The VLDB Journal (2007) 16: 293–316
DOI 10.1007/s00778-005-0168-2

REGULAR PAPER

Yufei Tao · Dimitris Papadias · Xiang Lian · Xiaokui Xiao

Multidimensional reverse *k*NN search

Received: 29 November 2004 / Accepted: 8 July 2005 / Published online: 20 December 2005

Abstract Given a multidimensional 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: they (i) do not support arbitrary values of k, (ii) cannot deal efficiently with database updates, (iii) are applicable only to 2D data but not to higher dimensionality, and (iv) retrieve only approximate results. Motivated by these shortcomings, we develop algorithms for *exact* R*k*NN processing 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. As a second step, we extend the proposed techniques to *continuous* R*k*NN search, which returns the R*k*NN results for every point on a line segment. We evaluate the effectiveness of our algorithms with extensive experiments using both real and synthetic datasets.

Keywords Reverse nearest neighbor · Continuous search · Spatial database

1 Introduction

Given a multidimensional dataset P and a point $q \notin P$, a *reverse k nearest neighbor* (R*k*NN) query retrieves all the points $p \in P$ which have q as one of their k nearest neighbors (NN) [10]. Formally, $RkNN(q) = \{p \in P \mid dist(p, q) < dist(p, p')\}$, where *dist* is a distance metric (we assume Euclidean distance), and p' the k-th farthest NN of p in P. Figure 1 shows a dataset with 4 points $p_1, p_2, \ldots, p_4$, where each point is associated with a circle covering its two NNs (e.g., the circle centered at p_4 encloses p_2 and p_3). The result of a R2NN query q includes the "owners" (i.e., p_3, p_4) of the circles that contain q. Let $kNN(q)$ be the set of k nearest neighbors of q. Note that $p \in kNN(q)$ does not necessarily imply $p \in RkNN(q)$, and vice versa. For instance, $2NN(q) = \{p_1, p_3\}$, but p_1 does not belong to $R2NN(q)$. On the other hand, although $p_4 \in R2NN(q)$, it is not in $2NN(q)$.

R*k*NN search is important both as a stand-alone query in Spatial Databases, and a component in applications involving profile-based marketing. For example, assume that the points in Fig. 1 correspond to records of houses on sale, and the two dimensions capture the size and price of each house. Given a new property q on the market, the real estate company wants to notify the customers potentially interested in q. An effective way is to retrieve the set $RkNN(q)$, and then contact the customers that have previously expressed interest in $p \in RkNN(q)$. Note that a RNN query is more appropriate than NN search, since $RkNN(q)$ is determined by the neighborhood of each data point p and not strictly by the distance between q and p. For instance, in Fig. 1, although p_4 is farther from q than p_1, customers interested in p_4 may be more attracted to q (than those of p_1) because they have fewer options matching their preferences. Clearly, the discussion applies to space of higher (>2) dimensionality, if more factors (e.g., security rating of the neighborhood, etc.) affect customers' decisions.

R*k*NN processing has received considerable attention [2, 10, 12, 13, 15, 16, 20] in recent years. As surveyed in Sect. 2, however, all the existing methods have at least one of the following deficiencies: they (i) do not support arbitrary values of k, (ii) cannot deal efficiently with database updates, (iii) are applicable only to 2D data but not to higher dimensionality, and (iv) 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 these shortcomings, we develop *dynamic* algorithms (i.e., supporting updates) for *exact* processing of R*k*NN queries with *arbitrary* values of k on *multidimensional* datasets. Our methods are based on a data-partitioning index (e.g., R-trees [1],

Y. Tao (✉) · X. Xiao
City University of Hong Kong, Tat chee Avenue, Hong Kong
E-mail: {taoyf, xkxiao}@cs.cityu.edu.hk

D. Papadias · X. Lian
Hong Kong University of Science and Technology, Clear Water Bay Hong Kong
E-mail: {dimitris, xlian}@cs.cityu.edu.hk

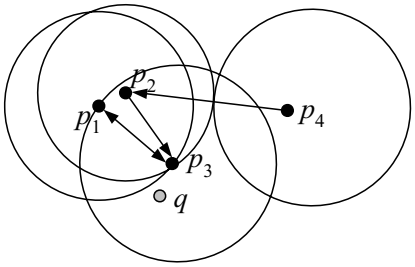


Fig. 1 2NN and R2NN examples

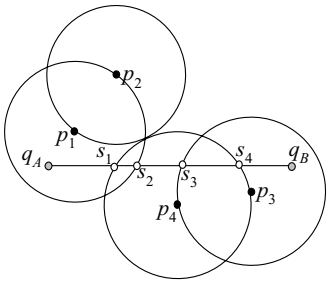


Fig. 2 A continuous RNN query

X-trees [3]), and do not require any preprocessing. Similar to the existing algorithms, 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 avoids multiple accesses to the same index node (i.e., each node is visited at most once).

As a second step, we extend our methodology to *continuous reverse k nearest neighbor* (C-RkNN) search, which retrieves the RkNNs of every point on a query segment q_Aq_B. Interestingly, although there are infinite points on q_Aq_B, the number of distinct results is finite. Specifically, the output of a C-RkNN query contains a set of $<R, T>$ tuples, where R is the set of RkNNs for (all the points in) the segment $T \subseteq q_Aq_B$. In Fig. 2, for instance, the C-RNN query returns $\{<\{p_1\}, [q_A, s_1)>, <\{p_1, p_4\}, [s_1, s_2)>, <\{p_4\}, [s_2, s_3)>, <\{p_3, p_4\}, [s_3, s_4)>, <p_3, [s_4, q_B]>\}$, which means that point p_1 is the RNN for sub-segment $[q_A, s_1)$, at s_1 point p_4 also becomes a RNN, and p_4 is the only RNN for $[s_2, s_3)$, etc. The points (i.e., s_1, s_2, s_3, s_4) where there is a change of the RNN set are called *split points*. Benetis et al. [2] solve the problem for single RNN retrieval in 2D space. Our solution applies to any dimensionality and value of k.

The rest of the paper is organized as follows. Section 2 surveys related work on NN and RNN search. Section 3 presents a new algorithm for single RNN ($k = 1$) retrieval, and Sect. 4 generalizes the solution to arbitrary values of k. Section 5 discusses continuous RkNN processing. Section 6 contains an extensive experimental evaluation that demonstrates the superiority of the proposed techniques over the previous algorithms. Section 7 concludes the paper with directions for future work.

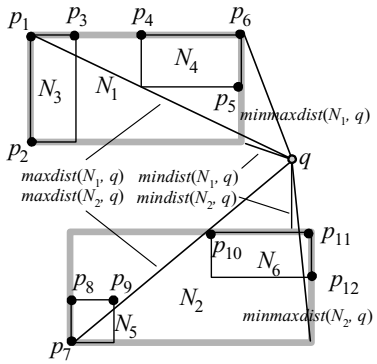
2 Background

Although our solutions can be used with various indexes, in the sequel, we assume that the dataset 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 surveys the previous studies on RkNN queries.

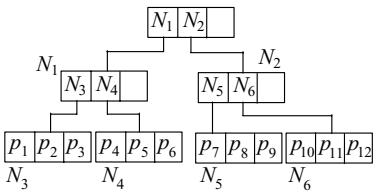
2.1 Algorithms for NN search using R-Trees

The R-tree [8] and its variants (most notably the R*-tree [1]) can be thought of as extensions of B-trees in multidimensional space. Figure 3a shows a set of points $\{p_1, p_2, \ldots, p_{12}\}$ indexed by an R-tree (Fig. 3b) assuming a capacity of three entries per node. Points close in space (e.g., p_1, p_2, p_3) are clustered in the same leaf node (e.g., 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 nearest neighbor query retrieves the data point p that is closest to q. The NN algorithms 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 N, and (iii) $minmaxdist(N, q)$, which gives



(a) Points and node extents



(b) The R-tree

Fig. 3 Example of an R-tree and a NN query

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 3a shows these pruning bounds between point q and nodes N_1, N_2.

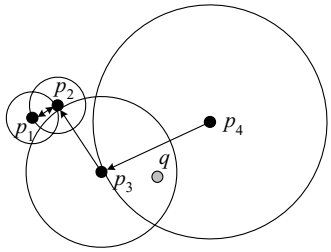
Existing NN methods are based on either depth-first (DF) or best-first (BF) traversal. DF algorithms [5, 14] start from the root and visit recursively the node with the smallest *mindist* from q. In Fig. 3, for instance, the first 3 nodes accessed are (in this order) the root, N_1 and N_4, where the first potential nearest neighbor p_5 is found. During backtracking to the upper levels, DF only descends entries whose minimum distances (to q) 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.

The BF algorithm [9] maintains a heap H containing the entries visited so far, sorted in ascending order of their *mindist*. In Fig. 3, for instance, BF starts by inserting the root entries into $H = \{N_1, N_2\}$. 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 its entries in H, after which $H = \{N_2, N_4, N_3\}$. 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, BF 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 extended for the retrieval of $k > 1$ nearest neighbors. Furthermore, BF is "incremental", i.e., it reports the nearest neighbors in ascending order of their distances to the query.

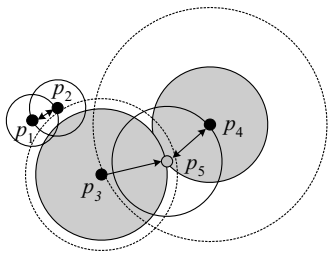
2.2 RNN algorithms

We first illustrate the RNN algorithms using 2D data and $k = 1$, and then clarify their applicability to higher dimensionality and k. We refer to each method using the authors' initials. KM [10] pre-computes, for every data point p, its nearest neighbor $NN(p)$. Then, p is associated with a *vicinity circle* centered at it with radius equal to the distance between p and $NN(p)$. The MBRs of all circles are indexed by an R-tree, called the RNN-tree. Using this structure, the reverse nearest neighbors of q can be efficiently retrieved by a point location query, which returns all circles containing q. Figure 4a illustrates the concept using four data points; since q falls in the circles of p_3 and p_4, $RNN(q) = \{p_3, p_4\}$.

Because the RNN-tree is optimized for RNN, but not NN search, Korn and Muthukrishnan [10] 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, YL [20] combines the two indexes in the RdNN-tree. Similar to the RNN-tree, a leaf entry of the RdNN-tree contains the vicinity circle of



(a) RNN processing with pre-computation



(b) Problem with updates

Fig. 4 Illustration of KM method

a data point. On the other hand, an intermediate entry contains the MBR of the underlying points (not their vicinity circles), together with the maximum distance from a point in the subtree to its nearest neighbor. As shown in the experiments of [20], the RdNN-tree is efficient for both RNN and NN queries because, intuitively, it incorporates all the information of the RNN-tree, and has the same structure (for node MBRs) as a conventional R-tree. MVZ [13] is another pre-computation method that is applicable only to 2D space and focuses on asymptotical worst case bounds (rather than experimental comparison with other approaches).

The problem of KM, YL, MVZ (and, in general, all techniques that rely on preprocessing) is that they cannot deal efficiently with updates. This is because each insertion or deletion may affect the vicinity circles of several points. Consider Fig. 4b, 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 neighbor. 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. [12] propose a technique for bulk insertions in the RdNN-tree.

Stanoi et al. [16] eliminate the need for pre-computing all NNs by utilizing some interesting properties of RNN retrieval. Consider Fig. 5, which divides the space around a query q into 6 equal regions S_1 to S_6. Let p be the NN of q in some region S_i ($1 \leq i \leq 6$); it can be proved that either $p \in RNN(q)$ or there is no RNN of q in S_i. For instance, in Fig. 5, 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. Similarly, no result can exist in S_2, S_3 (p_4, p_5 are NNs of each other), S_5 (the NN of

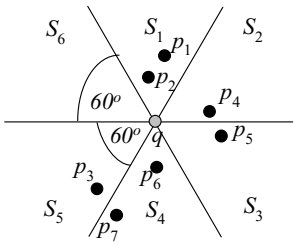


Fig. 5 Illustration of SAA method

p_3 is p_7), and S_6 (no data points). The actual $RNN(q)$ contains only p_6 (in S_4). Based on the above property, SAA [16] adopts a two-step processing method. First, six'constrained NN queries" [6] 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.

Singh et al. [15] show that the number of regions to be searched for candidate results increases exponentially with the dimensionality, rendering SAA inefficient even for three dimensions. Motivated by this, they propose SFT, a multi-step algorithm that (i) finds (using an R-tree) a large number K of NNs of the query q, which constitute the initial RNN candidates, (ii) eliminates the candidates that are closer to each other than to q, and (iii) determines the final RNNs from the remaining ones. The value of K should be larger than the number k of RNNs requested by every query. Consider, for instance, the (single) RNN query of Fig. 6, assuming $K = 4$. SFT first retrieves the 4 NNs of q: p_6, p_4, p_5 and p_2. The second step discards p_4 and p_5 since they are closer to each other than to q. The third step verifies whether p_2 (p_6) is a real RNN of q by checking if there is any point in the shaded circle centered at p_2 (p_6) crossing q. This involves a "boolean range query", which is similar to a range search except that it terminates as soon as (i) the first data point is found, or (ii) an edge of a node MBR lies within the circle entirely. For instance, as $minmaxdist(N_1, p_2) \leq dist(p_2, q)$, N_1 contains at least a point p with $dist(p_2, p) < dist(p_2, q)$, indicating that p_2 is a false hit. Since the boolean query of p_6 returns empty, SFT reports p_6 as the only RNN. The major shortcoming of the method is that it may incur false misses. In Fig. 6, although p_3 is a RNN of q, it does not belong to the 4 NNs of the query and will not be retrieved.

Table 1 Summary of the properties of RNN algorithms

	Support dynamic data	Arbitrary dimensionality	Exact result
KM, YL	No	Yes	Yes
MVZ	No	No	Yes
SAA	Yes	No	Yes
SFT	Yes	Yes	No

Table 1 summarizes the properties of each algorithm. As discussed before, pre-computation methods cannot efficiently handle updates. MVZ focuses exclusively on 2D space, 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 chance of false misses but increases significantly the processing cost. Regarding the applicability of the existing algorithms to arbitrary k, pre-computation methods only support a specific value (typically 1), used to determine the vicinity circles. SFT can support the retrieval of RkNNs by setting a large value of K ($\gg k$) and adapting boolean queries for deciding whether there are at least k objects in a search region. The extension of SAA to arbitrary k has not been studied before, but we will discuss it in Sect. 4.3.

The only existing method BJKS [2] for continuous RNN queries is based on the SAA algorithm. We illustrate the algorithm using Fig. 7, where the dataset consists of points $p_1, \ldots, p_4$ and the C-RNN query is the segment $q_A q_B$. In the filter step, BJKS considers (conceptually) every point q on segment $q_A q_B$. For each such point, it divides the data space into 6 partitions (based on q) and retrieves the NN of q in each partition. Due to symmetry, let us focus on the partition bounded by the two upward rays (see Fig. 7). When q belongs to the segment $[q_A, a_1)$, the NN of q is p_1. The NN is p_2 for q belonging to segment $[a_1, a_2)$, and p_3 for q in $[a_2, q_B)$ (position a_2 is equally distant to p_2 and p_3). For each of the candidates (p_1, p_2, p_3) returned by the filter phase, the refinement step of BJKS obtains its NN (in the entire data space), and examines the corresponding vicinity circle (e.g., the circle for p_1 crosses its NN p_2). The candidate is a final result if and only if its circle intersects the query segment. In Fig. 7, p_2 and p_3 are false hits because

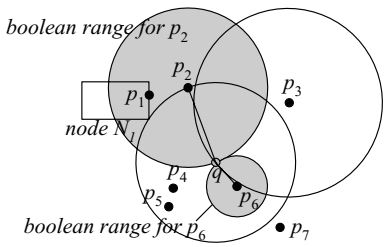


Fig. 6 Illustration of SFT method

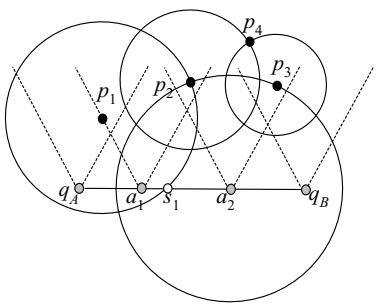


Fig. 7 Illustration of BJKS method

their circles are disjoint with $q_A q_B$. On the other hand, p_1 is the RNN for every point on segment $[q_A, s_1)$, where s_1 is the intersection between its circle and the query segment. There is no RNN for any point on $[s_1, q_B]$. Since BJKS is based on SAA, its applicability is restricted to 2D space.

