Relaxed Reverse Nearest Neighbors Queries

Arif Hidayat, Muhammad Aamir Cheema, and David Taniar

Faculty of Information Technology, Monash University, Australia
{arif.hidayat, aamir.cheema, david.taniar}@monash.edu

Abstract. Given a set of users U, a set of facilities F, and a query facility q, a reverse nearest neighbors (RNN) query retrieves every user u for which q is its closest facility. Since q is the closest facility of u, the user u is said to be influenced by q. In this paper, we propose a *relaxed* definition of influence where a user u is said to be influenced by not only its closest facility but also every other facility that is *almost* as close to u as its closest facility is. Based on this definition of influence, we propose relaxed reverse nearest neighbors (RRNN) queries. Formally, given a value of $x > 1$, an RRNN query q returns every user u for which $dist(u, q) \leq x \times NNDist(u)$ where $NNDist(u)$ denotes the distance between a user u and its nearest facility. Based on effective pruning techniques and several non-trivial observations, we propose an efficient RRNN query processing algorithm. Our extensive experimental study conducted on several real and synthetic data sets demonstrates that our algorithm is several orders of magnitude better than a naïve algorithm as well as a significantly improved version of the naïve algorithm.

1 Introduction

People usually prefer the facilities in their vicinity. Hence, they are influenced by nearby facilities. A *reverse nearest neighbors* (RNN) query [1–4] aims at finding every user that is influenced by a query facility q. Formally, given a set of users U, a set of facilities F and a query facility q, an RNN query returns every user $u \in U$ for which the query facility q is its closest facility. The set containing RNNs, denoted as $RNN(q)$, is also called the influence set of q.

Consider the example of Fig. 1 that shows four McDonald's restaurants (f_1 to f_4) and three users (u_1 to u_3). In the context of RNN queries, the users u_2 and u_3 are both influenced by f_1 because f_1 is their closest McDonald's. Therefore, u_2 and u_3 are the RNNs of f_1, i.e., $RNN(f_1) = \{u_2, u_3\}$. Similarly, it can be confirmed that $RNN(f_2) = \emptyset$, $RNN(f_3) = \emptyset$, $RNN(f_4) = \{u_1\}$.

A *reverse k nearest neighbors* (RkNN) query [5–10] is a natural extension of the RNN query and uses a *relaxed* notion of influence. Specifically, in the context of an RkNN query, a user u is considered to be influenced by its k closest facilities. Hence, an RkNN query q returns every user $u \in U$ for which q is among its k closest facilities. In the example of Fig. 1, assuming $k = 2$, $R2NN(f_2) = \{u_1, u_2, u_3\}$ because f_2 is one of the two closest facilities for all of the three users. Similarly, $R2NN(f_1) = \{u_2, u_3\}$, $R2NN(f_3) = \emptyset$ and $R2NN(f_4) = \{u_1\}$.

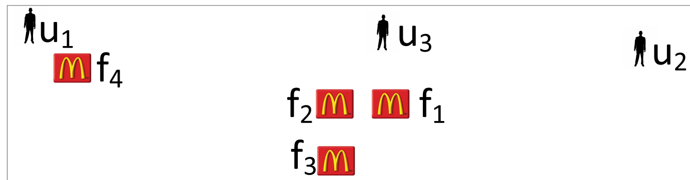


Fig. 1. Illustration of the reverse nearest neighbor query and its variants

RkNN queries have numerous applications [1] in location based services, resource allocation, profile-based management, decision support etc. Consider the example of a supermarket. The people for which this supermarket is one of the k closest supermarkets are its potential customers and may be influenced by targeted marketing or special deals. Due to its significance, RNN queries and its variants have received significant research attention in the past decade (see [6] for a survey).

In this paper, we propose an alternative definition of influence and propose a variant of RNN queries called *relaxed reverse nearest neighbors* (RRNN) query. This definition is motivated by our observation that an RkNN query may not properly capture the notion of influence as explained below.

1.1 Motivation

Consider the example of a person living in a suburban area (e.g., u_2 in Fig. 1) who does not have any McDonald's nearby. Her nearest McDonald's is f_1 which is say 30 Km from her location. In the context of R2NN query, u_2 is influenced by f_1 and f_2 – her two nearest facilities. However, we argue that it is also influenced by f_3 because a user who needs to travel a minimum of 30 Km to visit a McDonald's may also be willing to travel to a McDonald's store 31 Km far from her location.

Similarly, consider the example of another person living in a suburb (e.g., u_1 in Fig. 1) who has only one McDonald's nearby (f_4) assuming that all other McDonald's (e.g., f_1 to f_3) are in downtown area and are quite far. In the context of R2NN queries, the user u_1 is considered to be influenced by both f_4 and f_2 because these are her two closest facilities. However, we argue that the user u_1 is only influenced by f_4 because the other facilities are significantly farther than $dist(u_1, f_4)$, e.g., a user who has a McDonald's within 1 Km is not very likely to visit a McDonald's that is say 30 Km from her location.

As shown above, the definition of influence used in RkNN queries considers only the relative ordering of the facilities based on their distances from u and ignores the actual distances of the facilities from u. Motivated by this, in this paper, we propose a relaxed reverse nearest neighbors (RRNN) query that relaxes the definition of influence using a parameter x (called the x factor in this paper) and considers the relative distances between the users and the facilities.

Definition 1. *Let $NNdist(u)$ denote the distance between u and its nearest facility. Given a value of $x > 1$, a user u is said to be influenced by a facility f, if $dist(u, f) \leq x \times NNdist(u)$.*

Relaxed Reverse Nearest Neighbors (RRNN) query. Given a value of $x > 1$, an RRNN query q returns every user u for which $dist(u, q) \leq x \times NNdist(u)$, i.e., return every user u that is influenced by q according to Definition 1. The set of RRNNs of a query q is denoted as $RRNN_x(q)$. Note that an RRNN query is the same as an RNN query if $x = 1$.

In the example of Fig. 1, assuming $x = 1.2$, RRNN of f_2 are the users u_2 and u_3, i.e., $RRNN_{1.2}(f_2) = \{u_2, u_3\}$. Similarly, $RRNN_{1.2}(f_1) = \{u_2, u_3\}$, $RRNN_{1.2}(f_3) = \{u_2\}$ and $RRNN_{1.2}(f_4) = \{u_1\}$.

Remark. RkNN queries and RRNN queries assume that the distance is the main factor influencing a user. This assumption holds in many real world scenarios. For instance, the users looking for nearby fuel stations are usually not concerned about price (or even rating) because all fuel stations have similar price (or even the same price because, in some countries, the fuel prices are regulated by the government). Similarly, users interested in McDonald's restaurants are mainly influenced by the distance because other attributes such as price, menu, and ratings are the same for all stores. Nevertheless, in the case where the users are influenced by other attributes, reverse top-k queries [11, 12] can be used to compute the influence using a scoring function involving multiple attributes such as distance, price, and rating. This is a different line of research and is not within the scope of this paper.

