Reverse Nearest Neighbor Search in Metric Spaces

Yufei Tao, Man Lung Yiu, and Nikos Mamoulis

Abstract—Given a set $\mathcal{D}$ of objects, a *reverse nearest neighbor* (RNN) query returns the objects o in $\mathcal{D}$ such that o is closer to a query object q than to any other object in $\mathcal{D}$, according to a certain similarity metric. The existing RNN solutions are not sufficient because they either 1) rely on precomputed information that is expensive to maintain in the presence of updates or 2) are applicable only when the data consists of "Euclidean objects" and similarity is measured using the L_2 norm. In this paper, we present the first algorithms for efficient RNN search in generic metric spaces. Our techniques require no detailed representations of objects, and can be applied as long as their mutual distances can be computed and the distance metric satisfies the triangle inequality. We confirm the effectiveness of the proposed methods with extensive experiments.

Index Terms—Reverse nearest neighbor, metric space.

1 Introduction

A *reverse nearest neighbor* (RNN) query returns objects in a data set that have a query object q as their nearest neighbors (NN) respectively, according to some similarity metric. Consider, for example, Fig. 1a where the data set consists of 4 points $p_1, \ldots, p_4$. Assuming that the similarity between two points corresponds to their Euclidean distance, the RNN query q (the black dot) returns p_1 and p_2. In particular, p_1 (similarly, p_2) is a result because q is its NN—p_1 is closer to q than to any other object in the data set. It is important to note that the NN of q (i.e., p_3) is *not* necessarily the RNN of q (the NN of p_3 is p_4, instead of q). The concept can be easily generalized to "reverse k nearest neighbors" (RkNN), i.e., a point p is an RkNN of a query point q if q is one of the k NNs of p.

RNN processing has received considerable attention in the past few years [2], [11], [12], [17], [18], [19], [20] because it is a fundamental operation in data mining [14]. Intuitively, the RNNs of an object o are those objects on which o has significant "influence" (by being their nearest neighbors). Such "influence sets" may lead to useful observations on the correlation among data, as shown by Korn and Muthukrishnan in their pioneering paper [11]. Indeed, RNN search is inherent to any applications where the similarity between two objects can be quantified into a single value, using an appropriate evaluating process.

The first motivation of this paper is that, except for several methods relying on precomputation (and, thus, incurring expensive space/update overhead), the existing solutions have rather limited applicability. As discussed in the next section, they assume that similarity is computed according to the L_2 norm (i.e., Euclidean distance), which is not true for many applications in practice. We illustrate this using two representative applications given in [11].

Application 1 (business location planning). Consider evaluating the impact of opening a supermarket at a selected location. For this purpose, the manager would examine how many residential areas would find the new outlet as their nearest choice. The traveling distance, obviously, is not the Euclidean distance between a residential area and the supermarket. Its precise computation should take into account the underlying road network, i.e., the traveling distance is the length of the shortest path connecting the two locations.

Application 2 (profile-based marketing). Consider that a cell phone company has collected the profiles of customers regarding the services they prefer. To predict the popularity of a new plan, the market analyst would like to know how many profiles have the plan as their best match, against the existing plans in the market. Most likely, Euclidean distance is not the best metric for capturing the matching degree of a profile.

Our second motivation is that, somewhat surprisingly, the existing research on RNN processing targets only "Euclidean objects," which can be represented as points in a multidimensional space, where an RNN query can be accelerated by deploying various geometric properties to effectively prune the search space. Unfortunately, these properties are restricted to Euclidean spaces, rendering the previous methods inapplicable to complex data (such as time series, DNA sequences, etc.) that do not have obvious Euclidean modeling. This problem is currently preventing the analysis of such data using the data mining tools [14] based on RNN search.

Application 3 (clustering and outlier detection). Since it is difficult to visualize metric data, we use a Euclidean example in Fig 1b (the discussion extends to metric spaces directly). Points in the two clusters at the bottom of the figure are very close to each other, whereas objects' mutual distances are relatively larger for the cluster at the top. Nanopoulos et al. [14] suggest that, given an appropriate value of k, points that do not have any RkNN are *outliers* (an outlier is an object that does not belong to any cluster). For

- *Y. Tao is with the City University of Hong Kong, Tat Chee Avenue, Hong Kong. E-mail: taoyf@cs.cityu.edu.hk.*
- *M.L. Yiu and Nikos Mamoulis are with the University of Hong Kong, Pofulam Road, Hong Kong. E-mail: {myliu2, nikos}@cs.hku.hk.*

Manuscript received 11 May 2005; revised 5 Dec. 2005; accepted 7 Apr. 2006; published online 19 June 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0182-0505.

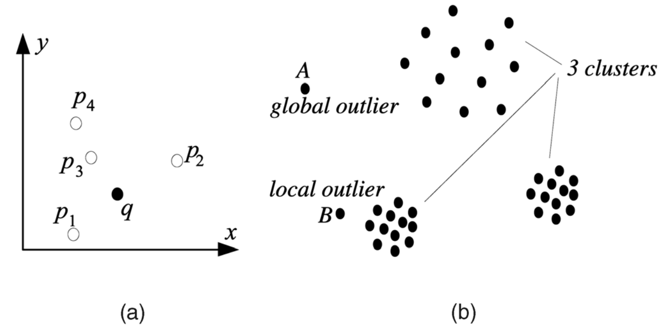


Fig. 1. RNN definition and its relevance to data mining. (a) An example. (b) Clustering and outlier detection.

example, for $k = 3$, points A and B are outliers in Fig. 1b. In particular, A is a global outlier whereas B is a local outlier.[1]

The rest of this paper is organized as follows: Section 2 reviews the previous results that are directly related to ours. Section 3 formally defines the problem and clarifies several properties fundamental to our techniques. Section 4 elaborates the detailed algorithms, whose performance is improved in Section 5 with statistics. Section 6 presents the results of our experimental evaluation, and Section 7 concludes the paper with directions for future work.

2 Related Work

Section 2.1 first describes the previous RNN solutions in Euclidean spaces and points out their defects. Then, Section 2.2 reviews the existing indexes in metric spaces.

2.1 Euclidean RNN Algorithms

Early RNN algorithms [11], [12], [13], [20] are based on precomputation. For each data point p, such a method materializes its *NN distance*, equal to the distance from p to its nearest point in the data set. Then, checking whether p is the RNN of a query q can be easily achieved by examining if q falls in the circle that centers at p and has a radius equal to the NN distance of p. The NN distances of all objects, however, occupy considerable space and require expensive maintenance overhead whenever an object is inserted or deleted.

We are interested in solutions that do not require precomputation. The first algorithm of this type is due to Stanoi et al. [18], who observe an interesting connection between the RNNs of a point and its "constrained nearest neighbors" [8] in various subregions of the universe. Unfortunately, their approach is limited to 2D spaces and cannot be applied to higher dimensionalities.

An any-dimensionality solution TPL is developed in [19]. To illustrate its rationale, consider points p and q in Fig. 2a. Let us denote $\perp(p, q)$ as the perpendicular bisector (according to Euclidean distance) of the segment connecting p and q. The bisector cuts the data space into two half-planes. Any other point p' that falls in the half-plane containing p cannot be an RNN of q. This is because at least p is closer to p' than q is, and hence, q is not the NN of p'.

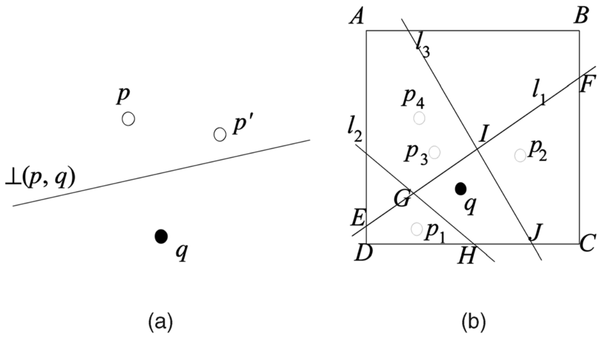


Fig. 2. The TPL algorithm. (a) The basic idea. (b) An example.

Based on this observation, TPL finds the RNNs of a query q by continuously truncating the search space. We illustrate its idea with Fig. 2b, using the same data as in Fig. 1b. Applying the best-first search paradigm [9], TPL assumes an R-tree [1] on the data set and examines the objects in ascending order of their distances to q. In this example, the first point inspected is p_3, which is added to a candidate set S_{can}. Then, TPL obtains the bisector $\perp(q, p_3)$ (i.e., line l_1), and truncates the data space (box $ABCD$) into a trapezoid $EDCF$, where E (F) is the intersection between l_1 and the left (right) boundary of the universe. Note that, other potential RNNs can lie only in $EDCF$ since, by the reasoning illustrated in Fig. 2a, no point in the half-plane bounded by l_1 containing p_3 can possibly be a result (therefore, p_4 does not need to be considered).

Next, among the objects in $EDCF$, TPL identifies the point p_1 with the smallest distance to q, and includes it in S_{can}. The bisector $\perp(p_1, q)$ (line l_2) further shrinks the search region (from $EDCF$) to quadrilateral $GHCF$, where G (or H) is the intersection between l_2 and l_1 (or the lower edge of the universe). Finally, TPL adds p_2 as a candidate (since it is the only point in $GHCF$) and cuts the search region into $GHJI$.