It is worth mentioning that all the above algorithms (as well as our solutions) aim at *monochromatic* RNN retrieval in [10]. Stanoi et al. [17] consider *bichromatic* RNN search: given two data sets P_1, P_2 and a query point $q \in P_1$, a bichromatic RNN query retrieves all the points $p_2 \in P_2$ that are closer to q than to any other object in P_1, i.e., $dist(q, p_2) < dist(p_1, p_2)$ for any $p_1 \in P_1$ and $p_1 \neq q$. If $VC(q)$ is the Voronoi cell covering q in the Voronoi diagram [4] computed from P_1, the query result contains all the points in P_2 that fall inside $VC(q)$. Based on this observation, SRAA [17] first computes $VC(q)$ using an R-tree on P_1, and then retrieves the query result using another R-tree on P_2. This approach is not directly applicable to monochromatic search (which involves a single dataset), but the concept of Voronoi cells is related to our solutions, as clarified in Sect. 3.3.

3 Single RNN processing

In this section, we focus on single RNN retrieval ($k = 1$). Section 3.1 illustrates some problem characteristics that motivate our algorithm, which is presented in Sect. 3.2. Section 3.3 analyzes the performance of the proposed techniques with respect to existing methods.

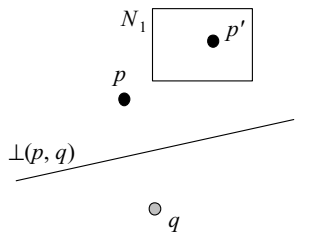
3.1 Problem characteristics

Consider the perpendicular bisector $\perp(p, q)$ between the query q and an arbitrary data point p as shown in Fig. 8a. The bisector divides the data space into two half-spaces: $HS_q(p, q)$ that contains q, and $HS_p(p, q)$ that contains p. Any point (e.g., p') in $HS_p(p, q)$ cannot be a RNN of q because it is closer to p than to q. Similarly, a node MBR (e.g., N_1) that falls completely in $HS_p(p, q)$ cannot contain any results. In some cases, the pruning of an MBR requires multiple half-spaces. For example, in Fig. 8b, although N_2 does not fall completely in $HS_{p_1}(p_1, q)$ or $HS_{p_2}(p_2, q)$, it can still be pruned since it lies entirely in the union of the two half-spaces.

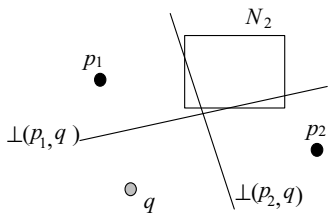
In general, if $p_1, p_2, \ldots, p_{n_c}$ are n_c data points, then any node N whose MBR falls inside $\cup_{i=1}^{n_c} HS_{p_i}(p_i, q)$ cannot contain any RNN of q. Let the *residual polygon* N^{resP} be the area of MBR N outside $\cup_{i=1}^{n_c} HS_{p_i}(p_i, q)$, i.e., the part of the MBR that may cover RNNs. Then, N can be pruned if and only if $N^{resP} = \emptyset$. A non-empty N^{resP} is a convex polygon bounded by the edges of N and the bisectors $\perp(p_i, q)$ $(1 \leq i \leq n_c)$. We illustrate its computation using Fig. 9a with $n_c = 3$. Initially, N^{resP} is set to N, and then we trim it incrementally with each bisector in turn. In particular, the trimming with $\perp(p_i, q)$ results in a new N^{resP} corresponding to the part of the previous N^{resP} inside the half-space $HS_q(p_i, q)$. The shaded trapezoid in Fig. 9a is the N^{resP} after being trimmed with $\perp(p_1, q)$. Figure 9b shows the final N^{resP} after processing all bisectors.

The above computation of N^{resP} has two problems. First, in the worst case, each bisector may introduce an additional vertex to N^{resP}. Consequently, processing the i-th $(1 \leq i \leq n_c)$ bisector takes $O(i)$ time because it may need to examine all edges in the previous N^{resP}. Thus, the total computation cost is $O(n_c^2)$, i.e., quadratic to the number of bisectors. Second, this method does not scale with the dimensionality because computing the intersection of a half-space and a hyper-polyhedron is prohibitively expensive in high-dimensional space [4].

Therefore, we propose a simpler trimming strategy that requires only $O(n_c)$ time. The idea is to bound N^{resP} by a

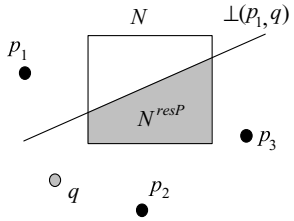


(a) Pruning with one point

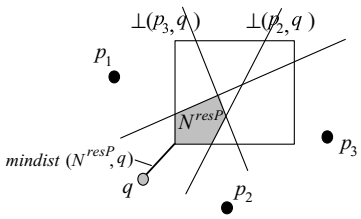


(b) Pruning with two points

Fig. 8 Illustration of half-space pruning



(a) After trimming with $\perp(p_1, q)$



(b) The final polygon

Fig. 9 Computing the residual region

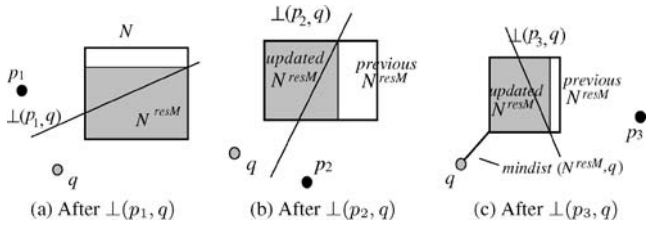


Fig. 10 Computing the residual MBR

residual MBR N^{resM}. Figure 10 illustrates the residual MBR computation using the example in Fig. 9. Figure 10a shows the trimming with $\perp(p_1, q)$ where, instead of keeping the exact shape of N^{resP}, 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 $HS_q(p_i, q)$. Figures 10b and c illustrate the residual MBRs after processing $\perp(p_2, q)$ and $\perp(p_3, q)$, respectively. Note that the final N^{resM} is not necessarily the MBR of the final N^{resP} (compare Figs. 10c and 9b). Trimmed MBRs can be efficiently computed (for arbitrary dimensionality) using the *clipping algorithm* of [7].

Figure 11 presents the pseudo-code for the above approximate trimming procedures. If N^{resM} exists, *trim* returns the minimum distance between q and N^{resM}; otherwise, it returns ∞. Since N^{resM} always encloses N^{resP}, $N^{resM} = \emptyset$ necessarily leads to $N^{resP} = \emptyset$. This property guarantees that pruning is "safe", meaning that *trim* never eliminates a node that may contain query results. The algorithm also captures points as MBRs with zero extents. In this case, it will return the actual distance between a point and q (if the point is closer to q than to all other candidates), or ∞ otherwise.

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

Lemma 1 *Given a query point q and an MBR N in 2D space, let N^{resP} be the part (residual polygon) of N satisfying a set S of half-spaces, and N^{resM} the residual MBR computed (by the algorithm in Fig. 11) using the half-spaces in S. Then, $mindist(N^{resM}, q) = mindist(N^{resP}, q)$ in all cases.*

Proof Presented in the appendix. □

Algorithm *Trim* $(q, \{p_1, p_2, ..., p_{n_c}\}, N)$
/* q is the query point; $p_1, p_2, ..., p_{n_c}$ are arbitrary data points;
N is a rectangle being trimmed */
1. $N^{resM} = N$
2. for i = 1 to n_c //consider each data point in turn
3. $N^{resM} = clipping(N^{resM}, HS_q(p_i, q))$
//algorithm of [7]: obtain the MBR for the part of N^{resM} in the half-space $HS_q(p_i, q)$
4. if $N^{resM} = \emptyset$ then return ∞
5. return $mindist(N^{resM}, q)$

Fig. 11 The trim algorithm

As an illustration of the lemma, note that *mindist* (N^{resP}, q) in Fig. 9b is equivalent to *mindist*(N^{resM}, q) in Fig. 10c. Our RNN algorithm, discussed in the next section, aims at examining the nodes N of an R-tree in ascending order of their *mindist*(N^{resP}, q). Since N^{resP} is expensive to compute in general, we decide the access order based on *mindist*(N^{resM}, q), which, as indicated by Lemma 1, has the same effect as using *mindist*(N^{resP}, q) in 2D space. It is worth mentioning that the lemma does not hold for dimensionalities higher than 2 (in this case, N^{resM} may exist even if N^{resP} does not [7]). Nevertheless, pruning based on *mindist*(N^{resM}, q) is still safe because, as mentioned earlier, N^{resM} is eliminated only if N^{resP} is empty.

3.2 The TPL algorithm

Based on the above discussion, we adopt a two-step framework that first retrieves a set of candidate RNNs (filter step) and then removes the false hits (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, 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-spaces is used to prune nodes (data points) that cannot contain (be) candidates. Next we discuss TPL using the example of Fig. 12, which shows a set of data points (numbered in ascending order of their distance from the query) and the corresponding R-tree (the contents of some nodes are omitted for clarity). The query result contains only point p_5.

Initially, TPL visits the root of the R-tree and inserts its entries N_{10}, N_{11}, N_{12} into a heap H sorted in ascending order of their *mindist* from q. Then, the algorithm de-heaps N_{10} (top of H), visits its child node, and inserts into H the entries there ($H = \{N_3, N_{11}, N_2, N_1, N_{12}\}$). Similarly, the next node accessed is leaf N_3, and H becomes (after inserting the points in N_3): $\{p_1, N_{11}, p_3, N_2, N_1, N_{12}\}$. Since p_1 is the top of H, it is the first candidate added to the candidate set S_{cnd}. The next de-heaped entry is N_{11}. As $S_{cnd} \neq \emptyset$, TPL uses *trim* (Fig. 11) to check if N_{11} can be

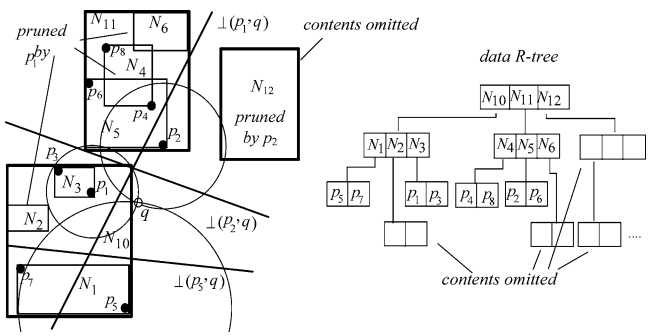


Fig. 12 Illustration of the TPL algorithm

pruned. Part of N_{11} lies in $HS_q(p_1, q)$ (i.e., *trim* returns $mindist(N_{11}^{resM}, q) \neq \infty$), and thus it has to be visited.

Among the three MBRs in node N_{11}, N_4 and N_6 fall completely in $HS_{p_1}(p_1, q)$, indicating that they cannot contain any candidates. Therefore, N_4 and N_6 are not inserted in H, but are added to the *refinement set* S_{rfn}. *In general, all the points and nodes that are not pruned during the filter step are preserved in* S_{rfn}*, and will be used in the refinement step to verify candidates.* On the other hand, N_5 (an MBR in node N_{11}) falls partially in $HS_q(p_1, q)$, and is inserted into H using $mindist(N_5^{resM}, q)$ as the sorting key ($H = \{N_5, p_3, N_2, N_1, N_{12}\}$). 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 MBR that cannot contain candidates.

TPL proceeds to de-heap the top N_5 of H, and retrieves its child node, where point p_2 is added to $H = \{p_2, p_3, N_2, N_1, N_{12}\}$, and p_6 to $S_{rfn} = \{N_4, N_6, p_6\}$ (p_6 is in $HS_{p_1}(p_1, q)$, and hence, cannot be a RNN of q). Then, p_2 is removed from H, and becomes the second candidate, i.e., $S_{cnd} = \{p_1, p_2\}$. Point p_3 (now top of H), however, is added to S_{rfn} because it lies in $HS_{p_1}(p_1, q)$. Similarly, the next processed entry N_2 is also inserted in S_{rfn} (without visiting node N_2). Part of N_1, on the other hand, appears in $HS_q(p_1, q) \cup HS_q(p_2, q)$ and TPL accesses its child node, leading to $S_{cnd} = \{p_1, p_2, p_5\}$ and $S_{rfn} = \{N_2, N_4, N_6, p_6, p_3, p_7\}$. Finally, N_{12} is also inserted into S_{rfn} as it falls completely in $HS_{p_2}(p_2, q)$. The filter step terminates when $H = \emptyset$.

Figure 13 illustrates the pseudo-code for the filter step. Note that *trim* is applied twice for each node N: when N is inserted into the heap and when it is de-heaped, respectively. The second test is necessary, because N may be pruned by some candidate that was discovered after the insertion of N 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. This heuristic 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.

After the termination of the filter step, we have a set S_{cnd} of candidates and a set S_{rfn} of node MBRs and data points. Let P_{rfn} (N_{rfn}) be the set of points (MBRs) in S_{rfn}. The refinement step is performed in *rounds*. Figure 14 shows the pseudo-code for each round, where we eliminate the maximum number of candidates from S_{cnd} without visiting additional nodes. Specifically, 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 a node MBR $N \in N_{rfn}$ such that $minmaxdist(p, N) < dist(p, q)$ (i.e., N is guaranteed to contain a point that is closer to p than q). For instance, in Fig. 12, 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 of Fig. 14 prune false hits according to the above observations.

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 $N \in N_{rfn}$, it holds that $mindist(p, N) > dist(p, q)$. In Fig. 12, candidate p_5 satisfies these conditions and is validated as a final RNN (also 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, the validation of p requires accessing the set $p.toVisit$ of nodes $N \in N_{rfn}$ that satisfy $mindist(p, N) < dist(p, q)$. After computing *toVisit* for all the candidates, P_{rfn} and N_{rfn} are reset to empty.

Next, TPL accesses a node selected from the *toVisit* of the candidates. Continuing the running example, after the first round p_1 is eliminated, p_5 is reported (as an actual result), and $S_{cnd} = \{p_2\}$. The nodes that may contain NNs of p_2 are $p_2.toVisit = \{N_4, N_{12}\}$. We choose to access a lowest

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 $S_{cnd} = \emptyset$, $S_{rfn} = \emptyset$
3. insert (R-tree root, 0) to H
4. while H is no longer empty
5. de-heap the top entry (e, key) of H
6. if ($trim(q, S_{cnd}, e) = \infty$) then $S_{rfn} = S_{rfn} \cup \{e\}$
7. else //entry may be or contain a candidate
8. if e is data point p
9. $S_{cnd} = S_{cnd} \cup \{p\}$
10. else if e points to a leaf node N
11. for each point p in N
12. if ($trim(q, S_{cnd}, p) = \infty$) then $S_{rfn} = S_{rfn} \cup \{p\}$
13. else insert (p, $dist(p, q)$) in H
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 ($mindist(N_i^{resM}, q) = \infty$) then $S_{rfn} \cup = \{N_i\}$
18. else insert ($N_i, mindist(N_i^{resM}, q)$) in H

Fig. 13 The TPL filter algorithm

Algorithm *refinement-round* (q, S_{cnd}, P_{rfn}, N_{rfn})
/* q is the query point; S_{cnd} is the set of candidates that have not been verified so far; P_{rfn} (N_{rfn}) contains the points (nodes) that will be used in this round for candidate verification */
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 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 $mindist(p, N) < dist(p, q)$ then add N to $p.toVisit$
12. if $p.toVisit = \emptyset$
13. $S_{cnd} = S_{cnd} - \{p\}$ and report p //actual result

Fig. 14 The refinement-round algorithm

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 at line 7 of Fig. 14 is more effective). In case of a tie (i.e., multiple nodes of the same low level), we access the one that appears in the *toVisit* lists of the largest number of candidates.