1.2 Contributions

We make the following contributions in this paper.

1. We complement the RkNN queries by proposing a new definition of influence that uses the x factor to provide more meaningful results by considering the relative distances between the users and the facilities.

2. As we show in Section 3, the pruning techniques used to solve RkNN queries cannot be applied or extended for RRNN queries. This is mainly because, in our problem settings, a facility f may not be able to prune the users that are quite far from f (see Section 3 for details). Based on several non-trivial observations, we propose efficient pruning techniques that are proven to be *tight*, i.e., given a facility f used for pruning, the pruning techniques guarantee to prune every point that can be pruned by f. We then propose an efficient algorithm that utilizes these pruning techniques to efficiently compute the RRNNs.

3. We conduct an extensive experimental study on three real data sets and several synthetic data sets to show the effectiveness of our proposed techniques. Since existing techniques cannot be extended to answer RRNN queries, we compare our algorithm with a naïve algorithm (called RQ) as well as a significantly improved version of RQ (called IRQ). The experimental results show that our algorithm is several orders of magnitude better than both of the competitors. Furthermore, we note that the results of an RRNN query are the same as the RkNN ($k = 1$) query when x is quite close to 1. Therefore, we also compare our algorithm (by setting $x = 1 + 10^{-0.6}$) with the most notable RNN algorithms.

Although our algorithm solves a more challenging version of the problem, our experiments show that it performs reasonably well compared to RNN algorithms.

The rest of the paper is organized as follows. We present the problem definition and an overview of the related work in Section 2. The pruning techniques are discussed in Section 3. Section 4 describes our algorithm to solve RRNN queries. An extensive experimental study is provided in Section 5 followed by conclusions and directions for future work in Section 6.

2 Preliminaries

2.1 Problem Definition

Similar to RkNN queries, RRNN queries can also be classified into *bichromatic* RRNN queries and *monochromatic* RRNN queries.

Bichromatic RRNN query. Given a set of users U, a set of facilities F, a query facility q (which may or may not be in F), and a value of $x > 1$, a bichromatic RRNN query returns every user $u \in U$ for which $dist(u, q) \leq x \times NNdist(u)$ where $NNDist(u)$ denotes the distance between u and its nearest facility in F.

Monochromatic RRNN query. Given a set of facilities F, a query facility q (which may or may not be in F), and a value of $x > 1$, a monochromatic RRNN query returns every facility $f \in F$ for which $dist(f, q) \leq x \times NNdist(f)$ where $NNDist(f)$ denotes the distance between f and its nearest facility in $\{F - f\}$.

In Fig. 1, the monochromatic RRNNs of f_2 (assuming $x = 1.5$) are f_1 and f_3. Monochromatic queries aim at finding the facilities that are influenced by the query facility. Consider a set of police stations. For a given police station q, a monochromatic query returns the police stations for which q is a nearby police station. Such police stations may seek assistance (e.g., extra policemen) from q in case of an emergency event.

Although our techniques can be easily applied to monochromatic RRNN queries, in this paper, we focus on bichromatic RRNN queries because the bichromatic version has more applications in real world scenarios. Similar to the existing work on RNN queries, we assume that both the facility and user data sets are indexed by R*-tree [13]. The R*-tree that indexes the set of facilities (resp. users) is called facility (resp. user) R*-tree. Since most of the applications of the RNN query and its variants are in location-based services, similar to the existing RNN algorithms [6], the focus of this paper is on two dimensional location data.

2.2 Related Work

The RkNN query has been extensively studied [3, 14, 15, 2, 7, 5, 16, 17, 4, 8, 9, 18, 19, 10] ever since it was introduced in [1]. Below, we briefly describe two widely used pruning strategies.

Half-space based pruning [5]. A perpendicular bisector between a facility f and a query q divides the whole space into two halves. Let $H_{f:q}$ denote the half-space that contains f and $H_{q:f}$ be the half-space that contains q. A user u that

lies in $H_{f:q}$ cannot be the RNN of q because $dist(u, f) < dist(u, q)$. Consider the example of Fig. 2, where the half-space $H_{a:q}$ is the shaded area. The users u_1 and u_2 cannot be the RNN of q because they lie in $H_{a:q}$. This observation can be extended for RkNN queries. Specifically, a user u cannot be the RkNN of q if it lies in at least k such half-spaces. In Fig. 2, assuming $k = 2$, the user u_2 cannot be R2NN of q because it lies in $H_{a:q}$ and $H_{b:q}$. In other words, the area $H_{a:q} \cap H_{b:q}$ (the dark shaded area) can be pruned.

Six-regions based pruning [2]. In six-regions based pruning approach, the space around q is divided into six equal regions of 60° each (see P_1 to P_6 in Fig. 3). Let d_i^k be the distance between q and its k-th nearest facility in a partition P_i. It can be proved that a user u lying in a partition P_i cannot be the RkNN of q if $dist(u, q) > d_i^k$. Based on this observation, the k-th nearest facility in each partition P_i is found and the distance d_i^k is used to prune the search space. For instance, in Fig. 3, the shaded area can be pruned if $k = 1$, i.e., the users u_1 and u_2 are pruned.

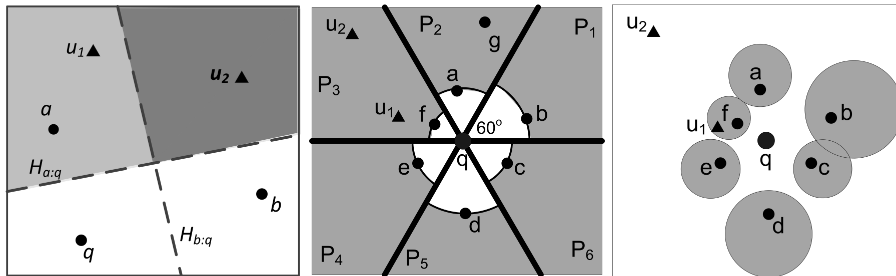


Fig. 2. Half-space pruning **Fig. 3.** Six-regions pruning **Fig. 4.** Challenges

It has been shown [5] that the half-space based approach prunes more area than the six-regions based pruning. However, the advantage of the six-regions based pruning is that it is computationally less expensive. Six-region [2] and Slice [10] are the most notable algorithms that use six-regions based pruning whereas TPL [5], FINCH [20], InfZone [8, 21], and TPL++ [6] are some of the remarkable algorithms that employ half-space based pruning. The details of these algorithms can be found in a recent survey paper [6].

To the best of our knowledge, none of the existing algorithms can be applied or trivially extended to answer RRNN queries studied in this paper. The idea of relative distances has been discussed in [22] in the context of k nearest neighbors queries. However, this is a survey study and a solution was not proposed.

3 Pruning Techniques

Given a facility f, a user u cannot be the RRNN of q if $dist(u, q) > x \times dist(u, f)$. In such case, we say that the facility f prunes the user u. In this section, we will present the pruning techniques that use a facility f or an MBR of the facility R*-tree to prune the users. First, we highlight the challenges.