Since $GHJI$ does not cover any data point, TPL enters the next "refinement step," where it verifies whether each candidate in S_{can} is indeed a result. For this purpose, it simply retrieves the actual NN of the candidate and compares it with q. In Fig. 2, for instance, p_3 is a "false hit" because it is closer to its NN p_4 than to q. The other candidates p_1 and p_2, on the other hand, are confirmed to be the final RNNs.[2]

Clearly, TPL is restricted to Euclidean objects because bisectors are simply not defined for objects that cannot be represented as multidimensional vectors. Furthermore, even in Euclidean spaces, TPL has a serious defect: truncating the current search region (using the bisector introduced by a new candidate) can be computationally expensive in high-dimensional spaces.

To explain this, note that, given two d-dimensional points p_1, p_2, their perpendicular bisector under an L_p norm conforms to

$$\sum_{i=1}^{d}(|x[i] - p_1[i]|)^p = \sum_{i=1}^{d}(|x[i] - p_2[i]|)^p, \tag{1}$$

where x is any point on the bisector and $x[i]$ ($1 \le i \le d$) its coordinate on the ith dimension (similarly for $p_1[i]$ and $p_2[i]$). The bisector in general is a complex hypercurve (a

[1] A global outlier has the largest distance to its nearest neighbor in the data set [15]. Such outliers can be found by performing a NN query for each object. A local outlier [4] does not have this property, e.g., B has a smaller NN-distance than the points in the upper cluster. Instead, a local outlier demonstrates irregular patterns compared to the data in its neighborhood. Nanopoulos et al. [14] propose an algorithm for discovering local outliers with RNN retrieval.

[2] TPL adopts more complex algorithms to combine the filter and refinement phases (see [19] for details).

d-dimensional polynomial with degree $p-1$) and, as a result, the truncated search region (maintained during the execution of TPL) becomes excessively complex polytopes bounded by these hypercurves. The truncating process requires computing the intersection between a curve (described by (1)) and such a polytope, which is extremely difficult in high-dimensional spaces (in fact, the problem is already very hard in a 3D space [3]).

The authors of [19] remedy this problem by approximating the polytope with a hyper-rectangle. Unfortunately, the approximation is applicable only to the L_2 norm. Deriving the corresponding approximation for L_p norms with $p \neq 2$ is much more challenging. Furthermore, even if the approximation for a particular L_p norm can be derived, it is not applicable to another $L_{p'}$ norm with $p' \neq p$, which, hence, requires its own approximation. The implication is that TPL cannot support arbitrary L_p norms in a uniform manner, but requires special (difficult) implementation for each value of p. The techniques proposed in this paper, on the other hand, handle all types (non-Euclidean/Euclidean) of objects and distance metrics (satisfying the triangle inequality) with exactly the same algorithms.

Among other RNN studies, Singh et al. [17] propose an approximate algorithm, which cannot guarantee the exact results. Finally, Benetis et al. [2] address RNN processing on moving objects. As with the above methods, these algorithms are applicable to Euclidean objects only.

2.2 Metric Space Indexing

The problem of indexing objects in a generic metric space has been extensively studied and numerous effective solutions exist, as discussed in an excellent survey [10]. These indexes utilize only objects' mutual distances, and do not require the detailed information of individual objects. In this paper, we focus on the M-tree [6], since it is a dynamic structure specifically designed for external-memory environments.

In an M-tree, an intermediate entry e records 1) a *routing object* $e.RO$ that is a selected object in the subtree sub_e of e, 2) a *covering radius* $e.r$ that equals the maximum distance between e and the objects in sub_e, and 3) a *parent distance* $e.pD$ corresponding to the distance between e and the routing object of the parent entry e_p referencing the node containing e. Obviously, all the objects in sub_e lie in the *cluster sphere* of e that centers at its routing object, and has radius $e.r$. A leaf entry o, on the other hand, stores the details of an object and its parent distance $o.pD$ with respect to its parent entry. No covering radius is defined for leaf entries.

We illustrate the above definitions using a Euclidean data set containing nine points $o_1, \ldots, o_9$ (Fig. 3). The circles demonstrate the cluster spheres of the nonleaf entries. For instance, e_1 is associated with routing object o_1 ($= e_1.RO$), its covering radius $e_1.r$ equals the distance $d(o_1, o_2)$ between o_1 and o_2, and parent distance $e_1.pD$ corresponds to $d(e_1.RO, e_5.RO)$. Similarly, the routing object of e_5 is o_4, $e_5.r$ is equivalent to $d(e_5.RO, o_2)$, and $e_5.pD = \infty$ (since e_5 is a root entry). As leaf entry examples, consider the child node of e_1, where the parent distance $o_1.pD$ for o_1 is 0 (because o_1 is the routing object of e_1), while $o_2.pD$ and $o_3.pD$ equal $d(e_1.RO, o_2)$ and $d(e_1.RO, o_3)$, respectively. Ciaccia et al. [7] point out that the M-tree construction algorithms aim at minimizing the overlap among the cluster spheres of the intermediate entries at the same level of the tree (e.g., in Fig. 3, there is little overlap among the cluster spheres of e_1, e_2, e_3 and e_4).

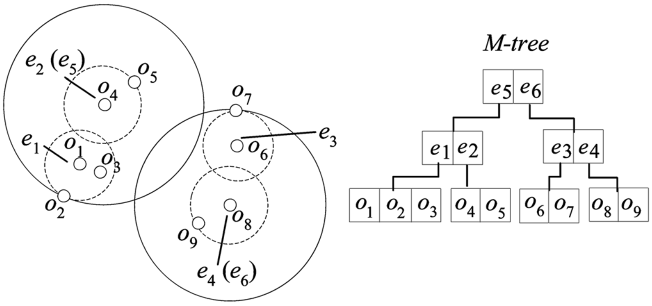


Fig. 3. An M-tree example.

All the query algorithms of M-trees are based on *minimum distances* formulated for intermediate entries. Specifically, the minimum distance $minD(e, q)$ of an entry e with respect to a query object q equals the smallest possible distance between q and any object in the subtree of e, *as can be derived only from the information stored in* e. Specifically (assuming the distance metric satisfies the triangle inequality),

$$minD(e, q) = \begin{cases} d(e.RO, q) - e.r & \text{if } d(e.RO, q) > e.r \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

The equation can be intuitively understood with an analogy to calculating the minimum distance between a point q and a circle centering at $e.RO$ with radius $e.r$.

We demonstrate query processing with $minD$ by explaining the best-first algorithm [9], [10] for nearest neighbor search. Assume that we want to find the NN of object o_9 in Fig. 3. The search starts by inserting the root entries of the M-tree into a min-heap H, using their minimum distances to o_9 as the sorting key (the sorted order is $H = \{e_6, e_5\}$). Then, the algorithm removes the top entry e_6 of H and accesses its child node whose entries are also added to H in the same manner (now $H = \{e_4, e_3, e_5\}$). Similarly, the next node visited is (the child of) e_4, en-heaping the objects encountered there (o_9 is not en-heaped because it is the query): $H = \{o_8, e_3, e_5\}$. Since the top of H is an object o_8, the algorithm reports it as the final answer and terminates. Note that the minimum distance from o_9 to any remaining entry in H is larger than $d(o_9, o_8)$, indicating that no other object can be closer to o_9 (than o_8).

Although the above discussion uses the M-tree as the representative metric index, it is worth mentioning that the NN algorithms of other metric indexes follow similar rationales. In particular, the essence of these algorithms is to derive upper and lower bounds about the distance between the query and any object in a subtree. The index is traversed in a depth-first [16] or best-first order, guided by the upper/lower bounds. A subtree is pruned if we can assert that it cannot contain any query result. We refer our readers to [10], where the authors present a detailed analysis of all the upper/lower bounds and capture them into four crucial lemmas.

3 Problem Definition and Characteristics

Consider a database $\mathcal{D}$ with $|\mathcal{D}|$ objects o_1, o_2, $\ldots$, $o_{|\mathcal{D}|}$. The similarity between two objects o_i, o_j is defined as their distance $d(o_i, o_j)$, i.e., o_i and o_j are more similar if $d(o_i, o_j)$ is smaller. Function d is a metric, meaning that it satisfies the triangle inequality $d(o_i, o_j) \leq d(o_i, o_k) + d(o_k, o_j)$, for arbitrary o_i, o_j, and o_k. For each object $o \in \mathcal{D}$, we define its

TABLE 1
Frequently Used Symbols

Symbol	Description
$\mathcal{D}$	dataset
f_{min}	minimum node fanout of an M-tree
$d(o_1, o_2)$	distance between objects o_1 and o_2
$e.RO$	routing object of an intermediate entry e
$e.r$	covering radius of an intermediate entry e
sub_e	subtree of e
$o.pD$ or $e.pD$	parent distance of an object o or an intermediate entry e
$maxND_k(o)$ or $maxND_k(e)$	max-k-nndist of an object o or an intermediate entry e
$minD(e, q)$	minimum distance between the query object q and any object in sub_e

kNN-distance as the distance between o and its kth NN in $\mathcal{D} - \{o\}$ (i.e., o cannot be a NN of itself).