If the node N visited is a leaf, then P_{rfn} contains the data points in N, and N_{rfn} is set to $\emptyset$. Otherwise (N is an intermediate node), N_{rfn} includes the MBRs of N, and P_{rfn} is $\emptyset$. In our example, the parameters for the second round are $S_{cnd} = \{p_2\}$, $P_{rfn} = \{p_4, p_8\}$ (points of N_4), and $N_{rfn} = \emptyset$. Point p_4 eliminates p_2, and the algorithm terminates. Figure 15 shows the pseudo-code of the TPL refinement step. Lines 2-4 prune 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.

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 Analytical comparison with the previous solutions

TPL and the existing techniques that do not require preprocessing (SAA, SFT) are based on the filter-refinement framework. Interestingly, the two steps are independent in the sense that the filtering algorithm of one technique can be combined with the refinement mechanism of another.

Algorithm *TPL-refinement* (q, S_{cnd}, S_{rfn})
/* q is the query point; S_{cnd} and S_{rfn} are the candidate and refinement sets returned from the filter step */
1. for each point $p \in S_{cnd}$
2. for each other point $p' \in S_{cnd}$
3. if $dist(p, p') < dist(p, q)$
4. $S_{cnd} = S_{cnd} - \{p\}$; goto 1
5. if p is not eliminated initialize $p.toVisit = \emptyset$
6. P_{rfn} = the set of points in S_{rfn}; N_{rfn} = MBRs in S_{rfn}
7. repeat
8. *refinement-round*(q, S_{cnd}, P_{rfn}, N_{rfn})
9. if $S_{cnd} = \emptyset$ return //terminate
10. let N be the lowest level node that appears in the largest number of $p.toVisit$ for $p \in S_{cnd}$
11. remove N from all $p.toVisit$ and access N
12. $P_{rfn} = N_{rfn} = \emptyset$ //for the next round
13. if N is a leaf node
14. $P_{rfn} = \{p \mid p \in N\}$ //P_{rfn} contains only the points in N
15. else
16. $N_{rfn} = \{N' \mid N' \in N\}$ //N_{rfn} contains the MBRs in N

Fig. 15 The TPL refinement algorithm

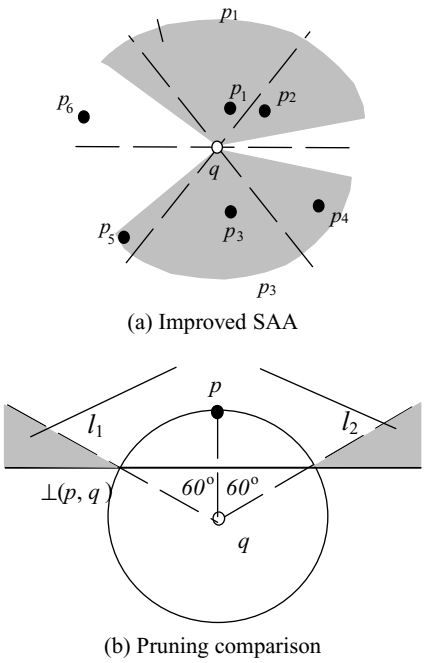


Fig. 16 Superiority of TPL over SAA

For instance, the boolean range queries 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 filter step with respect to SAA, we first present an improvement of that method. Consider the space partitioning of SAA in Fig. 16a 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 p_1q (upper shaded region in Fig. 16a). 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 6 (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).

Going one step further, the filter step of TPL is even more efficient than that of the improved SAA. Consider Fig. 16b 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-space $HS_p(p, q)$, which includes $\nabla(p)$ 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 best-first NN algorithms in general). In other words, all the non-visited nodes that can be pruned by $\nabla(p)$ can also be pruned by $HS_p(p, q)$. As a corollary, the maximum number of candidates retrieved by TPL is also bounded by a constant depending only on the dimensionality (e.g., 6 in 2D space). Furthermore, 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 filter 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 Fig. 12, 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 $p_1, \ldots, p_5$ 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 Fig. 17. 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 dimensional space.

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, and all 6 candidates of SAA are in N. Then, as suggested in [16] 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-spaces. 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. [15] 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 filter 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 or boolean queries).

The concept of bisectors is closely related to Voronoi cells (VC) used in [17] for bichromatic queries. In fact, a possible solution for finding RNNs in 2D space is to first obtain the set $S_V(q)$ of points from the dataset P, whose bisectors with the query point q contribute to an edge of the VC covering q (in the Voronoi diagram computed from $P \cup \{q\}$). For example, in Fig. 18, $S_V(q)$ equals $\{p_1, p_2, p_3, p_4\}$, and $VC(q)$ is the shaded region. Any point (e.g., p_5) that does not belong to $S_V(q)$ cannot be a RNN, because it lies outside $VC(q)$, and must be closer to at least one point (i.e., p_2) in $S_V(q)$ than to q. Therefore, in the refinement step, it suffices to verify whether the points in $S_V(q)$ are the true RNNs.

However, this approach is limited to 2D space because computing Voronoi cells in higher dimensional space is very expensive [4]. Furthermore, its application to $k > 1$ requires calculating order-k Voronoi cells, which is complex and costly even in 2D space [4]. TPL avoids these problems by retrieving candidates that are not necessarily points in $VC(q)$, but are sufficient for eliminating the remaining data. Furthermore, note that some objects in $VC(q)$ may not be discovered by TPL as candidates. For instance, in Fig. 18, TPL will process p_2 before p_3 since the former is closer to q. After adding p_2 to the candidate set, p_3 will be pruned because it falls in the half-space $HS_{p_2}(q, p_2)$. In this case, the candidate set returned by the filter step of TPL includes only p_1, p_2, and p_4.

Section 4.3 discusses an alternative solution based on the previous work, and clarifies the superiority of TPL.

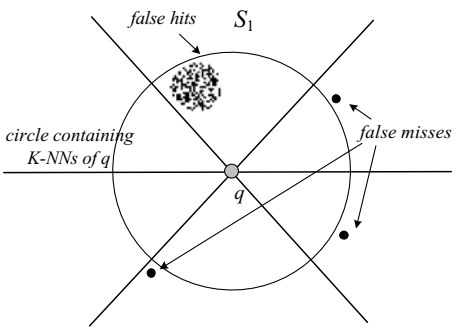


Fig. 17 False hits and misses of SFT

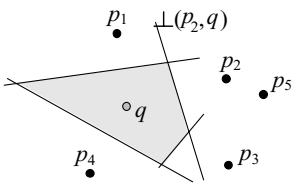


Fig. 18 The connection between TPL and Voronoi cells

4 RkNN processing

Section 4.1 presents properties that permit pruning of the search space for arbitrary values of k. Section 4.2 extends TPL to RkNN queries. "Section 4.3 discusses an alternative solution based on the previous work, and clarifies the superiority of TPL."

4.1 Problem characteristics

The half-space pruning strategy of Sect. 3.1 extends to arbitrary values of k. Figure 19a shows an example with $k = 2$, where the shaded region corresponds to the intersection $HS_{p_1}(p_1, q) \cap HS_{p_2}(p_2, q)$. 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 (e.g., N_1) inside the shaded area cannot contain any results. In some cases, several half-space intersections are needed to prune a node. Assume the R2NN query q and the three data points of Fig. 19b. Each pair of points generates an intersection of half-spaces: (i) $HS_{p_1}(p_1, q) \cap HS_{p_2}(p_2, q)$ (i.e., polygon $IECB$), (ii) $HS_{p_1}(p_1, q) \cap HS_{p_3}(p_3, q)$ (polygon $ADCB$), and (iii) $HS_{p_2}(p_1, q) \cap HS_{p_3}(p_3, q)$ (polygon $IFGHB$). 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.

In general, assume a RkNN query and $n_c \geq k$ data points $p_1, p_2, \ldots, p_{n_c}$ (e.g., in Fig. 19b $n_c = 3$ and $k = 2$). Let $\{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ be any subset of $\{p_1, p_2, \ldots, p_{n_c}\}$. The subset prunes the intersection area $\cap_{i=1}^{k} HS_{\sigma_i}(\sigma_i, q)$. The entire region that can be eliminated corresponds to the union of the intersection areas of all $\binom{n_c}{k}$ subsets. Examining $\binom{n_c}{k}$ subsets is expensive for large k and n_c. In order to reduce the cost, we restrict the number of inspected subsets using the following heuristic. First, all the points are sorted according to their Hilbert values; let the sorted order be $p_1, p_2, \ldots, p_{n_c}$. Then, we consider only the intersection areas computed from the n_c subsets $\{p_1, \ldots, p_k\}$, $\{p_2, \ldots, p_{k+1}\}, \ldots, \{p_{n_c}, \ldots, p_{k-1}\}$, based on the rationale that points close to each other tend to produce a large pruning area. The tradeoff is that we may occasionally mis-judge an MBR N to be un-prunable, while N could be eliminated if all the $\binom{n_c}{k}$ subsets were considered. Similar to *trim* in Fig. 11, *k-trim* aims at returning the minimum distance from query q to the part N^{resP} of N that cannot be pruned. Since N^{resP} is difficult to compute, we bound it with a residual MBR N^{resM}, and *k-trim* reports the *mindist* from q to N^{resM}. If N^{resM} does not exist, *k-trim* returns ∞, and N is pruned.

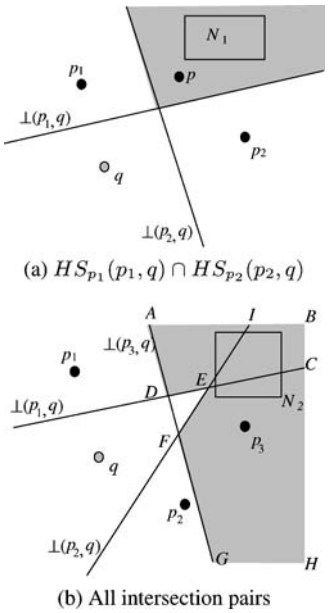


Fig. 19 Examples of R2NN queries

Algorithm *k-trim* $(q, \{p_1, p_2, ..., p_{n_c}\}, N)$
/* q is the query point; $p_1, p_2, ..., p_{n_c}$ are arbitrary data points, and $n_c \geq k$; N is the MBR being trimmed; */
1. sort the n_c data points in ascending order of their Hilbert values (assume the sorted order $p_1, p_2, ..., p_{n_c}$)
2. $N^{resM} = N$
3. for i = 1 to n_c //consider each subset containing k consecutive points $\{\sigma_1, \sigma_2, ..., \sigma_k\}$ in the sorted order
4. for j = 1 to k
5. N_j = *clipping*(N^{resM}, $HS_q(\sigma_j, q)$)
//algorithm of [7]: obtain the MBR for the part of N^{resM} in the half-space $HS_q(\sigma_j, q)$
6. $N^{resM} = \cup_{j=1}^{k} N_j$
7. if $N^{resM} = \emptyset$ then return ∞
8. return $mindist(N^{resM}, q)$

Fig. 20 The *k-trim* algorithm

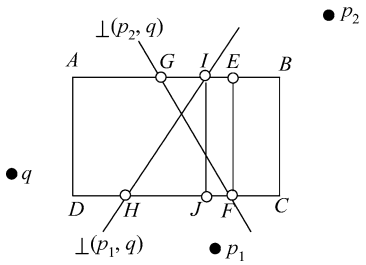


Fig. 21 Computing N^{resM} for R2NN processing

The above discussion leads to the *k-trim* algorithm in Fig. 20. Initially, N^{resM} is set to N, and is updated incrementally according to each of the n_c subsets examined. Specifically, given a subset $\{\sigma_1, \sigma_2, \ldots, \sigma_k\}$, we first compute, for each point σ_j ($1 \leq j \leq k$), the MBR N_j for the part of the current N^{resM} that falls in $HS_q(\sigma_j, q)$. Then, the new N^{resM} becomes the union of the k MBRs $N_1, N_2, \ldots, N_k$. We illustrate the computation using Fig. 21 where the current N^{resM} is rectangle $ABCD$, and the subset being examined is $\{p_1, p_2\}$ (i.e., $k = 2$). For bisector $\perp(p_1, q)$, we use the algorithm in [7] to obtain the MBR N_1 (polygon $ADJI$) for the area of N^{resM} falling in $HS_q(p_1, q)$. Similarly, for bisector $\perp(p_2, q)$, the algorithm of [7] returns $N_2 = ADFE$ (MBR for the part of N^{resM} in $HS_q(p_2, q)$). Hence, the new

Algorithm *k-refinement-round* (q, S_{cnd}, P_{rfn}, N_{rfn})
/* q is the query point; S_{cnd} is the set of candidates that have not been verified so far; P_{rfn} (N_{rfn}) contains the points (nodes) that will be used in this round for candidate verification */
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. $p.counter = p.counter - 1$
5. if $p.counter = 0$
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 $f^l_{min} \geq p.counter$ //l is the level of N
10. $S_{cnd} = S_{cnd} - \{p\}$
11. goto 1 //test next candidate
12. for each node MBR $N \in N_{rfn}$
13. if $mindist(p,N) < dist(p,q)$ then add N in $p.toVisit$
14. if $p.toVisit = \emptyset$
15. $S_{cnd} = S_{cnd} - \{p\}$ and report p //actual result

Fig. 22 The refinement round of TPL for $k > 1$

N^{resM} is the union of N_1 and N_2, i.e., rectangle *ADFE*. Notice that every point that is in the original N^{resM} but not in *ADFE* cannot be a R2NN of q, because it must lie in both $HS_{p_1}(p_1, q)$ and $HS_{p_2}(p_2, q)$.

4.2 The TPL algorithm for RkNN search

To solve a RkNN query, we adopt the framework of Sect. 3. Specifically, the filter step of TPL initially accesses the nodes of the R-tree in ascending order of their *mindist* to the query q, and finds an initial candidate set S_{cnd} which contains the k points nearest to q. Then, the algorithm decides the node access order (for the MBRs subsequently encountered) based on the distance computed by *k-trim*. MBRs and data points pruned (i.e., *k-trim* returns ∞) are kept in the refinement set S_{rfn}. The filter phase finishes when all the nodes that may include candidates have been accessed.

The refinement step is also executed in rounds, which are formally described in Fig. 22. The first round is invoked with P_{rfn} and N_{rfn} that contain the points and MBRs in S_{rfn} respectively, and we attempt to eliminate (validate) as many false hits (final RkNNs) from S_{cnd} as possible. The elimination and validation rules, however, are different from $k = 1$ because a point $p \in S_{cnd}$ can be pruned (validated) only if there are at least (fewer than) k points within distance $dist(p, q)$ from p. Thus, we associate p with a counter $p.counter$ (initially set to k), and decrease it every time we find a point p' satisfying $dist(p, p') < dist(p, q)$. We eliminate p as a false hit when its counter becomes 0.

Recall that, for $k = 1$, TPL claims a point p to be a false hit as long as $minmaxdist(p, N) < dist(p, q)$ for a node $N \in N_{rfn}$. For $k > 1$, this heuristic is replaced with an alternative that utilizes the *maxdist* between p and N, and a lower bound for the number of points in N. If $maxdist(p, N) < dist(p, q)$, then there are at least f^l_{min} points (in the subtree of N) that are closer to p than q, where f_{min} is the minimum node fanout (for R-trees, 40% of the node capacity), and l the level of N (counting from the leaf level as level 0). Hence, p can be pruned if $f^l_{min} \geq p.counter$.