3.1 Challenges

Existing pruning techniques cannot be applied or extended for the RRNN queries due to the unique challenges involved. For instance, the algorithms to solve RkNN queries can prune most of the search space by considering only the nearby facilities surrounding q. Consider the example of Fig. 4 where the six-regions approach finds the nearest facility to the query q in each of the six partitions and the shaded area can be pruned.

However, in the case of RRNN queries, the nearby facilities surrounding the query q are not sufficient to prune a large part of the search space. Assuming $x = 2$, in partition P_3 (see Fig. 4), while the user u_1 can be pruned by f the user u_2 cannot be pruned by f. In other words, the users that are further from a facility f are less likely to be pruned by it.

In Fig. 4, assuming $x = 2$, the six shaded circles show the maximum possible area that can be pruned by the six facilities a to f (the details on how to compute the circles will be presented later). Note that the facilities that are close to q prune a smaller area as compared to the farther facilities. Hence, the algorithm needs to access not only nearby facilities but also farther facilities to prune a large part of the search space. Also, note that RRNN queries are more challenging because the maximum area that can be pruned is significantly smaller.

In Section 3.2, we present the pruning techniques that prune the space using a data point, i.e. a facility f. In Section 3.3, we present the techniques to prune the space using an MBR of the facility R*-tree. Efficient implementation of the pruning techniques is discussed in Section 3.4.

3.2 Pruning using a facility point

Before we present our non-trivial pruning technique, we present the definition of a pruning circle.

Definition 2 (Pruning circle). *Given a query q, a multiplication factor $x > 1$ and a point p, the pruning circle of p (denoted as C_p) is a circle centered at c with radius r where $r = \frac{x \cdot dist(q,p)}{x^2 - 1}$ and c is on the line passing through q and p such that $dist(q, c) > dist(p, c)$ and $dist(q, c) = \frac{x^2 \cdot dist(q,p)}{x^2 - 1}$.*

Consider the example of Fig. 5 that shows the pruning circle C_f of a facility f assuming $x = 2$. The centre of c is located on the line passing through q and f such that $dist(q, c) = \frac{4 \cdot dist(q,f)}{3}$, $dist(q, c) > dist(f, c)$ and radius $r = \frac{2 \cdot dist(q,f)}{3}$. The condition $dist(q, c) > dist(f, c)$ ensures that c lies towards f on the line passing through q and f, i.e., f lies between the points c and q as shown in Fig. 5. Next, we introduce our first pruning rule in Lemma 1.

Lemma 1. *Every user u that lies in the pruning circle C_f of a facility f cannot be the RRNN of q, i.e., $dist(u, q) > x \times dist(u, f)$.*

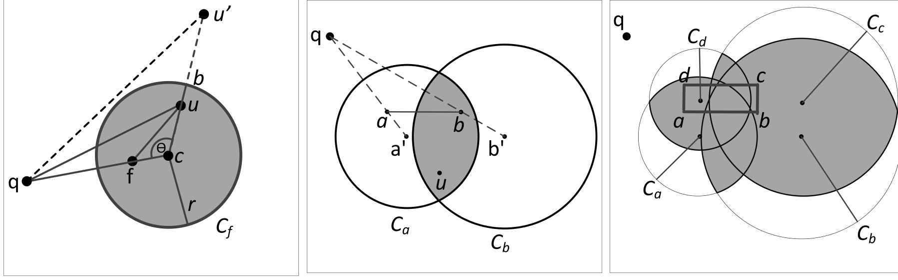


Fig. 5. Lemma 1 **Fig. 6.** Lemma 3 **Fig. 7.** Pruning using MBR

Proof. Given two points v and w, we use $\overline{vw}$ to denote $dist(v,w)$. Consider the example of Fig. 5. Since u is inside the circle C_f, $\overline{uc} < r$. Assume that $\overline{uc} = n \cdot r$ where $0 \le n < 1$. Since $r = \frac{x \cdot \overline{qf}}{x^2-1}$, we have $\overline{uc} = n \cdot r = n \cdot \frac{x \cdot \overline{qf}}{x^2-1}$.

Considering the triangle $\triangle quc$, $\overline{qu} = \sqrt{(\overline{qc})^2 + (\overline{uc})^2 - 2 \cdot \overline{uc} \cdot \overline{qc} \cdot \cos\theta}$. Since $\overline{uc} = n \cdot \frac{x \cdot \overline{qf}}{x^2-1}$ and $\overline{qc} = \frac{x^2 . \overline{qf}}{x^2-1}$, we have

$$\begin{aligned}\overline{qu} &= \sqrt{(\frac{x^2 \cdot \overline{qf}}{x^2-1})^2 + n^2(\frac{x \cdot \overline{qf}}{x^2-1})^2 - 2n(\frac{x \cdot \overline{qf}}{x^2-1})(\frac{x^2 \cdot \overline{qf}}{x^2-1}) \cdot \cos\theta} \\ &= \sqrt{(\frac{x \cdot \overline{qf}}{x^2-1})^2(x^2 + n^2 - 2 \cdot x \cdot n \cdot \cos\theta)} \\ &= (\frac{x \cdot \overline{qf}}{x^2-1})\sqrt{x^2+n^2-2xn\cos\theta}\end{aligned} \tag{1}$$

Similarly considering $\triangle fcu$, $\overline{fu} = \sqrt{(\overline{fc})^2 + (\overline{uc})^2 - 2 \cdot \overline{uc} \cdot \overline{fc} \cdot \cos\theta}$. Since $\overline{fc} = \overline{qc} - \overline{qf}$ and $qc = \frac{x^2 . \overline{qf}}{x^2-1}$, we get $\overline{fc} = \frac{\overline{qf}}{x^2-1}$. We can obtain the value of $\overline{fu}$ by replacing the values of $\overline{fc}$ and $\overline{uc}$.

$$\begin{aligned}\overline{fu} &= \sqrt{(\frac{\overline{qf}}{x^2-1})^2 + n^2(\frac{x \cdot \overline{qf}}{x^2-1})^2 - 2 \cdot n(\frac{x \cdot \overline{qf}}{x^2-1}) \cdot (\frac{\overline{qf}}{x^2-1}) \cdot \cos\theta} \\ &= (\frac{\overline{qf}}{x^2-1})\sqrt{1+n^2x^2-2nx\cos\theta}\end{aligned} \tag{2}$$

Note that the user u can be pruned if $dist(u,q) > x \times dist(u,f)$. Therefore, we need to show $\overline{qu} - x \cdot \overline{fu} > 0$. The left side of this inequality can be obtained using the values of $\overline{qu}$ and $\overline{fu}$ from Eq. (1) and Eq. (2), respectively.

$$\overline{qu} - x \cdot \overline{fu} = \frac{x . \overline{qf}}{x^2-1}(\sqrt{x^2+n^2-2xn\cos\theta} - \sqrt{1+x^2n^2-2xn\cos\theta}\ ) \tag{3}$$