Given a query object q and a parameter k, a *reverse k nearest neighbor* (RkNN) query retrieves all objects o from $\mathcal{D}$ such that $d(o, q)$ is smaller than the kNN-distance of o. Our objective is to minimize the I/O and CPU cost of such queries.

We make three assumptions: First, except for their mutual distances, no other information about the objects can be deployed to process a query. Second, an M-tree has been constructed on $\mathcal{D}$. Third, the parameter k of an RkNN query is smaller than the minimum node fanout f_{min} of the M-tree, i.e., the smallest number of entries in a node (if it is not the root). In fact, practical values of k are expected to be fairly small (e.g., less than 10 [11], [19]), while a typical f_{min} is at the order of 100. We will briefly explain how to solve a query with $k > f_{min}$ in Section 4.3, but such queries are not the focus of optimization.

Query processing in metric spaces is inherently more difficult (than in Euclidean spaces), due to the lack of geometric properties (e.g., the bisector rules in Fig. 2 are no longer applicable). In the sequel, we design alternative pruning heuristics following a distance-based strategy. Section 3.1 first introduces the basic concept of *max k*-nearest *neighbor distance*. Based on this concept, Sections 3.2 and 3.3 explain the detailed heuristics, focusing on intermediate and leaf entries, respectively. Table 1 lists the symbols to be used frequently in the discussion.

3.1 Max k Nearest Neighbor Distance

We define the *max-k-nndist* of an object o, denoted as $maxND_k(o)$, as any value larger than the kNN-distance of o. Similarly, for an intermediate entry e in the M-tree, its *max-k-nndist* $maxND_k(e)$ can be any value larger than the kNN-distances of all objects in the subtree sub_e of e. Obviously, the values of $maxND_k(e)$ and $maxND_k(o)$ are not unique.

Our RNN algorithms originate from a simple observation:

Lemma 1. *The subtree sub_e of an intermediate entry e cannot contain any result of an RkNN query q if the minimum distance $minD(e, q)$ between e and q is at least $maxND_k(e)$. Similarly, an object o cannot satisfy q if $d(o, q) \geq maxND_k(o)$.*

Proof. Obvious from the definitions of $minD(e, q)$, $maxND_k(e)$, and $maxND_k(o)$. □

Designing pruning heuristics based on Lemma 1 is not trivial. We must derive values of $maxND_k(o)$ and $maxND_k(e)$ that can be efficiently calculated and, yet, are as low as possible to obtain strong pruning power. Ideally, $maxND_k(o)$ and $maxND_k(e)$ should match their lower bounds, equal to the actual kNN distance of o and the largest kNN distance of all objects in sub_e, respectively. Unfortunately, as will be explained in Section 5.4, achieving the lower bounds is not practical, since they require expensive overhead to compute and maintain. In the next sections, we derive values of $maxND_k(e)$ and $maxND_k(o)$ that can be obtained with small cost and permit effective pruning.

3.2 Pruning Intermediate Entries

For $k = 1$, we can set $maxND_1(e)$ to $e.r$, i.e., the largest distance between the routing object $e.RO$ and any object o in sub_e. Thus, Lemma 1 becomes a concrete heuristic:

Rule 1. *For an RNN query q ($k = 1$), the subtree sub_e of an intermediate entry e can be pruned if $d(e.RO, q) \geq 2e.r$.*

The rule becomes clear by rewriting the above inequality as $d(e.RO, q) - e.r \geq e.r$, where the left side equals $minD(e, q)$ (see (2)), and the right side equals $maxND_1(e)$.

For any $k \in [2, f_{min})$, $maxND_k(e)$ can be set to $2e.r$. To verify this, notice that, for any objects o_1, o_2 in sub_e, by the triangle inequality $d(o_1, o_2)$ is at most

$$d(e.RO, o_1) + d(e.RO, o_2),$$

which, in turn, is bounded by $2e.r$. Since we consider $k < f_{min}$ (the minimum node fanout), there are at least f_{min} objects in sub_e. This means that, for any object in sub_e, its distances to any k other objects in sub_e are all bounded by $2e.r$, which establishes the correctness of $maxND_k(e) = 2e.r$.

Following the reasoning behind the derivation of Rule 1, we have:

Rule 2. *For an RkNN query q with any $k \in [2, f_{min})$, the subtree sub_e can be pruned if $d(e.RO, q) \geq 3e.r$, where f_{min} is the minimum node fanout.*

The previous formulation of $maxND_k(e)$ considers only e itself. Given, on the other hand, $k - 1$ arbitrary data objects $o_1, o_2, \ldots, o_{k-1}$ different from $e.RO$, we can compute $maxND_k(e)$ based on the distances $d(e.RO, o_1)$, $\ldots$, $d(e.RO, o_{k-1})$. Since the derivation is not trivial, we present it as a lemma.

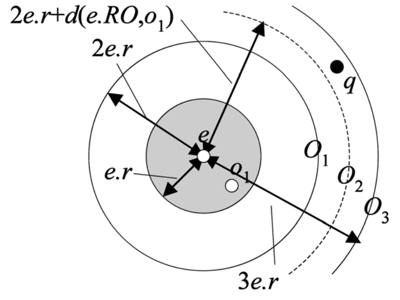


Fig. 4. Illustration of Rules 1-3.

Lemma 2. *Let $o_1, o_2, \ldots, o_{k-1}$ be $k-1$ ($2 \leq k < f_{min}$) objects in $\mathcal{D}$ that are different from the routing object of an intermediate entry e. Then, $maxND_k(e)$ can be set to*

$$\max_{i=1}^{k-1}(e.r + d(e.RO, o_i)).$$

Proof. Let o be an object in sub_e, and

$$\lambda = \max_{i=1}^{k-1}(e.r + d(e.RO, o_i)).$$

To prove the lemma, it suffices to find k objects different from o such that the distance between o and each object is at most λ. For this purpose, we proceed with three cases.

First, assume that o is different from $e.RO$, $o_1, \ldots, o_{k-1}$. For each o_i ($1 \leq i \leq k-1$),

$$d(o, o_i) \leq d(e.RO, o) + d(e.RO, o_i) \leq e.r + d(e.RO, o_i) \leq \lambda.$$

Since $d(o, e.RO) \leq e.r \leq \lambda$, we have found k objects $e.RO$, $o_1, \ldots, o_{k-1}$ whose distances to o are at most λ.

Second, if $o = o_1$ (the scenario $o = o_i$ for $2 \leq i \leq k-1$ is the same), by the reasoning of the first case, its distances to $o_2, \ldots, o_{k-1}$, and $e.RO$ are smaller than or equal to λ. Furthermore, let o' be any object in sub_e different from $e.RO$ and $o_1, o_2, \ldots, o_{k-1}$. Since sub_e contains at least $f_{min} > k$ objects, such an o' always exists. Then,

$$d(o_1, o') \leq d(o', e.RO) + d(e.RO, o_1) \leq e.r + d(e.RO, o_1) \leq \lambda.$$

Hence, we have also found k objects ($o_2, \ldots, o_{k-1}$, $e.RO$, and o') whose distances to o are bounded by λ.

Finally, if $o = e.RO$, $d(e.RO, o') \leq e.r \leq \lambda$ holds for any object o' in sub_e, which completes the proof. □

Combining Lemmas 1 and 2, we obtain the following heuristic:

Rule 3. *Let $o_1, o_2, \ldots, o_{k-1}$ be $k-1$ objects as described in Lemma 2. For an RkNN query q with $k \in [2, f_{min})$, the subtree sub_e can be pruned if*

$$d(e.RO, q) \geq \max_{i=1}^{k-1}(2e.r + d(e.RO, o_i)).$$

Example. Consider Fig. 4, where all the objects in the subtree sub_e of an intermediate entry e lie in the shaded circle that centers at $e.RO$, and has radius $e.r$. Also centering at $e.RO$, circles O_1, O_2, O_3 have radii $2e.r$, $2e.r + d(e.RO, o_1)$, and $3e.r$, respectively. Consider the query q denoted as the black dot. If the parameter k of q equals 1, sub_e does not need to be visited because it cannot contain any results according to Rule 1 (i.e., $dist(e.RO, q) > 2e.r$). For any $k \geq 2$, Rule 2 *cannot* eliminate e since q falls inside O_3 (i.e., $dist(e.RO, q) < 3e.r$). However, if we have encountered data point o_1 before (i.e., o_1 does not belong to sub_e, but has been retrieved from other parts of the tree), e can still be pruned for $k = 2$ based on Rule 3 (i.e., $dist(e.RO, q) > 2e.r + dist(e.RO, o_1)$, as q falls out of circle O_2). Notice that with only o_1, Rule 3 cannot be applied to any $k \geq 3$; in general, $k-1$ objects are needed for this rule to be useful.

