} we introduced earlier.

Let r_{min} and r_{max} be as defined in Lemma 8 and 9, respectively. Then, the circle centered at q with radius r_{max} (the big circle in Fig. 7(a)) completely contains the influence zone. Similarly, the circle centered at q with radius r_{min} (the shaded circle in Fig. 7(a)) is completely contained by the influence zone. Hence, any point p that has a distance greater than r_{max} from q is not contained by the influence zone and any point p' that lies within distance r_{min} of q is contained by the influence zone.

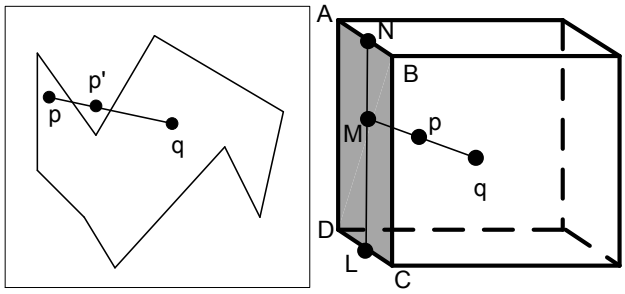
For the applications that allow relatively expensive pre-processing, the influence zone can be indexed (e.g., by a grid or a quad-tree) to efficiently check the containment. For example, for the continuous monitoring of RkNN queries, we use a grid to index the influence zone. The details are presented in next section.

3.4 Extension to higher dimensions

RkNN queries have various applications in higher dimensional space such as in classification, profile based advertisement, and document repositories [24]. For instance, in classification, the RkNN query is commonly used to select a suitable classifier. More specifically, an object o is a good classifier if its RkNNs also belong to the same cluster as of o [43]. Due to their importance, several approaches have been presented to compute RkNN in arbitrary dimensionality [36,1,13]. Although the focus of this paper is on developing techniques for two-dimensional data, we show that our proposed techniques can be extended for arbitrary dimensionality.

In dimensions higher than two, the bisectors are called half-spaces and the unpruned region is a polytope instead of a polygon [30]. The circle C_p centered at p with radius $dist(p, q)$ is called a hypersphere. It can be shown that Lemma 4 holds for higher dimensions. This can be proved by a projection on a two dimensional space for each point of the hypersphere.

The space is pruned in a similar way as in two dimensional space, i.e., the space that is pruned by at least k half-spaces is pruned. The following lemma holds for the unpruned region which is a polytope.



(a) Lemma 10 (b) Lemma 11

Fig. 8

Lemma 11 *A facility point f can be ignored if, for every vertex v of the unpruned polytope, f lies outside C_v.*

Proof We prove the lemma for a 3-dimensional polyhedron and the proof for the arbitrary dimensionality is similar. Let p be any point inside the polyhedron as shown in Fig. 8(b). We draw a line that passes through p and q and crosses a face (the shaded face $ABCD$) of the polyhedron at a point M. For such point M, we can always draw a line on this face of the polyhedron such that it passes through M and intersects the edges of the face at points L and N as shown in Fig 8(b). From Lemma 4, C_A and C_B contain C_N. Similarly, C_C and C_D contain C_L. Again, from Lemma 4, C_N and C_L contain C_M. Lastly, C_M contains C_p (Lemma 2). Hence, C_p is contained by the hyperspheres of the vertices of the face $ABCD$ (C_A, C_B, C_C and C_D). This holds for any arbitrary point p inside the polyhedron. Hence, we only need to check the containment in C_v for every vertex v of the polyhedron. □

Given Lemma 11, it can be immediately verified that Lemmas 6 and 7 also hold in dimensions higher than two.

4 Applications in RkNN Processing

4.1 Snapshot Bichromatic RkNN Queries

Our algorithm consists of two phases namely *pruning* phase and *containment* phase.

Pruning Phase. In this phase, the influence zone Z_k is computed using the given set of facilities.

Containment Phase. By the definition of influence zone Z_k, a user u can be the bichromatic RkNN if and only if it lies within the influence zone Z_k. We assume that the set of users are indexed by an R-tree. The R-tree is traversed and the entries that lie outside the influence zone are pruned. The objects that lie in the influence zone are RkNNs.

4.2 Snapshot Monochromatic RkNN Queries

By definition of a monochromatic RkNN query (see Section 2.1), a facility f is the RkNN iff $|C_f| < k+1$. Hence, a facility that lies in Z_{k+1} is the monochromatic RkNN of q where Z_{k+1} is the influence zone computed by setting k to $k+1$. Below, we highlight our technique.

Pruning Phase. In this phase, we compute the influence zone Z_{k+1} using the given set of facilities F. We also record the facility points that are accessed during the construction of the influence zone and call them the candidate objects.

Containment Phase. Please note that every facility point that is contained in the influence zone Z_{k+1} will be accessed during the pruning phase. This is because every facility that lies in the influence zone cannot be ignored during the construction of the influence zone (inferred from Lemma 1). Hence, the set of candidate object contains all possible RkNNs. For each of the candidate object, we report it as RkNN if it lies within the influence zone Z_{k+1}.

4.3 Continuous monitoring of RkNNs

In this section, we present our technique to continuously monitor bichromatic RkNN queries (see the problem definition in Section 2.1). The basic idea is to index the influence zone by a grid. Then, the RkNNs can be monitored by tracking the users that enter or leave the influence zone.

Initially, the influence zone Z_k of a query q is computed by using the set of facility points. We use a grid based data structure to index the influence zone. More specifically, a cell c of the grid is marked as an *interior* cell if it is completely contained by the influence zone. A cell c' is marked as a *border* cell if it overlaps with the boundary of the influence zone. Fig. 9(a) shows an example where the influence zone is the polygon $ABCDFEGHI$, interior cells are shown in dark shade and the border cells are the light shaded cells.

For each border cell, we record the edges of the polygon that intersect it. For example, in c_1, we record the edge AI and in c_2 we record the edges AI and HI. If a user $u \in U$ is in an interior cell, we report it as RkNN of the query. If a user lies in a border cell, we check if it lies outside the polygon by checking the edges stored in this cell. For example, if a user lies in c_1 and it lies inside AI, we report it as RkNN.

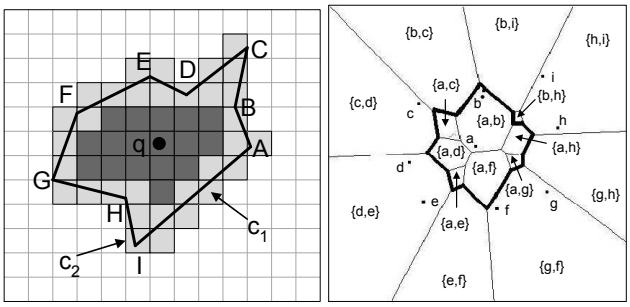
5 Theoretical Analysis

We assume that the facilities and the users are uniformly distributed in a unit space. The number of facilities is $|F|$. For bichromatic queries, the number of users is $|U|$.

5.1 Size of Influence Zone

Before we analyse the size of the influence zone, we show the relationship between an order k Voronoi cell and the influence zone. We utilize this relationship to analyse the area/volume of the influence zone.

Relationship with order k Voronoi cell: An order k Voronoi diagram divides the space into cells and we refer to each cell as a k-Voronoi cell. Each k-Voronoi cell is related to a set of k facility points (denoted as F_k) such that for any point p in this cell, the k closest facilities are F_k. Fig. 9(b) shows an order 2 Voronoi diagram computed on the facility points a to i. Each cell c is related to two facility points (shown as $\{f_i, f_j\}$ in Fig. 9(b)) and these are the two closest facilities for any point p in c. For example, for any point p in the cell marked as $\{a, e\}$ the two closest facilities are a and e.



(a) Continuous Monitoring (b) Order 2 Voronoi diagram

Fig. 9

Clearly, when $k = 1$ the k-Voronoi cell related to q is exactly the same as the influence zone. For $k > 1$, the influence zone corresponds to the union of all k-Voronoi cells that are related to q (i.e., have q in their F_k). For example, in Fig. 9(b), the influence zone of the facility a is shown in bold boundary and it corresponds to the union of the cells related to a.

Now, we analyse the area of the influence zone.

Consider the influence zones of all the facilities in the data set. Every point in the unit space lies in a cell that is related to k facilities. This implies that every point lies in exactly k influence zones (e.g., in Fig. 9(b), every point in the cell marked as $\{a, f\}$ lies in the influence zone of a as well as the influence zone of f). Hence, the sum of the areas of all the influence zones is k. Since the total number of facility points is $|F|$, the expected area of a randomly chosen facility point is $k/|F|$.

Note that the above discussion does not depend on the dimensionality. Hence, the volume of the influence zone is $k/|F|$ regardless of the dimensionality.

Remark: The above discussion shows that the influence zone can be computed by using a pre-computed order k Voronoi diagram. However, as mentioned in [49], a technique that uses a pre-computed order k Voronoi diagram may not be practical for the following reasons : i) the value of k may not be known in advance; ii) even if k is known in advance, order k Voronoi diagrams are very expensive to compute and incur high space requirement; iii) spatial indexes are useful for all query types and pre-computed Voronoi diagrams may not be used for all queries. In contrast, R-tree based indexes used by our algorithm are used for many important queries.