Since $x > 1$, $(\frac{x . \overline{qf}}{x^2-1})$ is always positive. Hence, we just need to prove that $(\sqrt{x^2+n^2-2xn\cos\theta} - \sqrt{1+x^2n^2-2xn\cos\theta} > 0$. In other words, we need to

show $(\sqrt{x^2 + n^2 - 2xn\cos\theta} > \sqrt{1 + x^2n^2 - 2xn\cos\theta}$. Note that both sides of this inequality are positive (otherwise $\overline{qu}$ and $\overline{fu}$ in Eq. (1) and Eq. (2) would be negative which is not possible). Hence, we can take the square of both sides resulting in $x^2 + n^2 - 2xn\cos\theta > 1 + x^2n^2 - 2xn\cos\theta$ which implies that we need to prove $(x^2 + n^2 - x^2n^2 - 1) > 0$. This inequality can be simplified as $(x^2-1)(1-n^2) > 0$. Since $x > 1$ and $n < 1$, it is easy to see that $(x^2-1)(1-n^2) > 0$ which completes the proof. □

Note that although the pruning technique itself is non-trivial, applying this pruning rule is not expensive, i.e., to check whether a user u can be pruned or not, we only need to compute its distance from the centre c and compare it with the radius r. Next, we show that this pruning rule is *tight* in the sense that any user u' that lies outside C_f is guaranteed not to be pruned by the facility f.

Lemma 2. *Given a facility f and a user u' that lies on or outside its pruning circle C_f, then $dist(u', q) \leq x \times dist(u', f)$, i.e. u' cannot be pruned by f.*

Proof. Consider the user u' in Fig. 5. Since u' is on or outside the pruning circle, it satisfies $\overline{u'c} = n \cdot r$, where $n \geq 1$. The proof is similar to the proof of Lemma 1 except that we need to show that $\overline{u'q} - x.\overline{fu'} \leq 0$, i.e., we need to show $(x^2 - 1)(1 - n^2) \leq 0$ which is obvious given that $x > 1$ and $n \geq 1$. □

Note that the pruning circle C_f is larger if $dist(q, f)$ is larger which implies that the facilities that are farther from the query prune larger area. For instance, in Fig. 6, the pruning circle C_b is bigger than the pruning circle C_a.

3.3 Pruning using the nodes of facility R*-tree

In this section, we present our techniques to prune the search space using the intermediate or leaf nodes of the facility R*-tree. These pruning techniques reduce the I/O cost of the algorithm because the algorithm may prune the search space using a node of the R*-tree instead of accessing the facilities in its sub-tree.

A node of the facility R*-tree is represented by a minimum bounding rectangle (MBR) that encloses all the facilities in its sub-tree. Without accessing the contents of the node, we cannot know the locations of the facilities inside it except that each side of the MBR contains at least one facility. We utilize this information to devise our pruning techniques. Specifically, we use all four sides of the MBR and use each side (i.e., line segment) to prune the search space. Lemma 3 presents the pruning rule and Fig. 6 provides an illustration.

Lemma 3. *Given a query q, a multiplication factor $x > 1$, and a line $\overline{ab}$ representing a side of an MBR, a user u cannot be the RRNN of q if it lies inside both of the pruning circles C_a and C_b, i.e., u can be pruned if u lies in $C_a \cap C_b$.*

Proof. Let $maxdist(p, \overline{ab})$ denote the maximum distance between a point p and a line $\overline{ab}$. Note that $maxdist(u, \overline{ab}) = max(dist(u, a), dist(u, b))$. Since u lies in both C_a and C_b, $dist(u, q) > x \times dist(u, a)$ and $dist(u, q) > x \times dist(u, b)$

(according to Lemma 1). In other words, $dist(u,q) > x \times maxdist(u,\overline{ab})$. Since there is at least one facility f on the line $\overline{ab}$, $dist(u,f) \leq maxdist(u,\overline{ab})$. Hence, $dist(u,q) > x \times dist(u,f)$ which implies that the user u can be pruned. □

In Fig. 6, the shaded area can be pruned by using the line $\overline{ab}$. The next lemma shows that this pruning rule is also tight.

Lemma 4. *Given a line $\overline{ab}$ such that the only information we have is that there is at least one facility f on $\overline{ab}$, a user u cannot be pruned if it lies outside either C_a or C_b.*

Proof. Without the loss of generality, assume that u lies outside C_a. Now assume that there is exactly one facility f on the line $\overline{ab}$ and it lies at the end point a. Since f lies on a, $C_a = C_f$ which implies that u is outside C_f. Hence, u cannot be pruned by f (Lemma 2). □

To prune the search space using an MBR, we apply Lemma 3 on each of side s_i of the MBR. Specifically, a user u can be pruned if, for *any* side s_i of the MBR, u lies in *both* of the pruning circles of the end points of s_i. Consider the example of Fig. 7 where an MBR *abcd* is shown along with the pruning circles of the corners of the MBR (see C_a to C_d). Let A_i denote the area pruned by a side s_i of the MBR. In Fig. 7, the shaded area can be pruned which corresponds to $\cup_{i=1}^{4} A_i$ where $A_1 = C_a \cap C_b$, $A_2 = C_b \cap C_c$, $A_3 = C_c \cap C_d$, and $A_4 = C_d \cap C_a$.

3.4 Implementation of the pruning techniques

In the previous sections, we discussed how to prune the search space using a facility point or an MBR of the facility R*-tree. In this section, we discuss how to efficiently and effectively implement the pruning techniques.

Assume that we have a set of facilities and MBRs to be used for pruning the search space. Let A_i denote the area pruned by a facility point or a side of an MBR. Let $\mathcal{A} = \{A_i, \cdots, A_n\}$ be the total area that can be pruned by using the set of facilities and MBRs. In this section, we present Algorithm 1 that efficiently checks whether an entry e of user R*-tree (i.e., a point or an MBR) can be pruned by $\mathcal{A}$ or not, i.e., whether e lies inside $\mathcal{A}$ or not. Before we discuss the details of Algorithm 1, we describe how to prune a user MBR e using a single pruning area $A_i \in \mathcal{A}$. Since e is an MBR, it is possible that e only partially lies in A_i. Ideally, we should be able to prune the part of the MBR that lies inside A_i. In our algorithm, we process the MBR e such that the area that lies inside A_i is trimmed. Below are the details on how to do this.

Case 1: A_i corresponds to the area pruned by a facility. Consider the example of Fig. 8 where A_i corresponds to the circle C_a. Note that only a part of the rectangle R lies in the circle. In such case, we conservatively approximate the area that can be pruned. Specifically, we use a function $\texttt{TrimEntry}(C_a, R)$ that trims the MBR R using a circle C_a and returns R_a that corresponds to the minimum bounding rectangle of the part of R that lies outside C_a, i.e., R_a cannot be pruned by C_a. In Fig. 8, R_a is the shaded area. In Fig. 9, R_b (the light shaded

area) is returned by `TrimEntry`(C_b, R). The function `TrimEntry`(C_a, R) can be implemented as follows. Let I be the set of intersection points between a circle C_a and a rectangle R. Let $\mathcal{C}$ be the corners of R that lie outside C_a. The trimmed entry R_a is the minimum bounding rectangle enclosing the points in $I \cup \mathcal{C}$.