Each of the above heuristics has a variation, which utilizes the parent entry e_p of the node containing the intermediate entry e. Note that the distance $d(e.RO, q)$ is at least $|d(e_p.RO, q) - d(e.RO, e_p.RO)|$ by the triangle inequality. Thus, Rules 1-3 are still correct by replacing $d(e.RO, q)$ with this lower bound:

Rule 4. *Let e_p be the parent entry of the node containing an intermediate entry e. Rules 1-3 are still valid by replacing $d(e.RO, q)$ with $|d(e_p.RO, q) - e.pD|$, where $e.pD$ is the parent distance of e (equal to $d(e.RO, e_p.RO)$).*

For each of Rules 1-3, its version in Rule 4 has weaker pruning power, i.e., if Rule 4 can prune e, so can the original rule. However, the advantage of Rule 4 is that its application does not require any distance computation. Specifically, during RNN search, when examining e, the algorithm must have already calculated $d(e_p.RO, q)$ (recall that e_p is at a level higher than e). Since $e.pD$ is associated with e in the M-tree, all the values needed in Rule 4 are directly available. On the other hand, applying Rules 1-3 requires evaluating the distance between $e.RO$ and q. In particular, Rule 3 also requires calculating the distances from $e.RO$ to the $k-1$ objects defined in Lemma 2.

3.3 Pruning Leaf Entries

Unlike the max-k-nndist of an intermediate entry e, which can be derived from e itself, the $maxND_k(o)$ of an object o must be formulated by taking into account other objects.

Lemma 3. *Consider a leaf node (whose parent entry is e) containing f objects $o_1, o_2, \ldots, o_f$, sorted in ascending order of their parent distances (i.e., o_1 is the routing object of e). Then:*

- *For $k = 1$: $maxND_1(o_1)$ can be set to $o_2.pD$, and $maxND_1(o_i)$ to $o_i.pD$ ($2 \leq i \leq f$);*
- *For $k \in [2, f_{min}]$: $maxND_k(o_i)$ can be set to $o_i.pD + \xi$, where ξ equals $o_{k+1}.pD$ for $i \in [1, k]$, and $o_k.pD$ for $i \in [k+1, f]$.*

Proof. The scenario of $k = 1$ is straightforward. Since o_1 is the routing object of e, its distance to o_2 equals the parent distance of o_2, which thus is a legal value for $maxND(o_1)$. For every other object o_i ($2 \leq i \leq f$), at least $e.RO$ is within distance $o_i.pD$ from o_i, validating the choice of $maxND_1(o_i)$.

For $k > 1$, we discuss only objects o_i for $1 \leq i \leq k$, since the case where $i > k$ can be proved in the same way. Denote λ as $o_i.pD + \xi$, where ξ is defined in the

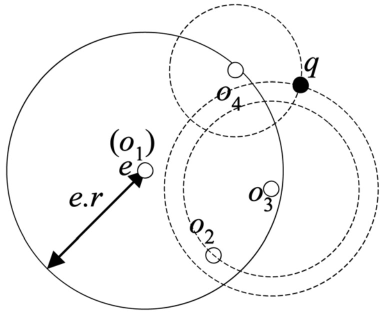


Fig. 5. Max-k-nndist derivation for leaf entries.

lemma. For any of the k objects o_j ($j = 1, \ldots, i-1, i+1, \ldots, k+1$), we have

$$d(o_i, o_j) \leq d(e.RO, o_i) + d(e.RO, o_j) = o_i.pD + o_j.pD.$$

Due to the way the objects are sorted, $o_j.pD \leq o_{k+1}.pD$. Adding $o_i.pD$ to both sides of the inequality leads to $o_i.pD + o_j.pD \leq \lambda$. Hence, setting $maxND_k(o_i)$ to λ is correct. □

Notice that obtaining the max-k-nndist of an object incurs no distance evaluation at all, since only the information stored in the M-tree is used. We explain the $k > 1$ case of Lemma 3 using a concrete example (the case of $k = 1$ is straightforward).

Example. Consider Fig. 5 where a leaf node contains objects $o_1, \ldots, o_4$, and o_1 is the routing object associated with the parent entry e. Assume k = 2; to compute objects' max-2-nndist, we sort the objects in ascending order of their parent distances; in this case, the sorted order is $\{o_1, o_2, o_3, o_4\}$. For o_1 (o_2), by Lemma 3, its max-2-nndist equals its parent distance plus that of o_3 (the third, or $(k+1)$st, object in the sorted list). For o_3 (o_4), on the other hand, its max-2-nndist is the sum of its parent distance and $o_2.pD$ (o_2 is the second, or the kth object in the sorted list).

Combining Lemmas 1 and 3 leads to:

Rule 5. *Given an* RkNN *query q with $k < f_{min}$, an object o cannot be a result if $d(o, q) \geq maxND_k(o)$, where $maxND_k(o)$ is calculated in Lemma 3.*

Similar to Rule 4, a weaker version of the above rule exists:

Rule 6. *Let e be the parent entry of the leaf node that contains o. The previous heuristic still holds by replacing $d(o, q)$ with $|d(e.RO, q) - o.pD|$.*

Similar to the reasons explained at the end of Section 3.2, the application of Rule 6 does not incur any distance evaluation.

4 RkNN Search Using M-Trees

Based on the results in the previous section, Fig. 6 presents the pseudocode of an algorithm RkNN, which follows a filter-refinement framework. In the filter step, RkNN retrieves a set S_{can} of candidates (any object not in S_{can} can be safely discarded), which are verified in the refinement step. In the next section, we discuss the details of the filter step. Then, Section 4.2 elaborates the refinement phase.

```
Algorithm RkNN($S_{can}$, $q$, $k$)
1. $S_{can} = \emptyset$
2. RkNN-Filter(root of the M-tree, $S_{can}$, $q$, $k$)
3. RkNN-Refine($S_{can}$, $q$, $k$)
4. return $S_{can}$
Algorithm RkNN-Filter($N$, $S_{can}$, $q$, $k$)
1. if $N$ is not the root
2.     $e_p$ = parent entry of $N$ //$d(e_p, q)$ must have been evaluated before
3.     sort the entries in $N$ in ascending order of their parent distances;
       let $\{e_1, e_2, ..., e_f\}$ be the sorted order, where $f$ is the number of
       entries in $N$
4.     if $d(e_p.RO, q) \geq 2e_p.r + e_k.pD$ then return //Rule 3
5. if $N$ is a leaf node
6.     Add-Cand($N$, $S_{can}$, $q$, $k$)
7. else for every entry $e_i$ in $N$ ($1 \leq i \leq f$)
8.     if $k = 1$ then $\lambda = 2e_i.r$ else $\lambda = 3e_i.r$
9.     if $|d(e_p.RO, q) - e_i.pD| \geq \lambda$
10.        continue the for-loop at Line 7 //Rule 4
11.    compute $d(e_i.RO, q)$;
12.    if $d(e_i.RO, q) \geq \lambda$ then continue at Line 7 //Rule 1 or 2
13.    RkNN-Filter(child of $e_i$, $S_{can}$, $q$, $k$)
Algorithm RkNN-Refine($S_{can}$, $q$, $k$)
1. for each object $o \in S_{can}$
2.     if Adapted-kNN($o$, $q$, $k$) = FALSE
3.         remove $o$ from $S_{can}$
```

Fig. 6. The RkNN algorithm.

4.1 The Filter Step

RkNN-Filter performs a depth-first traversal over the underlying M-tree, starting from the root. In general, assume that a node N has been fetched from the disk. If N is not the root (Line 1 of RkNN-Filter in Fig. 6, the algorithm attempts to prune it using Rule 3. For this purpose, Line 2 obtains the parent entry e_p of N. Then, Line 3 sorts all the entries in N in ascending order of their parent distances (i.e., their distances to $e_p.RO$). Let e_k be the kth entry in the sorted order. According to Rule 3, the subtree sub_{e_p} of e_p can be pruned if

$$d(e_p.RO, q) \geq 2e_p.r + e_k.pD$$

(Line 4). Note that $d(e_p.RO, q)$ must have been computed prior to accessing N. Hence, the application of Rule 3 here requires no distance evaluation.

If N is a leaf node (Line 5), RkNN-Filter invokes the algorithm Add-Cand to collect the candidate objects from N (Line 6). We defer the discussion of Add-Cand until later.

In case N is an intermediate node, RkNN-Filter examines each of its entries e_i ($1 \leq i \leq f$) in turn (Line 7), and decides whether it is necessary to visit its child node. If $k = 1$ (or > 1), the decision is based on Rule 1 (or Rule 2), and its weaker version in Rule 4. At Line 8 of the pseudocode, the value of λ controls the rules to be used (notice that λ is the constant on the right of the inequality in Rule 1 or 2). Line 9 first applies Rule 4 since, as discussed at the end of Section 3.2, application of this rule incurs no distance evaluation. If Rule 4 successfully prunes e_i (Line 10), the algorithm continues to check the next entry in N at Line 7.

```
Algorithm Add-Cand($N$, $S_{can}$, $q$, $k$)
/* $N$ is a leaf node with $f$ objects $o_1$, $o_2$, ..., $o_f$ which are sorted in ascending order of their parent distances
(the sorting is performed at Line 2 of RkNN-Filter in Figure 6) */
1.  $e_p$ = parent entry of $N$
2.  obtain $maxND_k(o_i)$ for each object $o_i$ ($1 \leq i \leq f$) in $N$ as described in Lemma 3
3.  for $i = 1$ to $f$
4.    if $|d(e_p.RO, q) - o_i.pD| \geq maxND_k(o_i)$ // $d(e.RO, q)$ has been evaluated before
5.      mark $o_i$ as pruned; continue at Line 2 //Rule 6
6.    compute $d(o_i, q)$
7.    if $d(o_i, q) \geq maxND_k(o_i)$ then mark $o_i$ as pruned //Rule 5
8.  for $i = 1$ to $f$
9.    if $o_i$ has not been pruned
10.     if $o_i$ is closer to $k$ other objects in $N$ than to $q$
11.       continue at Line 8
12.     $S_{can} = S_{can} \cup \{o_i\}$
```

Fig. 7. Algorithm for discovering candidates.