5.2 Result size of RkNN queries

First, we analyse the result size for bichromatic RkNNs queries. We assume that the users are uniformly distributed in the space. Recall that every user that lies in the influence zone is a bichromatic RkNN object. Hence, the expected result size (i.e., number of bichromatic RkNNs) can be obtained by multiplying $|U|$ with the expected area (volume for higher dimensionality) of the influence zone. Hence, the expected number of bichromatic RkNNs is $|U|.k/|F|$ (regardless of the dimensionality).

Now, we analyse the result size for monochromatic RkNN queries. The area/volume of the influence zone Z_{k+1} for a monochromatic RkNN query is $(k+1)/|F|$. The number of facilities in this zone is $(k+1)$ which includes the query. Hence the expected number of monochromatic RkNNs is k (regardless of the dimensionality).

5.3 IO cost of our algorithms

In this section, we present IO cost analysis for our algorithms which is applicable to arbitrary dimensionality. Before we analyse the IO costs of our proposed algorithms, we analyse the cost of a *circular range* query in d-dimensional space. Then, we analyse the costs of our algorithms by using the IO cost of the circular range queries.

5.3.1 IO cost of a circular range query

A circular range query [6] finds the objects that lie within distance r of the query location. We assume that the objects are indexed by an R-tree and analyse the number of nodes that lie within the range of the query.

The approach to analyse the IO cost of the circular range query is similar to the IO cost analysis of window queries presented in [38]. Assume a hypersphere in a d-dimensional space that has a radius R. Let V_R denote the volume of this hypersphere. Let N_l be the number of rectangles at level l of the R-tree. We assume that the centers of the rectangles at each level follow a uniform distribution. The expected number of rectangles at a level l that have their centers in the hypersphere is $V_R \times N_l$.

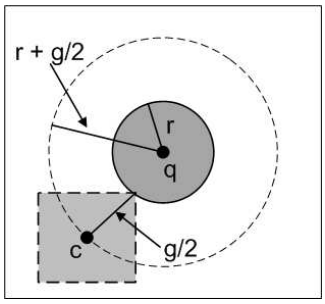


Fig. 10 Range query

Fig. 10 shows a two dimensional range query q with range r (the shaded circle). We first analyse the number of rectangles at level l that lie within the range r. Let s_l be the side length of each rectangle at level l (the rectangles of a good R-tree have similar sizes [22]). Let g_l be the diagonal length of each rectangle at level l. As shown in Fig. 10, any rectangle that has its center c further than $r + g_l/2$ from q cannot intersect the range query and should not be accessed. In other words, at a level l, the number of rectangles (nodes) that are to be accessed is the number of centers c falling in the hypersphere centered at q with radius $R = r + g_l/2$ (the large circle in Fig. 10). Hence, the number of nodes accessed at level l is $V_R \times N_l$ where $R = r + g_l/2$ and V_R denotes the volume of the hypersphere.

The total IO cost (the total number of nodes accessed) is obtained by adding the number of accessed

nodes for each level l. The total number of levels excluding the root is $\lfloor log_f S \rfloor$ where f is the fanout of R-tree and S is the total number of objects indexed by the R-tree. The root is accessed anyway, so one is added to this cost. Hence, the total IO cost is obtained by Eq. (3).

$$Range\ query\ cost = 1 + \sum_{l=1}^{\lfloor log_f S \rfloor} V_R \times N_l \quad (3)$$

Now, we need to compute V_R, and N_l for each level l. To compute V_R, we need to compute g_l. The number of rectangles N_l at level l of the R-tree is S/f^l (e.g., leaf nodes are at level 1 and the number of leaf level rectangles is S/f). Since we assume uniform distribution of points, each rectangle at level l contains f^l points. In other words, the area/volume of each node (rectangle) is f^l/S. Assuming that the side length of a rectangle on each dimension is the same, the side length s_l is $(f^l/S)^{1/d}$. Given s_l of a rectangle, the diagonal length g_l can be computed by using Eq. (4).

$$g_l = \sqrt{\sum_{i=1}^{d} s_l^2} = \sqrt{\sum_{i=1}^{d} (\frac{f^l}{S})^{\frac{2}{d}}} = \sqrt{d \times (\frac{f^l}{S})^{\frac{2}{d}}} \quad (4)$$

Finally, we need to compute V_R. Let R be the radius of a hypersphere. The volume V_R of the hypersphere is $V_R = C_d \times R^d$ where d denotes the dimensionality of the hypersphere[2]. C_d for even dimensionality is $\frac{\pi^{d/2}}{(d/2)!}$ where $x!$ denotes the factorial of a number x. C_d for odd dimensionality is $\frac{2^{(d+1)/2}\pi^{(d-1)/2}}{d!!}$ where $x!!$ denotes the *double factorial* of x. The double factorial of x is the multiplication of all odd numbers from 1 to x.

By using g_l shown in Eq. (4), we can compute R (and V_R). Plugging the values of V_R and N_l in Eq. (3) gives us the IO cost of the range query. Based on the IO cost of the range query, first we analyse the cost of computing the influence zone and then we analyse the costs of our RkNN algorithms.

5.3.2 IO cost of computing the influence zone

We approximate the influence zone to a hyperspherical region that has the same area/volume as that of the influence zone (we noted that as k gets larger the shape of the influence zone has more resemblance with a hypersphere). Since the area/volume of the influence zone Z_k is $k/|F|$, the radius r_k of the hypersphere can be computed. More specifically, $V_{r_k} = k/|F| = C_d \times r_k^d$ which implies that $r_k = (\frac{k}{|F| \times C_d})^{1/d}$. As implied by Lemma 5, an object can be ignored if it lies at a distance greater than $dist(q, v)$ from every vertex v of the unpruned region. Since we assume that each vertex is at the same distance r_k from the query (i.e., influence zone is a hypersphere), an object can be ignored if it lies at a distance greater than $2r_k$ from q. Hence, the objects within the range $2r_k$ of the query are accessed during the computation of the influence zone. The IO cost is then the cost of a range query with range $2r_k = 2(\frac{k}{|F| \times C_d})^{1/d}$.

[2] http://www.en.wikipedia.org/wiki/N-sphere

5.3.3 IO cost of a monochromatic RkNN query

The IO cost for a monochromatic RkNN query is the same as the IO cost of computing the influence zone Z_{k+1}. This is because the R-tree is traversed only during the construction of the influence zone (i.e., the containment phase does not access R-tree). Hence, IO cost of a monochromatic query is the IO cost of a range query with range set as $2r_{k+1} = 2(\frac{k+1}{|F| \times C_d})^{1/d}$.

5.3.4 IO cost of a bichromatic RkNN query

The cost of the pruning phase is the same as the cost of computing the influence zone Z_k which we have analysed earlier. The cost of the containment phase is the cost of accessing the users that lie within the influence zone which can be computed in a similar way. More specifically, only the users that lie within distance r_k (the radius of the influence zone) of q are accessed. Hence, the cost of the containment phase is the IO cost of the range query with range set to $r_k = 2(\frac{k}{|F| \times C_d})^{1/d}$.

5.4 Complexity Analysis

The complexity of Algorithm 2 for arbitrarily dimensionality d is exponential in d because the number of intersection points of m half-spaces in d dimensional space is $O(m^d)$. Nevertheless, our experimental results demonstrate that the performance of our algorithms is reasonable as compared to other approaches (for dimensionality up to 4). In this section, we show that the complexity of Algorithm 2 in two dimensional space can be reduced from $O(m^2)$ to $O(km)$ where m is the number of facilities (or bisectors) considered so far. In the rest of the paper, our discussion is based on two dimensional space unless specifically mentioned otherwise. Below, we define a few terms and notations and present a lemma that helps us in analysing the complexity.

5.4.1 Preliminaries

Valid intersection point. An intersection point that has a counter less than k is called a valid intersection point.

Left/right pruning intersection. Consider the example of Fig. 11 where two bisectors B_2 and B_3 intersect a bisector B_1 at points l_1 and r_2, respectively. The bisector B_2 prunes every point on B_1 that lies on the left side of l_1 (as shown with an arrow). For example, B_2 prunes the points l_2, r_2 and r_1. The intersection point l_1 is called a *left pruning* intersection of B_1 because it prunes every point on B_1 that lies on its left side. The bisector B_3 prunes every point on B_1 that lies on the right side of r_2 (e.g., the points l_2, l_1 and r_3). The intersection point r_2 is called a *right pruning* intersection of B_1. In Fig. 11, the right pruning intersection points are shown as r_i (black circles) and the left pruning intersection points are shown as l_i (the hollow circles). To keep Fig. 11 simple, we do not show the bisectors related to r_1, r_3 and l_2.

Note that the counter of any point p that lies on B_1 is at least equal to the number of left pruning intersections on its right side plus the number of right pruning intersections on its left side. For example, the counter of point l_2 is $1 + 2 = 3$ because it is pruned by l_1, r_1 and r_2. Lemma 12 shows that each existing bisector can have at most $2k$ valid intersection points.

Lemma 12 *For any bisector B_1, the number of valid intersection points*[3] *on it is at most $2k$.*

Proof Let the number of right pruning intersection points of B_1 be u. We denote the right pruning intersections of B_1 by $r_1, ..., r_u$ such that for any intersection r_i there are $i-1$ right pruning intersections on its left. For example, in Fig. 11, there are two right pruning intersections (r_1 and r_2) on the left side of r_3. For any right pruning intersection point r_i, its counter is at least equal to $i-1$ because r_i is pruned by at least $i-1$ right pruning intersections. Hence, only the intersections r_i for $0 < i \leq k$ can have counters less than k. This implies that the number of right pruning intersection points that are valid is at most k. Following similar arguments, it can be shown that at most k left pruning intersection points are valid intersections. Hence, the total number of valid intersection points on B_1 is at most $2k$. □

[3] The proof of this lemma assumes that each intersection point is unique, i.e., two bisectors do not intersect B_1 at the same point. However, the complexity analysis remains the same even in the absence of this assumption. This is because such intersection points can be merged and treated as one intersection point because their counters would be exactly the same.

