Reverse *k*NN Search in Arbitrary Dimensionality

Yufei Tao[§] Dimitris Papadias[†] Xiang Lian[†]

[§]Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
taoyf@cs.cityu.edu.hk

[†]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{*dimitris, xlian*}*@cs.ust.hk*

Abstract

Given a point *q*, a *reverse k nearest neighbor* (R*k*NN) query retrieves all the data points that have *q* as one of their *k* nearest neighbors. Existing methods for processing such queries have at least one of the following deficiencies: (i) they do not support arbitrary values of *k* (ii) they cannot deal efficiently with database updates, (iii) they are applicable only to 2D data (but not to higher dimensionality), and (iv) they retrieve only approximate results. Motivated by these shortcomings, we develop algorithms for *exact* processing of R*k*NN with *arbitrary* values of *k* on *dynamic multidimensional* datasets. Our methods utilize a conventional data-partitioning index on the dataset and do not require any pre-computation. In addition to their flexibility, we experimentally verify that the proposed algorithms outperform the existing ones even in their restricted focus.

1. INTRODUCTION

Given a multi-dimensional dataset *P* and a point *q*, a *reverse nearest neighbor* (RNN) query retrieves all the points $p \in P$ that have *q* as their nearest neighbor. Formally, RNN(*q*) = {$p \in P$ | $\neg\exists p' \in P$ such that *dist*(*p*,*p'*) < *dist*(*p*,*q*)}, where *dist* is a distance metric (in this paper we assume Euclidean distance). Although the problem was proposed recently [KM00], it has already received considerable attention due to its importance in several applications involving decision support, resource allocation, profile-based marketing, etc. Other versions of the problem include (i) *continuous* RNN [BJKS02], where *P* contains linearly moving objects with fixed velocities, and the goal is to retrieve all RNNs of *q* for a future interval; (ii) *bichromatic* RNN [SRAA01] where, given a set *Q* of queries, the goal is to find the objects $p \in P$ that are closer to some $q \in Q$ than any other point of *Q*; (iii) *stream* RNN [KMS02], where data arrive in the form of streams, and the goal is to report aggregate results over the RNNs of a set of query points.

This paper focuses on conventional (i.e., *monochromatic*) reverse nearest neighbor queries. In addition to single RNN search, we deal with *reverse k nearest neighbor* (R*k*NN) queries, which retrieve all the points $p \in P$ that have *q* as one of their *k* nearest neighbors. Specifically, R*k*NN(*q*) = {$p \in P$ | $dist(p,q) \leq dist(p,p_k)$, where p_k is the *k*-th farthest NN of *p*}. Figure 1.1 shows four 2D points, where each point *p* is associated with a circle covering its two nearest neighbors For example, the two NNs of p_4 (p_2, p_3) are in the circle centered at p_4. Accordingly, $p_4 \in$ R2NN(p_2) and $p_4 \in$ R2NN(p_3). Let *k*NN(*p*) be the set of *k* nearest neighbors of point *p*. It is important to note that $p \in k$NN(*q*) does not necessarily imply $p \in$ R*k*NN(*q*) and vice versa. For instance, 2NN(p_4)={p_2,p_3}, while R2NN(p_4)=∅ (i.e., p_4 is not contained in the circles of p_1, p_2, or p_3).

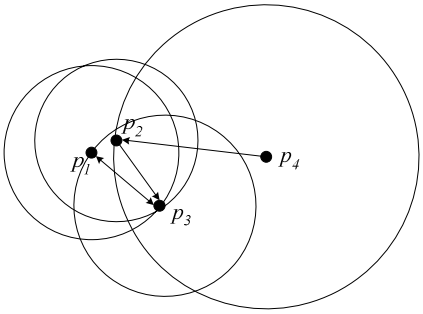


Figure 1.1: 2NN and R2NN examples

As discussed in Section 2.2, all the previous methods for RNN search have at least one of the following deficiencies: (i) they do not support arbitrary values of *k* (ii) they cannot deal efficiently with database updates, (iii) they are applicable only to 2D data (but not to higher dimensionality), and (iv) they retrieve only approximate results (i.e., potentially incurring false misses). In other words, these methods address restricted versions of the problem without providing a general solution. Motivated by this, we develop algorithms for *exact* processing of R*k*NN queries with *arbitrary* values of *k* on *dynamic multidimensional* datasets.

Our methods do not require any pre-processing besides a data-partitioning index (e.g., R-tree [BKSS90], X-tree [BKK96]). Similar to the existing algorithms for

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

dynamic data, we follow a filter-refinement framework. Specifically, the filter step retrieves a set of candidate results that is guaranteed to include all the actual reverse nearest neighbors; the subsequent refinement step eliminates the false hits. The two steps are integrated in a seamless way that eliminates multiple accesses to the same index node (i.e., each node is visited at most once). Our experimental comparison verifies that the proposed techniques outperform the previous ones, even in their restricted focus.

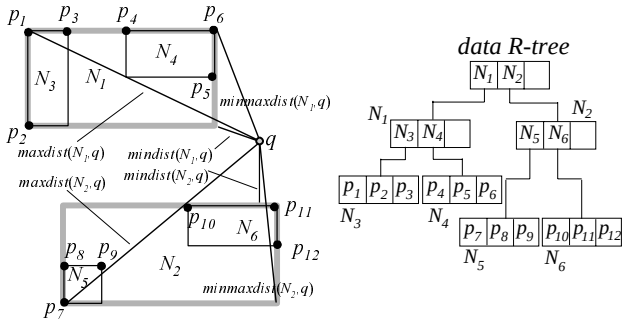
The rest of the paper is organized as follows. Section 2 surveys related work on NN and RNN search. Section 3 presents some interesting problem characteristics, and proposes a new algorithm for single RNN (k=1) retrieval. Section 4 extends the solution to arbitrary values of k. Section 5 experimentally evaluates the proposed methods, and Section 6 concludes the paper with directions for future work.

2. BACKGROUND

Although the proposed algorithms can be used with various indexes, in the sequel, we assume that the dataset P is indexed by an R-tree due to the popularity of this structure in the literature. Section 2.1 briefly overviews the R-tree and algorithms for nearest neighbor search. Section 2.2 describes previous work on monochromatic RNN queries.

2.1 Algorithms for NN search using R-trees

The R-tree [G84] and its variants (most notably the R*-tree [BKSS90]) can be thought of as extensions of B-trees in multi-dimensional spaces. Figure 2.1 shows a 2D point set $P=\{p_1,p_2,\ldots,p_{12}\}$ indexed by an R-tree assuming a capacity of three entries per node. Points that are close in space (e.g., p_1, p_2, p_3) are clustered in the same leaf node (N_3). Nodes are then recursively grouped together with the same principle until the top level, which consists of a single root. An intermediate index entry contains the minimum bounding rectangle (MBR) of its child node, together with a pointer to the page where the node is stored. A leaf entry stores the coordinates of a data point and (optionally) a pointer to the corresponding record.



(a) Points and node extents (b) The R-tree

Figure 2.1: Example of an R-tree and a NN query

Given a d-dimensional set P and a point q, a nearest neighbor query retrieves the point $p \in P$ that is closest to q. The algorithms for NN queries on R-trees utilize some bounds to prune the search space: (i) *mindist*(N,q), which corresponds to the minimum possible distance between q and any point in the subtree of node N, (ii) *maxdist*(N,q), which denotes the maximum possible distance between q and any point in the subtree of N, and (iii) *minmaxdist*(N,q), which gives an upper bound of the distance between q and its closest point in N. In particular, the derivation of *minmaxdist*(N,q) is based on the fact that each edge of the MBR of N contains at least one data point. Hence, *minmaxdist*(N,q) equals the smallest of the maximum distances between all edges (of N) and q. Figure 2.1a shows these pruning bounds between point q and nodes N_1, N_2.

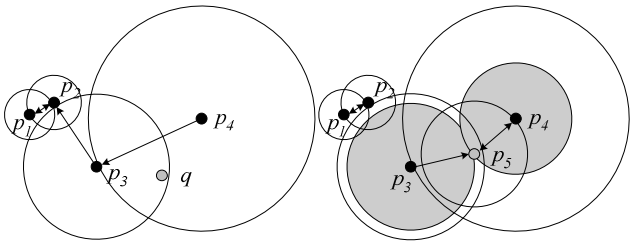
Existing NN methods follow either depth-first (DF), or best-first (BF) traversal. DF algorithms [RKV95, CF98] start from the root and visit recursively the node with the smallest *mindist* from q. In Figure 2.1, for instance, the first 3 nodes accessed are (in this order) root, N_1 and N_4, where the first potential nearest neighbor is found (p_5). During backtracking to the upper levels, DF only visits entries whose minimum distances are smaller than the distance of the NN already retrieved. For example, after discovering p_5, DF backtracks to the root level (without visiting N_3 because *mindist*(N_3,q) > *dist*(p_5,q)), and then follows the path N_2, N_6 where the actual NN p_{11} is found.