If Rule 4 cannot prune e_i, then it is necessary to compute the distance between $e_i.RO$ and q (Line 11) in order to (Line 12) deploy Rule 1 or 2 (depending on $k = 1$ or $k > 1$), which has stronger pruning power than Rule 4. Again, if pruning succeeds, RkNN-Filter goes back to Line 7 to consider the next entry in N. Otherwise, the algorithm recursively visits the child node of e_i (Line 13).

As mentioned earlier, if a leaf node N cannot be pruned by Rule 3, Add-Cand (Fig. 7) is invoked to identify the candidate objects in N that may satisfy the query. Note that, at this time, the objects in N have been sorted in ascending order of their parent distances (the sorting is performed at Line 3 of RkNN-Filter).

Line 1 of Fig. 7 identifies the parent entry e_p of N. Then, Line 2 computes the $\mathrm{max\text{-}k\text{-}nndist}$ for all objects in N based on Lemma 3. Next, Add-Cand inspects (Line 3) each object o_i ($1 \leq i \leq f$, where f is the number of objects in N), and attempts to disqualify o_i using Rule 5 and its weaker version, Rule 6. Specifically, at Line 4, Rule 6 is first applied because it does not demand any distance computation. If Rule 6 successfully prunes o_i, the algorithm proceeds with the next object in N (Line 5). Otherwise, $d(o_i, q)$ is evaluated (Line 6), and Rule 5 is applied (Line 7).

Finally, Lines 8 through 12 perform the following tasks: For each object o_i that has not been pruned by Rule 5, we check whether it is closer to k other objects in N than to q. If yes, o_i cannot be a query result; otherwise, it is added to S_{cnd} to be refined, as discussed in the next section.

4.2 The Refinement Step

To verify if a candidate o is an RkNN of q, a straightforward approach is to find the kth NN o' of o. Then, o can be confirmed as a result if and only if $d(o, q) < d(o, o')$.

Fig. 8 presents an alternative solution Adapted-kNN, which requires fewer I/O accesses and distance computations. Adapted-kNN is very similar to the best-first kNN algorithm reviewed in Section 2, but differs from that algorithm in its ability to terminate earlier if the candidate is a false hit.

Specifically, Adapted-kNN achieves early termination with two optimizations. To illustrate the first optimization, let e be an intermediate entry; a candidate o must be a false hit if $d(e.RO, o) + e.r \leq d(o, q)$ holds (Line 10 in Fig. 8). To understand this, consider any object o' in the subtree sub_e of e. The distance $d(o, o')$ between o and o' is bounded by $d(e.RO, o) + d(e.RO, o')$, which, in turn, is bounded by $d(e.RO, o) + e.r$. Hence, the previous inequality establishes the fact that $d(o, o') \leq d(o, q)$. Since there are at least $f_{min} > k$ objects in sub_e, q cannot be closer to o than the kth NN of o.

The second optimization utilizes the property of M-trees that the routing object of each intermediate entry e is an object in sub_e. Specifically, Adapted-kNN maintains a set S, which stores all the objects that have been seen so far and are closer to o than q. Note that these objects may have been collected from the leaf nodes accessed (Line 14), or from the routing objects of the intermediate entries visited (Line 9). Once the size of S reaches k, Adapted-kNN terminates (Line 15), since o cannot be a query result in this case.

4.3 Discussion

I/O-pruning in RkNN-Filter essentially relies on Rules 1 and 2. In particular, for an RkNN query q with $k = 1$ (or > 1), the child node of an entry e is visited if and only if $d(e.RO, q) < 2e.r$ (or $d(e.RO, q) < 3e.r$) holds. Rule 3 is applied only to reduce CPU time; as mentioned in Section 4.1, its application does not require any distance evaluation.

```
Algorithm Adapted-kNN($o$, $q$ $k$)
//$o$ is a candidate, and $q$ is a R$k$NN query
1.  initialize a min-heap $H$ whose entries are in the form $[e, key]$
2.  $S = \emptyset$
3.  insert $[e_{dum}, \infty]$ into $H$ //a dummy pointer to the root
4.  while $H$ is not empty
5.    de-heap $[e, key]$ from $H$
6.    for each entry $e'$ in the child of $e$ //$e'$ can be a leaf or
      intermediate entry
7.      if $e'$ is an intermediate entry
8.        compute $d(e'.RO, o)$
9.        if $d(e'.RO, o) \leq d(o, q)$ then $S = S \cup \{e'\}$
10.       if $d(e'.RO, o) + e'.r \leq d(o, q)$ return FALSE
11.       if $d(e'.RO, o) - e'.r \leq d(o, q)$ add $[e', minD(e', o)]$ to $H$
12.     else //$e'$ is an object
13.       compute $d(e', o)$
14.       if $d(e', o) \leq d(o, q)$ then $S = S \cup \{e'\}$
15.   if $|S| = k$ then return FALSE
16.return TRUE
```

Fig. 8. Algorithm for verifying candidates.

In fact, a natural attempt to use Rule 3 for I/O pruning is to deploy the following heuristic: Given a query with $k > 1$, when trying to prune an intermediate entry e, we may resort to S_{can} (the set of candidates discovered so far). Specifically, if there are k objects $o \in S_{can}$ satisfying $d(e.RO, q) \geq 2e.r + d(e.RO, o)$, then e can be eliminated according to Rule 3. It turns out that this heuristic offers almost no I/O improvement, while increasing the number of distance computations significantly.[3]

The phenomenon is caused by the fact that the I/O-pruning power of Rule 3 is usually subsumed by Rule 2. This is because, as explained shortly, the distance between $e.RO$ and any candidate o not in sub_e is most likely larger than $e.r$. In this case, the inequality $d(e.RO, q) \geq 2e.r + (e.RO, o)$ leads to $d(e.RO, q) \geq 3e.r$, i.e., if Rule 3 can prune e, so can Rule 2.

Why is $d(e.RO, o) > e.r$ usually true (when o is not in the subtree of e)? As mentioned in Section 2.2, M-trees aim at minimizing the overlap among the cluster spheres of the intermediate entries at the same level. Let e' be the entry at the same level as e, such that the subtree of e' contains o. Since the cluster sphere of e' has little overlap with that of e, (with a high probability) o falls out of the cluster sphere of e, resulting in $d(e.RO, o) > e.r$.

Our discussion so far considers that k is smaller than the minimum node fanout f_{min} (remember that f_{min} is on the order of 100 in practice). In the unlikely event where $k > f_{min}$, we can extend the above algorithms using a simple observation: All the pruning rules are still valid by replacing f_{min} with the smallest number n_{min} of leaf entries in the subtree of an intermediate entry e. For example, if e is at the ith level (leaves are at level 0), then n_{min} equals f^i_{min}.

We close this section by pointing out the differences between RkNN and TPL (reviewed in Section 2.1). First, RkNN is not based on truncating the data space, which is the core of TPL. In fact, "truncating" is simply impossible in metric spaces, because there is no such a concept as "the intersection between two regions." Second, RkNN does not deploy any geometric properties (e.g., the bisector rule in Fig. 2a), but utilizes only the distances among objects.

5 Statistics-Aided Algorithms

In this section, we aim at reducing the I/O cost of the filter step, which dominates the overall query overhead as shown in the experiments. The reduction is made possible by maintaining a small number of statistics. In Section 5.1, we explain the type of statistics needed and why they can be used to improve search performance. Then, Section 5.2 derives a cost model that quantifies the I/O performance of the filter step. Based on the model, Section 5.3 elaborates the statistics computation and the improved RkNN algorithm. Finally, Section 5.4 presents a special optimization for static data.

5.1 k-Covering Distances

We target RkNN queries with $k \geq 2$. For these queries, I/O pruning in RkNN-Filter of Fig. 6 is achieved using only Rule 2, where the subtree sub_e of an intermediate entry e is pruned if $d(e.RO, q) \geq 3e.r$. Although RkNN-Filter also applies Rule 3, the application is limited to reducing CPU time. Next, we show how this rule can be used to improve I/O performance too.

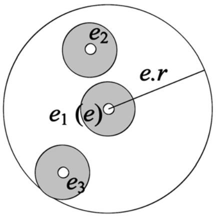


Fig. 9. Illustration of a 2-covering distance.