After a round, TPL accesses a node N selected from the *toVisit* lists of the remaining candidates by the same criteria as in the case of $k = 1$. Then, depending on whether N is a leaf or intermediate node, P_{rfn} or N_{rfn} is filled with the entries in N, and another round is performed. The refinement phase terminates after all the points in S_{cnd} have been eliminated or verified. We omit the pseudo-codes of the filter and main refinement algorithms for $k > 1$ because they are (almost) the same as those in Figs. 13 and 15 respectively, except for the differences mentioned earlier.

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 as in Fig. 5. Then, the* k *NNs of q in each partition are the only possible results of q. Furthermore, in the worst case, all these points may be the actual R*k*NNs.*

Proof Presented in the appendix. □

As a corollary, for any query point in 2D space, the maximum number of RkNNs equals $6k$. Figure 23 illustrates the lemma using an example with $k = 2$. The candidates of q include $\{p_1, p_2, p_4, p_5, p_6\}$ (e.g., p_3 is not a candidate since it is the 3rd NN in partition S_1). Based on Lemma 2, the filter step of SAA may execute 6 constrained kNN queries [6] in each partition. Then, the refinement step verifies or eliminates each candidate with a kNN search. 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 Sect. 2.2, SFT can support RkNN by setting a large value of K ($\gg k$), and adapting a boolean range query to verify whether there are at least k points closer to a candidate than the query point q. Similar to the case of $k = 1$, various boolean queries may access the same node multiple times, which is avoided in TPL.

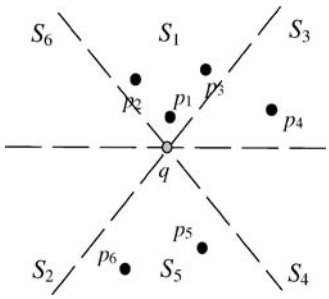


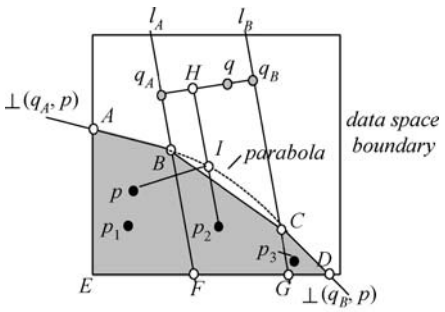
Fig. 23 Illustration of Lemma 2

5 Continuous R*k*NN processing

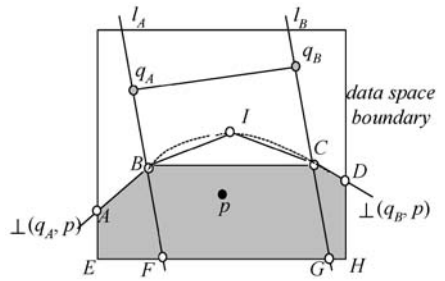
Given a segment q_Aq_B, a C*k*NN query aims at reporting the R*k*NNs for every point on the segment. As discussed in Sect. 1, the objective is to find a set of split points that partition q_Aq_B into disjoint sub-segments, such that all points in the same sub-segment have identical R*k*NNs. Section 5.1 first explains the pruning heuristics, and then Sect. 5.2 illustrates the concrete algorithms.

5.1 Problem characteristics

We first provide the rationale behind our solutions assuming $k = 1$ and 2D space, before presenting the formal results for arbitrary k and dimensionality. Consider Fig. 24a, where we draw lines l_A and l_B that are vertical to the query segment, and cross the two end points q_A and q_B, respectively. These two lines divide the data space into 3 areas: to the left of l_A, between l_A and l_B, and to the right of l_B. Let p be a data point to the left of l_A. The bisector $\perp(q_A, p)$ intersects the left boundary of the data space at A, and it intersects line l_A at B. Then, the polygon $ABFE$ cannot contain any query result. To understand this, consider an arbitrary point p_1 in $ABFE$, and any point q on segment q_Aq_B. The distance between p_1 and q is at least $dist(p_1, q_A)$ (the minimum distance from p_1 to the query segment), which is larger than $dist(p_1, p)$ (since p_1 is in $HS_p(q_A, p)$). Therefore, p is closer to p_1 than q, i.e., p_1 is not a RNN of q. Bisector $\perp(q_B, p)$, on the other hand, intersects the bottom of the data space at D and line l_B at C. By the same reasoning (of eliminating $ABFE$), no point (e.g., p_3) in the triangle CGD can be a query result.



(a) Point p lies to the left of l_A



(b) Point p lies between l_A and l_B

Fig. 24 Pruning regions for continuous RNN

Point p also prunes a region between lines l_A and l_B. To formulate this region, we need the locus of points (between l_A and l_B) that are equi-distant to p and q_Aq_B. The locus is a parabola, i.e., the dashed curve in Fig. 24a. All points (e.g., p_2) bounded by l_A, l_B, and the parabola can be safely discarded. In fact, for any point q on q_Aq_B, $dist(p_2, q)$ is at least $dist(p_2, H)$, where H is the projection of p_2 on q_Aq_B. Segment p_2H intersects the parabola at point I and, by the parabola definition, $dist(p, I) = dist(I, H)$. Since $dist(p_2, H)$ is the sum of $dist(p_2, I)$ and $dist(I, H)$, $dist(p_2, H) = dist(p_2, I) + dist(p, I) > dist(p, p_2)$ (triangle inequality). Therefore, p is closer to p_2 than q, or equivalently, p_2 is not a RNN of q.

Therefore, p prunes a region that is bounded by two line segments AB, CD, and curve BC, i.e., any node N that falls completely in this region does not need to be accessed. Unfortunately, checking whether N lies in this region is inefficient due to the existence of a non-linear boundary BC. We avoid this problem by examining if N is contained in the intersection of half-spaces, which has been solved in the previous sections. Specifically, we decrease the pruning region by replacing the boundary curve BC with a line segment BC, resulting in a new region corresponding to the shaded area in Fig. 24a. All points/MBRs falling in the area can be safely eliminated because it is entirely contained in the exact pruning region.

By symmetry, a point p lying to the right of l_B produces a pruning area that can be derived in the same way as in Fig. 24a. Next, we elaborate the case where p is between l_A and l_B (see Fig. 24b). Bisectors $\perp(q_A, p)$ and $\perp(q_B, p)$ define polygons $ABFE$ and $GCDH$ that cannot contain query results (the reasoning is the same as eliminating $ABFE$ and CDG in Fig. 24a). The curve BC in Fig. 24b is a parabola including points that are equally distant from q_Aq_B and p. Similar to Fig. 24a, all the points between l_A and l_B that are below the parabola can be pruned. To facilitate processing, we again approximate curve BC with a line segment BC, and as a result, the pruning region introduced by p is also a polygon (the shaded area) bounded by $\perp(q_A, p)$, $\perp(q_B, p)$ and segment BC.

As a heuristic, the parabolas in Figs. 24a and 24b can be more accurately approximated using multiple segments in order to reduce the difference between the approximate and exact pruning regions. For example, in Fig. 24b, instead of segment BC, we can bound the approximate pruning region with segments BI and IC, where I is an arbitrary point on the parabola. For simplicity, in the sequel, we always approximate a parabola using a single segment, but extending our discussion to using multiple segments is straightforward.

In general, for any dimensionality d, the pruning region defined by a point p is decided by three d-dimensional planes, two of which are the bisector planes $\perp(q_A, p)$ and $\perp(q_B, p)$, respectively. To identify the third one, we first obtain two d-dimensional planes L_A, L_B that are perpendicular to segment q_Aq_B, and cross q_A, q_B respectively (in Fig. 24 where $d = 2$, L_A and L_B are lines l_A and l_B, respectively). Planes L_A and $\perp(q_A, p)$ intersect into a $(d-1)$-dimensional

plane L'_A, and similarly, L_B and $\perp(q_B, p)$ produce a $(d-1)$-dimensional plane L'_B (in Fig. 24, L'_A and L'_B are points B and C, respectively). As shown in the following lemma, there exists a d-dimensional plane passing both L'_A and L'_B, and this is the 3rd plane bounding the pruning region.

Lemma 3 *Both L'_A and L'_B belong to a d-dimensional plane satisfying the following equation:*

$$\sum_{i=1}^{d}(2p[i] - q_A[i] - q_B[i]) \cdot x[i] + \sum_{i=1}^{d}\left(q_A[i] \cdot q_B[i] - \frac{p[i]^2}{2}\right) = 0 \quad (1)$$

where $x[i]$ denotes the i-th ($1 \le i \le d$) coordinate of a point in the plane, and similarly, $p[i]$, $q_A[i]$, $q_B[i]$ describe the coordinates of p, q_A, and q_B, respectively.

Proof Presented in the appendix. □

The next lemma establishes the correctness of the pruning region formulated earlier.

Lemma 4 *Given a query segment q_Aq_B and a data point p, consider half-spaces $HS_p(q_A, p)$, $HS_p(q_B, p)$ (decided by bisectors $\perp(q_A, p)$ and $\perp(q_B, p)$), and the half-space $HS_p(L)$ that is bounded by the plane L of Eq. (1) and contains p. Then, no point in $HS_p(q_A, p) \cap HS_p(q_B, p) \cap HS_p(L)$ can be a RNN of any point q on q_Aq_B.*

Proof Presented in the appendix. □

We are ready to clarify the details of pruning an MBR N, given a set S of n_c points $\{p_1, \ldots, p_{n_c}\}$. At the beginning, we set N^{resM} (the residual MBR) to N. For each p_i ($1 \le i \le n_c$), we incrementally update N^{resM} using 3 half-spaces H_1, H_2, H_3 that are "complement" to those in Lemma 4. Specifically, H_1 and H_2 correspond to $HS_{q_A}(q_A, p_i)$ and $HS_{q_B}(q_B, p_i)$ respectively, and H_3 is the half-space that is decided by the d-dimensional plane of Eq. (1) (replacing p with p_i), and contains q_A (H_3 can be represented with an inequality that replaces the "=" in Eq. (1) with "$\le$"). For every H_j ($1 \le j \le 3$), we apply the clipping algorithm of [7] to obtain the MBR N_j for the part of the previous N^{resM} lying in H_j, after which N^{resM} is updated to $\cup_{i=1}^{3} N_j$. To understand the correctness of the resulting N^{resM}, notice that any point p, which belongs to the original N^{resM} but not $\cup_{j=1}^{3} N_j$, does not fall in any of H_1, H_2, and H_3, indicating that p lies in the pruning region formulated in Lemma 4. If N^{resM} becomes empty, no query result can exist in the subtree of N, and N can be eliminated.

The extension to general values of k is straightforward. Following the methodology of Sect. 4.1, we sort the points in S according to their Hilbert values. Given the sorted list $\{p_1, \ldots, p_{n_c}\}$, we examine the n_c subsets $\{p_1, \ldots, p_k\}$, $\{p_2, \ldots, p_{k+1}\}, \ldots, \{p_{n_c}, \ldots, p_{k-1}\}$ in turn, and update

Algorithm *c-k-trim* $(q_Aq_B, \{p_1, p_2, ..., p_{n_c}\}, N)$
/* q_Aq_B is the query segment; $p_1, p_2, ..., p_{n_c}$ are arbitrary data points, and $n_c \ge k$; N is the MBR being trimmed; */
1. sort the n_c data points in ascending order of their Hilbert values (assume the sorted order $p_1, p_2, ..., p_{n_c}$)
2. $N^{resM} = N$
3. for i = 1 to n_c //consider each subset containing k consecutive points $\{\sigma_1, \sigma_2, ..., \sigma_k\}$ in the sorted order
4. for j = 1 to k
5. N_{j1} = *clipping*$(N^{resM}, HS_{q_A}(\sigma_j, q_A))$ //algorithm of [7]: obtain the MBR for the part of N^{resM} in the half-space $HS_{q_A}(\sigma_j, q_A)$
6. N_{j2} = *clipping*$(N^{resM}, HS_{q_B}(\sigma_j, q_B))$
7. N_{j3} = *clipping*(N^{resM}, H) /* H is the half-space that is bounded by the plane of Equation 1 (replacing p with σ_j), and contains q_A */
8. $N^{resM} = \cup_{j=1}^{k}(N_{j1} \cup N_{j2} \cup N_{j3})$
9. if $N^{resM} = \emptyset$ then return ∞
10. return $mindist(N^{resM}, q_Aq_B)$ //algorithm of [2]: obtain the minimum distance between N^{resM} and q_Aq_B

Fig. 25 The trimming algorithm for C-RkNN search

N^{resM} incrementally after each examination. Specifically, given a subset $\{\sigma_1, \sigma_2, \ldots, \sigma_k\}$, we obtain, for each point σ_i ($1 \le i \le k$), three half-spaces H_{i1}, H_{i2}, H_{i3} as described earlier for the case of $k = 1$. For each of the $3k$ half-spaces H_{ij} ($1 \le i \le k$, $1 \le j \le 3$), we compute the MBR N_{ij} for the part of (the previous) N^{resM} in H_{ij}. Then, the new N^{resM} (after examining $\{\sigma_1, \sigma_2, \ldots, \sigma_k\}$) equals the union of the $3k$ MBRs N_{ij} ($1 \le i \le k$ and $1 \le j \le 3$). Figure 25 presents the trimming algorithm for any value of k. This algorithm returns ∞ if the final N^{resM} after considering all the n_c subsets of S is empty. Otherwise ($N^{resM} \ne \emptyset$), it returns the minimum distance between N^{resM} and the query segment q_Aq_B (see [2] for computing the distance between a segment and a rectangle for arbitrary dimensionality).

5.2 The C-TPL algorithm

We proceed to elaborate the proposed algorithm, C-TPL, for C-RkNN queries. As with TPL, C-TPL also has a filter and a refinement step for retrieving and verifying candidates, respectively. However, unlike conventional RkNN search where the actual NNs of the verified candidates do not need to be retrieved, as illustrated shortly, this is necessary for C-RkNN retrieval in order to obtain the split points. Therefore, C-TPL includes a third phase, the *splitting step*, for computing the split points. In the sequel, we explain C-TPL using a 2D example with $k = 1$. Since C-TPL is similar to TPL, our discussion focuses on clarifying the differences between the two algorithms.

Consider Fig. 26a, which shows part of a dataset and the MBRs of the corresponding R-tree in Fig. 26c. The filter step of C-TPL visits the entries in ascending order of their *mindist* to q_Aq_B, and maintains the encountered entries in a heap H. The first few nodes accessed are the root, N_1, and N_4, leading to $H = \{p_1, N_2, N_5, p_3, N_3, N_6\}$. Then, p_1 is removed from H, and becomes the first candidate in S_{cnd}.

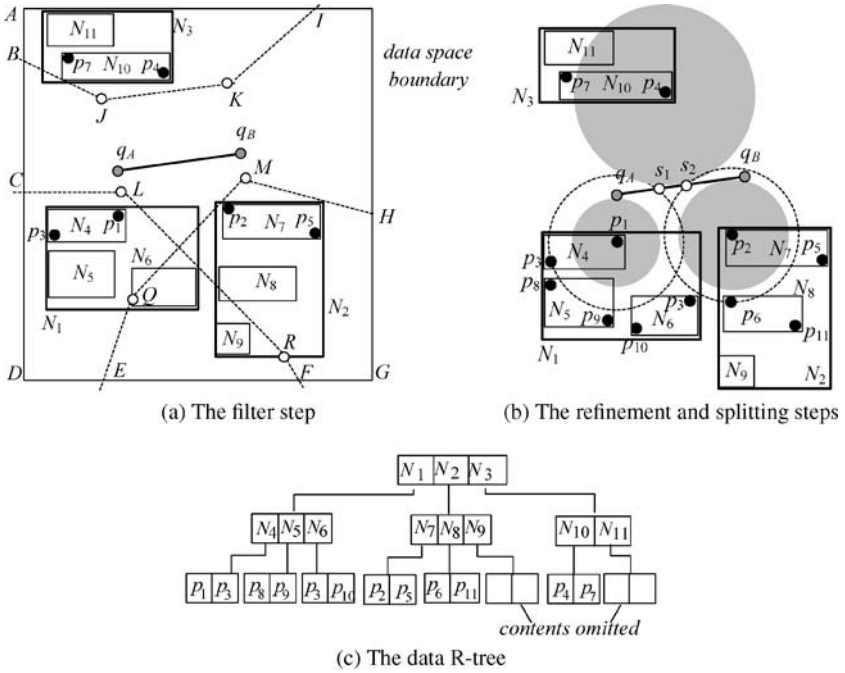


Fig. 26 Illustration of the C-TPL algorithm

By the reasoning of Fig. 24a, p_1 prunes polygon *CLRFD*, where segments *CL* and *RF* lie on $\perp(p_1, q_A)$ and $\perp(p_1, q_B)$ respectively, and point L (R) is on the line perpendicular to $q_A q_B$ passing q_A (q_B).