Best-first (BF) algorithms [H94, HS99] maintain a heap H with the entries visited so far, sorted by their *mindist*. As with DF, BF starts from the root, and inserts all its entries into H (together with their *mindist*), e.g., in Figure 2.1, $H=\{<N_1, mindist(N_1,q)>, <N_2, mindist(N_2,q)>\}$. Then, at each step, BF visits the node in H with the smallest *mindist*. Continuing the example, the algorithm retrieves the content of N_1 and inserts all its entries in H, after which $H=\{<N_2, mindist(N_2,q)>, <N_4, mindist(N_4,q)>, <N_3, mindist(N_3,q)>\}$. Similarly, the next two nodes accessed are N_2 and N_6 (inserted in H after visiting N_2), in which p_{11} is discovered as the current NN. At this time, the algorithm terminates (with p_{11} as the final result) since the next entry (N_4) in H is farther (from q) than p_{11}. Both DF and BF can be easily extended for the retrieval of k>1 nearest neighbors. *Maxdist* and *minmaxdist* can be applied to speed up the search process. Furthermore, BF is *incremental*, i.e., it reports the nearest neighbors in ascending order of their distance to the query, so that k does not have to be known in advance.

2.2 RNN Algorithms

Algorithms for RNN processing can be classified in two categories depending on whether they require pre-processing, or not. For simplicity, we describe all methods for single RNN retrieval in 2D space. At the end of the section we discuss their applicability to arbitrary values of k and dimensionality.

The original RNN method [KM00] pre-computes for each data point p its nearest neighbor NN(p). Then, it represents p as a *vicinity circle* (p, *dist*(p,NN(p))) centered at p with radius equal to the Euclidean distance between p and its NN. The MBRs of all circles are indexed by an R-tree, called the RNN-tree. Using the RNN-tree, the reverse nearest neighbors of q can be efficiently retrieved by a point location query, which returns all circles that contain q. Figure 2.2a illustrates the concept using four data points, each associated with a vicinity circle. Since q falls in the circles of p_3 and p_4, the result of the query is RNN(q) = {p_3, p_4}.



(a) RNN query (b) Insertion of p_5

Figure 2.2: Illustration of KM[1]

Because the RNN-tree is optimized for RNN, but not NN search, Korn and Muthukrishnan [KM00] use an additional (conventional) R-tree on the data points for nearest neighbors and other spatial queries. In order to avoid the maintenance of two separate structures, Yang and Lin [YL01] combine the two indexes in the RdNN-tree. Similar to the RNN-tree, a leaf node of the RdNN-tree contains vicinity circles of data points. On the other hand, an intermediate node contains the MBR of the underlying points (not their vicinity circles), together with the maximum distance from every point in the sub-tree to its nearest neighbor. As shown in the experiments of [YL01], the RdNN-tree is efficient for both RNN and NN queries because, intuitively, it contains the same information as the RNN-tree and has the same structure (for node MBRs) as a conventional R-tree. Another, solution based on pre-computation is proposed in [MVZ02]. The methodology, however, is applicable only to 2D spaces and focuses on asymptotical worst case bounds (rather than experimental comparison with other approaches).

The problem of KM, YL, MVZ, and all techniques that rely on pre-processing, is that they cannot deal efficiently with updates. This is because each insertion or deletion may affect the vicinity circles of several points. Consider Figure 2.2b, where we want to insert a new point p_5 in the database. First, we have to perform a RNN query to find all objects (in this case p_3 and p_4) that have p_5 as their new nearest neighbors. Then, we update the vicinity circles of these objects in the index. Finally, we compute the NN of p_5 (i.e., p_4) and insert the corresponding circle. Similarly, each deletion must update the vicinity circles of the affected objects. In order to alleviate the problem, Lin et al. [LNY03] propose a method for bulk insertions in the RdNN-tree.

Stanoi et al. [SAA00] eliminate the need for pre-computing all NNs by utilizing some interesting properties of RNN retrieval. Consider Figure 2.3, which divides the space around a query q into six equal regions S_1 to S_6. Let p be the NN of q in some region S_i; it can be proven that (i) either $p \in$ RNN(q) or (ii) there is no RNN of q in S_i. For instance, in Figure 2.3 the NN of q in S_1 is point p_2. However, the NN of p_2 is p_1. Consequently, there is no RNN of q in S_1 and we do not need to search further in this region. The same is true for S_2 (no data points), S_3, S_4 (p_4, p_5 are NNs of each other) and S_6 (the NN of p_3 is p_1). The actual result is RNN(q) = {p_6}. Based on the above property SAA adopts a two-step processing method. First, six constrained NN queries [FSAA01] retrieve the nearest neighbors of q in regions S_1 to S_6. These points constitute the *candidate* result. Then, at a second step, a NN query is applied to find the NN p' of each candidate p. If *dist*(p,q)< *dist*(p,p'), p belongs to the actual result; otherwise, it is a false hit and discarded.

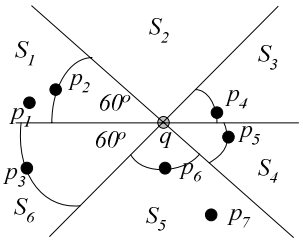


Figure 2.3: Illustration of SAA

The number of regions to be searched for candidate results increases exponentially with the dimensionality[2], rendering SAA inefficient even for three dimensions. Motivated by this, Singh et al. [SFT03] propose a multi-step algorithm that: (i) finds (using an R-tree) the K NNs of the query q, which constitute the initial candidates; (ii) it eliminates the points that are closer to some other candidate than q; (iii) it applies *boolean range queries* on the remaining candidates to determine the actual RNNs. Consider, for instance, the query of Figure 2.4 assuming that K (a system parameter) is 4. The algorithm first retrieves the 4 NNs of q: p_6, p_5, p_4 and p_2. The second step discards p_4 and p_5 since they are closer to each other than q. The third step uses the circles (p_2,*dist*(p_2,q)) and (p_6,*dist*(p_6,q)) to perform two boolean ranges on the data R-tree. The difference with respect to conventional range queries is that a boolean range terminates immediately when (i) the first data point is found, or (ii) the entire side of a node MBR lies within the circle. For instance, *minmaxdist*(N_1,p_2) ≤ *dist*(p_2,q), meaning that N_1 contains at least a point within the range (i.e.,). Thus, p_2 is a false

[1] We refer to the algorithms according to the author initials.

[2] Determining the number of space partitions in SAA is analogous to the *sphere packing* and the *kissing number* problems. For a discussion see [SFT03].

hit and SFT returns p_6 as the only RNN of q. The major shortcoming of the method is that it may incur false misses. In Figure 2.4, although p_3 is a RNN of q, it does not belong to the 4 NNs of the query and will not be retrieved.

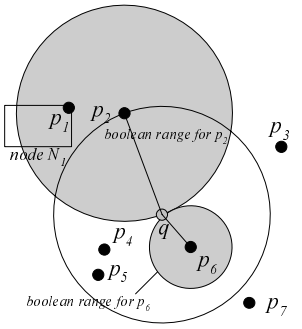


Figure 2.4: Illustration of SFT

Table 2.1 summarizes the properties of each algorithm. As discussed before, pre-computation methods cannot efficiently handle updates. MVZ is suitable only to 2D spaces, while SAA is practically inapplicable for 3 or more dimensions. SFT incurs false misses, the number of which depends on the parameter K: a large value of K decreases the false misses but increases significantly the processing cost.

	dynamic data	arbitrary dimensionality	exact result
KM,YL	No	Yes	Yes
MVZ	No	No	Yes
SAA	Yes	No	Yes
SFT	Yes	Yes	No

Table 2.1: Summary of algorithm properties

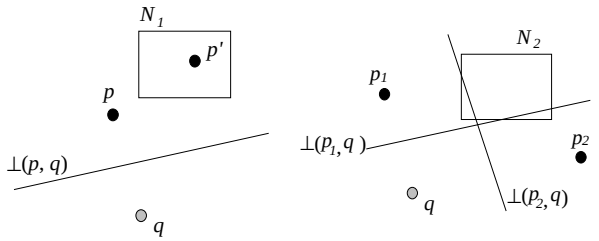
Regarding the applicability of the existing algorithms to arbitrary values of k, pre-computation methods only support a specific value (typically equal to 1), used to determine the vicinity circles. SFT can be adapted for retrieval of RkNN by setting a large value of K (>>k) and replacing the boolean with *count* queries (that return the number of objects in the query range instead of their actual ids). The extension of SAA to arbitrary k has not been studied before, but we will discuss it in Section 4.3. In the rest of the paper, we propose algorithms that return the exact results for dynamic datasets of any dimensionality. We start with single (i.e., k=1) RNN queries in Section 3, before proceeding to arbitrary values of k in Section 4.