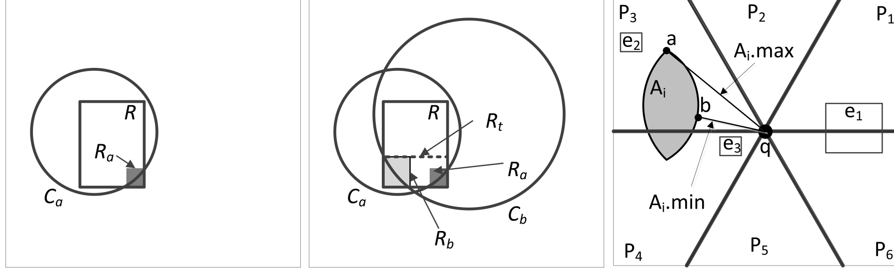


Fig. 8. Trimming an MBR **Fig. 9.** Pruning an entry **Fig. 10.** Observations 1&2

Case 2: A_i corresponds to the area pruned by a side of an MBR. Consider the example of Fig. 9 where A_i corresponds to the area pruned by a line $\overline{ab}$, i.e., $A_i = C_a \cap C_b$. In this case, we find the part of the MBR R that cannot be pruned by A_i as follows. Let R_a =TrimEntry(C_a, R) (see the dark shaded area) and R_b=TrimEntry(C_b, R) (see the light dotted area) in Fig. 9. The unpruned part of R is the minimum bounding rectangle enclosing both R_a and R_b, e.g., R_t shown in thick broken lines in Fig. 9 cannot be pruned by $C_a \cap C_b$.

Algorithm 1 PruneEntry$(e, \mathcal{A})$

Input: e: the entry to be pruned, $\mathcal{A}$: the set of pruned areas
Output: Return the part of e that cannot be pruned by $\mathcal{A}$

1: **for each** $A_i \in \mathcal{A}$ **do**
2: **if** A_i is related to a facility f **then**
3: $R \leftarrow$ `TrimEntry`(C_f,e)
4: **else if** A_i is related to a line $\overline{ab}$ **then**
5: $R_a \leftarrow$ `TrimEntry`(C_a,e)
6: $R_b \leftarrow$ `TrimEntry`(C_b,e)
7: $R \leftarrow$ minimum bounding rectangle enclosing both R_a and R_b
8: $e \leftarrow R$
9: **if** e is empty **then**
10: return ϕ
11: return e

Algorithm 1 shows the details of how to prune an entry e using a set of pruned areas $\mathcal{A}$. The output of the algorithm is the part of e that cannot be pruned by $\mathcal{A}$. Each entry A_i is iteratively accessed from $\mathcal{A}$ and the entry e is trimmed using the details described earlier (lines 2 to line 7). The trimmed part R is assigned to e which is to be further trimmed in the next iteration (line 8). At any stage, if e is empty, the algorithm terminates by returning ϕ (line 10) which indicates that the whole entry e can be pruned by $\mathcal{A}$. When all entries A_i in $\mathcal{A}$ have been accessed, the algorithm returns e.

We remark that although the trimming significantly improves the I/O cost (2 to 3 times) of the algorithm, the overall CPU time is also increased due to

the overhead of trimming. This must be taken into consideration when making the decision on whether to use trimming or not, e.g., the trimming should not be used if the main focus is to optimize CPU cost.

Improving Algorithm 1. Note that Algorithm 1 accesses every entry $A_i \in \mathcal{A}$ regardless of whether A_i can prune a part of e or not. Now, we discuss how to improve the efficiency of Algorithm 1 by ignoring the entries A_i that cannot prune e. Similar to six-regions approach [2] and SLICE [10], we divide the whole space around q in t equally sized partitions, e.g., see the partitions P_1 to P_6 in Fig. 10. Our technique is based on the following two simple observations.

Observation 1. Let $\mathcal{P}$ be the set of partitions overlapped by a pruned area A_i. An entry e can be pruned by A_i only if e overlaps with at least one partition in $\mathcal{P}$. Consider the example of Fig. 10 where the area A_i is shown shaded and overlaps with partitions P_3 and P_4. Since the entry e_1 does not overlap with P_3 or P_4, it cannot be pruned by A_i.

Observation 2. Let $A_i.max$ and $A_i.min$ denote the maximum and minimum distances between q and the pruned area A_i, respectively. Fig. 10 shows $A_i.max = dist(q, a)$ and $A_i.min = dist(q, b)$. We remark that $A_i.max$ and $A_i.min$ can be computed following the ideas presented in [23, 24]. Note that an entry e cannot be pruned by A_i if $mindist(q, e) > A_i.max$ or $maxdist(q, e) < A_i.min$. For instance, the entry e_2 cannot be pruned by A_i because $mindist(q, e_2) > A_i.max$. Similarly, the entry e_3 cannot be pruned because $maxdist(q, e_3) < A_i.min$.

Let $A_i.interval$ denote an interval from $A_i.min$ to $A_i.max$ and $e.interval$ denote an interval from $mindist(q, e)$ to $maxdist(q, e)$. Observation 2 shows that an entry e can be pruned by A_i only if $e.interval$ overlaps with $A_i.interval$. We use an interval tree [25] to efficiently retrieve every A_i for which $A_i.interval$ overlaps with $e.interval$. Specifically, for each partition P_i, we maintain an interval tree $\mathcal{T}_i$ that contains $A_j.interval$ for every $A_j \in \mathcal{A}$ that overlaps with P_i. To check whether an entry e (that overlaps with a partition P_i) can be pruned by $\mathcal{A}$, we issue an interval query on $\mathcal{T}_i$ with input interval $e.interval$. Let $\mathcal{A}_e$ denote the set containing every area A_j returned by the interval query $e.interval$. In Algorithm 1, we use $\mathcal{A}_e$ instead of $\mathcal{A}$. Note that the cost of interval query is $O(m + \log n)$ where n is the number of intervals stored in the interval tree and m is the number of intervals that overlap with the input interval.

4 Algorithm

Our algorithm consists of three phases namely *pruning*, *filtering* and *verification*. In the pruning phase, we use the facility R*-tree to prune the search space, i.e., compute $\mathcal{A}$. In the filtering phase, the users that lie in the pruned space are pruned and the remaining users are inserted in a candidate list called L_{cnd}. Finally, in the verification phase, each candidate user $u \in L_{cnd}$ is verified to check whether it is a RRNN of q or not.