Now, we analyse the complexity of Algorithm 2. For each line of Algorithm 2, we show that its complexity is at most $O(km)$.

5.4.2 Complexity of line 1: compute new intersection points and their counters

The number of new intersection points is $O(m)$ because each existing bisector intersects the new bisector $B_{f:q}$ at most once. To compute the counter of a new intersection point p, we count the number of existing bisectors that prune p. Hence, computing the counter of a new intersection point takes $O(m)$. Since there are $O(m)$ new intersection points, the complexity of computing the counters of these points is $O(m^2)$. Next, we show that the complexity can be reduced to $O(km)$.

Let p be an intersection point between $B_{f:q}$ and an existing bisector. If p lies outside the current influence zone (the unpruned polygon) then its counter is at least equal to k and p can be discarded for this reason. Hence, the counters of only the intersection points that lie inside the unpruned polygon are to be computed. We implement the whole procedure in two steps: 1) for each intersection point p, check whether p lies inside the unpruned polygon or not; 2) for each intersection point p that lies inside the unpruned polygon, compute its counter.

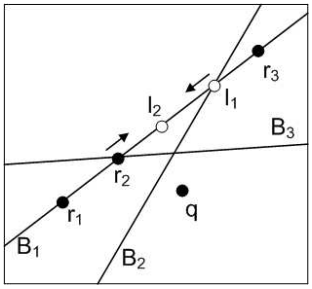


Fig. 11 Lemma 12

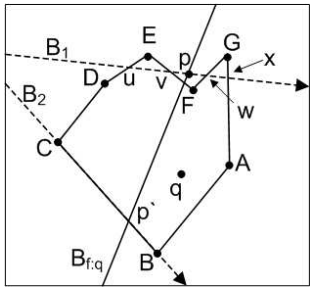


Fig. 12 Even-odd test

First we show that the step 1 can be implemented in $O(km)$ by using even-odd test [16] to determine if p lies inside the unpruned polygon or not. According to an even-odd test, a point p lies inside a polygon if and only if, for any ray starting from the point p, there is an odd number of crossings of this ray with the edges of the polygon. For example, in Fig. 12, point p lies outside the polygon $ABCDEFG$ and any ray starting from p intersects the edges of the polygon even number of times. For instance, the ray starting from p in the direction of x intersects the polygon at two points (w and x). Hence, p lies outside the polygon. Now, we show that for any intersection point p, we can conduct even-odd test in $O(k)$. Since we have at most $O(m)$ new intersection points, this ensures the overall complexity of $O(km)$.

Assume that p is an intersection point of $B_{f:q}$ and an existing bisector B_1 as shown in Fig. 12. Note that B_1 is an existing bisector and the algorithm maintains the existing valid intersection points of B_1. For example, the system maintains the intersection points u, v, w and x. To determine the number of intersections of the ray starting from p with the boundary of the unpruned polygon, we simply count the number of valid intersection points of B_1 that lie on the right side of p and lie on the boundary of the unpruned polygon. In Fig. 12, such intersection points are w and x. Since B_1 has at most $2k$ intersection points (Lemma 12), we need to choose the boundary points among at most $2k$ points. Later in this section, we show that we can determine whether an intersection point lies on the boundary or not in constant time by using its counter (Lemma 15). Hence, determining the intersection points that lie on the right side of p and lie on the boundary of the unpruned polygon takes $O(k)$.

As a special case, if the intersection point p lies on the boundary of the unpruned polygon we assume as if it lies inside the unpruned polygon. In Fig. 12, the intersection point p' between $B_{f:q}$ and B_2 lies on the boundary of the unpruned polygon. Note that any bisector B_2 can contribute at most $O(k)$ edges to the unpruned polygon (a direct implication of Lemma 12). Hence, to check whether p' lies on an edge of the polygon, we check if it intersects with any edge of the polygon contributed by B_2. It takes $O(k)$.

Now, we show that step 2 can be done in $O(km)$. As inferred from Lemma 12, the number of new intersection points that lie in the unpruned polygon is at most $O(k)$. Computing the counter of one intersection point takes $O(m)$. Hence, the total complexity of step 2 is $O(km)$.

5.4.3 Complexity of line 2: update the counters of existing intersection points

Lemma 12 shows that each existing bisector can have at most $O(k)$ valid intersection points. Since m is the number of existing bisectors, the total number of valid intersection points is $O(km)$. Recall that, to update the counter of an intersection point p, we only need to check whether it is pruned by $B_{f:q}$ or not where f is the new facility being considered. This can be done in constant time. Hence, the complexity of line 2 of Algorithm 2 is $O(km)$.

5.4.4 Complexity of line 3: discard intersection points with counters at least equal to k

We scan the list of intersection points and remove any intersection point that has a counter at least equal to k. Clearly, the complexity is $O(km)$.

5.4.5 Complexity of line 4: find the convex vertices

We show that we only need to scan the list of the intersection points once to determine the convex vertices. Since the total number of intersection points is $O(km)$, the complexity of this step is $O(km)$. Lemma 13 is the key to obtain the required complexity.

Lemma 13 *Among the intersection points that do not lie on the boundary of the data universe, only the intersection points with counters equal to $k-1$ can be the convex vertices.*

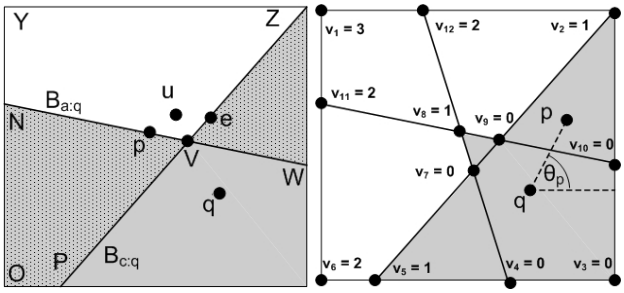
Proof Any intersection that has a counter greater than $k-1$ is pruned by at least k objects hence cannot be on the boundary of the influence zone (hence, cannot be a convex vertex). Now, we show that the intersections that have counters less than $k-1$ cannot be the convex vertices.

Consider the example of Fig. 13(a) where a vertex V has been shown which is the intersection point of two bisectors $B_{a:q}$ and $B_{c:q}$. Suppose that the counter of the vertex V is n. Now, imagine a point p that lies on the line VN and is infinitely close to the vertex V. Clearly, the point p is pruned by at most $n+1$ bisectors[4]. This is because it is pruned by n bisectors that prune V and the bisector $B_{c:q}$. Following the similar argument, we can say that any point e that lies on the line VZ and is infinitely close to V has a counter at most $n+1$. The counter of any point u that lies in the polygon $VNYZ$ (white area) and is infinitely close to V is at least $n+2$ (it is pruned by $B_{c:q}$ and $B_{a:q}$ in addition to all the bisectors that prune V).

If the counter n of the vertex V is less than or equal to $k-2$, then the line VN has at least one point p that has counter at most $k-1$ (i.e., $n+1$ as shown above). Hence, the line VN has at least one point p that lies in the influence zone. Similarly, the line VZ has at least one point e that lies in the influence zone. Clearly, the angle eVp is at least 180°. By definition of a convex hull, no internal angle of a convex hull can be greater

[4] In this proof, we assume that only two bisectors pass through the intersection point V. For the special case, when more than two bisectors pass through a vertex V, we may choose to treat V as a convex vertex. Note that this does not affect the correctness of the algorithm because checking containment in a vertex that is not a convex vertex does not affect the correctness.

than 180^0. Hence, the vertex V is not a convex vertex if its counter is less than or equal to $k-2$. □



(a) Lemma 13 (b) Counters

Fig. 13 Finding convex vertices

In Fig. 13(b), we revisit the example of Fig. 6. The vertices v_7 and v_9 do not lie on the boundary of the data universe and have counters less than $k-1$ (where $k=2$). Hence, they are not the convex vertices. Among the points that lie on the boundary of the data universe and have counters less than k, only the two extreme points for each boundary line can be the convex vertices. For example, in Fig. 6, the lower horizontal boundary line contains 4 vertices (v_3, v_4, v_5 and v_6). The vertex v_6 has counter not less than k and can be ignored. Among the remaining vertices, we consider the extreme vertices (v_3 and v_5) as the convex vertices. Following the above strategy, the convex vertices in Fig. 6 are v_3, v_2, v_8 and v_5.

The above discussion shows that the convex vertices can be found by scanning the list of intersection points once. Hence, the cost of finding the convex vertices is $O(km)$.

5.4.6 Complexity of line 5: compute the unpruned polygon

For any point p, we use θ_p to denote the angle formed by the horizontal line passing through q and the line segment pq (see Fig. 13(b)). Recall that line 1 adds $O(k)$ new intersection points. These intersection points are always inserted in sorted order of θ_p and this takes $O(k \cdot log\ m)$ because $O(k)$ points are inserted and each insertion takes $O(log\ m)$ (the maximum number of existing intersection points is $O(m^2)$). Next, we show that the unpruned polygon can be computed in $O(km)$ if all the intersection points are sorted according to θ_p.