3. SINGLE RNN PROCESSING

Section 3.1 illustrates some problem characteristics that permit the development of efficient algorithms presented in Section 3.2. Section 3.3 analyzes the performance of the proposed techniques with respect to existing methods.

3.1 Problem Characteristics

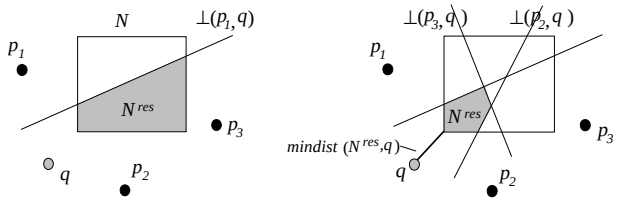
Consider the perpendicular bisector $\perp(p,q)$ between the query q and an arbitrary data point p as shown in Figure 3.1a. The bisector divides the data space into two half-planes: $PL_q(p,q)$ that contains q, and $PL_p(p,q)$ that contains p. Any point (e.g., p') in $PL_p(p,q)$ cannot be a RNN of q because it is closer to p than q. Similarly, a node MBR (e.g., N_1) that falls completely in $PL_p(p,q)$ cannot contain any candidate. In some cases, the pruning of an MBR requires multiple half-planes. For example, in Figure 3.1b, although N_2 does not fall completely in $PL_{p_1}(p_1,q)$ or $PL_{p_2}(p_2,q)$, it can still be pruned since it lies entirely in the *union* of the two half-planes.



(a) Pruning with one point (b) Pruning with two points

Figure 3.1: Illustration of half-plane pruning

In general, if p_1, p_2, …, pn_c are n_c data points, then any node whose MBR falls inside $\cup_{i=1\sim}n_c PL_{p_i}(p_i,q)$ cannot contain any RNN result. Let the *residual region* N^{res} be the area of node N outside $\cup_{i=1\sim}n_c PL_{p_i}(p_i,q)$ (i.e., the part of the MBR that may contain candidate RNNs of q). Then, N can be pruned if and only if $N^{res}=\emptyset$. Typically, N^{res} is a convex polygon bounded by the edges of N and the bisectors $\perp(p_i,q)$ ($1\le i\le n_c$). Consider Figure 3.2a that contains n_c=3 data points p_1, p_2, p_3. We can compute the residual region N^{res} by *trimming* N with each bisector in turn. Specifically, initially we set $N^{res}=N$ and use $\perp(p_1,q)$, after which N^{res} becomes the shaded trapezoid. In general, trimming with $\perp(p_i,q)$ reduces the *previous* N^{res} to the region inside the half-plane $PL_q(p_i,q)$. Figure 3.2b shows the final N^{res} after processing all bisectors. Given p_1, p_2 and p_3, N^{res} is the only part of the node MBR N that may contain RNNs of q.

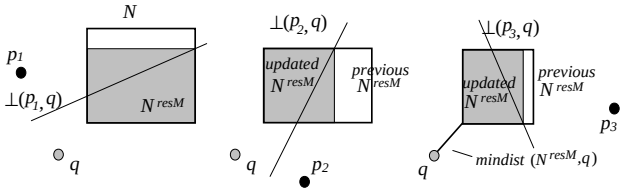


(a) After processing $\perp(q,p_1)$ (b) The final polygon

Figure 3.2: Computing the residual region

The above computation of N^{res} has two problems. First, in the worst case, each bisector may introduce an additional vertex to N^{res}. Consequently, the trimming of the i-th ($1\le i\le n_c$) bisector takes $O(i)$ time because it may need to examine all edges in the previous N^{res}. Thus, the total processing cost is $O(n_c^2)$, i.e., quadratic to the number of half-planes. Second, this method does not scale with the dimensionality because computing the intersection of a

half-space and a hyper-polyhedron becomes increasingly complex [BKOS97]. Motivated by this, we propose a simpler alternative that requires only $O(n_c)$ time. The idea is to bound N^{res} by a *residual* MBR N^{resM}. Figure 3.3 illustrates the residual MBR computation using the example in Figure 3.2. Initially N^{resM} is set to N and then it is trimmed incrementally by each bisector. Figure 3.3a shows trimming with $\perp(p_1,q)$, where, instead of keeping the exact shape of N^{res}, we compute N^{resM} (i.e., the shaded rectangle). In general, bisector $\perp(p_i,q)$ updates N^{resM} to the MBR of the region in the previous N^{resM} that is in $PL_q(p_i,q)$. Figures 3.3b, 3.3c illustrate the residual MBRs after trimming with $\perp(p_2,q)$, $\perp(p_3,q)$, respectively. Note that the final N^{resM} is not necessarily the MBR of the final N^{res} (compare Figure 3.3c and Figure 3.2b). Trimmed MBRs can be efficiently computed (in arbitrary dimensionality) using the *clipping* algorithm of [GRSY97].



(a) After $\perp(p_1,q)$ (b) After $\perp(p_2,q)$ (c) After $\perp(p_3,q)$

Figure 3.3: Computing the residual MBR

Figure 3.4 presents the pseudo-code for the above approximate trimming algorithm. If N^{resM} exists, *trim* returns the minimum distance between q and N^{resM}; otherwise, it returns ∞. Since N^{resM} always encloses N^{res}, $N^{resM} = \emptyset$ necessarily implies that $N^{res} = \emptyset$. This property guarantees that *pruning is "safe", meaning that trim never eliminates a node that may contain candidates*. The algorithm also captures points as MBRs with zero extents. In this case it will return the actual distance between the point and the query (if the point falls in the half-space of the query), or ∞ otherwise.

Algorithm trim (q, { p_1, p_2, …, pn_c}, N)
/* q is the query point, p_1, p_2, …, pn_c are arbitrary data points, and N is a rectangle being trimmed*/
1. $N^{resM} = N$
2. for i=1 to n_c // consider each data point in turn
3. N^{resM} =*clipping*(N^{resM}, PLq(p_i,q)) //algorithm of [GRSY97]
4. if $N^{resM} = \emptyset$ then return ∞
5. return *mindist*(N^{resM}, q)

End trim

Figure 3.4: The *trim* algorithm

An interesting question is: if $N^{resM} \neq \emptyset$, can N^{res} be $\emptyset$? (i.e., *trim* fails to prune an MBR that can be discarded). Interestingly, it turns out that the answer is negative in the 2D space, as illustrated in the next lemma (which proves an even stronger result):

Lemma 1: Given a 2D query q, an arbitrary number of half-planes and a node N, the residual MBR N^{resM} of N returned by *trim* exists if and only if N^{res} exists. Furthermore, if $N^{resM} \neq \emptyset$, *mindist*(N^{res},q) = *mindist*(N^{resM}, q), where N^{res} is the residual region of N.

In other words, the residual MBR N^{resM} preserves the minimum distance between N^{res} and q (compare *mindist*(N^{res},q) and *mindist*(N^{resM},q) in Figures 3.2b and 3.3c, respectively). It is worth mentioning that the lemma does not hold for arbitrary half-planes (the half-planes in RNN are constrained to contain q). Further, the lemma does not apply to arbitrary dimensionality. However, as mentioned earlier, we can still use *trim* to safely eliminate MBRs that do not contain candidates.

3.2 The TPL Algorithm

Based on the above discussion, we adopt a two-step framework that retrieves a set of candidate RNNs (*filtering step*) and then removes the false misses (*refinement step*). As opposed to SAA and SFT that require multiple queries for each step, the filtering and refinement processes are combined into a single traversal of the R-tree. In particular, our algorithm (hereafter, called TPL) traverses the R-tree in a best-first manner (see Section 2.1), retrieving potential candidates in ascending order of their distance to the query point q because the RNNs are likely to be near q. The concept of half-planes (half-spaces in high dimensions) is used to prune node MBRs (data points) that cannot contain (be) candidates. Each pruned entry is inserted in a *refinement set* S_{rfn}. In the refinement step, the entries of S_{rfn} are used to eliminate false hits. Next we discuss TPL using the example of Figure 3.5, which shows a set of data points (numbered in ascending order of their distance from the query) and the corresponding R-tree (the content of some nodes is omitted for clarity). The query result contains only point p_5.