Given an intermediate entry e, we define its k-covering distance, denoted as $cD_k(e)$, as a value such that at least $k - 1$ objects different from $e.RO$ in sub_e are within distance $cD_k(e)$ from $e.RO$. Our objective is to obtain a $cD_k(e)$ that is much smaller than the covering radius $e.r$ of e. As a result, in processing a query q, sub_e can be eliminated if $d(e.RO, q) \geq 2e.r + cD_k(e)$, according to Rule 3. This inequality is stronger than the inequality $d(e.RO, q) \geq 3e.r$ that underlies Rule 2.

Due to the reasons discussed in Section 4.3,, a $cD_k(e)$ that fulfills our objective cannot be derived during query execution. Hence, we compute it in a preprocessing step. Specifically, for each intermediate entry e, this step calculates $cD_k(e)$ for all $k \in [2, f_{min})$, i.e., e is associated with $f_{min} - 2$ covering distances. These precomputed values are retained in memory and deployed as described earlier in query processing.

The value of $cD_k(e)$ is not unique. In our implementation, $cD_k(e)$ is set to the kth smallest parent distance of the entries in the child node of e. As a result, all the covering distances can be obtained in a single traversal of the M-tree. As an example, consider Fig. 9, where e is the parent entry of an intermediate node containing entries e_1, e_2, and e_3, whose routing objects are represented as white dots. In particular, the routing object of e_1 is identical to that of e. By sorting these entries in ascending order of their parent distances, we obtain the sorted order $\{e_1, e_2, e_3\}$. Hence, according to our formulation, the value of $cD_2(e)$ equals $e_2.pD$. This value is consistent with the definition of $cD_2(e)$, because at least $e_2.RO$ (which resides in sub_e and is different from $e.RO$) is within distance $e_2.pD$ from $e.RO$.

In general, since $k < f_{min}$, $cD_k(e)$ is always bounded by $e.r$. For smaller k, the difference between $cD_k(e)$ and $e.r$ is more significant, resulting in larger difference in the pruning power of Rules 3 and 2. Once calculated, $cD_k(e)$ can be easily maintained during object insertions and deletions: Whenever the content of the child node of e changes, $cD_k(e)$ is modified accordingly.

Let S^{full}_{cD} be the set of $cD_k(e)$ for all intermediate entries e and $k \in [2, f_{min})$. In practice, it may not be possible to preserve the entire S^{full}_{cD} in memory, if the memory has a limited size. Denote M as the largest number of covering distances that can be accommodated in memory. When M is smaller than $|S^{full}_{cD}|$, we identify a subset S_{cD} of S^{full}_{cD} that has M values (the other $|S^{full}_{cD}|M$ covering distances in S^{full}_{cD} are discarded) and leads to the smallest I/O cost of the filter step. For this purpose, in the next section, we derive a cost model that represents the I/O cost as a function of S_{cD}.

[3] Note that we should not introduce a large number of distance computations simply because they enable us to save a couple of I/O accesses, since, for complex distance metrics, computing a distance may also be expensive.

5.2 A Cost Model of the Filter Step

Given an $i \in [2, f_{min})$, an i-covering distance $cD_i(e)$ can be used to accelerate RkNN queries with $k \leq i$. For example, imagine that we keep $cD_3(e)$ in memory but not $cD_2(e)$. For an R2NN query q, the subtree of e can still be eliminated if $d(e.RO, q) \geq 2e.r + cD_3(e)$ holds. Since $cD_2(e) \leq cD_3(e)$, the previous inequality implies $d(e.RO, q) \geq 2e.r + cD_2(e)$, which is the basis of using Rule 3 for pruning, as discussed in the previous section. The implication is that, if two covering distances $cD_i(e)$ and $cD_j(e)$ $(i < j)$ are close enough, we may keep only $cD_j(e)$ (i.e., the larger one), without compromising I/O performance significantly, since the I/O-pruning for RiNN search can be performed (almost) equally well using $cD_j(e)$.

Assume that we have already selected a subset S_{cD}. Given an integer i, let $\lceil i \rceil$ be the smallest integer j such that 1) $j \geq i$, and 2) $cD_j(e) \in S_{cD}$. Then, for an RiNN query q, the child node of e needs to be visited if and only if q is within distance $2e.r + cD_{\lceil i \rceil}(e)$ from $e.RO$. In case $\lceil i \rceil$ does not exist, the child node of e is accessed if $dist(e.RO, q) \leq 3e.r$ according to Rule 2.

Now, we are ready to predict the I/O cost of the filter step when an S_{cD} is available. Let $P_q(o, r)$ be the probability that the distance between a query q and a particular object o is at most r. Then, the probability $P_{acs}(e, i|S_{cD})$ that the child node of e is visited by an RiNN query, given the covering distances in S_{cD}, is

$$P_{acs}(e.RO, i|S_{cD}) = \begin{cases} P_q(e.RO, 2e.r + cD_{\lceil i \rceil}(e)) & \text{if } \lceil i \rceil \text{ exists} \\ P_q(e.RO, 3e.r) & \text{otherwise.} \end{cases} \tag{3}$$

Denote $P_k(i)$ $(2 \leq i < f_{min})$ as the probability that an RiNN query is issued. Then, the overall probability $P_{acs}(e|S_{cD})$ that the child node of e is visited in an arbitrary RkNN query (with any $k \in [2, f_{min})$), equals

$$P_{acs}(e|S_{cD}) = \sum_{i=2}^{f_{min}-1} (P_{acs}(e, i|S_{cD}) \cdot P_k(i)). \tag{4}$$

Summing up $P_{acs}(e|S_{cD})$ for all intermediate entries e, we obtain the expected number $C_{IO}^{filter}(S_{cD})$ of nodes accessed in the filter step:

$$C_{IO}^{filter}(S_{cD}) = \sum_{\text{nonleaf } e} P_{acs}(e|S_{cD}). \tag{5}$$

The cost model requires 1) $P_q(o, r)$, the probability that q appears within distance r from a given object o, and 2) $P_k(i)$, the probability of receiving an RiNN query. Both probabilities can be accurately estimated by maintaining a sample set of the previously answered queries. Specifically, $P_q(o, r)$ (or $P_k(i)$) can be approximated as the percentage of the sampled queries that are within distance r from o (or, the percentage of RiNN queries).

5.3 Statistics Computation and an Improved RkNN Algorithm

As mentioned at the end of Section 5.1, when the amount M of available memory is smaller than $|S_{cD}^{full}|$, we compute an $S_{cD} \subseteq S_{cD}^{full}$ such that S_{cD} consists of M covering distances and minimizes $C_{IO}^{filter}(S_{cD})$.

We achieve this purpose with a greedy approach. Initially, S_{cD} is set to S_{cD}^{full}. Then, we perform $|S_{cD}^{full}| - M$ iterations. Every iteration expunges the element (a covering distance) in S_{cD} whose removal causes the smallest increase of $C_{IO}^{filter}(S_{cD})$. After all iterations, the remaining M covering distances constitute the final S_{cD}.

We associate each element in S_{cD} with a *penalty*, which equals the growth of $C_{IO}^{filter}(S_{cD})$ if the element is expunged. Hence, the element to expunge at the next iteration is the one having the smallest penalty among all the elements in S_{cD}. Specifically, let $cD_x(e)$ be an arbitrary covering distance in S_{cD}. Based on (5), the penalty of $cD_x(e)$, denoted as $pen_x(e)$, is represented:

$$\begin{aligned} pen_x(e) = & \sum_{\text{nonleaf } e\prime} P_{acs}(e'|S_{cD} - \{cD_x(e)\}) \\ & - \sum_{\text{nonleaf } e\prime} P_{acs}(e'|S_{cD}). \end{aligned}$$

To simplify notation, denote S'_{cD} as $S_{cD} - \{cD_x(e)\}$. For any $e' \neq e$, $P_{acs}(e'|S'_{cD})$ is equivalent to $P_{acs}(e'|S_{cD})$, because neither of them is related to $cD_x(e)$. Hence, the above equation can be converted to

$$\begin{aligned} & pen_x(e) = P_{acs}(e|S'_{cD}) - P_{acs}(e|S_{cD}) \\ & (\text{By } (4)) = \sum_{i=2}^{f_{min}-1} \Big(P_{acs}(e, i|\S'_{cD}) - P_{acs}(e, i|S_{cD}) \Big) \cdot P_k(i). \end{aligned} \tag{6}$$

$P_{acs}(e, i|S'_{cD})$ and $P_{acs}(e, i|S_{cD})$ are computed according to (3). In particular, for any $i \in [x + 1, f_{min})$, the value of $\lceil i \rceil$ in (3) remains the same after removing $cD_x(e)$ from S_{cD} and, hence, $P_{acs}(e, i|S'_{cD}) = P_{acs}(e, i|S_{cD})$.