Lemma 14 *The unpruned polygon is always a star-shaped polygon and q is its kernel point.*

Proof Consider that $F' \subset F$ is a set of facilities that consist of only the facilities that have been considered so far. Clearly, the current unpruned polygon is the influence zone of q for the data set F'. Hence, Lemma 6 can be immediately applied to prove that the unpruned polygon is always a star-shaped polygon. □

Since the unpruned polygon P is a star-shaped polygon and q is its kernel point, every point on its boundary is visible from q [20]. This implies that θ_p is unique for every point p on the boundary of P, i.e., $\theta_p \neq \theta_{p'}$ for any two points p and p' that lie on the boundary of P. Hence, given a list of points that lie on the boundary of P, we can construct the polygon P by connecting the points in sorted order of the angles they make with q. Finally, we need to determine the intersection points that lie on the boundary of the unpruned polygon.

Lemma 15 *Among the intersection points that do not lie on the boundary of the data universe, any intersection point V that has a counter less than $k-2$ does not lie on the boundary of the unpruned polygon. Secondly, any intersection point V that has a counter equal to $k-2$ lies on the boundary of the unpruned polygon.*

Proof Consider the vertex V as shown in Fig. 13(a) and assume that it has a counter equal to n. The counter of any point u that lies infinitely close to V and lies in the white area is $n+2$. This is because it is pruned by the n bisectors that prune V and the bisectors $B_{a:q}$ and $B_{c:q}$. Note that any point u that is infinitely close to V can be pruned by *at most* $n+2$ bisectors (n bisectors that prune V and $B_{a:q}$ and $B_{c:q}$). If the counter of V is less than $k-2$ then the counter of any such point u is always smaller than k. Hence, u is a point inside the unpruned polygon. Since every u that lies infinitely close to V (in any direction) is a point of the unpruned polygon, V does not lie on the boundary of the unpruned polygon.

Now, we prove the second part of the lemma. Assume that the counter of V is equal to $k-2$. Clearly, the counter of u is k. Hence, u lies outside the unpruned polygon. Since u is infinitely close to V, V is a point on the boundary of the unpruned polygon. □

Lemma 15 along with Lemma 13 show that the boundary of the unpruned polygon consists of only the valid intersection points that either lie on the boundary of the data universe or have counters equal to $k-1$ or $k-2$. Hence, the list containing all intersection points sorted according to θ_p is scanned and the points that do not lie on the boundary of the polygon are ignored. Remaining points are connected in sorted order of θ_p to obtain the unpruned polygon. For example, in Fig. 13(b), the unpruned polygon is obtained by connecting the vertices in counter clock wise order, i.e., v_{10}, v_2, v_9, v_8, v_7, v_5, v_4 and v_3 in this order.

6 Handling data updates

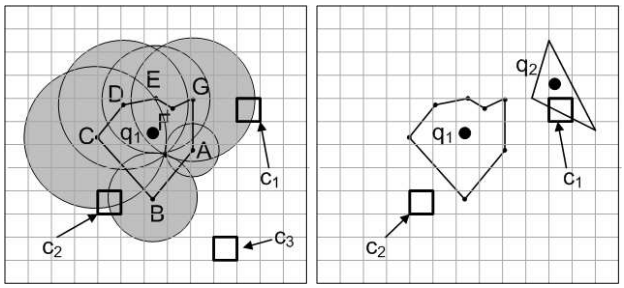
In this section, we present techniques to efficiently update the influence zone for two dimensional data sets for the case when the facilities issue updates, e.g., facilities are added or deleted in/from the data set or facilities stop/resume providing the service. Such data updates are frequent in many real world applications. For instance, a facility must be ignored (treated as a deletion) if it is already providing service to its maximum capacity and is unable to provide the service to more users. If the load on the facility reduces and it can accommodate new users, the server must start considering it for the influence zone computation and it is treated as an insertion.

Consider the example of a restaurant that sends promotional SMS to the people in its influence zone. Note that its influence zone may change when one or more restaurants close or open due to different business timings or when a restaurant stops/resumes providing the service constrained by its seating capacity. Similarly, consider the example of parking space problem where the server informs the cars about their nearby available parking slots. A parking space may be considered as a deletion if it is occupied. When the parking space becomes available, it may be treated as an insertion.

It may seem that a materialized approach that is based on *all* of the facilities is feasible for handling the case when facilities stop/resume providing the service. However, a careful consideration reveals that the materialized approach has several serious limitations. For instance, this approach suffers from the problems mentioned in Section 5.1. In addition, it requires computing and storing all possible order k Voronoi diagrams (or influence zones) for every possible combination of facilities. This is because an order k Voronoi diagram is different for every unique combination of facilities. Hence, a huge number of order k Voronoi diagrams (or influence zones) will have to be pre-computed. Next, we present a technique to efficiently update the influence zone that does not require pre-processing.

6.1 Solution overview

Assume a set of facilities and a set of queries where each query may have a different value of k. Note that a single update (insertion or deletion) may or may not affect the influence zone of a particular query q. Hence, it is important to identify the queries that are affected by an update. To enable us to quickly identify the affected queries, we define *impact region* of a query. The impact region of a query q is the area covered by C_v for every convex vertex v of the influence zone of q. Fig. 14(a) shows an example where the influence zone of q_1 is the polygon $ABCDEFG$ and the impact region is shown shaded.



(a) Impact region of q_1 (b) An update in c_2 does not affect the influence zone of q_2

Fig. 14 Finding the queries affected by an update

As inferred from Lemmas 6 and 7, a facility f affects the influence zone of a query q_1 if and only if f lies in C_v for at least one convex vertex v of the influence zone. Hence, a query is affected by a facility if and only if f lies in the impact region of the query. In the example of Fig. 14(a), a facility that lies outside of the shaded area does not affect the influence zone. Hence, the update issued by every such facility can be safely ignored.

To quickly identify the queries that are affected by an update, we index the impact regions of all the queries by a grid data structure. Each cell c of the grid has a list called *qList* which is denoted as $c.qList$. The *qList* of a cell c contains every query q for which the impact region of q overlaps or contains the cell c. In Fig. 14(a), $c_1.qList$ and $c_2.qList$ contain q_1. On the other hand, $c_3.qList$ does not contain q_1. Note that an update in c_3 cannot affect the influence zone of q_1. On the other hand, an update in c_1 or c_2 may or may not affect the influence zone of q_1. For instance, if a facility f is inserted in c_1 and lies inside the impact region of q_1 then it affects the influence zone. Otherwise, if f lies in c_1 but is outside the impact region of q_1 then it does not affect the influence zone. Hence, for each facility update in c_1, we specifically check if f lies inside the impact region of q_1 or not.

The *qList* of a cell c helps in pruning the queries that are not affected by an update of a facility in cell c. Consider the example of Fig. 14(b) where, in addition to q_1, another query q_2 along with its influence zone is shown. To keep the illustration simple, the impact regions of the queries are not shown. However, we remark that the impact region of q_2 overlaps with c_1 and does not overlap with c_2. Hence, $c_1.qList$ contains q_1 and q_2 whereas $c_2.qList$ contains only q_1. If a facility f is inserted or deleted in c_1 then the influence zones of the queries in $c_1.qList$ (i.e., q_1 and q_2) may be affected. If f is inserted or deleted in c_2 then only q_1 may be affected because $c_2.qList$ contains only q_1.

Algorithm 3 Handling update

Input: an update issued by f
Output: updated influence zone of every query in the system
1: identify the cell c that contains f
2: **for** each query q in $c.qList$ **do**
3: **if** f lies inside the impact region of q **then**
4: **if** f is a new facility or has resumed service **then**
5: update influence zone of q as stated in Section 6.2
6: **else** /* f `is deleted or has stopped its service` */
7: update influence zone of q as stated in Section 6.3

Algorithm 3 presents the details. For an update issued by a facility f, we first identify the cell c of the grid that contains f (line 1). The update of f may affect only the queries in $c.qList$. For each such query q, we specifically check if f lies inside its impact region or not (line 3). If f lies inside the impact region of q, we update the influence zone accordingly (lines 4 to line 7). Otherwise, the influence zone of q is not required to be updated.

Note that when the influence zone of a query q is updated, the $qList$ of several cells may have changed and must be updated. We add q in every cell c of the grid that overlaps or is contained by the new impact region of q. We also delete q from every cell c' that is not overlapped by the impact region of q but previously contained q in its $qList$. To efficiently do this, we use *conceptual grid tree* which we introduced in [10] and then further studied in [19,17]. For details, please see one of [10,19,17].

In Section 6.2, we show how to update the influence zone of a query q if f is a new facility or has resumed its service (e.g., the update is an insertion). In Section 6.3, we show the procedure to update the influence zone of q when a facility f is deleted or stops providing the service (e.g., the update is a deletion).

6.2 Handling an insertion

As stated earlier, we use qList to identify every query that contains f in its impact region. We update the influence zone of each of such query by calling Algorithm 2. As shown earlier, the complexity of Algorithm 2 is $O(km)$. Next, we present few geometric observations that although do not reduce the complexity but help to give more insight into the properties of the problem.

Recall that, at line 1 of Algorithm 2, we compute new intersection points between $B_{f:q}$ and all existing bisectors and then compute their counters. Next, we present few geometric observations that show that we do not need to consider the intersection points of the new bisector $B_{f:q}$ with *all* of the existing bisectors.