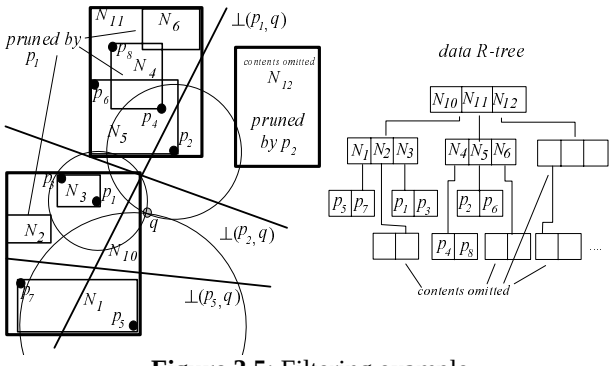


Figure 3.5: Filtering example

Initially, the algorithm visits the root of the R-tree and inserts its entries N_{10}, N_{11}, N_{12} into a heap H sorted on their *mindist* from q. Then it de-heaps N_{10}, visits its child node and inserts into H the corresponding entries: H = {N_3, N_{11}, N_2, N_1, N_{12}}. The next node accessed is N_3, where the first point p_1 (i.e., the one closest to q) has *dist*(p_1,q)< *dist*(N_{11},q) (N_{11} is at the top of the heap) and is added to the candidate set S_{cnd}. The second point p_3 in N_3

lies in $PL_{p_1}(p_1,q)$ (i.e., it cannot be a RNN of q) and is inserted into the refinement set S_{rfn}. In general, any point or node examined during the filter step is not discarded because it may influence (i.e., be a NN of) some candidate. In this example, p_3 will invalidate p_1 (during the refinement step) because $dist(p_1,p_3) < dist(p_1,q)$.

The next de-heaped entry is N_{11}. *Trim* checks if N_{11} can be pruned. Since part of N_{11} lies in $PL_q(p_1,q)$, it has to be visited. Its child nodes N_4 and N_6 fall completely out of $PL_q(p_1,q)$. Therefore, they cannot contain any candidates and are added to S_{rfn}. On the other hand, N_5 falls partially in $PL_q(p_1,q)$, i.e., *trim* will return a $mindist(N_5^{resM},q)$ that is different from ∞. Thus, N_5 is inserted into H together with its $mindist(N_5^{resM},q)$. The rationale of this choice, instead of $mindist(N_5,q)$, is that since our aim is to discover candidates according to their proximity to q, the node visiting order should not take into account the part of the node that cannot contain candidates. Assuming that $mindist(N_5^{resM},q) < mindist(N_2,q)$, N_5 is at the top of H and immediately de-heaped. Inside N_5, point p_2 is added to $S_{cnd}=\{p_1,p_2\}$ and p_6 to $S_{rfn}=\{p_3,N_4,N_6,p_6\}$. The next heap entry N_2 lies in $PL_{p_1}(p_1,q)$ and is added to S_{rfn}, without being visited. On the other hand, part of N_1 lies in $PL_q(p_1,q)$ and is accessed, leading to $S_{cnd} = \{p_1,p_2,p_5\}$ and $S_{rfn} = \{p_3,N_4,N_6,N_2,p_6,p_7\}$. Finally, N_1 is also inserted into S_{rfn} as it lies completely in $PL_{p_2}(p_2,q)$. The filtering step terminates when $H=\emptyset$.

The contents of the heap at each phase of the filtering process are shown in Table 3.1. Although omitted in the table, the heap entry for N also contains $mindist(N^{resM},q)$, if N has been trimmed, or $mindist(N,q)$, otherwise. In addition, the heap may include (non-pruned) data points (for simplicity, in the example we assumed that such points were processed immediately).

Action	Heap	S_{cnd}	S_{rfn}
visit root	$\{N_{10},N_{11},N_{12}\}$	$\emptyset$	$\emptyset$
visit N_{10}	$\{N_3,N_{11},N_2,N_1,N_{12}\}$	$\emptyset$	$\emptyset$
visit N_3	$\{N_{11},N_2,N_1,N_{12}\}$	$\{p_1\}$	$\{p_3\}$
visit N_{11}	$\{N_5,N_2,N_1,N_{12}\}$	$\{p_1\}$	$\{p_3,N_4,N_6\}$
visit N_5	$\{N_2,N_1,N_{12}\}$	$\{p_1,p_2\}$	$\{p_3,N_4,N_6,p_6\}$
process N_2	$\{N_1,N_{12}\}$	$\{p_1,p_2\}$	$\{p_3,N_4,N_6,p_6,N_2\}$
visit N_1	$\{N_{12}\}$	$\{p_1,p_2,p_5\}$	$\{p_3,N_4,N_6,p_6,N_2,p_7\}$
visit N_{12}	$\emptyset$	$\{p_1,p_2,p_5\}$	$\{p_3,N_4,N_6,p_6,N_2,p_7,N_{12}\}$

Table 3.1: Heap contents during filtering

Figure 3.6 illustrates the pseudo-code for the filtering step. Note that *trim* is applied twice for each node N; when N is inserted into the heap and when it is de-heaped. The second test is necessary, because N may be pruned by some candidate that was discovered after N's insertion into H. Similarly, when a leaf node is visited, its non-pruned points are inserted into H (instead of S_{cnd}) and processed in ascending order of their distance to q. Although this may increase the heap size (and the CPU cost of heap operations), it maximizes the chance that some points will be subsequently pruned by not-yet discovered candidates that are closer to the query, hence reducing the size of S_{cnd} (and the cost of the subsequent refinement step).

```
Algorithm TPL-filter(q) /* q is the query point */
1.  initialize a min-heap H accepting entries of the form (e, key)
2.  initialize sets S_cnd=∅, S_rfn=∅
3.  insert (R-tree root, 0) to H
4.  while H is not empty
5.    (e, key)=de-heap H
6.    if (trim(q, S_cnd, e)=∞) then S_rfn=S_rfn∪{e}
7.    else // entry may be or contain a candidate
8.      if e is data point p
9.        S_cnd=S_cnd∪{p}
10.     else if e points to a leaf node N
11.       for each point p in N (sorted on dist(p,q))
12.         if (trim(q,S_cand,p)≠∞) then insert (p,dist(p,q)) in H
13.         else S_rfn=S_rfn∪{p}
14.     else // e points to an intermediate node N
15.       for each entry N_i in N
16.         mindist(N_i^resM, q)=trim(q, S_cnd, N_i)
17.         if (mindist(N_i^resM, q)=∞) then S_rfn=S_rfn∪{N_i}
18.         else insert (N_i, mindist(N_i^resM, q)) in H
End TPL-filter
```

Figure 3.6: TPL filtering algorithm

After the termination of the filter step we have a set S_{cnd} of candidates and a set S_{rfn} of node MBRs or data points. Let $P_{rfn} \subseteq S_{rfn}$ be the set of points and $N_{rfn} \subseteq S_{rfn}$ be the set of MBRs in S_{rfn}. The refinement step is performed in *rounds*. Figure 3.7 shows the pseudo-code for each round, where we eliminate the maximum number of candidates from S_{cnd} without visiting additional nodes. Intuitively, a point $p \in S_{cnd}$ can be discarded as a false hit, if (i) there is a point $p' \in P_{rfn}$ such that $dist(p,p') < dist(p,q)$, or (ii) there is an node MBR $N \in N_{rfn}$ such that $minmaxdist(p,N) < dist(p,q)$ (i.e., N is guaranteed to contain a point p' such that $dist(p,p') < dist(p,q)$). For instance, in Figure 3.5, the first condition prunes p_1 because $p_3 \in P_{rfn}$ and $dist(p_1,p_3) < dist(p_1,q)$. Lines 2-9 prune false hits according to the above observations.

```
Algorithm refinement_round(q, S_cnd, P_rfn, N_rfn)
1. for each point p in S_cnd
2.     for each point p' in P_rfn
3.         if dist(p,p')< dist(p,q)
4.                 S_cnd= S_cnd -{p} //false hit
5.                 goto 1 //test next candidate
6.     for each node MBR N in N_rfn
7.         if minmaxdist(p,N)< dist(p,q)
8.                 S_cnd= S_cnd -{p} //false hit
9.                 goto 1 //test next candidate
10.    for each node MBR N in N_rfn
11.        if mindist(p,N)< dist(p,q) add N in toVisit(p)
12.    if (toVisit(p)= ∅)
13.                S_cnd= S_cnd -{p} and report p // actual result
End refinement_round
```

Figure 3.7: The *refinement_round* algorithm

On the other hand, a point $p \in S_{cnd}$ can be reported as an actual result without any extra node accesses, if (i) there is no point $p' \in P_{rfn}$ such that $dist(p,p') < dist(p,q)$ and (ii)

for every node $N \in N_{rfn}$: $mindist(p,N) \geq dist(p,q)$. In Figure 3.5, candidate p_5 satisfies these conditions and is removed from S_{cnd}. Each remaining point p in S_{cnd} (e.g., p_2) must undergo additional refinement rounds because there may exist points (p_4) in some not-yet visited nodes (N_4) that invalidate it. In this case, p requires accessing some nodes N such that $mindist(p,N) < dist(p,q)$, which are inserted in $toVisit(p)$, i.e., $toVisit(p)$ is the set of nodes that need to be visited before verifying p as a result.