Since S_{cnd} is not empty, for every MBR/point de-heaped subsequently, C-TPL attempts to prune it using the algorithm *c-k-trim* of Fig. 25. Continuing the example, C-TPL visits node N_2, where N_9 cannot contain any query result (it falls in polygon *CLRFD*, i.e., *c-k-trim* returns $mindist(q_A q_B, N_9^{resM}) = \infty$), and is added to the refinement set S_{rfn}. On the other hand, N_7 and N_8 (MBRs in node N_2) are inserted to H ($= \{N_7, N_5, p_3, N_3, N_6, N_8\}$), using $mindist(q_A q_B, N_7^{resM})$ and $mindist(q_A q_B, N_8^{resM})$ as the sorting keys. The algorithm proceeds by accessing N_7, taking p_2 (found in N_7) as the second candidate (which eliminates polygon *HMQEG*), and inserting N_5, p_3, p_5 to S_{rfn} (they fall in the union of polygons *CLRFD* and *HMQEG*). Then, C-TPL visits nodes N_3, N_{10}, includes p_4 as a candidate (which prunes polygon *ABJKI*), and adds all the remaining entries in H to $S_{rfn} = \{N_9, N_5, N_6, N_8, N_{11}, p_3, p_5, p_7\}$, terminating the filter step.

We illustrate the refinement step using Fig. 26b (which shows some data points hidden from Fig. 26a). A candidate p is a final result if and only if no other data point exists in the circle centered at p with a radius $mindist(q_A q_B, p)$ (the shaded areas represent the circles of the 3 candidates p_1, p_2, p_4). C-TPL invalidates a candidate immediately if its circle contains another candidate. In Fig. 26b, since no candidate can be pruned this way, C-TPL associates each candidate with a value *NNdist*, initialized as its distance to another nearest candidate. In particular, $p_1.NNdist = p_2.NNdist = dist(p_1, p_2)$ (they are closer to each other than to p_4), and $p_4.NNdist = dist(p_4, p_1)$.

The remaining refinement is performed in rounds. The first round is invoked with $P_{rfn} = \{p_3, p_5, p_7\}$ and $N_{rfn} = \{N_9, N_5, N_6, N_8, N_{11}\}$ including the points and MBRs in S_{rfn}, respectively. For every point $p \in P_{rfn}$, C-TPL checks (i) if it falls in the circle of any candidate (i.e., eliminating the candidate), and (ii) if it can update the *NNdist* of any candidate. In our example, no point in P_{rfn} satisfies (i), but $p_1.NNdist$ is modified to $dist(p_1, p_3)$ (i.e., p_3 is the NN of p_1 among all the data points seen in the refinement step). Similarly, $p_2.NNdist$ and $p_4.NNdist$ become $dist(p_2, p_5)$ and $dist(p_4, p_7)$, respectively.

For each MBR $N \in N_{rfn}$, C-TPL first examines whether its *minmaxdist* to a candidate is smaller than the radius of the candidate's shaded circle. In Fig. 26b, the right edge of N_{11} lies inside the circle of p_4, which discards p_4 as a false hit (N_{11} is guaranteed to contain a point that is closer to p_4 than q). Then, C-TPL populates the *toVisit* list of each remaining candidate with those MBRs intersecting its circle, i.e., $toVisit(p_1) = N_5$ and $toVisit(p_2) = \emptyset$ (indicating that p_2 is a final result). In TPL, nodes of N_{rfn} that do not appear in any *toVisit* list can be discarded, while C-TPL collects (among such nodes) into a set S_{splt} those that may contain the NN of a candidate. In our example, S_{splt} contains N_6 and N_8 because $mindist(N_6, p_1)$ and $mindist(N_8, p_1)$ are smaller than the current $p_1.NNdist$ and $p_2.NNdist$, respectively. Note that N_5 is not collected (even though $mindist(p_1, N_5) < p_1.NNdist$) because it belongs to $p_1.toVisit$.

The refinement round finishes by selecting a node (i.e., N_5) from the *toVisit* lists to be visited next, using the same criteria as in TPL. The next round is carried out in the same way with an empty N_{rfn} and a P_{rfn} containing the points p_8, p_9 in N_5. Both of them fall out of the circle of p_1 (i.e., they cannot invalidate p_1). Furthermore, they do not affect the

NNdist of the current candidates. Since all *toVisit* lists are empty, the refinement step is completed, and confirms p_1 and p_2 as the final results.

C-TPL now enters the splitting step which obtains the true *NNdist* for every candidate with respect to the entire dataset. Towards this, it performs a best-first NN search for every confirmed candidate in turn, using the nodes preserved in $S_{splt} = \{N_6, N_1\}$. For every data point encountered during the NN search of one candidate, C-TPL also attempts to update the *NNdist* for the other candidates. Assume that in Fig. 26, the algorithm performs the NN search for p_1 first, and processes the MBRs of S_{splt} in ascending order (i.e., $\{N_6, N_1\}$) of their *mindist* to p_1. Since $mindist(p_1, N_6) < p_1.NNdist$, node N_6 is visited. Although the points p_3 and p_{10} in N_6 do not affect $p_1.NNdist$, $p_2.NNdist$ (originally equal to $dist(p_2, p_5)$) is decreased to $dist(p_2, p_3)$. Since $p_1.NNdist = dist(p_1, p_3)$ is smaller than the *mindist* between p_1 and the next entry N_1 in S_{splt}, the NN search of p_1 finishes. Next, a similar search is performed for p_2 (using the remaining MBRs in S_{splt}), accessing N_8 and finalizing $p_2.NNdist$ to $dist(p_2, p_6)$.

To decide the split points, C-TPL draws 2 circles centering at p_1 and p_2 with radii equal to $p_1.NNdist$ and $p_2.NNdist$, respectively. As shown in Fig. 26, these circles intersect q_Aq_B at s_1 and s_2. Hence, the final result of the C-RNN query is: $\{<\{p_1\}, [q_A, s_1)>, <\emptyset, [s_1, s_2)>, <\{p_2\}, [s_2, q_B]>\}$ (points in $[s_1, s_2)$ do not have any RNN).

Extending C-TPL for $k = 1$ to other values of k requires modifications similar to those discussed in Sect. 4.2 (for extending TPL to $k > 1$). First, in the filter step, *c-k-trim* can be applied for pruning only after S_{cnd} has included at least k points. Second, in the refinement step, the *NNdist* corresponds to the distance between the candidate and its k-th NN among all the points that have been examined in refinement.[1] Furthermore, a candidate p can be invalidated (verified) if there are at least (less than) k points in the circle centered at p with radius $mindist(q_Aq_B, p)$. Third, in the splitting step, a kNN search is needed for each verified candidate. The detailed implementations of the above modifications are illustrated in Figs. 13 (replacing q with q_Aq_B, and *trim* with *c-k-trim*), 27 and 28, which present the pseudo-codes for the filter, refinement, and splitting steps respectively, covering arbitrary k and dimensionality.

We close this section by explaining how BJKS (originally designed for $k = 1$) can be extended to the case of $k > 1$, based on the adapted SAA in Sect. 4.3. Conceptually, for every point q on the query segment q_Aq_B, the filter step of BJKS retrieves as candidates the k NNs of q in each of the 6 partitions around q. All the NN searches can be performed in a single traversal of the R-tree using the technique of [2]. For each candidate p, the refinement step obtains its k-th NN (in the entire data space) p', and confirms p as a result only if the circle centered at p with radius $dist(p, p')$ intersects

Algorithm *C-k-refinement-round*
(q_Aq_B, S_{rslt}, S_{cnd}, P_{rfn}, N_{rfn}, S_{splt})
/* q_Aq_B is the query segment; S_{rslt} is the set of points confirmed to be the query results; S_{cnd} is the set of candidates that have not been verified; P_{rfn} (N_{rfn}) contains the points (nodes) that will be used for candidate verification in this round; S_{splt} is the set of nodes to be processed in the splitting step*/
1. for each point $p \in S_{cnd} \cup S_{rslt}$
2. for each point $p' \in P_{rfn}$
3. if $dist(p, p') < p.NNdist$ then
4. update $p.S_{NN}$, which contains the k NNs of p among the points seen in the refinement step so far
5. if $p.S_{NN}$ has less than k points then $p.NNdist = \infty$
6. else $p.NNdist$ = the distance between p and the farthest point in $p.S_{NN}$
7. if $p \in S_{cnd}$
8-18. these lines are identical to Lines 3-13 in Figure 22
19. if $p.toVisit = \emptyset$ then $S_{cnd} -= \{p\}$, $S_{rslt} \cup = \{p\}$
20. for each node N that is in N_{rfn} but not in any $toVisit$ list
21. for each point p in S_{cnd} and S_{rslt}
22. if $mindist(N, p) < p.NNdist$
23. $S_{splt} = S_{splt} \cup \{N\}$ and go to 19

Algorithm *C-TPL-refinement* (q_Aq_B, S_{cnd}, S_{rfn})
/* q_Aq_B is the query segment; S_{cnd} and S_{cnd} are the candidate and refinement sets returned by the filter step; */
1. $S_{rslt} = S_{splt} = \emptyset$ /* the semantics of S_{rslt} and S_{splt} are explained at the beginning of *c-k-refinement-round*/
2. for each point $p \in S_{cnd}$
3. $p.S_{NN}$ = the k other points in S_{cnd} closest to p
4. set $p.NNdist$ as in Lines 5-6 in *c-k-refinement-round*.
5. $p.counter = k$
6. for each other point p' in S_{cnd}
7. if $dist(p, p') < mindist(p, q_Aq_B)$
8. $p.counter = p.counter - 1$
9. if $p.counter = 0$
10. $S_{cnd} = S_{cnd} - \{p\}$, $S_{splt} = S_{splt} \cup \{p\}$, and goto 6
11. P_{rfn} = the set of points in S_{rfn}; N_{rfn} = the MBRs in S_{rfn}
12. repeat
13. ***C-k-TPL-refinement-round***(q,S_{rslt},S_{cnd},P_{rfn},N_{rfn},S_{splt})
Lines 14-21 are identical to Lines 9-16 in Figure 15

Fig. 27 The refinement algorithm of C-TPL

q_Aq_B. Finally, the split points on q_Aq_B can be computed in the same way as C-TPL, i.e., taking the intersection between the circle of p and q_Aq_B. The extended BJKS, however, is also restricted to 2D space, due to the limitation of SAA.

6 Experiments

In this section, we experimentally evaluate the efficiency of the proposed algorithms, using a Pentium IV 3.4GHz CPU. We deploy 5 real datasets[2], whose statistics are summarized in Table 2. Specifically, *LB*, *NA*, and *LA* contain 2D points representing geometric locations in Long Beach County, North America, and Los Angeles, respectively.

[1] Since points are not encountered in ascending order of their distances to a candidate, in order to maintain *NNdist*, C-TPL also keeps the coordinates of the k NNs for each candidate, as illustrated in Fig. 27.

[2] *LB*, *NA*, and *LA* can be downloaded from *http://www.census.gov/geo/www/tiger*, *Wave* from *http://www.ndbc.noaa.gov*, and *Color* from *http://www.cs.cityu.edu.hk/~taoyf/ds.html*.

Algorithm *C-TPL-splitting* (q_Aq_B, S_{rslt}, S_{splt})
/* q_Aq_B is the query segment; S_{rslt} contains all the query results; S_{splt} involves the nodes that have not be accessed in the previous steps */
1. for each point $p \in S_{rslt}$
2. organize the MBRs in S_{splt} into a heap H using their *mindist* to p as the sorting keys
3. repeat
4. de-heap the top entry (N, key) from H //N is an MBR, and key is its *mindist* to p
5. if $p.NNdist < key$ //the end of the NN search for p
6. $S_{splt} = \{N\} \cup$ {the remaining nodes in H}
7. go to line 1
8. if N is a leaf node
9. for each point p' in N
10. update the S_{NN} and $NNdist$ of every point in S_{rslt} as in Lines 4-6 of *c-k-refinement-round* in Figure 27
11. else //N is an intermediate node
12. for each MBR N' in N
13. insert $(N', mindist(N', p))$ in H
14. obtain the circle that is centered at each point $p \in S_{rslt}$, and its radius equals $p.NNdist$
15. obtain all the split points as the intersection between q_Aq_B and the circles
16. for each sub-segment s of q_Aq_B separated by the split points
17. report <{owners of circles covering s}, s>

Fig. 28 The splitting step algorithm of C-TPL

Wave includes the measurements of wave directions at the National Buoy Center, and *Color* consists of the color histograms of 65k images. For all datasets, each dimension of the data space is normalized to range [0, 10000]. We also create synthetic data following the uniform and Zipf distributions. The coordinates of each point in a uniform dataset are generated randomly in [0, 10000], whereas, for a Zipf dataset, the coordinates follow a Zipf distribution skewed towards 0 (with a skew coefficient[3] 0.8). In both cases, a point's coordinates on various dimensions are mutually independent.

Every dataset is indexed by an R*-tree [1] where the node size is fixed to 1k bytes (we choose a smaller page size to simulate practical scenarios where the dataset cardinality is much larger). Accordingly, the node capacity (i.e., the maximum number of entries in a node) equals 50, 36, 28, and 23 entries for dimensionalities 2, 3, 4, and 5, respectively. The query cost is measured as the sum of the I/O and CPU time, where the I/O time is computed by charging 10ms for each page access. We present our results in two parts, focusing on conventional RkNN search in Sect. 6.1 and continuous retrieval in Sect. 6.2, respectively.

6.1 Results on conventional RkNN queries

We compare TPL against SAA (for 2D data) and SFT because, as discussed in Sect. 2.2, these are the only methods applicable to dynamic datasets. Our implementation of SAA incorporates the optimization of [16] that performs the $6k$ constrained NN queries[4] (in the filter step) with a single traversal of the R-tree. Recall that the filter phase of SFT performs a KNN search where K should be significantly larger than k. In the following experiments, we set K to $10d \cdot k$, where d is the dimensionality of the underlying dataset[5], e.g., $K = 20$ for a RNN query on a 2D dataset. We remind that SFT is approximate, i.e., false misses cannot be avoided unless K is as large as the dataset size.

Table 2 Statistics of the real datasets used

	LB	*NA*	*LA*	*Wave*	*Color*
Dimensionality	2	2	2	3	4
Cardinality	123k	569k	1314k	60k	65k

The experiments in this section aim at investigating the influence of these factors: data distribution, dataset cardinality, dimensionality, value of k, and buffer size. In particular, the first three factors are properties of a dataset, the next one a query parameter, and the last factor is system-dependent. A "workload" consists of 200 queries with the same k whose locations follow the distribution of the underlying dataset. Each reported value in the following diagrams is averaged over all the queries in a workload. Unless specifically stated, the buffer size is 0, i.e., the I/O cost is determined by the number of nodes accessed.

Figure 29 evaluates the query cost (in seconds) of alternative methods as a function of k using the real datasets. The cost of each method is divided in two components, corresponding to the overhead of the filter and refinement steps, respectively. The number on top of each column indicates the 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 omitted for SAA (SFT) because they are fixed to $6k$ ($10d \cdot k$). SAA is not tested on *Wave* and *Color* because the datasets have 3 and 4 dimensions, respectively.