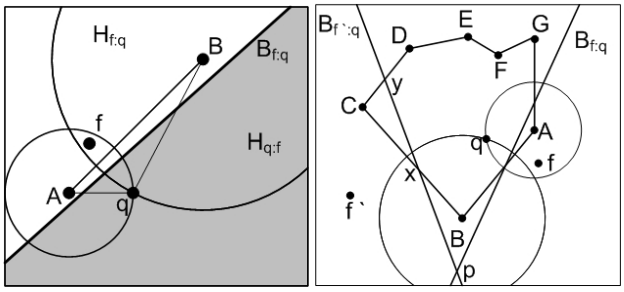
Lemma 16 *Given a line segment AB and a facility f, the bisector $B_{f:q}$ intersects the line segment AB if and only if exactly one of C_A or C_B contains f, i.e., if both of C_A and C_B contain f or none of C_A and C_B contain f then $B_{f:q}$ does not intersect AB.*

Proof First, we show that $B_{f:q}$ intersects AB only if exactly one of C_A or C_B contains f. We prove this by showing that $B_{f:q}$ does not intersect AB if either both of C_A and C_B contain f or none of C_A or C_B contains f.

Consider the example of Fig. 15(a) where the line segment AB and the circles C_A and C_B are shown. Recall that the bisector $B_{f:q}$ divides the space in two half planes. $H_{f:q}$ denotes the plane that contains f (the white area) and $H_{q:f}$ denotes the plane that contains q (the shaded area). If both C_A and C_B contain f then it means that the bisector $B_{f:q}$ prunes both A and B, i.e., both A and B lie in $H_{f:q}$ (as shown in Fig. 15(a)). Since $B_{f:q}$ is a line, the whole line segment AB lies in the plane $H_{f:q}$ which implies that $B_{f:q}$ does not intersect AB.

If none of C_A or C_B contains f then the bisector $B_{f:q}$ does not prune any of A or B. In other words, both A and B lie in $H_{q:f}$. Since $B_{f:q}$ is a line, the whole line segment AB lies in the plane $H_{q:f}$. This implies that $B_{f:q}$ does not intersect AB.

Now, we show that $B_{f:q}$ intersects AB if exactly one of C_A or C_B contains f. Without loss of generality, assume that C_A contains f and C_B does not contain f. This means that the bisector $B_{f:q}$ prunes A and does not prune B. In other words, A lies in $H_{f:q}$ and B lies in $H_{q:f}$. Since $B_{f:q}$ is a line, the line segment AB intersects $B_{f:q}$. □



(a) Lemma 16 (b) Lemma 17

Fig. 15

The above lemma shows that we may not need to compute the intersection of $B_{f:q}$ with all of the existing bisectors. Let A and B be two end points of a bisector within the influence zone. We only need to compute the intersection of $B_{f:q}$ with the bisector if exactly one of C_A or C_B contains f. However, we first need to efficiently identify such bisectors. Before we show how to identify such bisectors, we define few terms and notations.

Let v be a vertex such that C_v contains f. We call such a vertex v a container vertex. In Fig. 15(b), A is a container vertex because C_A contains f. Any edge XY of the influence zone is called a container edge if *at least* one of C_X or C_Y contains f. Any edge that is not a container edge is called a non-container edge. In Fig. 15(b), AB and AG are the only container edges. The next lemma shows that we only need to consider the intersection of $B_{f:q}$ with the existing bisectors that intersect with a container edge.

Lemma 17 *Let $B_{f':q}$ be a bisector that does not intersect with any of the container edges of the influence zone. The intersection point of $B_{f':q}$ and $B_{f:q}$ lies outside the influence zone, i.e., the intersection has a counter at least equal to k and can be ignored for this reason.*

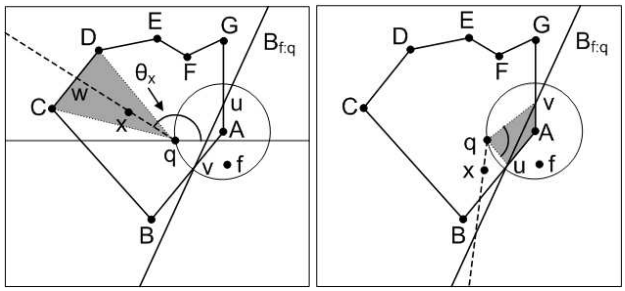
Proof Consider the example of Fig. 15(b) where a polygon $ABCDEFG$ is shown. A is the only container vertex of the polygon. The bisector $B_{f':q}$ does not intersect any of the container edges AB or AG. Without loss of generality, assume that the two end points of the bisector $B_{f':q}$ that lie within the influence zone are x and y (see Fig. 15(b)). We prove the lemma by showing that the bisector $B_{f:q}$ does not intersect the line segment xy. We show that both C_x and C_y do not contain f which implies (see Lemma 16) that $B_{f:q}$ does not intersect xy.

We prove that C_x does not contain f and the proof for C_y is similar. As inferred by Lemma 4, C_x is contained by $C_B \cup C_C$. Since BC is a non-container edge, both C_B and C_C do not contain f. This implies that C_x does not contain f because $C_x \subseteq C_B \cup C_C$. □

As inferred from Lemma 17, we only need to check the intersection of $B_{f:q}$ with the bisectors that intersect with any of the container edges. Next issue is to determine the container edges efficiently. Recall that, in our grid structure, we maintain $q.vList$ for each cell c that contains the list of the vertices that overlap or contain the cell c. If a facility f lies in the cell c, we use $q.vList$ and can identify the vertices of the influence zone of q that contain f. These vertices are the container vertices and the related container edges can be easily determined.

Recall that line 2 of Algorithm 2 requires updating the counters of all existing intersection points. As stated earlier, we increment the counter of an intersection point p if and only if $B_{f:q}$ prunes p. The number of existing intersection points is $O(km)$. Next, we show that we may not need to check whether $B_{f:q}$ prunes p for all of the intersection points.

First, we define few terms and notations. Let x be a point inside the influence zone. *Beam* of x is a line starting from q that passes through the point x. Fig. 16(a) shows the beam of a point x in broken line.



(a) Lemma 18 (b) Optimization

Fig. 16

Lemma 18 *An intersection point x is not pruned by a bisector $B_{f:q}$ if the beam of x does not intersect with any container edge of the influence zone.*

Proof Consider the example of Fig. 16(a) where A is the only container vertex. Recall that a point x is pruned by a bisector $B_{f:q}$ if and only if C_x contains f. Without loss of generality, assume that the beam of x intersects the influence zone at a non-container edge CD at a point w (see Fig. 16(a)). From Lemma 2, C_x is contained by C_w. From Lemma 4, C_w is contained by $C_C \cup C_D$. The object f is not contained in $C_C \cup C_D$ because CD is a non-container edge. Hence, f is not contained by C_x which implies that $B_{f:q}$ does not prune x. □

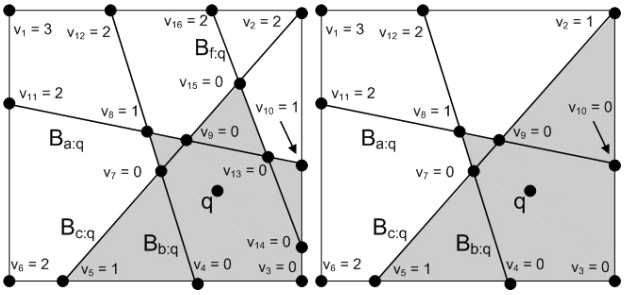
From above lemma, we know that we only need to update the counters of an intersection point if its beam intersects a container edge. Next issue is to efficiently determine the intersection points for which their beams intersect with a container edge. Recall that, for any point x, θ_x is the angle between line qx and the horizontal line passing through q (see Fig. 16(a)). For the edge CD in the Fig. 16(a), note that the beam of any intersection point x intersects CD if and only if θ_x lies between the angle range θ_D and θ_C (i.e., x lies in the shaded area). Hence, we can use the θ_p of an intersection point p to test if its beam intersects an edge or not.

We further improve the above observation. Consider the example of Fig. 16(b), where the intersection point x is shown and its beam intersects a container edge AB. Although the beam of x intersects a container edge, x is not pruned by $B_{f:q}$ as shown in Fig. 16(b). Assume that the bisector $B_{f:q}$ intersects the influence zone at two points u and v as shown in Fig. 16(b). An intersection point x can be pruned by $B_{f:q}$ only if θ_x is greater than θ_u and is smaller than θ_v (i.e., x lies in the shaded area of Fig. 16(b)). The proof is straight forward and is omitted.

We can quickly identify the intersection points that lie within the shaded area as follows. Recall that we keep the list of intersection points sorted in order of their θ_p. We do a binary search on this list and obtain the first intersection point p that has θ_p just greater than θ_u. Then, the list is scanned in sorted order until the next intersection point p' has $\theta_{p'}$ greater than θ_v. Let n be the number of intersection points that lie in the shaded area of Fig. 16(b), the above procedure can find all such intersection points in $O(n+log\ m)$. Hence, the complexity of updating the counters of existing intersection points is $O(n + log\ m)$ where n is at most equal to $O(km)$ (the number of all existing intersection points).

6.3 Handling a deletion

If the deleted facility f lies inside the impact region of q then it means that the facility f contributes a bisector to the influence zone. Assume that the influence zone was determined by considering m facilities. When f is deleted, we create the new unpruned polygon P by considering the bisectors of remaining $m-1$ facilities. During the creation of the new unpruned polygon P, we use the following optimizations to improve the efficiency.