Our next goal is to access the nodes of $toVisit(p)$ (for $p \in S_{cnd}$) in an order that achieves quick elimination of the remaining candidates. Continuing the running example, after the first round $S_{cnd} = \{p_2\}$ and the nodes that may contain NNs of p_2 are $toVisit(p_2)=\{N_4,N_{12}\}$. We choose to access a lowest level node first (in this case N_4), because it can achieve better pruning since it either encloses data points or MBRs with small extents (therefore the *minmaxdist* pruning is more effective). In case of a tie (i.e., multiple nodes of the same low level), we access the one that may prune the largest number of candidates.

If the node N to be visited is a leaf, then P_{rfn} contains only the data points in N, and N_{rfn} is set to $\emptyset$. Otherwise (N is an intermediate node), N_{rfn} contains only the child nodes of N, and P_{rfn} is $\emptyset$. In our example, the parameters for the second round are $S_{cnd} = \{p_2\}$, $P_{rfn} = \{$points of $N_4\}$ and $N_{rfn} = \emptyset$. Inside N_4, point p_4 eliminates p_2 and the algorithm terminates. Figure 3.8 shows the pseudo-code of the TPL refinement step. Lines 2-4 eliminate candidates that are closer to each other than the query point (i.e., similar to the second step of SFT). This test is required only once and therefore, is not included in *refinement_round* in order to avoid repeating it for every round.

Algorithm TPL-refinement (q, S_{cnd}, P_{rfn}, N_{rfn})
1. for each point p in S_{cnd}
2. for each other point $p' \neq p$ in S_{cnd}
3. if $dist(p,p') < dist(p,q)$
4. $S_{cnd} = S_{cnd} - \{p\}$; goto 1
5. if p is not eliminated initialize $toVisit(p)=\emptyset$
6. repeat
7. **refinement_round**(q, S_{cnd}, P_{rfn}, N_{rfn})
8. if ($S_{cnd}=\emptyset$) return // terminate
9. $P_{rfn}=N_{rfn}=\emptyset$ //initialization of next round
10. Let N be the lowest level node that appears in the largest number of sets $toVisit(p)$, where $p \in S_{cnd}$
11. remove N from all $toVisit(p)$ and access N
12. if N is a leaf node
13. $P_{rfn}=\{p / p \in N\}$ //P_{rfn} contains only the points of N
14. if N is an intermediate node
15. $N_{rfn}=\{N' / N' \in N\}$//$N_{rfn}$ contains the child nodes of N
End TPL-refinement

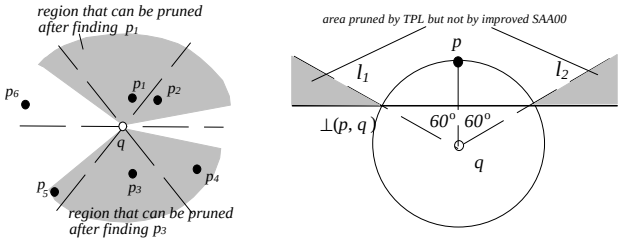
Figure 3.8: TPL refinement algorithm

In order to verify the correctness of TPL, observe that the filter step always retrieves a superset of the actual result (i.e., it does not incur false misses), since *trim* only prunes node MBRs (data points) that cannot contain (be) RNNs. Every false hit p is subsequently eliminated during the refinement step by comparing it with each data point retrieved during the filter step and each MBR that may potentially contain NNs of p. Hence, the algorithm returns the exact set of RNNs.

3.3 Discussion

TPL and the existing techniques that do not require pre-processing (SAA, SFT) are based on the filter-refinement framework. Interestingly, the two steps are independent in the sense that the filtering algorithms of one technique can be combined with the refinement mechanisms of another. For instance, the concept of boolean ranges of SFT can replace the conventional NN queries in the second step of SAA and vice versa. In this section we show that, in addition to being more general, TPL is more effective than SAA and SFT in terms of both filtering and refinement, i.e., it retrieves fewer candidates and eliminates false hits with lower cost.

In order to compare the efficiency of our filtering step with respect to SAA, we first present an improvement of that method. Consider the space partitioning of SAA in Figure 3.9a and the corresponding NNs in each partition (points are numbered according to their distance from q). Since the angle between p_1 and p_2 is smaller than 60 degrees and p_2 is farther than p_1, point p_2 cannot be a RNN of q. In fact, the discovery of p_1 (i.e., the first NN of the query) can prune all the points lying in the region $\nabla(p_1)$ extending 60 degrees on both sides of line segment qp_1 (upper shaded region in Figure 3.9a). Based on this observation, we only need to search for other candidates outside $\nabla(p_1)$. Let p_3 be the next NN of q in the constrained region of the data space (i.e., not including $\nabla(p_1)$). Similar to p_1, p_3 prunes all the points in $\nabla(p_3)$. The algorithm terminates when the entire data space is pruned. Although the maximum number of candidates is still six (e.g., if all candidates lie on the boundaries of the 6 space partitions), in practice it is smaller (in this example, the number is 3, i.e., p_1, p_3 and p_6).



(a) Improved SAA (b) Pruning comparison

Figure 3.9: Superiority of TPL over SAA

Going one step further, the filter step of TPL is even more efficient than that of the improved SAA. Consider Figure 3.9b where p is the NN of q. The improved SAA prunes the region $\nabla(p)$ bounded by rays l_1 and l_2. On the other hand, our algorithm prunes the entire half-plane $PL_p(p,q)$, which *includes* $\nabla(p_1)$ except for the part below $\perp(p,q)$.

Consider the circle centered at q with radius $dist(p,q)$. It can be easily shown that the circle crosses the intersection point of $\perp(p,q)$ and l_1 (l_2). Note that all the nodes intersecting this circle have already been visited in order to find p (a property of our filter step and all BF NN algorithms in general). In other words, *all the non-visited nodes that can be pruned by* $\nabla(p)$ *can also be pruned by* $PL_p(p,q)$. As a corollary, the maximum number of candidates retrieved by TPL is also bounded by the dimensionality (i.e., $|S_{cnd}| \leq 6$ is 2D space). Further, TPL supports arbitrary dimensionality in a natural way, since it does not make any assumption about the number or the shape of space partitions (as opposed to SAA).

The comparison with the filtering step of SFT depends on the value of K, i.e., the number of NNs of q that constitute the candidate set. Assume that in Figure 3.5 we know in advance that the actual RNNs of the query (in this case p_5) are among the K=5 NNs of q. SFT would perform a 5NN query and insert all the retrieved points p_1 , .., p_5 to S_{cnd}, whereas TPL inserts only the non-pruned points $S_{cnd} = \{p_1,p_2,p_5\}$. Furthermore, the number of candidates in TPL is bounded by the dimensionality, while the choice of K in SFT is arbitrary and does not provide any guarantees about the quality of the result. Consider, for instance, the (skewed) dataset and query point of Figure 3.10. A high value of K will lead to the retrieval of numerous false hits (e.g., data points in partition S_1), but no actual reverse nearest neighbors of q. The problem becomes more serious in higher dimensionality.

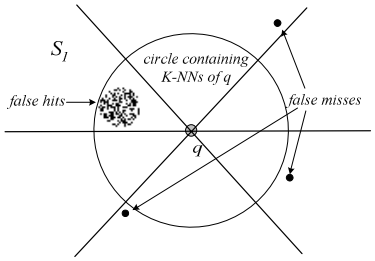


Figure 3.10: False hits and misses of SFT

One point worth mentioning is that although TPL is expected to retrieve fewer candidates than SAA and SFT, this does not necessarily imply that it incurs fewer node accesses during the filter step. For instance, assume that the query point q lies within the boundary of a leaf node N (in fact, q can be a data point) and all six candidates of SAA are in N. Then, as suggested in [SAA00] the NN queries can be combined in a single tree traversal, which can potentially find all these candidates by following a single path from the root to N. A similar situation may occur with SFT if all K NNs of q are contained in the same leaf node. On the other hand, the node accesses of TPL depend on the relative position of the candidates and the resulting half-planes. Nevertheless, the small size of the candidate set reduces the cost of the refinement step since each candidate must be verified.