Clearly[6], TPL is the best algorithm for all datasets, especially for large k. In particular, the maximum speedup of TPL over SAA (SFT) is 37 (10), which occurs for *LB* (*NA*) and $k = 16$. Notice that TPL is especially efficient in the refinement step. Recall that TPL performs candidate verification using directly the refinement set (containing the points and nodes pruned) from the filter step, avoiding duplicate accesses to the same node. Furthermore, most candidates are invalidated directly by other candidates or points in

[3] When the skew coefficient equals 1, all numbers generated by the Zipf distribution are equivalent. When the coefficient equals 0, the Zipf distribution degenerates to uniformity.

[4] Stanoi et al. [16] discuss only RNN search ($k = 1$). For $k > 1$, we use the extension of SAA presented in Sect. 4.3.

[5] In the experiments of [15], SFT used $K = 50$ even for $k = 1$. We use a relatively lower K to reduce the cost of this method.

[6] The cost is different from the results reported in the short version [18] of this paper, where query points uniformly distributed in the data space, instead of following the underlying data distribution. Furthermore, all methods consume less CPU time because we used a more powerful machine.

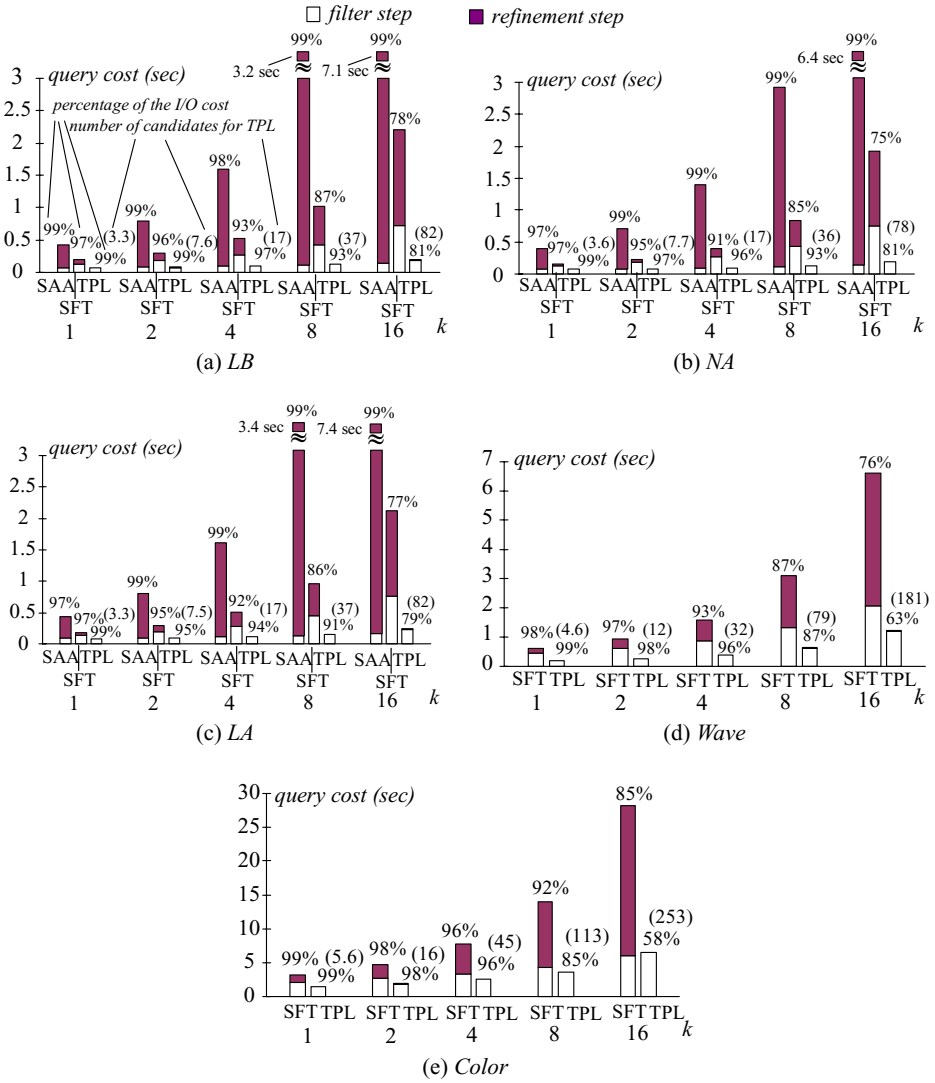


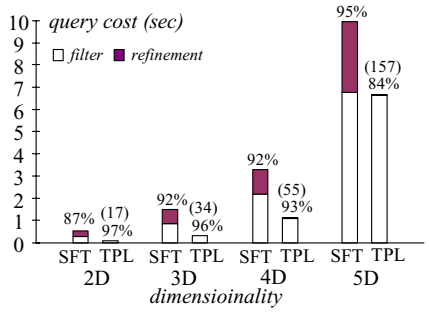
Fig. 29 RkNN cost vs. k (real data)

the refinement set. The remaining candidates can be verified by accessing a limited number of additional nodes.

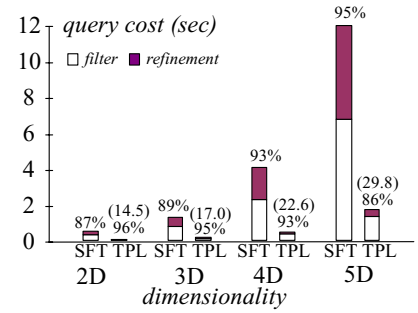
The performance of SAA is comparable to that of TPL in the filter step, because SAA retrieves a small number (compared to the node capacity) of NNs of the query point q, which requires visiting only the few nodes around q. However, SAA is expensive in the refinement phase since it invokes one NN search for every candidate. SFT is most costly in filtering because it retrieves numerous ($10d \cdot k$) candidates; its refinement is more efficient than SAA (due to the superiority of boolean range queries over NN search), but is still less effective than TPL. All the algorithms are I/O bounded. However, as k increases, the CPU cost of TPL occupies a larger fraction of the total time (indicated by its decreasing I/O percentage as k grows) due to the higher cost of *k-trim* which needs to process more half-spaces.

The next set of experiments inspects the impact of the dimensionality. Towards this, we deploy synthetic datasets (*Uniform* and *Zipf*) containing 512k points of dimensionalities 2–5. Figure 30 compares the cost of TPL and SFT (SAA is not included because it is restricted to 2D only) in answering R4NN queries (the parameter $k = 4$ is the median value used in Fig. 29). The performance of both algorithms degrades because, in general, R-trees become less efficient as the dimensionality grows [19] (due to the larger overlap among the MBRs at the same level). Furthermore, the number of TPL candidates increases, leading to higher cost for both the filter and refinement phases. Nevertheless, TPL is still significantly faster than SFT.

To study the effect of the dataset cardinality, we use 3D *Uniform* and *Zipf* datasets whose cardinalities range from 128k to over 2 million. Figure 31 measures the performance of TPL and SFT (in processing R4NN queries) as a function of the dataset cardinality. TPL incurs around a quarter of the overhead of SFT in all cases. The step-wise cost growth corresponds to an increase of the tree height (from 4 to 5). Specifically, for *Uniform* (*Zipf*) data, the increase occurs at cardinality 1024k (2048k).

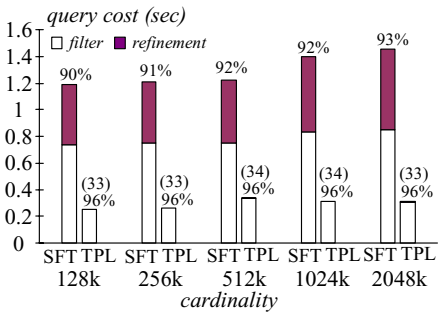


(a) *Uniform*

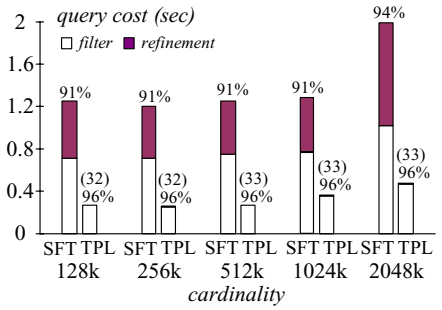


(b) *Zipf*

Fig. 30 RkNN cost vs. dimensionality ($k = 4$, cardinality = 512k)



(a) *Uniform*



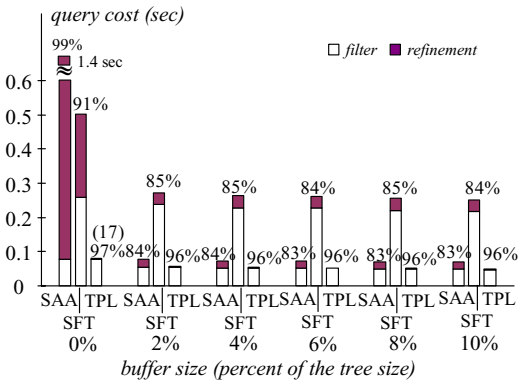
(b) *Zipf*

Fig. 31 RkNN cost vs. cardinality ($k = 4$, dimensionality = 3)

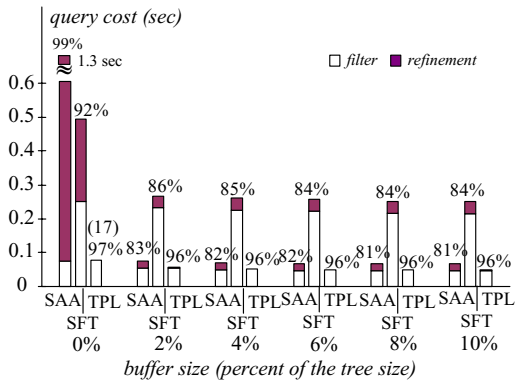
The last set of experiments in this section examines the performance of alternative methods in the presence of a LRU buffer. We process R4NN queries on the 2D synthetic datasets with cardinality 512k, varying the buffer size from 0% to 10% of the R-tree size. Given a workload, we measure the average cost of the last 100 queries (i.e., after "warming up" the buffer with the first 100 queries). Figures 32a and 32b demonstrate the results for *Uniform* and *Zipf* data, respectively. The refinement step of SAA and SFT requires multiple NN/boolean searches that repetitively access several nodes (e.g., the root of the R-tree), and a (small) buffer ensures loading such nodes from the disk only once, leading to dramatic reduction in the overall cost. Similar phenomena are not observed for TPL because it never accesses the same node twice in a single query. For buffer sizes larger than 2%, the cost of all algorithms is decided by their filter phase, and SAA becomes more efficient than SFT. TPL again outperforms its competitors in all cases.

6.2 Results on continuous RkNN queries

Having demonstrated the efficiency of TPL for conventional RNN search, we proceed to evaluate C-TPL for continuous retrieval. The only existing solution BJKS [2] assumes $k = 1$. For $k > 1$, we compare C-TPL against the extended version of BJKS explained at the end of Sect. 5.2. In addition to the parameter k, the query performance is also affected by the length l of the query segment. We generate a segment by first deciding its starting point q_A following the underlying data distribution, and then selecting the ending point q_B randomly on the circle centered at q_A with radius l. A workload contains 200 queries with the same k and l.



(a) *Uniform*



(b) *Zipf*

Fig. 32 RkNN cost vs. buffer size ($k = 4$, dimensionality = 2)

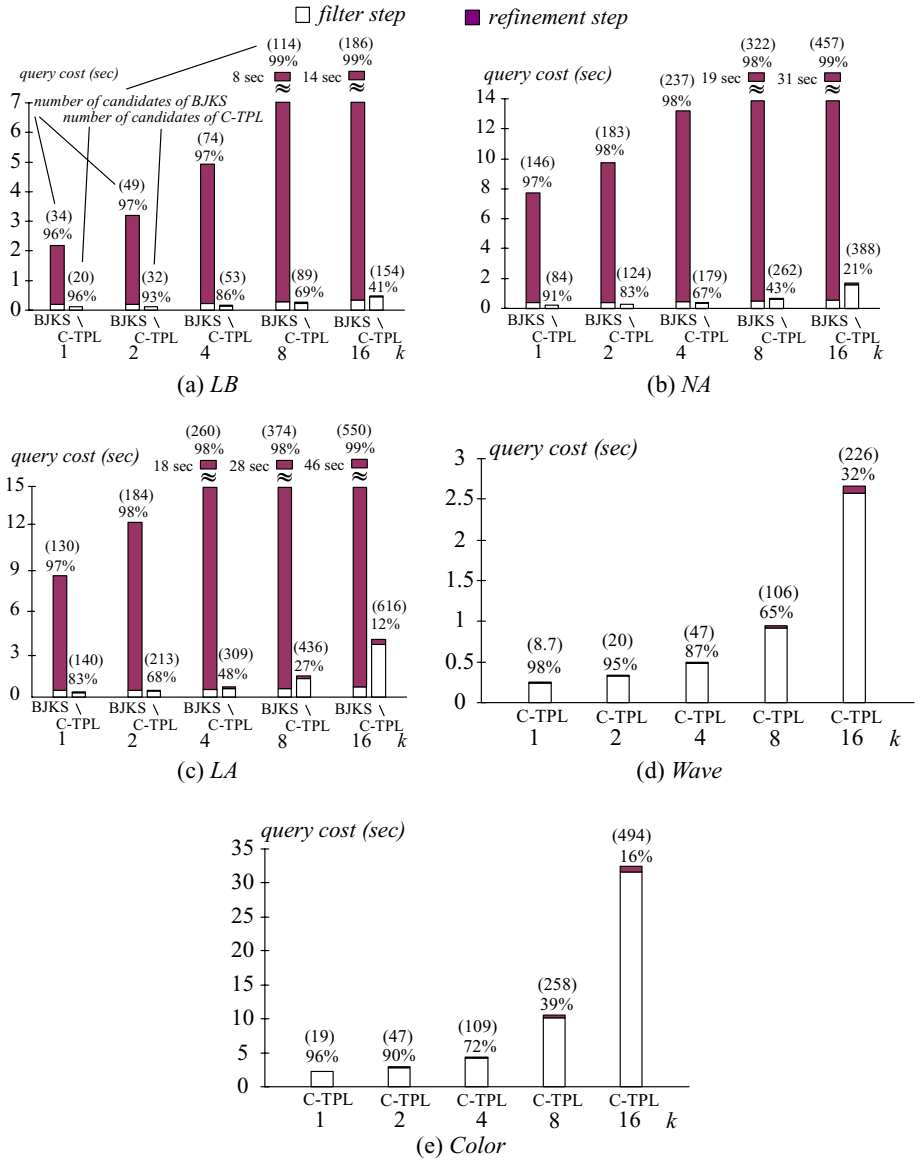


Fig. 33 C-RkNN cost vs. k (real data, $l = 100$)

The first set of experiments fixes l to 100 (recall that each axis of the data space has length 10000), and measures the cost of BJKS and C-TPL as a function of k, using the real datasets. Figure 33 illustrates the results (BJKS is not applicable to *Wave* and *Color*). Similar to the previous diagrams, the percentage of the I/O time in the total cost is shown on top of each column. We also demonstrate the average number of candidates returned by the filter step of C-TPL and BJKS in brackets. C-TPL is the better method in all the experiments, and its comparison with BJKS is similar to that between TPL and SAA in Fig. 29. Compared to conventional RkNN search, the number of candidates retrieved by C-TPL is much higher, which increases the overhead of trimming, and explains why the CPU cost of C-TPL accounts for a larger fraction of the total query time than TPL. Setting k to the median value 4, Fig. 34 examines the performance of BJKS and C-TPL by varying l from 10 to 200. As l increases, both algorithms retrieve more candidates and incur higher overhead. C-TPL still outperforms BJKS significantly.

Next we study the effects of dimensionality and cardinality on the performance of C-TPL. Figure 35 plots the results as a function of the dimensionality using synthetic datasets containing 512k points and workloads with $k = 4$ and $l = 100$. C-TPL is more expensive in high dimensional space because of the degradation of R-trees, and the larger number of query results. In Fig. 36, we focus on 3D space, and evaluate the performance for various dataset sizes. Similar to Fig. 30, the cost growth demonstrates a step-wise behavior due to the increase of the R-tree height at 1024k and 2048k for *Uniform* and *Zipf*, respectively.