(a) Before deleting $B_{f:q}$ (b) After deleting $B_{f:q}$

Fig. 17 Handling a deletion

1. The counter of any intersection point p that is not pruned by $B_{f:q}$ remains unaffected. Hence, the counters of all such intersection points are not required to be recomputed. This also implies that the part of the influence zone that lies in $H_{q:f}$ remains unaffected.

Consider the example of Fig. 17(a) that shows the influence zone ($k = 2$), intersection points and their counters before a facility f and its corresponding bisector $B_{f:q}$ is deleted. The influence zone is shown shaded and it contains the intersection points that have counters less than k. Fig. 17(b) shows the new unpruned polygon P, intersection points and their counters after f is deleted. Note that the counters of all the intersection points that are not pruned by $B_{f:q}$ (i.e., the intersection points on the left side of $B_{f:q}$) remain unchanged. Also, the part of the influence zone that lies on the left side of $B_{f:q}$ remains unaffected.

2. The counter of any existing intersection point that is pruned by $B_{f:q}$ is decremented by 1. Hence, the counter of such intersection point is not needed to be computed from scratch. The counter of any new intersection point that is pruned by $B_{f:q}$ is recomputed.

In the example of Fig. 17(a), there is only one valid intersection point v_{10} that is pruned by $B_{f:q}$. Its counter is decremented by one after the deletion. Note that the intersection point v_2 had a counter equal to $k = 2$ before f was deleted. Hence, v_2 was not maintained before the deletion of f. The counter of such intersection point is needed to be recomputed.

Note that the new unpruned polygon P is always larger than the previous influence zone. Hence, there may be a facility f' that affects the new unpruned polygon P but was not considered before. To identify all such facilities, we check if there exists a new facility f' that lies in any C_v for any convex vertex of the new unpruned polygon. We can do this by calling Algorithm 1 with two small changes. Firstly, at line 1, the influence zone Z_k is initialized to the new unpruned polygon P instead of initializing it to the whole data universe. Secondly, the algorithm ignores any facility f that had already been considered to construct the influence zone.

Finally, we present another minor optimization. Note that at line 5 of Algorithm 1, we check if an entry e of R-tree is contained by C_v for every convex vertex of the influence zone. However, note that there are some convex vertices of the unpruned polygon P (see Fig. 17(b)) that existed in previous influence zone (see Fig. 17(a)). For example, the convex vertex v_8 is a convex vertex of the previous influence zone as well as the new unpruned polygon. Hence, we do not need to consider v_8 at line 5 of Algorithm 1. This is because if there was a facility in the circle of such convex vertex, that would have been considered before. Hence, the convex vertices that existed in the influence zone before the deletion can be ignored at line 5 of Algorithm 1.

7 Experiments

In Section 7.1, we evaluate the performance of our algorithms for snapshot RkNN queries. Since computation of the influence zone is a sub-task of our snapshot RkNN algorithm, we evaluate the cost of computing influence zone while evaluating the performance of RkNN algorithms. In Section 7.2, we evaluate the performance of our algorithm for continuous monitoring of RkNN queries.

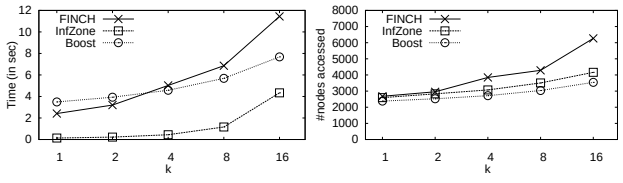
7.1 Snapshot RkNN queries

We use both synthetic and real datasets. Each synthetic dataset consists of 50000, 100000, 150000 or 200000 points following either Uniform or Normal distribution. The real dataset consists of 175,812 extracted locations in North America[5] and we randomly divide these points into two sets of almost equal sizes. One of the sets corresponds to the set of facilities and the other to the set of users. We use the two real datasets to evaluate the performance unless mentioned otherwise. We vary k from 1 to 16 and the default value is 8. From the set of facilities, we randomly choose 500 points as the query points. The experiment results correspond to the total cost of processing these 500 queries.

7.1.1 Monochromatic RkNN queries

We compare our algorithm with two best known algorithms FINCH [41] and Boost [13]. Boost is an optimized version of the algorithm presented in [1] and uses more powerful and cheaper pruning techniques. The page size is set to 4096 bytes. Following the experimental settings used in [41] for FINCH, the buffer size for FINCH is set to 10 pages which uses random eviction strategy. We remark that our algorithm and Boost both do not need any buffer and the cost remains unaffected even if no buffer is used.

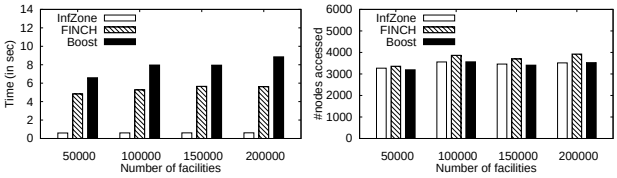
In Fig. 18, we vary the value of k and study the effect on the algorithms. As shown in Fig. 18(a), our algorithm outperforms the other two algorithms in terms of CPU time consumption. FINCH performs better than Boost for smaller values of k. CPU cost of our algorithm is lower mainly because we use efficient checks to prune the entries of the R-tree and because we do not need to compute the convex hull (in contrast to FINCH that computes a convex polygon to approximate the unpruned area). Fig. 18(b) shows that the IO cost of Boost is the lowest and the cost of our algorithm is reasonably close.



(a) CPU time (b) Nodes accesses

Fig. 18 Effect of k (monochromatic queries)

Fig. 19 studies the effect of number of facilities. Locations of the facilities in each data set follow Normal distribution. Fig. 19(a) shows that the computation cost of each algorithm slightly increases with the increase in the number of facilities. However, our algorithm performs significantly better and scales well. Fig. 19(b) shows that the number of facilities do not significantly affect the IO cost of the algorithms. Also, the IO cost of the three algorithms is reasonably close to each other.



(a) CPU time (b) Nodes accesses

Fig. 19 Effect of number of facilities (monochromatic queries)

7.1.2 Bichromatic RkNN queries

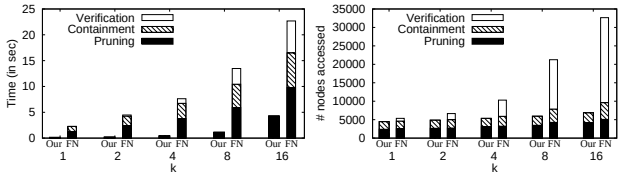
Boost [13] is designed for monochromatic queries and uses self pruning and mutual pruning techniques. Unfortunately, it is not trivial to extend it for efficiently processing bichromatic queries. This is because self pruning cannot be applied and also because Boost uses a special order for R-tree traversal which cannot be extended for bichromatic queries that uses two R-trees. We tried a straight forward extension of Boost and it performed quite poor. Therefore, we compare our algorithm only with FINCH [41] which is the best known existing algorithm for two dimensional bichromatic RkNN queries.

As stated in Section 2.2, FINCH has three phases namely pruning, containment and verification. Our algorithm has only pruning and containment phases. We show the CPU and IO cost of each phase for both of the algorithms. Experiment results demonstrate that our algorithm outperforms FINCH in terms of both CPU time and the number of nodes accessed. FINCH is denoted as FN in the experiment figures.

Fig. 20 studies the effect of k on the cost of bichromatic RkNN queries. The CPU time taken by containment phase of our algorithm is much smaller as compared to FINCH. This is mainly because i) the unpruned area of our algorithm is smaller and ii) we use efficient containment checking to prune the entries and the objects. IO cost of the containment phase is also smaller for our algorithm because the unpruned area of our algorithm is smaller. Our algorithm does not require the verification. On the other hand, FINCH consumes significant amount of CPU time and IOs in the verification phase.

Fig. 21 studies the effect of the number of the users on both of algorithms. The set of facilities corresponds to the real dataset and the locations of the users follow normal distribution. Our algorithm scales much better. On the other hand, the cost of FINCH degrades with

[5] http://www.cs.fsu.edu/ lifeifei/SpatialDataset.htm

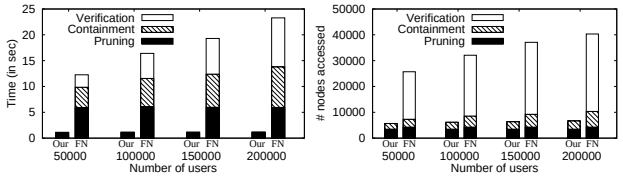


(a) CPU time (b) Nodes accesses

Fig. 20 Effect of k (bichromatic queries)

the increase in the number of users because a larger number of users are within the unpruned area and require verification.

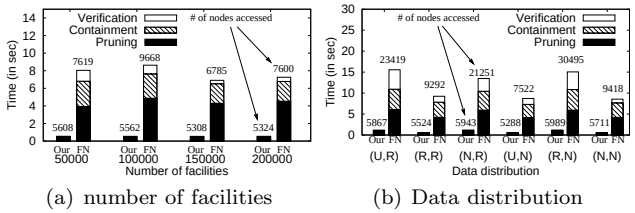
In Fig. 22(a), we study the effect of the number of the facilities. The set of the users correspond to the real dataset and the locations of the facilities follow normal distribution. Both of the algorithms are not significantly affected by the increase in the number of the facilities and our algorithm performs significantly better than FINCH.