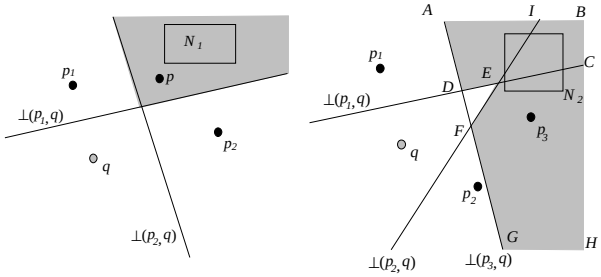
Regarding the refinement step, it suffices to compare TPL with SFT, since boolean ranges are more efficient than the conventional NN queries of SAA. Although Singh et al. [SFT03] propose some optimization techniques for minimizing the number of node accesses, a boolean range may still access a node that has already been visited during the filter step or by a previous boolean query. On the other hand, the seamless integration of the filtering and refinement steps in TPL (i) re-uses information about the nodes visited during the filter step, and (ii) eliminates multiple accesses to the same node. In other words, a node is visited at most once. This integrated mechanism can also be applied to the methodologies of SAA and SFT. In particular, all the nodes and points eliminated by the filter step (constrained NN queries in SAA, a KNN query in SFT) are inserted in S_{rfn} and our refinement algorithm is performed directly (instead of NN queries or boolean ranges).

4. RKNN PROCESSING

Section 4.1 presents properties that permit pruning of the search space for arbitrary values of k and Section 4.2 extends our methods for this problem. Finally, 4.3 discusses the application of previous techniques.

4.1 Problem Characteristics

The half-plane pruning strategy of Section 3.1 extends to arbitrary values of k. Figure 4.1a shows an example with k=2, where the shaded region corresponds to the intersection $\text{PL}_{p_1}(p_1,q) \cap \text{PL}_{p_2}(p_2,q)$. Point p is not a R2NN of q, since both p_1 and p_2 are closer to it than q. Similarly, a node MBR inside the shaded area cannot contain any candidate (i.e., N_1 can be pruned at the filter step). In some cases, several half-plane intersections are needed to prune a node. Assume the R2NN query q and the three data points of Figure 4.1b. Each pair of points generates an intersection of half-planes: (i) $\text{PL}_{p_1}(p_1,q) \cap \text{PL}_{p_2}(p_2,q)$ (i.e., polygon *IECB*), (ii) $\text{PL}_{p_1}(p_1,q) \cap \text{PL}_{p_3}(p_3,q)$ (*ADCB*) and (iii) $\text{PL}_{p_2}(p_1,q) \cap \text{PL}_{p_3}(p_3,q)$ (*IFGHB*). The shaded region is the *union* of these 3 intersections (i.e., $IECB \cup ADCB \cup IFGHB$). A node MBR (e.g., N_2) inside this region can be pruned, although it is not totally covered by any individual intersection area.



(a) $\text{PL}_{p_1}(p_1,q) \cap \text{PL}_{p_2}(p_2,q)$ (b) All intersection pairs

Figure 4.1: Examples of R2NN queries

In the general case, assume a R*k*NN query and $n_c \geq k$ data points $p_1, p_2, \ldots, pn_c$ (e.g., in Figure 4.1b n_c=3 and k=2). Let $\{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ be a subset of $\{p_1, p_2, \ldots, pn_c\}$. Each subset prunes the intersection area $\cap_{i=1\sim k}\text{PL}_{\sigma_i}(\sigma_i,q)$. The entire region that can be eliminated corresponds to the union of all $\binom{n_c}{k}$ intersections. Given a node N under consideration, the *k-trim* algorithm (Figure 4.2) computes, for each subset (in turn), the residual MBR N^{resM} of N. If at some point N^{resM} becomes $\emptyset$, it prunes N and terminates. Otherwise ($N^{resM} \neq \emptyset$), it continues with the next subset, by setting $N=N^{resM}$. Similar to *trim*, for qualifying nodes *k-trim* returns the minimum distance between q and N^{resM}; for the eliminated ones it returns ∞. The only complication concerns the computation of $\cap_{i=1\sim k}\text{PL}_{\sigma_i}(\sigma_i,q)$ in line 3. For this, we use a variant of [GRSY97], which returns a conservative approximation of the intersection region for each subset.

Algorithm *k*-trim (q, k, $\{p_1, p_2, \ldots, pn_c\}$, N)
/* $p_1, p_2, \ldots, pn_c$ are candidate data points, $n_c \geq k$ */
1. $N^{resM}=N$
2. for i=1 to $\binom{n_c}{k}$ //consider each subset in turn
   //assume the subset is $\{\sigma_1, \sigma_2, \ldots, \sigma_k\}$
3. N^{resM}= *clipping*(N^{resM}, $\cap_{i=1\sim k}\text{PL}_{\sigma_i}(\sigma_i,q)$)
4. if $N^{resM} = \emptyset$ then return ∞
5. return *mindist*(N^{resM}, q)
End *k*-trim

Figure 4.2: The *k-trim* algorithm

Examining all $\binom{n_c}{k}$ subsets is prohibitive for large k and S_{cnd}. In order to reduce the cost, we can restrict the number of inspected subsets using the following heuristic. We sort all the candidates in S_{cnd} according to their Hilbert values. Let the sorted order be $\{p_1, p_2, \ldots, pn_c\}$. Then, *k-trim* examines only the n_c subsets $\{p_1, \ldots, p_k\}$, $\{p_2, \ldots, p_{k+1}\}$, …, $\{pn_c, \ldots, p_{k-1}\}$. The rationale of this choice is that points close to each other tend to produce intersections with large areas. The trade-off is that this method may increase the number of candidates, since it fails to prune nodes that can be pruned by some non-inspected subset. In any case, as with *trim*, pruning with *k-trim* is always safe, meaning that it will never eliminate nodes that potentially contain candidates.

4.2 The *k*-TPL Algorithm

The filtering step of *k*-TPL follows exactly that of the TPL algorithm in Figure 3.6. Specifically, at the beginning, *k*-TPL uses BF traversal to locate a set of candidates close to the query q. After the size of S_{cnd} reaches k, *k-trim* prunes nodes (data points) that cannot contain (be) candidates. The pruned nodes and points are kept in S_{rfn}. The refinement step is more complex because a candidate p can only be pruned if we find k points within distance *dist*(p,q) from p. Thus, we associate p with a counter (initially set to k), and decrease it every time we find such a point. We can eliminate p as a false hit, when its counter becomes 0. The *minmaxdist* pruning cannot be applied in this case, because even if *minmaxdist*(p,N) < *dist*(p,q), we do not know how many points in N are within distance *dist*(p,q) from N, unless we visit the node. Instead, we use the *maxdist* and the minimum cardinality of N, i.e., the smallest possible number of points in N, given the minimum node utilization (typically, 40% for R-trees) and the level of N. In particular, a candidate p can be pruned if *maxdist*(p,N) < *dist*(p,q) and *min_card*(N) ≥ counter(p). Figure 4.3 shows the pseudo-code for *refinement_round* in the case of R*k*NN. The main refinement algorithm is similar to the one shown in Figure 3.7 and omitted.

Algorithm *k*-refinement_round(q, S_{cnd}, P_{rfn}, N_{rfn})
1. for each point p in S_{cnd}
2. for each point p' in P_{rfn}
3. if *dist*(p,p')< *dist*(p,q)
4. counter(p)--
5. if counter(p)=0
6. S_{cnd}= S_{cnd} -{p} //false hit
7. goto 1 //test next candidate
8. for each node MBR N in N_{rfn}
9. if *maxdist*(p,N)<*dist*(p,q) and *min_card*(N)≥ counter(p)
10. S_{cnd}= S_{cnd} -{p}; goto 1 //test next candidate
11. for each node MBR N in N_{rfn}
12. if *mindist*(p,N)< *dist*(p,q) add N in set *toVisit*(p)
13. if (*toVisit*(p)=$\emptyset$)
14. S_{cnd}= S_{cnd} -{p} and report p // actual result
End *k*-refinement_round

Figure 4.3: The *k-refinement_round* algorithm

4.3 Discussion

Although SAA was originally proposed for single RNN retrieval, it can be extended to arbitrary values of k based on the following lemma:

Lemma 2: Given a 2D R*k*NN query q, divide the space around q into 6 equal partitions using 6 rays emanating from q, such that each partition is bounded by two rays. Then, the k NNs of q in each partition are the only possible results of q. Further, in the worst case, all these points may be the actual results.