Let us define $\lfloor x \rfloor$ as the largest integer j such that $j < x$ and $cD_j(e)$ exists in S_{cD} (in case j does not exist, $\lfloor x \rfloor = 1$). Then, for any $i \in [2, \lfloor x \rfloor]$, the value of $\lceil i \rceil$ also remains the same after discarding $cD_x(e)$. As a result, (6) can be rewritten as

$$pen_x(e) = \sum_{i=\lfloor x \rfloor + 1}^{x} \Big(P_{acs}(e, i|S'_{cD}) - P_{acs}(e, i|S_{cD}) \Big) \cdot P_k(i). \tag{7}$$

In fact, for any $i \in [\lfloor x \rfloor + 1, x]$, $\lceil i \rceil = x$ and, hence, by (3), $P_{acs}(e, i|S_{cD})$ equals $P_q(e.RO, 2e.r + cD_x(e))$. On the other hand, $P_{acs}(e, i|S'_{cD})$ depends on two cases. First, if $\lceil x \rceil$ exists, after removing $cD_x(e)$, $\lceil i \rceil = \lceil x \rceil$ and, thus, $P_{acs}(e, i|S'_{cD})$ evaluates to $P_q(e.RO, 2e.r + cD_{\lceil x \rceil}(e))$. Otherwise ($\lceil x \rceil$ does not exist), $P_{acs}(e, i|S'_{cD})$ is $P_q(e.RO, 3e.r)$. As a result, (7) becomes:

$$pen_x(e) = \begin{cases} \begin{aligned} & \Big(P_q(e.RO, 2e.r + cD_{\lceil x \rceil}(e)) \\ & - P_q(e.RO, 2e.r + cD_x(e)) \Big) \\ & \cdot \textstyle\sum_{i=\lfloor x \rfloor + 1}^{x} P_k(i) \end{aligned} & \text{if } \lceil x \rceil \text{ exists,} \\ \begin{aligned} & \Big(P_q(e.RO, 3e.r) \\ & - P_q(e.RO, 2e.r + cD_x(e)) \Big) \\ & \cdot \textstyle\sum_{i=\lfloor x \rfloor + 1}^{x} P_k(i) \end{aligned} & \text{otherwise.} \end{cases} \tag{8}$$

```
Algorithm Select-Stat
1.  $S_{cD} = S_{cD}^{full}$ //$S_{cD}^{full}$ is derived as discussed in Section 5.1
2.  for each intermediate entry $e$
3.      for $i = 2$ to $f_{min}$
4.          compute $pen(cD_i(e))$ by Equation 8
5.  while $|S_{cD}| > M$
6.      $cD_x(e)$ = the element in $S_{cD}$ with the smallest penalty
7.      if $\lceil x \rceil$ exists
8.          re-compute the penalty $pen_{\lceil x \rceil}(e)$ by Equation 8
9.      if $\lfloor x \rfloor \geq 2$
10.         re-compute the penalty $pen_{\lfloor x \rfloor}(e)$ by Equation 8
11.     expunge $cD_x(e)$ from $S_{cD}$
```

Fig. 10. Algorithm for computing S_{cD}.

It is clear that, after each iteration, the removal of an element, say $cD_y(e)$, from S_{cD} affects the penalties of, at most, two remaining covering distances $cD_{\lfloor y \rfloor}(e)$ and $cD_{\lceil y \rceil}(e)$ in S_{cD}. For any other remaining element $cD_x(e)$, its $\lceil x \rceil$ and $\lfloor x \rfloor$ remain unchanged and, hence, its penalty stays identical.

Fig. 10 formally summarizes the algorithm Select-Stat for computing S_{cD}, based on the above discussion. Fig. 11 presents the modified RkNN-Filter that deploys Rule 3 for I/O pruning, using the S_{cD} returned by Select-Stat. We refer to the modified RkNN-Filter as Stat-Filter.

After its computation, S_{cD} can be maintained along with object insertions/deletions, since its elements (covering distances) can be incrementally updated, as explained in Section 5.1. The efficiency of RkNN-Filter, however, may deteriorate with the number of updates. This is because S_{cD} is selected to optimize the query performance only at its computation time. Nevertheless, as will be demonstrated in the experiments, the performance degradation is very slow.

5.4 Optimization for Static Data

Recall that Lemma 1, the foundation of all our pruning heuristics, achieves highest efficiency if $maxND_k(o)$ and $maxND_k(e)$ reach their lower bounds for all objects o and intermediate entries e, respectively. The lower bound of $maxND_k(o)$ is the actual kNN-distance of o, while the lower bound of $maxND_k(e)$ equals the maximum of the kNN-distances of all objects in the subtree of e. In the rest of this section, all occurrences of $maxND_k(o)$ and $maxND_k(e)$ represent their lower bounds.

Maintaining these lower bounds is unrealistic for data sets with frequent updates, since it entails considerable overhead. For example, the insertion of an object o must be followed by retrieval of its kNN-distance, whereas deleting an object o demands recomputing the kNN-distances of all the RkNNs of o. Similarly, a single object update may influence the $maxND_k(e)$ of multiple intermediate entries.

However, if the data set is static, computing the lower bounds for the max-k-NNdist of all intermediate entries becomes "one-time cost." In this case, it is worthy because search performance may be improved significantly. Specifically, by Lemma 1, the subtree of an intermediate entry can be pruned if $minD(e, q) \geq maxND_k(e)$, which is more effective than all the pruning rules proposed earlier. Naturally, for each intermediate entry e, we may precompute $maxND_k(e)$ for all $k \in [2, f_{min})$. Let S_{ND}^{full} be the set of resulting values, which are stored in memory for query processing.

```
Algorithm Stat-Filter($N$, $S_{can}$, $q$, $k$)
Lines 1-7 are identical to those of RkNN-Filter in Figure 6
8.  if $k = 1$ then $\lambda = 2e_i.r$ //Rule 1
9.  else if $\lceil i \rceil$ exists
10.     $\lambda = 2e.r + cD_{\lceil i \rceil}(e_i)$ //Rule 3
11.  else $\lambda = 3e_i.r$ //Rule 2
Lines 12-16 are the same as Lines 9-13 of RkNN-Filter of Figure 6
```

Fig. 11. Algorithm of the filter step with statistics.

Here, we encounter a problem similar to the one solved in the previous sections: the available memory may not be large enough to hold the entire S_{ND}^{full}. We tackle the problem using the same approach. Let M be the number of values that can be retained in memory. We aim at obtaining a subset S_{ND} of S_{ND}^{full} that contains M values and minimizes the expected I/O cost. In fact, S_{ND} can be computed using exactly the same algorithm in Fig. 10, replacing S_{cD} and $cD_i(e)$ with S_{ND} and $maxND_i(e)$, respectively. Accordingly, the RkNN-Filter of Fig. 11 needs to be slightly revised: Line 10 should be changed to "$\lambda = e.r + maxND_{\lceil i \rceil}(e)$."

6 Experiments

This section experimentally evaluates the efficiency of the proposed techniques, using both real and synthetic data. The first data set *SF* contains points representing 174 k locations in San Francisco.[4] The similarity between two points is measured as their L_1 distance (which simulates their shortest road network distance, when most road segments are axis parallel). The second data set *TS* is a time series containing 76 k values corresponding to Dow Jones indexes[5] norms. The third data set *Color* involves 4D vectors[6] representing the color histograms of 68 k images, where the similarity is evaluated with the L_∞ norm.

Following the experiment settings of [5], we also generate a *Signature* data set, where each object is a string with 65 English letters. We first obtain 20 "anchor signatures," whose letters are randomly chosen from the alphabet. Then, each anchor produces a cluster with 2.5 k objects (resulting in the total cardinality 50 k), each of which is obtained by randomly changing x positions in the corresponding anchor signature to other random letters, where x is uniformly distributed in range $[1, 18]$. The similarity between two strings is calculated as their edit distance, i.e., the smallest number of editorial changes (e.g., adding, removing, or modifying a letter) required to convert one string to the other.

We index each data set using an M-tree. The disk page size is fixed to 4 k bytes, such that the maximum node capacities for *SF*, *TS*, *Color*, *Signature* equal 255, 146, 170, and 56 entries, respectively. A *workload* contains 500 queries. The query objects are sampled directly from the underlying data set. The distribution of the parameter k of the queries in a workload depends on concrete experiments and will be clarified later. Unless specifically stated, each reported value is the average result of all the queries in a workload. All experiments

[4] Available at http://www.census.gov/geo/www/tiger/.

[5] http://finance.yahoo.com.

[6] *Color* and *Signature* (to be introduced shortly) can be downloaded at http://www.cs.cityu.edu.hk/~taoyf/ds.html.

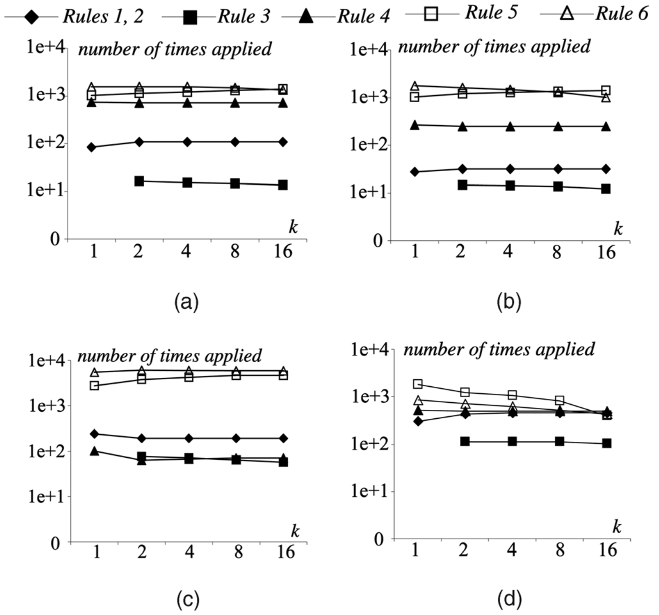


Fig 12. Application frequencies of pruning rules. (a) *SF*. (b) *TS*. (c) *Color*. (d) *Signature*.

are performed on a machine with a Pentium IV 2.8 GHz CPU and 512 Megabytes memory.