Pruning Phase Algorithm 2 presents the details of the pruning phase. The algorithm initializes a heap h with the root of the facility R*-tree. The entries are iteratively de-heaped from the heap and are processed as follows. If a de-

heaped entry e is pruned (i.e., the entry e' returned by Algorithm 1 is empty), we ignore it (lines 5 and 6). Otherwise, we process it as follows.

Algorithm 2 Pruning

Input: facility R*-tree, and a query q
Output: The set of pruned areas $\mathcal{A}$

```
 1: A ← φ
 2: insert root of facility R-tree in a h
 3: while h is not empty do
 4:    de-heap an entry e
 5:    e′ ← PruneEntry(e, A)                              ▷ Algorithm 1
 6:    if e′ ≠ φ then                                    ▷ e is not pruned
 7:       if e is an intermediate or leaf node then
 8:          for each side ab of e do
 9:             create A_i = C_a ∩ C_b and insert in A
10:          for each child c of e do
11:             if c overlaps with e′ then insert c in the heap
12:       else                                          ▷ e is a facility point
13:          create A_i = C_e and insert in A
```

If e is an intermediate or leaf node of the R*-tree, for each side of e, we create a pruning area A_i and insert it in $\mathcal{A}$ (line 9). We also insert its children in the heap h. Note that a child c of e that does not overlap with e' can be pruned because it lies in the pruned area. Hence, only the children that overlap with e' are inserted in the heap (line 11). If e is a facility point, we create the pruning circle C_e and add it to $\mathcal{A}$ (line 13). The algorithm terminates when the heap becomes empty.

Filtering Phase Algorithm 3 describes the filtering phase. A stack S is initialized with the root entry of the user R*-tree. Each entry e is iteratively retrieved from S and processed as follows. If e can be pruned by $\mathcal{A}$, it is ignored (lines 5 and 6). Otherwise, if it is an intermediate or leaf node, its children that overlap with e' are inserted in the stack (line 9). If e is a user, it is inserted in L_{cnd} (line 11). The algorithm stops when the stack S becomes empty.

Verification Phase In the verification phase, each candidate user $u \in L_{cnd}$ is verified as follows. Note that a user u is a RRNN if and only if there is no facility f for which $dist(u, f) < \frac{dist(u,q)}{x}$. A circular boolean range query is issued with centre at u and radius $r = \frac{dist(u,q)}{x}$ that returns true if and only if there exists a facility in the circle. The boolean range query is conducted on the facility R*-tree as in previous works [7] and u is reported as an answer if it returns false.

5 Experiments

5.1 Experimental Setup

To the best of our knowledge, there is no prior algorithm to solve RRNN queries. We consider a naïve algorithm (RQ) and make reasonable efforts to devise a significantly improved version of RQ, as explained below.

Algorithm 3 Filtering

Input: user R*-tree, query q, and $\mathcal{A}$
Output: a list of candidates L_{cnd}

```
1: L_cnd ← φ
2: insert root of user R*-tree in a stack S
3: while S is not empty do
4:    retrieve top entry e from S
5:    e' ← PruneEntry(e, A)                          ▷ Algorithm 1
6:    if e' ≠ φ then                                 ▷ e is not pruned
7:       if e is an intermediate or leaf node then
8:          for each child c of e do
9:             if c overlaps with e' then insert c in stack S
10:      else                                        ▷ e is a user
11:         insert e in L_cnd
```

Range Query (RQ). For each user u, a boolean range query with range $dist(u, q)/x$ is issued on the facility R*-tree (as described in the verification phase above).

Improved Range Query (IRQ). Note that an intermediate or leaf node entry e_u of the user R*-tree cannot contain any RRNN if there exists at least one facility f such that $mindist(e_u, q) > x \times maxdist(e_u, f)$, i.e., e_u can be pruned. Based on this, to check whether e_u can be pruned or not, we use a function `isPruned`(e_u) that is implemented as follows. The facility R*-tree is traversed in ascending order of $maxdist(e_u, e_f)$ where e_f denotes an entry in the facility R*-tree. The entry e_u is pruned as soon as we find an entry e_f for which $mindist(e_u, q) > x \times maxdist(e_u, e_f)$. To further improve the I/O and CPU cost of `isPruned`(e_u), we do not access the sub-tree of a facility entry e_f if $mindist(e_u, q) < x \times mindist(e_u, e_f)$ because no child of e_f can prune e_u.

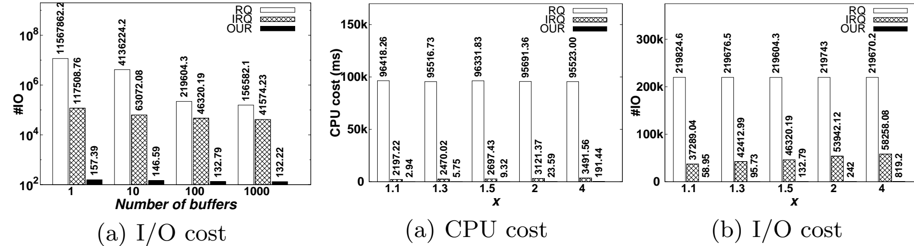
The IRQ algorithm is the same as Algorithm 3 except that 1) "**if** `isPruned`(e) **then**" replaces lines 5 and 6 of Algorithm 3; and 2) at line 11, the user is reported as an answer instead of inserting it in L_{cnd}. Note that IRQ does not have a pruning and verification phase because it merges all these phases in one algorithm. In our experiments, we observed that the performance of IRQ can be further improved if `isPruned`(e_u) is only applied to leaf entries of the user R*-tree. This is because the intermediate nodes are highly unlikely to be pruned and result in un-necessary I/O. We included this optimization in IRQ.

All algorithms were implemented in C++ and experiments were run on Intel Core $I5$ 2.3GHz PC with 8GB memory running on Debian Linux. Experimental settings are quite similar to the existing work [6]. Specifically, we use the same real data sets containing $175,812$ points from North America (called NA data set hereafter), 2.6 million points from Los Angeles (LA) and 25.8 million points from California (CA). We also generate several synthetic data sets containing $1,000$ to 20 million points following normal distributions. The default real data set is LA containing 2.6 million points. Unless mentioned otherwise, each data set is randomly divided into two sets of almost equal size, one corresponding to the facilities and the other to the users. The page size of each R*-Tree [13] is set to $4,096$ Bytes. We randomly select 100 points from the facility data set and

treat them as query points. The cost reported in the experiments correspond to the average cost of a single RRNN query. We vary the value of x from 1.1 to 4 and the default value is 1.5.

5.2 Evaluating Performance

Effect of buffers. All three algorithms need to traverse facility R*-tree every time a boolean range query is issued to verify a candidate user. Hence, the buffers may reduce the I/O cost. We study the effect of the number of buffers on each algorithm. Each buffer page can hold one node of the R*-tree and we use random eviction strategy. In Fig. 11, we report the I/O cost of each algorithm on LA data set for different number of buffers. As expected, the I/O cost of each algorithm decreases with the increase in number of buffers. Note that IRQ is up to two orders of magnitude better than RQ and our algorithm is up to three orders of magnitude better than IRQ. Similar to [6], we use 100 buffer pages for each algorithm in the rest of the experiments.