As a corollary, the maximum number of reverse k NNs of q in 2D space equals 6k. Figure 4.4 illustrates the lemma using an example with k=2. The candidates of q include $\{p_1, p_2, p_4, p_5, p_6\}$; p_3 (i.e., the 3rd NN in S_2) cannot be a candidate, since p_1 and p_2 are both closer to it than q. Based on Lemma 2, the filtering step of SAA executes 6 constrained *k*NN queries in each partition. Then, the refinement step verifies or eliminates each of the 6k candidates by performing a *k*NN query. This approach, however, has the same problem as the original SAA, i.e. the number of partitions to be searched increases exponentially with the dimensionality.

As mentioned in Section 2, SFT can be adapted for R*k*NN by setting a large value of K (>>k). Nevertheless, the concept of boolean ranges cannot be applied for

arbitrary values of k for the same reason that *minmaxdist* pruning cannot be applied in our *k-refinement_round* algorithm. Thus, Singh et al. [SFT03] suggest performing the *count* query, which decides if there are at least k points inside the query range. Similar to boolean range queries, count queries may also access the same node multiple times, which is avoided in *k*-TPL.

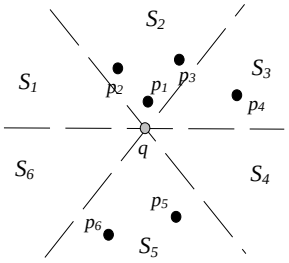


Figure 4.4: Example of Lemma 2

5. EXPERIMENTS

Our evaluation is performed using both real and synthetic data. In particular, we deploy the five real datasets summarized in Table 5.1. *LB*, *NA* and *LA* contain spatial data[3] corresponding to geometric locations in the Long Beach county, North America, and Los Angeles, respectively. *Wave*[4] is obtained from the measurements of wave directions at the National Buoy Center. *Color* includes data from the color histograms of 65k images. The synthetic data follow uniform and Zipf distributions. Their cardinalities range from 128k to 2048k (i.e., over 2 million points), and their dimensionalities vary from 2 to 5.

	LB	***NA***	***LA***	***Wave***	***Color***
dimensionality	2	2	2	3	4
cardinality	123k	569k	1314k	60k	65k

Table 5.1: Statistics of the real datasets used

Each dataset is indexed by an R*-tree [BKSS00] with node (disk page) size of 1k bytes. The capacity (i.e., the maximum number of entries in a node) equals 50, 36, 28 and 23 entries, for 2, 3, 4, and 5 dimensions, respectively. TPL is compared with SAA (for 2D data) and SFT, since as discussed in Section 2.2, these are the only methods applicable to dynamic data. For SAA, we implemented the optimization of [SAA00] that performs the six constraint NN queries (i.e., the filter step) with a single traversal of the R-tree. The filter step of SFT requires a KNN query, where K should be significantly larger than the number k of requested RNNs to avoid false misses[5]. In our experiments, we set K to $10 \cdot d \cdot k$, where d is the dimensionality of the dataset being examined, e.g., for single RNN in 2D space K=20.

The experiments investigate the effect of the following parameters: (i) data distribution, (ii) dataset cardinality, (iii) dimensionality d, and (iv) k (for RkNN). The reported results represent the average cost per query for a workload of 200 queries with the same parameters. The locations of the queries are uniformly generated in the data space. The cost includes both the I/O overhead (by charging 10ms for each node access) and CPU time. All the experiments are executed on a Pentium IV CPU at 2.4GHz with 512 Mbytes memory. Section 5.1 presents the results for single RNN retrieval, and Section 5.2 discusses RkNN.

5.1 Results of Single RNN Search

Figure 5.1 compares the performance of TPL, SAA and SFT using real datasets. The cost of each method is divided in two parts, corresponding to the filter and refinement steps. The number above each column indicates the average percentage of I/O time in the total query cost. For TPL, we also demonstrate (in brackets) the average number of candidates retrieved by the filter step. These numbers are fixed for SAA (6) and SFT ($10 \cdot d$) and omitted. SAA is not performed on *Wave* and *Color*, because the datasets have 3 and 4 dimensions, respectively.

Clearly, TPL is the best algorithm for all datasets. The similar performance of the filter steps is due to the fact that SAA and SFT perform a 6NN or 10dNN query, which, in general, visits a limited number of R-tree nodes around the query point. Similarly, the pruning of TPL is very effective in low dimensional spaces. On the other hand, the refinement step of TPL is much faster than that of the other algorithms. Recall that TPL integrates the filter and refinement steps to avoid accessing the same node twice. Further, due to the small number (at most 6.2) of candidates retrieved (by the filter step), the refinement is usually accomplished with only 1 or 2 node accesses. The refinement step of SAA performs 6 NN queries, which traverse the tree multiple times, thus incurring high overhead. The boolean ranges of SFT achieve better performance (than SAA) but are less effective than the refinement step of TPL. All the algorithms are I/O bounded.

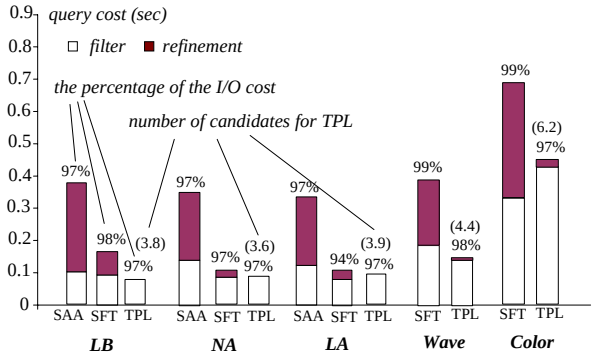
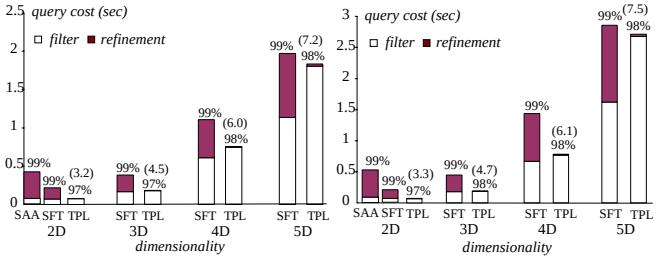


Figure 5.1: Costs for real datasets

Figure 5.2 shows the total cost of TPL and SFT as a function of the dimensionality for synthetic datasets with

[3] http://www.census.gov/geo/www/tiger/
[4] http://www.ndbc.noaa.gov/historical_data.shtml
[5] Singh et al. [SFT03], in their evaluation, used K=50 for single RNN retrieval.

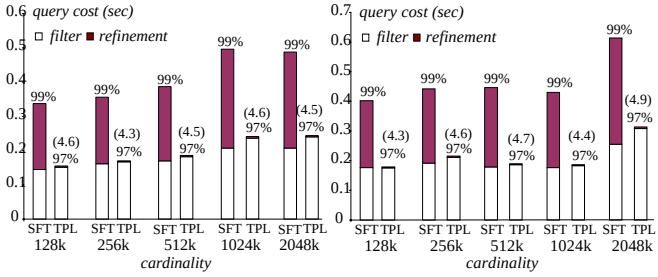
cardinality 512k. The performance of both algorithms degrades because, in general, R-trees become less efficient as the dimensionality grows [TS96] (due to the large overlap among the node MBRs). In addition, the number of potential candidates for TPL increases (see parenthesis on top of the TPL column), leading to higher cost, especially for the filter step. The data distribution does not have a significant effect on the performance; this observation is confirmed by all experiments (including the real data) despite the different settings.



(a) *Uniform* (b) *Zipf*

Figure 5.2: Cost vs. dimensionality (512k)

Figure 5.3 fixes the dimensionality to 3, and shows the query cost as a function of the dataset cardinality. TPL incurs around half the cost of SFT in all cases. The step-wise growth corresponds to an increase of the tree height. Specifically, for uniform data, the step occurs at 1024k, whereas for Zipf data at cardinality 2048k. The height increase has a similar effect on both algorithms.