(a) CPU time (b) Nodes accesses

Fig. 21 Effect of number of users

Fig. 22(b) studies the effect of the data distribution on both of the algorithms. The data distributions of the facilities and the users are shown in the form ($Dist_1$,$Dist_2$) where $Dist_1$ and $Dist_2$ correspond to the data distribution of the facilities and the users, respectively. U, R and N correspond to Uniform, Real and Normal distributions, respectively. For example, (U,R) corresponds to the case where the facilities follow uniform distribution and the users correspond to the real dataset. Each dataset contains around 88,000 objects. Our algorithm outperforms FINCH both in terms of CPU time and the number of nodes accessed for all of the data distributions.



(a) number of facilities (b) Data distribution

Fig. 22 Effect of data size and distribution

Fig. 23 studies the effect of the buffer size on both of the algorithms. As the pruning and the containment phases do not visit a node twice, our algorithm is not affected by the buffer size. FINCH issues multiple range queries to verify the candidate objects. For this reason, the cost of its verification phase depends on the buffer size. Note that FINCH performs worse than our algorithm even when it uses large buffer size. Number of nodes accessed by FINCH is around 194,000 and 61,000 when the buffer size is 2 and 5, respectively.

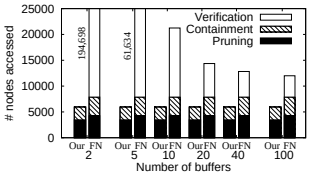
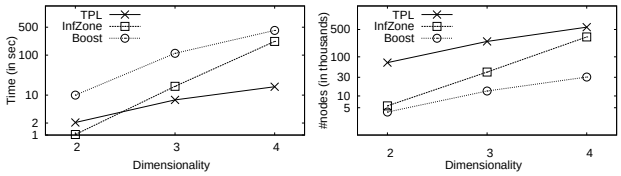


Fig. 23 Effect of buffer size

7.1.3 RkNN queries in higher dimensionality

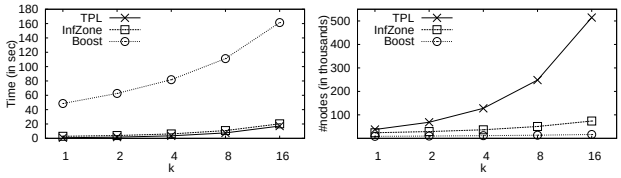
In this section, we evaluate the performance of our algorithm for multidimensional data sets. We compare our algorithm with TPL [36] and Boost [13]. These two are the best known algorithms that are applicable to arbitrary dimensionality. We remark that, in two dimensional space, FINCH was shown [41] to be superior to TPL. However, FINCH [41] is designed only for two dimensional data and, for this reason, is not considered as a competitor in this section. The points in each data set follow Normal distribution and the number of points (users/facilites) in the default data sets is 100,000. The default value of k is 8.



(a) CPU time (b) Nodes accesses

Fig. 24 Effect of dimensionality (monochromatic queries)

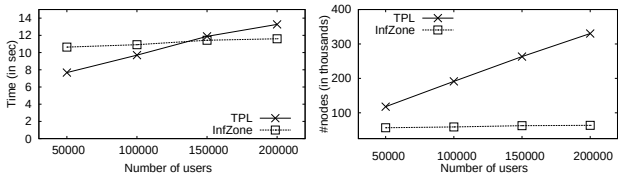
In Fig. 24, we process monochromatic RkNN queries on 2d, 3d and 4d data sets. The performance of each algorithm is significantly affected as the dimensionality increases. However, the performance of our algorithm deteriorates more seriously. This is because the geometry of the influence zone becomes significantly complex in higher dimensionality. However, it is worth noting that the CPU cost of our algorithm is still lower than the CPU cost of Boost (see Fig. 24(a)). On the other hand, Fig. 24(b) shows that the IO cost of our algorithm is lower than the IO cost of TPL when the dimensionality is at most 4.



(a) CPU time (b) Nodes accesses

Fig. 25 Effect of k (3d monochromatic queries)

Fig. 25 shows the effect of k on each of the three algorithms (on $3d$ data sets). Fig. 25(a) shows that the computational cost of our algorithm is significantly lower than the cost of Boost and is quite close to the cost of TPL. Fig. 25(b) shows that the IO cost of our algorithm is reasonably close to the IO cost of Boost and is signifcantly lower than the IO cost of TPL. Note that Boost performs poor in terms of CPU cost and TPL performs poor in terms of IO cost. In contrast, the IO and CPU cost of our algorithm is reasonably low.



(a) CPU time (b) Nodes accesses

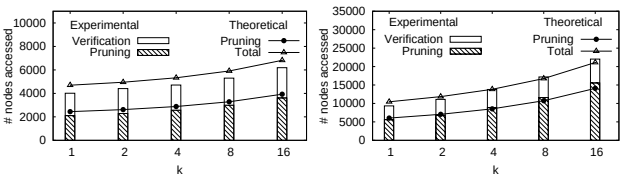
Fig. 26 Effect of number of users ($3d$ bichromatic queries)

Next, we evaluate the performance of our algorithm for bichromatic RkNN queries. As discussed in Section 7.1.2, Boost cannot be efficiently extended for bichromatic queries. For this reason, we compare our algorithm only with TPL. The experimental results for varying dimensionality and k are similar to the results presented in Fig. 24 and Fig. 25. Therefore, we omit the results for varying dimensionality and k.

Fig. 26 shows the effect of number of users on both of the algorithms (on $3d$ data sets). As the number of users increases, the cost of TPL increases significantly. This is because TPL needs to verify more objects. On the other hand, the cost of our algorithm is not significantly affected. This is because our algorithm does not need verification. We omit the results for varying number of facilities because both of the algorithms are not significantly affected by the change in the number of facilities (similar to the results in Fig. 22(a)).

7.1.4 Verification of theoretical analysis

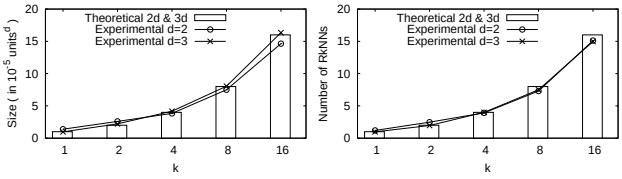
In this section, we evaluate our theoretical analysis presented in Section 5. In all the experiments, we run bichromatic RkNN queries on uniform data sets consisting of $100,000$ facilities and the same number of users.



(a) 2d data set (b) 3d data set

Fig. 27 Verification of theoretical analysis (IO cost)

In Fig. 27, we compare the experimental value of total number of nodes accessed with the theoretical value. Fig. 27(a) and Fig. 27(b) show the results for two and three dimensional data sets, respectively. Recall that the pruning phase of our algorithm corresponds to the computation of the influence zone. Fig. 27 shows the accuracy of our theoretical analysis for the IO cost of computing the influence zone and the total cost of our RkNN algorithm.



(a) Size of influence zone (b) Number of RkNNs

Fig. 28 Theoretical Analysis

In Fig. 28(a) and Fig. 28(b), we vary k and verify our theoretical analysis of the area/volume of the influence zone and the results size of RkNN queries, respectively. As mentioned in Section 5, our theoretical analysis of the area/volume of the influence zone and the result size does not depend on the dimensionality. Fig. 28 shows the theoretical values and the experimental values on $2d$ and $3d$ data sets. The experimental results verify the theoretical analysis.

7.2 Continuous Monitoring of RkNN

As mentioned earlier, the problem addressed by the influence zone based algorithm is a special case of the continuous RkNN queries. Hence, it is not fair to use the existing best known algorithms without making any obvious changes that improve the performance. As stated earlier in Section 2.2, Lazy Updates [10] is the best known algorithm for continuous monitoring of RkNN queries (even for this special case, we find that it outperforms other algorithms after necessary changes are made to all the existing algorithms). Hence, we compare our algorithm with Lazy Updates.

To conduct a fair evaluation, we set the size of the *safe region* for the Lazy Updates algorithm to zero. This is because the facilities do not move and the safe regions will not be useful in this case. We tested different possible sizes of the safe region and confirmed that this is the best possible setting for Lazy Updates for this special case of the continuous RkNN query.

Our experiment settings are similar to the settings used in [10] by Lazy Updates. More specifically, we use Brinkhoff generator [5] to generate the users moving on the road map of Texas (data universe is approximately 1000Km×1000Km). The facilities are randomly generated points in the same data universe. Table 2 shows

Table 2 System Parameters

Parameter	Range
Number of users (×1000)	40, 60, 80, **100**, 120
Number of facilities (×1000)	40, 60, 80, **100**, 120
Number of queries	100, 300, **500**, 700, 1000
k	1, 2, 4, **8**, 16
Speed of objects (users) in km/hr	40, 60, **80**, 100, 120
Mobility of objects (users) in %	5, 20, 40, 60, **80**, 100

the parameters used in our experiments and the default values are shown in bold.