(a) I/O cost

Fig. 11. Number of buffers

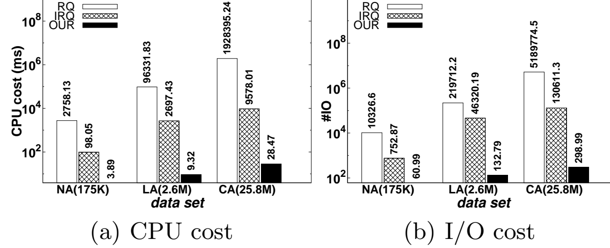
(a) CPU cost (b) I/O cost

Fig. 12. Effect of the x factor (LA data set)

Effect of the x factor. In Fig. 12, we study the effect of the x factor on the three algorithms. Specifically, Fig. 12(a) shows the CPU cost and Fig. 12(b) shows the I/O cost of the three algorithms for varying values of x. In terms of both CPU and I/O cost, our algorithm is up to three orders of magnitude better than IRQ and up to four orders of magnitude better than RQ. The cost of our algorithm and IRQ is higher for larger x factor because the pruning area shrinks as the x factor increases which results in a larger number of candidates and RRNNs. Note that the cost of RQ is not significantly affected by the x factor mainly because it needs to verify every user regardless of the value of x.

Effect of data set size. In Fig. 13(a) and 13(b), we study the effect of data set size on the performance of the three algorithms. Specifically, we conduct experiments on three real data sets: NA (175,000 points), LA (2.6 million points) and CA (25.8 million points). Our algorithm outperforms the other two algorithms and the gap between the three algorithms increases as the data set size increases (please note that log-scale is used in both figures). For example, Fig. 13(a) shows that our algorithm is around 25 times faster than IRQ on NA data set and 330 times faster on CA data set. Similarly, Fig. 13(b) shows that the I/O cost of our algorithm is around 12 times lower than IRQ for NA data set and almost 430 times lower for CA data set. Also, as expected the cost of each algorithm

increases as the data set size increases. This is mainly because the size of each R*-tree increases and more entries are required to be processed.

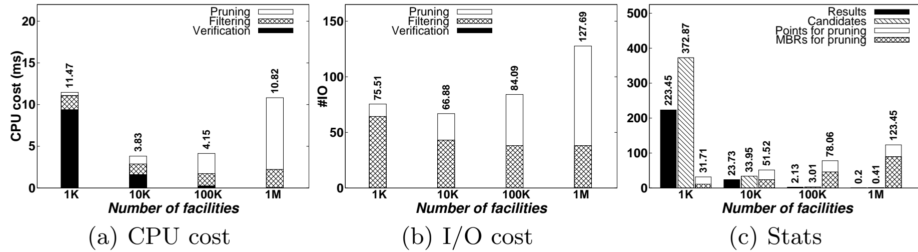


(a) CPU cost (b) I/O cost

Fig. 13. Performance comparison on different real data sets

Since our algorithm is up to several orders of magnitude better than the other algorithms, in the rest of the experiments, we focus on analysing the behavior of our algorithm and omit the cost of the other algorithms for better illustration.

Effect of relative data size. In the previous experiments, each data set contained almost the same number of users and facilities. Next, we analyse the performance of our algorithm where the number of users and the number of facilities are different. Specifically, in Fig. 14 we vary the number of facilities from 1000 to 1 million and the number of users is fixed to $100K$. The sets of facilities and users are generated using normal distribution. Fig. 14(a) and Fig. 14(b) show the CPU and I/O cost of our algorithm, respectively. Fig. 14(c) shows the number of candidates, number of RRNNs and the number of entries (facility points and MBRs) used for pruning.

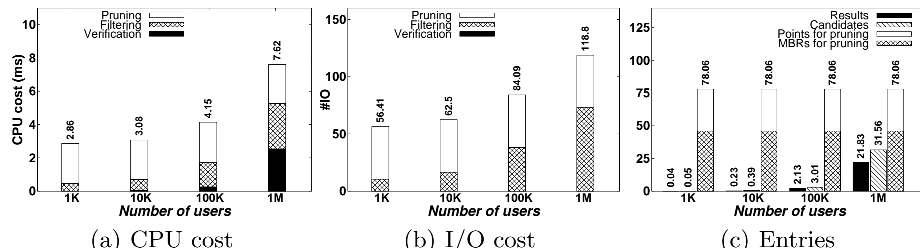


(a) CPU cost (b) I/O cost (c) Stats

Fig. 14. Effect of varying the number of facilities ($100K$ users)

Fig. 14(a) shows that the CPU cost of our algorithm is larger if the number of facilities is too small or too large as compared to the number of users. The reason is as follows. When the number of facilities is too small (e.g., $1,000$), the total area that can be pruned is smaller due to the lower density of the facilities. This results in a larger number of candidates and RRNNs (as shown in Fig. 14(c)). Hence, the verification cost of the algorithm is larger as shown in Fig. 14(a). On the other hand, when the number of facilities is too large (e.g., 1 million), the pruning phase is the dominant cost of the algorithm. This is because the algorithm needs to access a larger number of entries to prune the search space (see Fig. 14(c)).

Fig. 14(b) shows the I/O cost of our algorithm. When the number of facilities is too small, the I/O cost of the filtering phase is larger because the area that can be pruned is smaller due to the lower density of facilities data set. The I/O cost of pruning phase increases as the number of facilities increases. This is because the size of facility R*-tree increases and more entries are required to be accessed to prune the search space.



(a) CPU cost (b) I/O cost (c) Entries

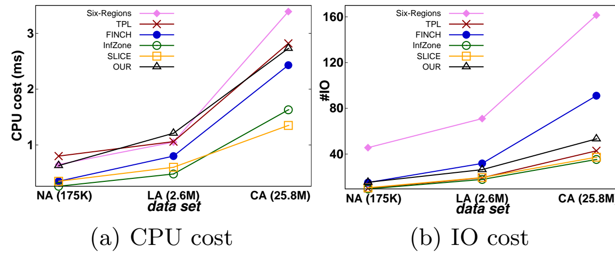
Fig. 15. Effect of varying the number of users ($100K$ facilities)

In Fig. 15, we vary the number of users from $1,000$ to 1 million and fix the number of facilities to $100K$. Fig. 15(a) shows that the CPU cost of the algorithm increases as the number of users increases. This is because the filtering and verification cost of the algorithm increases for larger set of users, e.g., the number of candidate users and RRNNs increases (as shown in Fig. 15(c)). Similarly, Fig. 15(b) shows that the I/O cost of the algorithm also increases for larger number of users. This is because the filtering requires traversing a larger user R*-tree which results in requiring to access more nodes of the users.