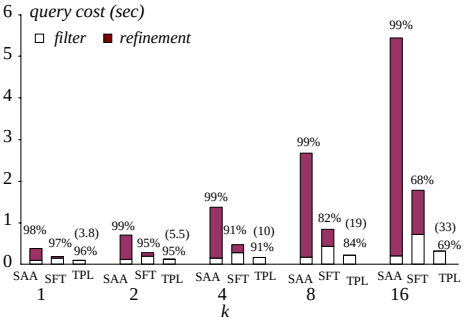


(a) *Uniform* (b) *Zipf*

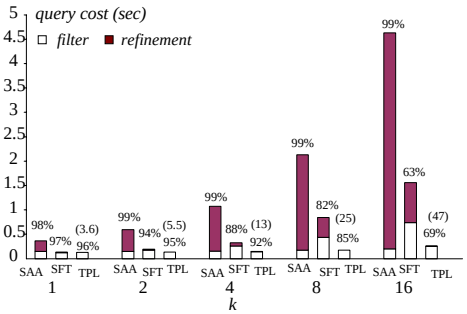
Figure 5.3: Cost vs. cardinality (3D)

5.2 Results of R*k*NN Search

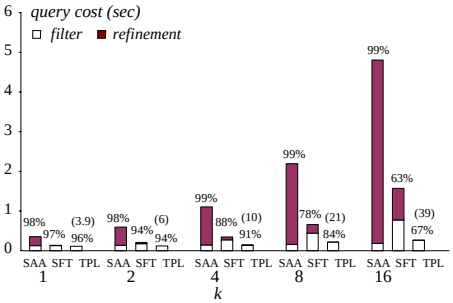
Having confirmed the superiority of TPL for single RNN retrieval, we proceed to evaluate its performance for R*k*NN queries. Our implementation of TPL applies the Hilbert heuristic discussed in Section 4.1 to reduce the number of inspected subsets. Figure 5.4 illustrates the performance of alternative algorithms as a function of k (in the range [1,16]) for real datasets. SAA here refers to the R*k*NN extension discussed in Section 4.3. Similar to the diagrams in the previous section, we also demonstrate the percentage of I/O costs, and the number of candidates for TPL. As expected, the overhead of each algorithm grows with k, due to the significant increase in CPU time (observe that the I/O percentage decreases with k). TPL again outperforms its competitors, and the difference increases with k. Note that for the 2D datasets, the average number of candidates retrieved by TPL is smaller than 4 for k=1 and increases almost linearly with k.



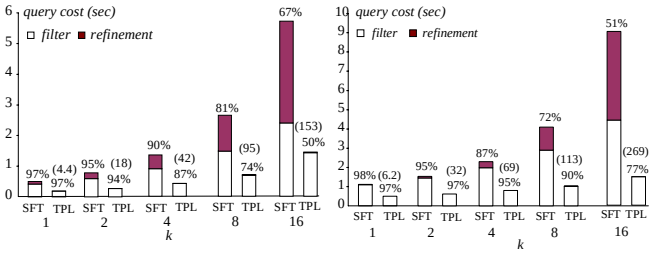
(a) *LB*



(b) *NA*



(c) *LA*

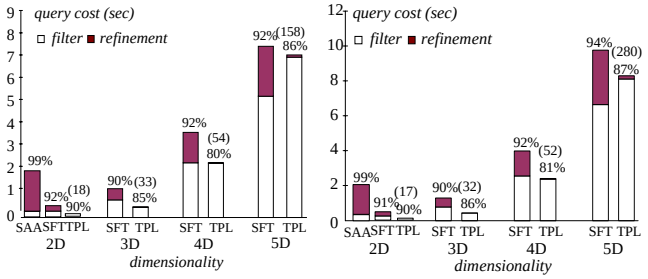


(d) *Wave* (e) *Color*

Figure 5.4: Cost vs. k (real data)

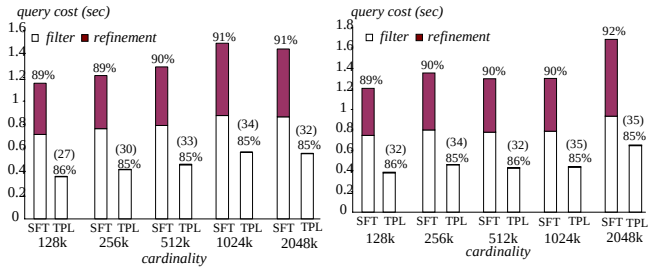
Next we explore the effects of dimensionality and cardinality using synthetic data. In the following experiments, k is fixed to its median value 4. Figure 5.5 shows the performance as a function of dimensionality for uniform (Figure 5.5a) and Zipf (Figure 5.5b) distributions, respectively (the cardinality is set to 512k). The diagrams and their explanations are similar to those in Figure 5.2.

The last set of experiments (Figure 5.6) illustrates the query overhead as a function of cardinality (the dimensionality equals 3), confirming the observations of Figure 5.3.



(a) *Uniform* (b) *Zipf*

Figure 5.5: Cost vs. dimensionality (k=4, 512k)



(a) *Uniform* (b) *Zipf*

Figure 5.6: Cost vs. cardinality (k=4, 3D)

6. CONCLUSIONS

The existing algorithms for RNN search are applicable only in restricted scenarios. This paper develops the first general methodology for retrieval of an arbitrary number of reverse nearest neighbors in multiple dimensions. In addition to its applicability and flexibility, our solution is better than the previous approaches also in terms of efficiency and scalability. An interesting direction for future work is to adapt the proposed methodology to other variations of RNN problems. Further, currently there does not exist any cost model for estimating the execution time of RNN techniques. The development of such a model will not only facilitate query optimization, but may also reveal new problem characteristics that could lead to even faster algorithms.

ACKNOWLEDGMENTS

This work was supported by grant HKUST 6180/03E from Hong Kong RGC. We would like to thank Kyriakos Mouratidis for proof-reading the paper.

REFERENCES

[BJKS02] Benetis, R., Jensen, C., Karciauskas, G., Saltenis, S. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. *IDEAS*, 2002.

[BKK96] Berchtold, S., Keim, D., Kriegel, H. The X-tree: An Index Structure for High-Dimensional Data. *VLDB*, 1996.

[BKOS97] Berg, M., Kreveld, M., Overmars, M., Schwarzkopf, O. Computational Geometry: Algorithms and Applications. ISBN 3-540-65620-0. *Springer*, 1997

[BKSS90] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[CF98] Cheung, K., Fu, A. Enhanced Nearest Neighbour Search on the R-tree. *SIGMOD Record* 27(3): 16-21, 1998.

[FSAA01] Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., Abbadi, A. Constrained Nearest Neighbor Queries. *SSTD*, 2001.

[G84] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD*, 1984.

[GRSY97] Goldstein, J., Ramakrishnan, R., Shaft, U., Yu, J. Processing Queries By Linear Constraints. *PODS*, 1997.

[H94] Henrich, A. A Distance Scan Algorithm for Spatial Access Structures. *ACM GIS*, 1994.

[HS99] Hjaltason, G., Samet, H. Distance Browsing in Spatial Databases. *TODS*, 24(2), 265-318, 1999.

[KM00] Korn, F., Muthukrishnan, S. Influence Sets Based on Reverse Nearest Neighbor Queries. *SIGMOD*, 2000.

[KMS02] Korn, F., Muthukrishnan, S. Srivastava, D. Reverse Nearest Neighbor Aggregates Over Data Streams. *VLDB*, 2002.

[LNY03] Lin, K., Nolen, M., Yang, C. Applying Bulk Insertion Techniques for Dynamic Reverse Nearest Neighbor Problems. *IDEAS*, 2003.

[MVZ02] Maheshwari, A., Vahrenhold, J., Zeh, N. On Reverse Nearest Neighbor Queries. *CCCG*, 2002.

[RKV95] Roussopoulos, N., Kelly, S., Vincent, F. Nearest Neighbor Queries. *SIGMOD*, 1995.

[SAA00] Stanoi, I., Agrawal, D., Abbadi, A., Reverse Nearest Neighbor Queries for Dynamic Databases. *SIGMOD* Workshop on Research Issues in Data Mining and Knowledge Discovery, 2000.

[SFT03] Singh, A., Ferhatosmanoglu, H., Tosun, A. High Dimensional Reverse Nearest Neighbor Queries. *CIKM*, 2003.

[SRAA01] Stanoi, I., Riedewald, M., Agrawal, D., Abbadi, A. Discovery of Influence Sets in Frequently Updated Databases. *VLDB*, 2001.

[TS96] Theodoridis, Y., Sellis, T. A Model for the Prediction of R-tree Performance. *PODS*, 1996.

[YL01] Yang, C., Lin, K. An Index Structure for Efficient Reverse Nearest Neighbor Queries. *ICDE*, 2001.