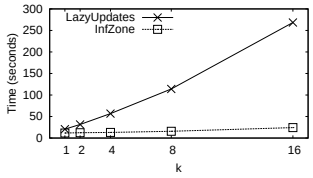
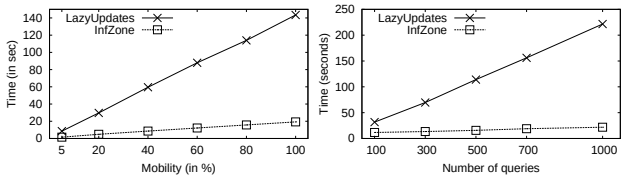


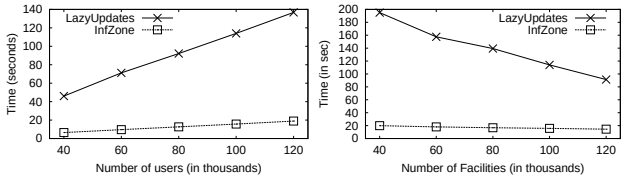
Fig. 29 Effect of k

The locations of the users are reported to the server after every one second (i.e., timestamp length is one second). The mobility of the objects refers to the percentage of the objects that report location updates at a given timestamp. In accordance with [10], the grid cardinality of both of the algorithms is set to 64×64. Each query is monitored for 5 minutes (300 timestamps) and the total time taken by all the queries is reported.



(a) Mobility (b) # of queries

Fig. 30 Effect of mobility and number of queries



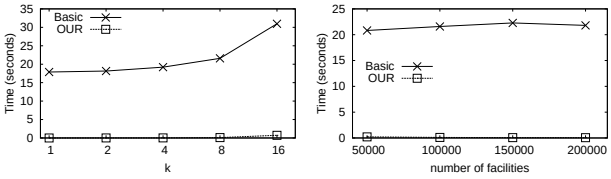
(a) # of users (b) # of facilities

Fig. 31 Effect of data size

In Fig. 29, 30(a), 30(b), 31(a) and 31(b), we study the effect of k, the data mobility, the number of the queries, the number of the users and the number of the facilities, respectively. Influence zone based algorithm is shown as *InfZone*. Clearly, the influence zone based algorithm outperforms Lazy Updates for all the settings and scales better. In Fig. 31(b), both of the algorithms perform better as the number of facilities increases. This is because the unpruned area becomes smaller when the number of facilities is large. Hence, a smaller area is to be monitored by both the algorithms and it results in lower cost.

7.3 Handling data updates

We compare our proposed technique with BASIC algorithm. BASIC calls Algorithm 2 whenever a new facility is added and recomputes the influence zone from scratch whenever a facility that contributes to the existing influence zone is deleted. We randomly generate 1000 updates such that half of the updates are insertions and the other half consists of deletions. The default value of k is 8, number of facilities in the default data set is $100,000$ and the number of queries is 500. The influence zone of each of the query is updated after every data update and the results show the total cost of handling all data updates.



(a) effect of k (b) # of facilities

Fig. 32 Handling data updates

Fig. 32(a) shows that our proposed technique not only performs significantly better than BASIC approach but also scales better as the value of k increases. Fig. 32(b) shows that both of the algorithms are not significantly affected as the number of facilities increases.

8 Conclusion

We introduce the concept of an influence zone which does not only have applications in target marketing and market analysis but can also be used to answer snapshot and continuous RkNN queries. We present a detailed theoretical analysis to study different aspects of the problem. Extensive experiment results verify the theoretical analysis and demonstrate that influence zone based algorithm outperforms existing algorithms. We also extend our technique to compute influence zone in dimensionality higher than two. and present efficient techniques to update the influence zone as the underlying data set is updated by insertions or deletion.

Acknowledgements We would like to thank the editors and the anonymous reviewers for their very helpful comments that have significantly improved this paper. The work was partially done when the corresponding author was taking a Chinese academic program at East China Normal University; and the first and second authors were visiting East China Normal University, and supported by NSFC61021004. The research of Wenjie Zhang was supported by ARC DP120104168 and DE120102144. The research of Xuemin Lin was also supported by ARC DP120104168, DP110102937 and DP0987557. The research of Ying Zhang was supported by ARC DP110104880 and UNSW ECR PSE1799.

References

1. E. Achtert, H.-P. Kriegel, P. Kröger, M. Renz, and A. Züfle. Reverse k-nearest neighbor search in dynamic and general metric databases. In *EDBT*, pages 886–897, 2009.
2. R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS*, 2002.
3. T. Bernecker, T. Emrich, H.-P. Kriegel, N. Mamoulis, M. Renz, and A. Züfle. A novel probabilistic pruning approach to speed up similarity queries in uncertain databases. In *ICDE*, pages 339–350, 2011.
4. T. Bernecker, T. Emrich, H.-P. Kriegel, M. Renz, , and S. Z. A. Züfle. Efficient probabilistic reverse nearest neighbor query processing on uncertain data. In *PVLDB*, 2011.
5. T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 2002.
6. M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Multi-guarded safe zone: An effective technique to monitor moving circular range queries. In *ICDE*, 2010.
7. M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Continuous monitoring of distance-based range queries. *IEEE TKDE*, 23(8):1182–1199, 2011.
8. M. A. Cheema, X. Lin, W. Wang, W. Zhang, and J. Pei. Probabilistic reverse nearest neighbor queries on uncertain data. *IEEE Trans. Knowl. Data Eng.*, 22(4):550–564, 2010.
9. M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang. Influence zone: Efficiently processing reverse k nearest neighbors queries. In *ICDE*, pages 577–588, 2011.
10. M. A. Cheema, X. Lin, Y. Zhang, W. Wang, and W. Zhang. Lazy updates: An efficient technique to continuously monitoring reverse knn. *PVLDB*, 2(1):1138–1149, 2009.
11. M. A. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li. Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks. *VLDB J.*, 21(1):69–95, 2012.
12. T. Emrich, H.-P. Kriegel, P. Kröger, M. Renz, N. Xu, and A. Züfle. Reverse k-nearest neighbor monitoring on mobile objects. In *GIS*, pages 494–497, 2010.
13. T. Emrich, H.-P. Kriegel, P. Kröger, M. Renz, and A. Züfle. Boosting spatial pruning: on optimal pruning of mbrs. In *SIGMOD Conference*, pages 39–50, 2010.
14. B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.
15. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, 1984.
16. E. Haines. *Graphics Gems IV*, chapter Point in Polygon Strategies. Academic Press Professional, Cambridge, 1994.
17. M. Hasan, M. A. Cheema, X. Lin, and W. Zhang. A unified algorithm for continuous monitoring of spatial queries. In *DASFAA (2)*, pages 104–118, 2011.
18. M. Hasan, M. A. Cheema, X. Lin, and Y. Zhang. Efficient construction of safe regions for moving knn queries over dynamic datasets. In *SSTD*, 2009.
19. M. Hasan, M. A. Cheema, W. Qu, and X. Lin. Efficient algorithms to monitor continuous constrained nearest neighbor queries. In *DASFAA (1)*, pages 233–249, 2010.
20. C. Icking and R. Klein. Searching for the kernel of a polygon - a competitive strategy. In *SoCG*, pages 258–266, 1995.
21. G. S. Iwerks, H. Samet, and K. P. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB*, pages 512–523, 2003.
22. I. Kamel and C. Faloutsos. On packing r-trees. In *CIKM*, 1993.
23. J. M. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang. Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In *ICDE*, 2007.
24. F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD*, 2000.
25. I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic queries over mobile objects. In *EDBT*, pages 269–286, 2002.
26. G. Li, Y. Li, J. Li, S. LihChyun, and F. Yang. Continuous reverse k nearest neighbor monitoring on moving objects in road networks. *Inf. Syst.*, 35:860–883, December 2010.
27. X. Lian and L. C. 0002. Efficient processing of probabilistic reverse nearest neighbor queries over uncertain data. *VLDB J.*, 18(3):787–808, 2009.
28. K.-I. Lin, M. Nolen, and C. Yang. Applying bulk insertion techniques for dynamic reverse nearest neighbor problems. *IDEAS*, 2003.
29. K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.
30. A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Wiley, 1999.
31. F. P. Preparata and M. I. Shamos. *Computational Geometry An Introduction*. Springer, 1985.
32. M. Safar, D. Ebrahimi, and D. Taniar. Voronoi-based reverse nearest neighbor query processing on spatial networks. *Multimedia Syst.*, 15(5):295–308, 2009.
33. I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop*, 2000.
34. I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of influence sets in frequently updated databases. In *VLDB*, 2001.
35. H.-L. Sun, C. Jiang, J.-L. Liu, and L. Sun. Continuous reverse nearest neighbor queries on moving objects in road networks. In *WAIM*, pages 238–245, 2008.
36. Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB*, 2004.
37. Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, pages 287–298, 2002.
38. Y. Theodoridis, E. Stefanakis, and T. K. Sellis. Efficient cost models for spatial queries using r-trees. *IEEE TKDE*, 2000.
39. Q. T. Tran, D. Taniar, and M. Safar. Reverse k nearest neighbor and reverse farthest neighbor search on spatial networks. *T. Large-Scale Data- and Knowledge-Centered Systems*, 1:353–372, 2009.
40. W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan. Continuous reverse k-nearest-neighbor monitoring. In *MDM*, 2008.
41. W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan. Finch: Evaluating reverse k-nearest-neighbor queries on location data. In *VLDB*, 2008.
42. T. Xia and D. Zhang. Continuous reverse nearest neighbor monitoring. In *ICDE*, page 77, 2006.
43. Z. Xing, J. Pei, and P. S. Yu. Early prediction on time series: A nearest neighbor approach. In *IJCAI*, 2009.
44. X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.
45. C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *ICDE*, 2001.
46. M. L. Yiu, N. Mamoulis, and P. Karras. Common influence join: A natural join operation for spatial pointsets. In *ICDE*, 2008.
47. M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse nearest neighbors in large graphs. *IEEE TKDE*, 2006.
48. X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, 2005.
49. J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD*, 2003.