Finally, we demonstrate the influence of LRU buffers on BJKS and C-TPL. As with the settings of Fig. 32, the two algorithms are used to process R4NN queries on a 2D

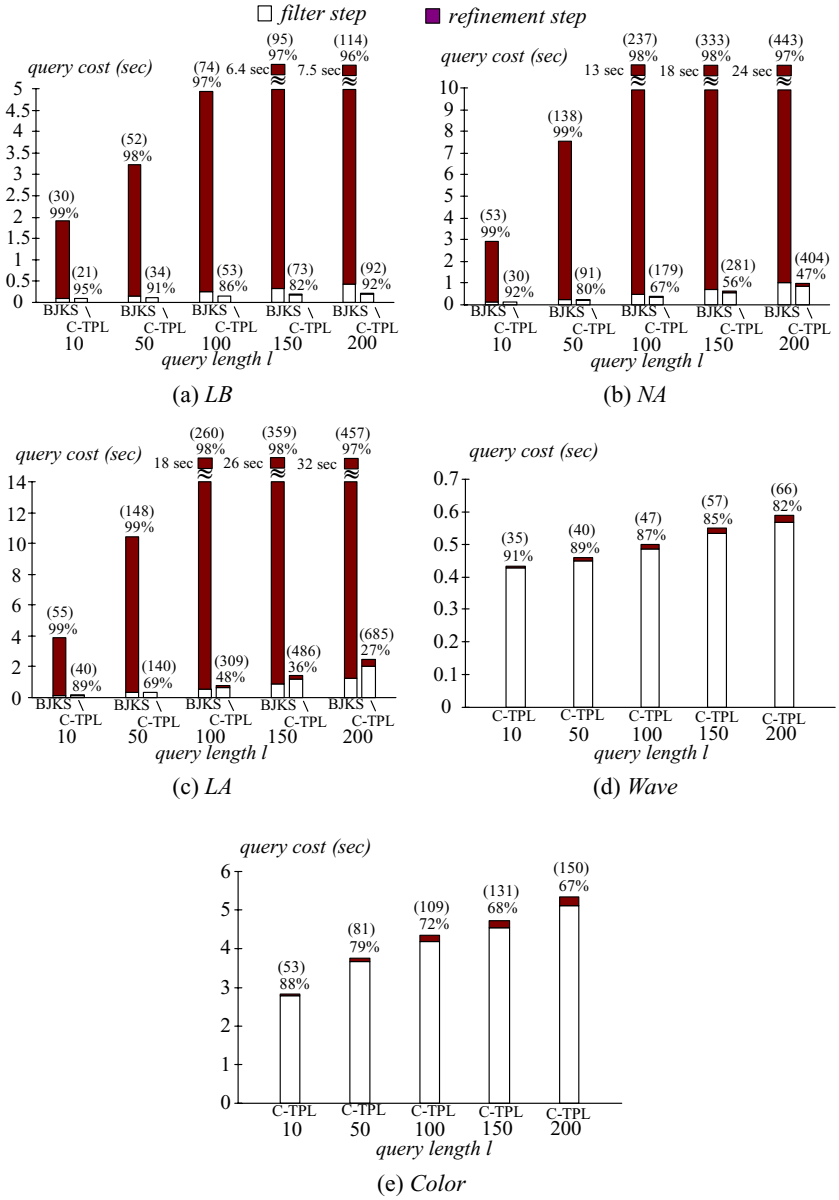


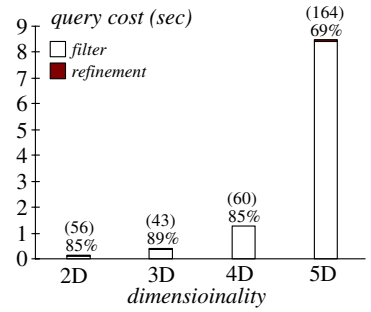
Fig. 34 C-RkNN cost vs. l (real data, $k = 4$)

dataset with cardinality 512k, as the buffer size changes from 0% to 10% of the size of the corresponding R-tree. For each workload, only the cost of the last 100 queries is measured. Figures 37 illustrates the overhead as a function of the buffer size, demonstrating phenomena similar to those in Fig. 32. Specifically, BJKS improves significantly given a small buffer, but C-TPL is consistently faster regardless of the buffer size.

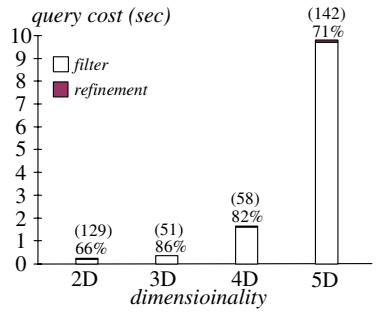
7 Conclusions

Existing methods for reverse nearest neighbor search focus on specific aspects of the problem, namely static datasets, retrieval of single ($k = 1$) RNNs or 2D space. This paper proposes TPL, the first general algorithm for exact RkNN search on dynamic, multidimensional datasets. TPL follows a filter-refinement methodology: a 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. An extensive experimental comparison verifies that, in addition to applicability, TPL outperforms the previous techniques, even in their restricted focus. Furthermore, it leads to a fast algorithm for answering continuous RkNN queries (again, for arbitrary k and dimensionality).

A promising direction for future work concerns the extension of the general framework of TPL to alternative versions of the problem. One such example refers to metric

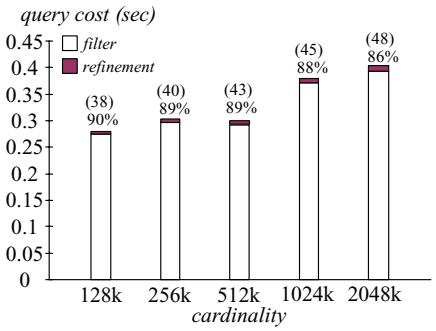


(a) *Uniform*

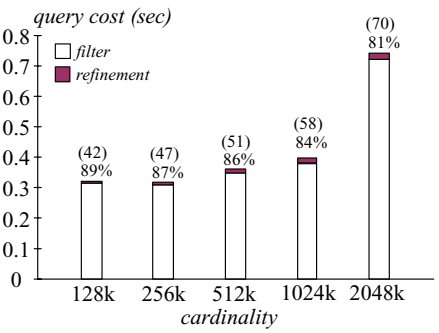


(b) *Zipf*

Fig. 35 C-TPL cost vs. dimensionality ($k = 4$, $l = 100$, cardinality = 512k)

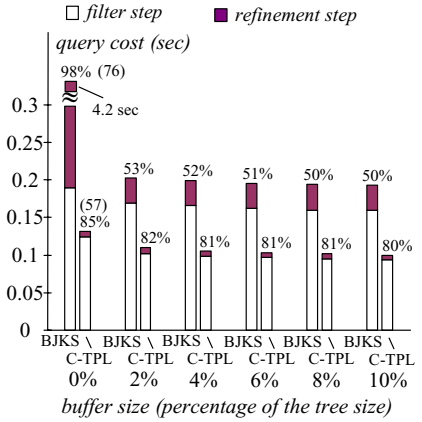


(a) *Uniform*

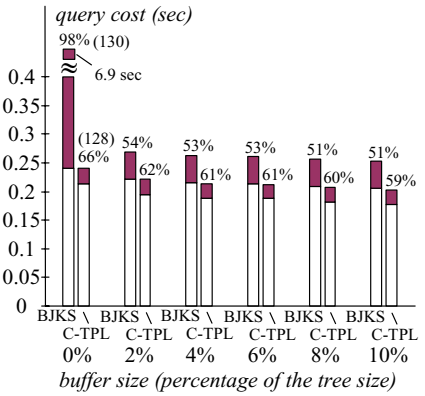


(b) *Zipf*

Fig. 36 C-TPL cost vs. cardinality ($k = 4$, $l = 100$, dimensionality = 3)



(a) *Uniform*



(b) *Zipf*

Fig. 37 C-RkNN cost vs. buffer size ($k = 4$, dimensionality = 2)

space where the triangular inequality has to be used (instead of bisectors) for pruning the search space. We also plan to investigate the application of the proposed methodology to other forms of RNN retrieval, particularly, bichromatic [10, 17] and aggregate [11] RNN queries. Finally, it would be interesting to develop analytical models for estimating (i) the expected number of RNNs depending on the data properties (e.g., dimensionality, distribution, etc.) and (ii) the execution cost of RNN algorithms. Such models will not only facilitate query optimization, but may also reveal new problem characteristics that could lead to even better solutions.

Appendix: proofs for lemmas

Lemma 1 *Given a query point q and an MBR N in 2D space, let N^{resP} be the part (residual polygon) of N satisfying a set S of half-spaces, and N^{resM} the residual MBR computed (by the algorithm in Fig. 11) using the half-spaces in S. Then, $mindist(N^{resM}, q) = mindist(N^{resP}, q)$ in all cases.*

Proof Since N^{resM} always contains N^{resP}, if N^{resM} is empty, N^{resP} is also empty, in which case $mindist(N^{resP}, q) = mindist(N^{resM}, q) = \infty$. Hence, it suffices to discuss the possibility where N^{resM} exists. We use the name "contact" for the point in N^{resM} (N^{resP}) nearest to q. The

following analysis shows that the contacts of N^{resM} and N^{resP} are the same point, which, therefore, validates the lemma.

We achieve this by induction. First, if S is empty, both N^{resP} and N^{resM} are equal to the original MBR N, and obviously, their contacts are identical. Assuming that N^{resP} and N^{resM} have the same contact for all sets S whose cardinalities are no more than m (≥ 0), next we prove that they have the same contact for any set S with a cardinality $m+1$. Let R^P (R^M) be the residual polygon (MBR) with respect to the first m half-spaces in S, and c^P (c^M) be the contact of R^P (R^M). By the inductive assumption, $c^P = c^M$. Since R^M is a rectangle, c^M appears either at a corner or on an edge of R^M. We discuss these two cases separately.

Case 1: (c^M is a corner of R^M): Without loss of generality, assume that the coordinates of c^M are larger than or equal to those of q on both dimensions. Denote h as the $(m+1)$-st half-space in S (recall that h contains q). If c^M satisfies h, then c^M and c^P remain the contacts of N^{resM} and N^{resP} respectively, i.e., the final residual MBR and polygon (after applying all the half-spaces in S) still have the same contact.

Now let us consider the scenario that c^M violates h. Hence, the boundary of h (a line) must intersect segment $c^M q$, and cross either the left or bottom edge of R^M (if not, R^{resM} becomes empty). Due to symmetry, it suffices to consider that line h intersects the left edge, as in Fig. 38a. The intersection point is exactly the contact c^{resM} of N^{resM} (note that the part of R^M lower than c^{resM} will not appear in N^{resM}). Thus, it remains to prove that this intersection is also the contact c^{resP} of N^{resP}.

Observe that c^{resP} must lie inside the shaded area, due to the fact that R^P (which contains c^{resP}) is entirely bounded by R^M. Actually, c^{resP} definitely falls on the part of line h inside R^M. To prove this, assume, on the contrary, that c^{resP} is at some position (e.g., c_1) above line h. Then, the segment connecting c_1 and c^P intersects line h at a point c_2. Since both c_1 and c^P belong to R^P, and R^P is convex, c_2 also lies in R^P, indicating that c_2 belongs to N^{resP}, too (N^{resP} is the part of R^P qualifying half-space h). Point c_2, however, is closer to q than c_1, contradicting the assumption that c_1 is the contact c^{resP} of N^{resP}.

It follows that if $c^{resP} \neq c^{resM}$ (e.g., $c^{resP} = c_2$ in Fig. 38a), then the x-coordinate of c^{resP} must be larger than that of c^{resM}, which, in turn, is larger than that of q. As c^{resM} is closer to q than c_2, the hypothesis that c^{resM} is not the contact of N^{resP} also implies that c^{resM} does not belong to N^{resP}, meaning that c^{resM} does not fall in a half-space h' (one of the first m planes) in S. However, since both c^M (the contact of the residue MBR R^M after applying with the first m planes) and c_2 qualify h', the boundary of h' must cross segment $c^{resM}c_2$ and $c^M c^{resM}$, but not $c^M q$. This is impossible (see Fig. 38a), thus verifying $c^{resP} = c^{resM}$.

Case 2: (c^M is on an edge of R^M): Assume that c^M ($= c^P$) lies on the left edge of R^M as illustrated in Fig. 38b (the scenarios where c^M is on other edges can be proved in the same manner). As in Case 1, if c^M satisfies the $(m+1)$-st half-space h in S, both c^M and c^P remain the contacts of N^{resM} and N^{resP}, respectively. Otherwise, line h intersects segment $c^M q$, and may cross the left edge of R^M above or below c^M. Due to symmetry, let us focus on the scenario where h intersects the left edge at a point above c^M (Fig. 38b), which is the contact c^{resM} of N^{resM}. The goal is to show that c^{resM} is also the contact c^{resP} of N^{resP}. Since R^M completely encloses R^P, c^{resP} falls in the shaded triangle of Fig. 38b. Then, $c^{resP} = c^{resM}$ can be established in exactly the same way as in Case 1 (notice that the relative positions of q, c^M, c^{resM}, h, and the shaded area are identical in Figs. 38a and 38b).

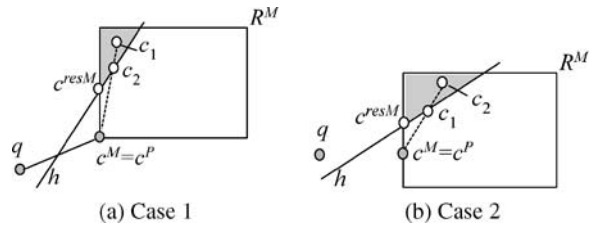


Fig. 38 Illustration of the proof of Lemma 1

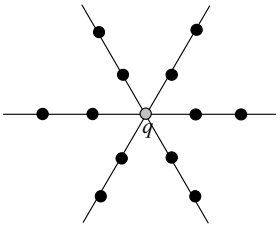


Fig. 39 The worst case of SAA

Note that Lemma 1 is also useful to "constrained k nearest neighbor search" [6], which finds the k data points in a polygonal constraint region that are closest to a query point q (recall that such queries are the building-block for SAA and its extended version discussed in Sect. 4.3). As shown in [6], the best-first algorithm can process a constrained kNN search optimally (i.e., accessing only the nodes of an R-tree that need to be visited by any algorithm), provided that it is possible to compute the minimum distance from q to the part of an MBR N inside the polygon. Lemma 1 provides an efficient way for obtaining this distance in 2D space, which is equal to the *mindist* from q to the residue MBR of N, after trimming N using the half-spaces bounding the constraint region.

Lemma 2 *Given a 2D Rk*NN *query q, divide the space around q into 6 equal partitions as in* Fig. 39. *Then, the k NNs of q in each partition are the only possible results of q. Furthermore, in the worst case, all these points may be the actual Rk*NNs.

Proof We first prove the first part of the lemma: if a point p is not among the k NNs of q in a partition, p cannot be a query result. Its correctness for $k = 1$ has been established in [16], which also shows an interesting corollary: if p' is the closest NN of q in the same partition S that contains p, then $dist(p, p') < dist(p, q)$. Utilizing these properties, in the sequel, we will prove that, if the first part of the lemma is true for $k = m$ (where m is an arbitrary integer), it also holds for $k = m+1$. In fact, if p' is removed from the dataset, we know that p is not among the m NNs of q in S. By our inductive assumption, there exist at least m points (different from p') that are closer to p than q. Since we already have $dist(p, p') < dist(p, q)$, there are at least $m+1$ points in the original dataset closer to p than q, i.e., p is not a R$(m+1)$NN of q.