Fig. 15(c) also shows the effectiveness of the proposed pruning techniques. Note that the number of candidates is much smaller as compared to the total number of users. Furthermore, almost 65% of the candidates are the relaxed reverse nearest neighbors. We remark that the verification I/O cost of our algorithm is negligible mainly because most of the nodes accessed during verification are already present in the buffer (from pruning phase or the previously issued boolean range queries).

Efficiency compared with RNN algorithms. As stated earlier, there is no previous algorithm to solve RRNN queries and the existing algorithms to solve RNN queries cannot be trivially extended. Although we made significant efforts to devise the second competitor IRQ, our algorithm is up to three orders of magnitude better than it. In the absence of a well-known competitor, readers may find it harder to evaluate the efficiency of an algorithm. Therefore, we compare our algorithm with the most well-known RNN algorithms, namely SLICE [10], InfZone [8], TPL [5], FINCH [20] and six-regions [2]. For our algorithm, we set $x = 1 + 10^{-6}$ because we note that the results of an RRNN query is the same as those of an RNN query if x is very close to 1.

Fig. 16 shows that the performance of our algorithm is comparable to the most popular RNN algorithms which shows the effectiveness of the techniques proposed in this paper. We remark that this experiment is conducted only to demonstrate that our algorithm is efficient and *it should not be used to draw any*



(a) CPU cost (b) IO cost

Fig. 16. Comparison with state-of-the-art RNN algorithms

conclusion regarding the superiority of our algorithm over any other algorithm and vice versa. This is because our algorithm solves an inherently different and arguably more challenging problem.

6 Conclusions and Future Work

In this paper, we propose a variant of RNN queries called relaxed reverse nearest neighbors (RRNN) queries. An RRNN query relaxes the definition of influence using the relative distances between the users and the facilities. RRNN queries are motivated by our observation that RkNN queries may be unable to properly capture the notion of influence. We propose an efficient algorithm based on several efficient and effective pruning techniques and non-trivial observations. The pruning techniques are proved to be tight. The extensive experimental study demonstrates that our algorithm is several orders of magnitude better than the competitors.

There are several interesting directions for future work. For example, it will be interesting to study the relaxed version of reverse top-k queries by using the idea of relative scores, i.e., return every user for whom the query product is almost as good as her most preferred product. Also, continuous RRNN queries for moving objects is another interesting research direction, e.g., continuously report the drivers for which my fuel station is an RRNN. RRNN queries for other distance metrics such as road network distances also need to be explored.

Acknowledgments. The research of Muhammad Aamir Cheema is supported by ARC DE130101002 and DP130103405.

References

1. Korn, F., Muthukrishnan, S.: Influence sets based on reverse nearest neighbor queries. In: SIGMOD. (2000) 201–212
2. Stanoi, I., Agrawal, D., Abbadi, A.E.: Reverse nearest neighbor queries for dynamic databases. In: ACM SIGMOD Workshop. (2000) 44–53
3. Cheema, M.A., Lin, X., Wang, W., Zhang, W., Pei, J.: Probabilistic reverse nearest neighbor queries on uncertain data. IEEE Trans. Knowl. Data Eng. (2010)
4. Stanoi, I., Riedewald, M., Agrawal, D., Abbadi, A.E.: Discovery of influence sets in frequently updated databases. PVLDB (2001) 99–108

5. Tao, Y., Papadias, D., Lian, X.: Reverse knn search in arbitrary dimensionality. PVLDB (2004) 744–755
6. Yang, S., Cheema, M.A., Lin, X., Wang, W.: Reverse k nearest neighbors query processing: Experiments and analysis. In: PVLDB. (2015) 605–616
7. Wu, W., Yang, F., Chan, C.Y., Tan, K.L.: FINCH: Evaluating reverse k-nearest-neighbor queries on location data. PVLDB (2008) 1056–1067
8. Cheema, M.A., Lin, X., Zhang, W., Zhang, Y.: Influence zone: Efficiently processing reverse k nearest neighbors queries. In: ICDE. (2011) 577–588
9. Cheema, M.A., Zhang, W., Lin, X., Zhang, Y., Li, X.: Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks. VLDB J. (2012) 69–95
10. Yang, S., Cheema, M.A., Lin, X., Zhang, Y.: SLICE: Reviving regions-based pruning for reverse k nearest neighbors queries. In: ICDE. (2014) 760–771
11. Vlachou, A., Doulkeridis, C., Kotidis, Y., Nørvåg, K.: Reverse top-k queries. In: ICDE. (2010) 365–376
12. Cheema, M.A., Shen, Z., Lin, X., Zhang, W.: A unified framework for efficiently processing ranking related queries. In: EDBT. (2014) 427–438
13. Beckmann, N., Kriegel, H., Schneider, R., Seeger, B.: The r*-tree: An efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990. (1990) 322–331
14. Emrich, T., Kriegel, H.P., Kröger, P., Renz, M., Züfle, A.: Incremental reverse nearest neighbor ranking in vector spaces. In: SSTD. (2009)
15. Singh, A., Ferhatosmanoglu, H., Tosun, A.S.: High dimensional reverse nearest neighbor queries. In: CIKM. (2003)
16. Achtert, E., Kriegel, H.P., Kröger, P., Renz, M., Züfle, A.: Reverse k-nearest neighbor search in dynamic and general metric databases. In: EDBT. (2009) 886–897
17. Sharifzadeh, M., Shahabi, C.: Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries. PVLDB **3**(1) (2010) 1231–1242
18. Cheema, M.A., Lin, X., Zhang, Y., Wang, W., Zhang, W.: Lazy updates: An efficient technique to continuously monitoring reverse knn. PVLDB (2009) 1138–1149
19. Bernecker, T., Emrich, T., Kriegel, H.P., Renz, M., , Züfle, S.Z.A.: Efficient probabilistic reverse nearest neighbor query processing on uncertain data. PVLDB (2011) 669–680
20. Wu, W., Yang, F., Chan, C.Y., Tan, K.L.: Continuous reverse k-nearest-neighbor monitoring. In: MDM. (2008) 132–139
21. Cheema, M.A., Zhang, W., Lin, X., Zhang, Y.: Efficiently processing snapshot and continuous reverse k nearest neighbors queries. VLDB J. **21**(5) (2012) 703–728
22. Taniar, D., Rahayu, W.: A taxonomy for nearest neighbour queries in spatial databases. J. Comput. Syst. Sci. **79**(7) (2013) 1017–1039
23. Cheema, M.A., Brankovic, L., Lin, X., Zhang, W., Wang, W.: Multi-guarded safe zone: An effective technique to monitor moving circular range queries. In: ICDE. (2010) 189–200
24. Cheema, M.A., Brankovic, L., Lin, X., Zhang, W., Wang, W.: Continuous monitoring of distance-based range queries. IEEE Trans. Knowl. Data Eng. **23**(8) (2011) 1182–1199
25. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., et al.: Introduction to algorithms. Volume 2. MIT press Cambridge (2001)