In order to prove the second part of the lemma (i.e., the number of RkNNs in 2D space can be $6k$), it suffices to construct such an example. Consider the 6 rays that partition the data space around q. On each ray, we place k points in ascending order of their distances to q as follows: the first point has distance 1 to q, and every subsequent point has distance 1 to the previous one. Figure 39 shows such an example for $k = 2$. These $6k$ points constitute a dataset where all the points have q as one of their k NNs. □

Lemma 3 *Both L'_A and L'_B belong to a d-dimensional plane satisfying the following equation:*

$$\sum_{i=1}^{d}(2p[i] - q_A[i] - q_B[i]) \cdot x[i] + \sum_{i=1}^{d}\left(q_A[i] \cdot q_B[i] - \frac{p[i]^2}{2}\right) = 0 \quad (2)$$

where $x[i]$ denotes the i-th ($1 \le i \le d$) coordinate of a point in the plane, and similarly, $p[i]$, $q_A[i]$, $q_B[i]$ describe the coordinates of p, q_A, and q_B, respectively.

Proof We prove the lemma only for L'_A because the case of L'_B is similar. We achieve this by representing L'_A using the coordinates of p, q_A, and q_B. For this purpose, we obtain the equation of L_A:

$$\sum_{i=1}^{d}((q_B[i]-q_A[i])\cdot x[i]) - \sum_{i=1}^{d}((q_B[i]-q_A[i])\cdot q_A[i]) = 0 \quad (3)$$

and the equation of $\perp(q_A, p)$:

$$\sum_{i=1}^{d}((q_A[i]-p[i])\cdot x[i]) - \sum_{i=1}^{d}\frac{q_A[i]^2-p[i]^2}{2} = 0 \quad (4)$$

Therefore, L'_A includes the points x that satisfy Eqs. (3) and (4) simultaneously. Hence, all the d-dimensional planes[7] that cross L'_A are captured by:

$$\sum_{i=1}^{d}(q_B[i]-q_A[i])x[i] - \sum_{i=1}^{d}(q_B[i]-q_A[i])q_A[i] + \lambda\left(\sum_{i=1}^{d}(q_A[i]-p[i])x[i] - \sum_{i=1}^{d}\frac{q_A[i]^2-p[i]^2}{2}\right) = 0 \quad (5)$$

where various planes are distinguished with different λ (an arbitrary real number). The plane of Eq. (2) corresponds to setting $\lambda = 2$. □

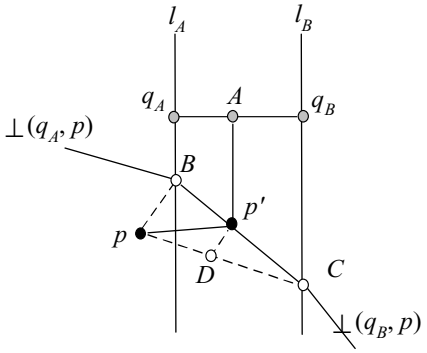
Lemma 4 *Given a query segment q_Aq_B and a data point p, consider half-spaces $HS_p(q_A, p)$, $HS_p(q_B, p)$ (decided by bisectors $\perp(q_A, p)$ and $\perp(q_B, p)$), and the half-space $HS_p(L)$ that is bounded by the plane L of Eq. (1) and contains p. Then, no point in $HS_p(q_A, p) \cap HS_p(q_B, p) \cap HS_p(L)$ can be a RNN of any point q on q_Aq_B.*

Proof Let L_A (L_B) be a d-dimensional plane that is perpendicular to q_Aq_B and crosses q_A (q_B). Plane L_A defines two half-spaces: $HS_{q_B}(L_A)$ that contains q_B, and $HS_{\neg q_B}(L_A)$ that does not (e.g., in Fig. 24a, $HS_{q_B}(L_A)/HS_{\neg q_B}(L_A)$ is the area on the right/left of line l_A). Similarly, L_B also introduces two half-spaces $HS_{q_A}(L_B)$ and $HS_{\neg q_A}(L_B)$, respectively. Note that $HS_{\neg q_B}(L_A)$, $HS_{\neg q_A}(L_B)$, and $HS_{q_B}(L_A) \cap HS_{q_A}(L_B)$ are 3 disjoint regions whose union constitutes the entire data space. For a point p' that falls in $HS_{\neg q_B}(L_A)$ or $HS_{\neg q_A}(L_B)$, its minimum distance to segment q_Aq_B equals $dist(q_A, p')$ or $dist(q_B, p')$, respectively. For a point p' that lies in $HS_{q_B}(L_A) \cap HS_{q_A}(L_B)$, however, the distance from p' to q_Aq_B equals the distance from p' to its projection on q_Aq_B.

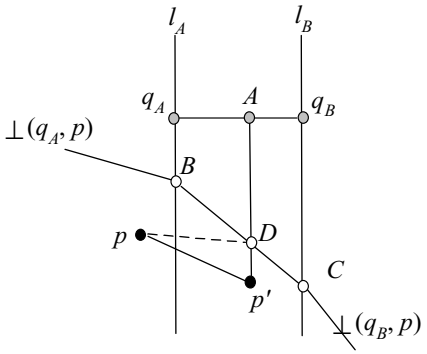
As shown in the proof of Lemma 3, L_A, $\perp(q_A, p)$, and L intersect at the same $(d-1)$-dimensional plane, and similarly, L_B, $\perp(q_B, p)$, and L intersect at another $(d-1)$-dimensional plane. Since L_A and L_B are parallel to each other, they divide $HS_p(q_A, p) \cap HS_p(q_B, p) \cap HS_p(L)$ into 3 disjoint regions: (i) $HS_p(q_A, p) \cap HS_{\neg q_B}(L_A)$, (ii) $HS_p(q_B, p) \cap HS_{\neg q_A}(L_B)$, and (iii) $HS_{q_B}(L_A) \cap HS_{q_A}(L_A) \cap HS_p(L)$. For example, in Fig. 24a, the 3 regions are polygons $ABFE$, CDG, and $BCGF$, respectively. Let p' be a point in region (i), which satisfies $dist(p, p') < dist(q_A, p')$ (because p' is in $HS_p(q_A, p)$). Since $dist(q_A, p')$ is the minimum distance from p' to q_Aq_B (recall that p' is in $HS_{\neg q_B}(L_A)$), for any point q on q_Aq_B, it holds that $dist(p, p') < dist(q, p')$, meaning that p' cannot be a RNN of q. By symmetry, no point p' in region (ii) can be the RNN of any point on q_Aq_B.

The remaining part of the proof will show that no point p' in region (iii) can be a query result either. We first prove this in 2D space,

[7] Strictly speaking, Eq. (5) does not include $\perp(q_A, p)$. We ignore this special case because it does not affect the subsequent discussion.



(a) Point p is on segment BC



(b) Point p is below segment BC

Fig. 40 Illustration of the proof in 2D space

where region (iii) is the area bounded by lines l_A, l_B, and segment BC in Fig. 24a. Our analysis distinguishes two cases, depending on whether p' lies on or below segment BC, respectively. Figure 40a demonstrates the first case (p' is on BC), where A is the projection of p' on q_Aq_B. Our goal is to establish, for any point q on q_Aq_B, the inequality $dist(p, p') \le d(p', q)$ (which indicates p' is not a RNN of q). Next, we derive an ever stronger result: $dist(p, p')$ is actually no more than $d(p', A)$, which is a lower bound for $d(p', q)$.

Denote r as the ratio between the lengths of segments Bp' and $p'C$, i.e., $r = dist(B, p')/dist(p', C)$. Then:

$$dist(p', A) = r \cdot dist(q_B, C) + (1-r) \cdot dist(q_A, B) \quad (6)$$

Let D be the intersection between segment pC and a line that passes p' and is parallel to Bp. Since $\frac{dist(p,D)}{dist(p,C)} = \frac{dist(B,p')}{dist(B,C)} = r$ and $\frac{dist(p',D)}{dist(p,B)} = \frac{dist(p',C)}{dist(B,C)} = 1-r$, we have:

$$dist(p, D) = r \cdot dist(p, C) \quad (7)$$
$$dist(p', D) = (1-r) \cdot dist(p, B) \quad (8)$$

Since B and C are on the bisectors $\perp(q_A, p)$ and $\perp(q_B, p)$ respectively, it holds that $dist(p, B) = dist(q_A, B)$ and $dist(p, C) = dist(q_B, C)$, leading to:

$$\begin{aligned} &dist(p', A) \\ &= r \cdot dist(p, C) + (1-r) \cdot dist(p, B) \quad \text{(by Eq. (6))} \\ &= dist(p, D) + dist(p', D) \quad \text{(by Eqs. (7) and (8))} \\ &\ge dist(p, p') \quad \text{(by triangle inequality)} \end{aligned}$$

The equality in the above formula holds only if p' is at B or C.

Next, we discuss the second case, namely, point p' appears below segment BC (meanwhile, between lines l_A and l_B), shown in Fig. 40b

where A is again the projection of p' on q_Aq_B. Similar to the first case, we aim at proving $dist(p, p') < dist(p', A)$ (which results in $dist(p, p') \leq dist(p', q)$ for any point q on segment q_Aq_B). Let D be the intersection between segments BC and $p'A$. As proved earlier, $dist(p, D) \leq dist(D, A)$, and hence:

$$dist(p, D) + dist(p', D) \leq dist(D, A) + dist(p', D) = dist(p', A) \quad (9)$$

By triangle inequality, the left part of the above inequality is larger than $dist(p, p')$, thus verifying $dist(p, p') < dist(p', A)$. Although the position of p in Figs. 40a and 40b is to the left of l_A, it is not hard to observe that the above analysis holds for any position of p.

So far we have proved that, in 2D space, no point in region (iii), i.e., $HS_{q_B}(L_A) \cap HS_{q_A}(L_A) \cap HS_p(L)$ can be a query result. Now we proceed to show that this is also true for arbitrary dimensionality d, through a careful reduction to the 2D scenario. Specifically, let us construct a coordinate system as follows. The origin of the system is point q_A, and the first axis coincides with segment q_Aq_B. This axis and point p decide a 2D sub-space, and in this sub-space, the line perpendicular to q_Aq_B is taken as the second axis. Then, the remaining $d-2$ dimensions are decided arbitrarily with the only requirement that all the resulting d dimensions are mutually orthogonal. The rationale for introducing such a coordinate system is that the coordinates of p, q_A, and q_B are 0 on all the dimensions except the first two, i.e., they lie on the d-dimensional plane L_c: $x[3] + x[4] + \cdots + x[d] = 0$. As a result, Eq. (2), the representation of plane L, is simplified to (note the upper limits of the two summations):

$$\sum_{i=1}^{2}(2p[i] - q_A[i] - q_B[i]) \cdot x[i] + \sum_{i=1}^{2}\left(q_A[i] \cdot q_B[i] - \frac{p[i]^2}{2}\right) = 0 \quad (10)$$

The above formula implies that (i) L is perpendicular to L_c, and (ii) every point x in the half-space $HS_p(L)$ (i.e., the half-space bounded by L containing p) satisfies the inequality that results from changing the equality sign in Eq. 10 to "$\geq$". Another benefit from the constructed coordinate system is that planes L_A and L_B are described concisely by equations $x_1[1] = 0$ and $x_1[1] = q_B[1]$, respectively ($q_B[1]$ is the coordinate of q_B on the first axis).

Consider any point p' that is in $HS_{q_B}(L_A) \cap HS_{q_A}(L_A) \cap HS_p(L)$; let A be its projection on q_Aq_B. As mentioned earlier, q_A and q_B belong to plane L_c, and hence, A also lies on this plane, implying $A[i] = 0$ for $3 \leq i \leq d$. To prove that p' is not a RNN of any point on q_Aq_B, (following the reasoning in the 2D case) we will show that $dist(p, p') \leq dist(p', A)$. Since

$$dist(p, p') = \sum_{i=1}^{d}(p[i] - p'[i])^2 = \sum_{i=1}^{2}(p[i] - p'[i])^2 + \sum_{i=3}^{d} p'[i]^2 \quad (p[i] = 0 \text{ for } 3 \leq i \leq d) \quad (11)$$

and

$$dist(p', A) = \sum_{i=1}^{d}(p'[i] - A[i])^2 = \sum_{i=1}^{2}(p'[i] - A[i])^2 + \sum_{i=3}^{d} p'[i]^2 \quad (12)$$

it suffices to show that $\Sigma_{i=1}^{2}(p[i] - p'[i])^2 \leq \Sigma_{i=1}^{2}(p[i] - A[i])^2$. Proving this inequality can be reduced to the 2D case we solved earlier, by projecting L_A, L_B, L, and p' into a 2D sub-space that involves only the first 2 dimensions. Specifically, the projection of L_A (L_B) is a line l_A (l_B) that crosses q_A (q_B), and is perpendicular to segment q_Aq_B. The projection of L is a line l that intersects l_A (l_B) at a point equidistant to p and q_A (q_B). Finally, p' is projected into a point between l_A and l_B that falls either on l, or on the same side of l as p. This leads to the situation in Fig. 40a or 40b, where l is the line passing segment BC. Thus, we complete the proof.

Acknowledgements This work was supported by two CERG grants from the government of HKSAR. In particular, Yufei Tao and Xiaokui Xiao were supported by Grant CityU 1163/04E, and Dimitris Papadias and Xiang Lian by Grant HKUST 6180/03E.

References

1. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-tree: An efficient and robust access method for points and rectangles. In: SIGMOD, pp. 322–331 (1990)
2. Benetis, R., Jensen, C.S., Karciauskas, G., Saltenis, S.: Nearest neighbor and reverse nearest neighbor queries for moving objects. In: IDEAS, pp. 44–53 (2002)
3. Berchtold, S., Keim, D.A., Kriegel, H.-P.: The X-tree: An index structure for high-dimensional data. In: VLDB, pp. 28–39 (1996)
4. Berg, M., Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications. Springer (2000)
5. Cheung, K.L., Fu, A.W.-C.: Enhanced nearest neighbour search on the R-tree. SIGMOD Record **27**(3), 16–21 (1998)
6. Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., Abbadi, A.E.: Constrained nearest neighbor queries. In: SSTD, pp. 257–278 (2001)
7. Goldstein, J., Ramakrishnan, R., Shaft, U., Yu, J.-B.: Processing queries by linear constraints. In: PODS, pp. 257–267 (1997)
8. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: SIGMOD, pp. 47–57 (1984)
9. Hjaltason, G.R., Goldstein, H.: Distance browsing in spatial databases. TODS **24**(2), 265–318 (1999)
10. Korn, F., Muthukrishnan, S.: Influence sets based on reverse nearest neighbor queries. In: SIGMOD, pp. 201–212 (2000)
11. Korn, F., Muthukrishnan, S., Srivastava, D.: Reverse nearest neighbor aggregates over data streams. In: VLDB, pp. 814–825 (2002)
12. Lin, K.-I., Nolen, M., Yang, C.: Applying bulk insertion techniques for dynamic reverse nearest neighbor prob lems. In: IDEAS, pp. 290–297 (2003)
13. Maheshwari, A., Vahrenhold, J., Zeh, N.: On reverse nearest neighbor queries. In: CCCG, pp. 128–132 (2002)
14. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: SIGMOD, pp. 71–79 (1995)
15. Singh, A., Ferhatosmanoglu, H., Tosun, A.S.: High dimensional reverse nearest neighbor queries. In: CIKM, pp. 91–98 (2003)
16. Stanoi, I., Agrawal, D., Abbadi, A.E.: Reverse nearest neighbor queries for dynamic databases. In: ACM SIGMOD workshop, pp. 744–755 (2000)
17. Stanoi, I., Riedewald, M., Agrawal, D., Abbadi, A.E.: Discovery of influence sets in frequently updated databases. In: VLDB, pp. 99–108 (2001)
18. Tao, Y., Papadias, D., Lian, X.: Reverse knn search in arbitrary dimensionality. In: VLDB, pp. 744–755 (2004)
19. Theodoridis, Y. Sellis, T.K.: A model for the prediction of R-tree performance. In: PODS, pp. 161–171 (1996)
20. Yang, C., Lin, K.-I.: An index structure for efficient reverse nearest neighbor queries. In: ICDE, pp. 485–492 (2001)