6.1 Effectiveness of Pruning Rules

In the experiments of this section, a workload consists of RkNN queries with the same k, which is varied as a workload parameter. The first experiment aims at illustrating the effectiveness of the pruning heuristics. We measure the effectiveness of a heuristic by how often it is successfully applied in RkNN-Filter (Fig. 6). For Rules 1 through 4, a successful application is counted when they prune an intermediate entry, whereas, for Rules 5 and 6, one success is counted when they disqualify an object without computing its distance from the query.

Focusing on data set *SF*, Fig. 12a shows the average number of times that each heuristic is successfully applied as a function of k. The efficiency of Rules 1 and 2 is illustrated together with the same curve because they are based on the same rationale but are deployed for $k = 1$ and $k > 1$, respectively. There is no result for Rule 3 at $k = 1$, since it is applicable only for $k > 1$. Figs. 12b, 12c, and 12d demonstrate the results of the same experiments for *TS*, *Color*, and *Signature*, respectively. Evidently, all heuristics are utilized a large number of times in a query, confirming their usefulness. Rules 5 and 6 are applied more frequently because RkNN-Filter encounters more leaf entries than intermediate ones.

To study the behavior of individual queries, we focus on the workloads with $k = 4$. From each workload, we randomly sample 50 queries (i.e., 10 percent of the workload), and examine their application frequencies of each heuristic. Fig. 13 demonstrates the results for the four data sets, confirming that all the heuristics are important because each heuristic is applied a large number of times in all queries.

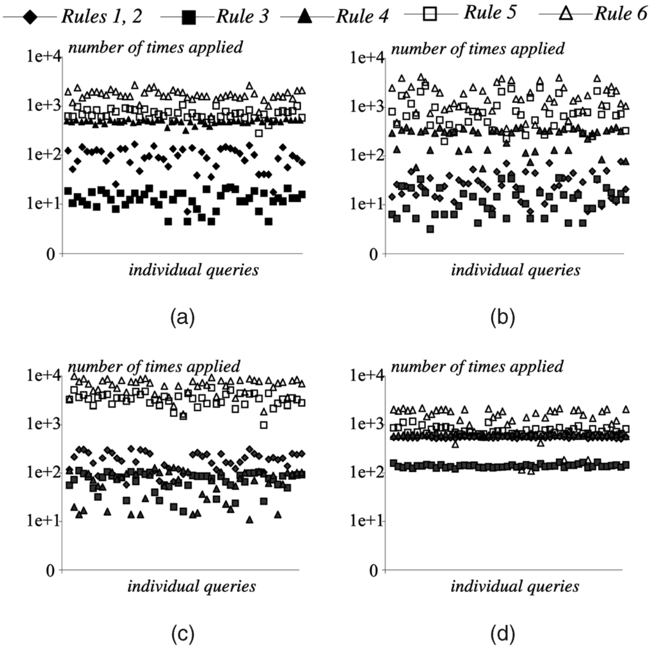


Fig. 13. Rule application frequencies of individual queries ($k = 4$). (a) *SF*. (b) *TS*. (c) *Color*. (d) *Signature*.

6.2 Query Performance of the Basic RkNN

We proceed to evaluate the efficiency of the proposed algorithm RkNN in Section 4. Since no existing algorithm is applicable to RNN search in metric spaces (due to the reasons discussed in Section 2.1), we compare RkNN against a baseline approach that retrieves the kth NN of each object, and reports the object as a result if it is closer to the query than to its kth NN. All queries in a workload have the same parameter k, as with the workloads used in the previous sections.

Concentrating on *SF*, Fig. 14a shows the number of node accesses in the filter/refinement step, as a function of k. The

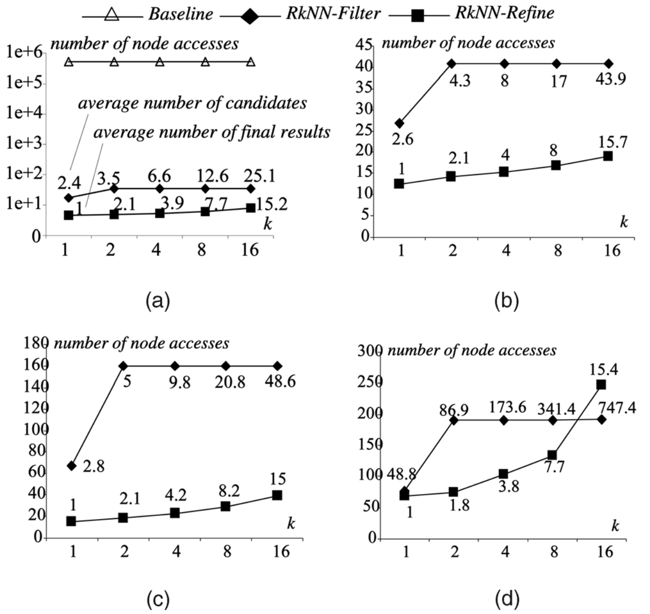


Fig. 14. I/O cost versus k. (a) *SF*. (b) *TS*. (c) *Color*. (d) *Signature*.

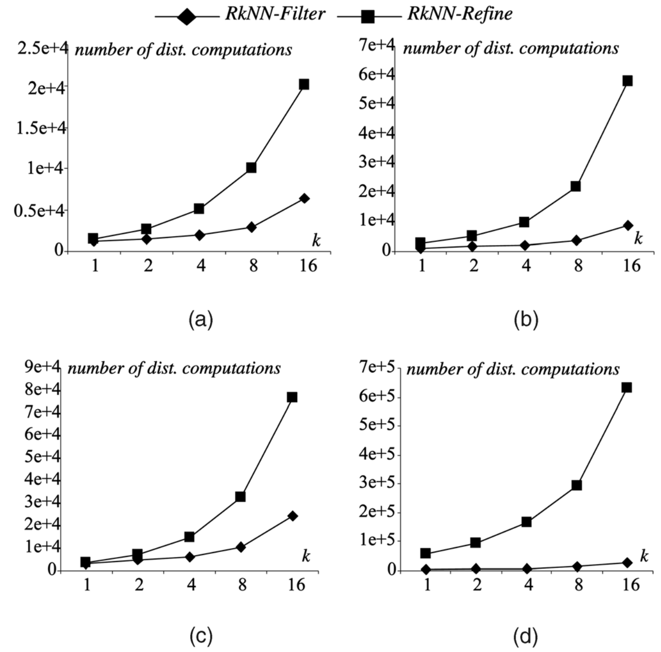


Fig. 15. CPU performance versus k. (a) *SF*. (b) *TS*. (c) *Color*. (d) *Signature*.

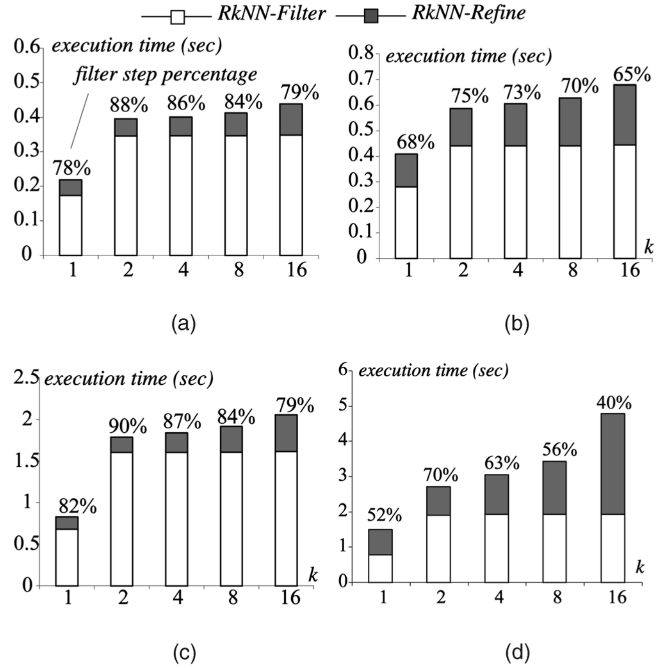


Fig. 16. Overall execution time versus k. (a) *SF*. (b) *TS*. (c) *Color*. (d) *Signature*.

number beside each dot in the curve of RkNN-Filter (or RkNN-Refine) indicates the average number of candidates (or final results) for the queries in the corresponding workload.

The cost of RkNN-Filter demonstrates stepwise behavior. Specifically, it increases as k changes from 1 to 2 but then stabilizes as k grows further. Recall that, as mentioned in Section 4.1, I/O pruning in RkNN-Filter is performed only by Rule 1 or 2 for $k = 1$ or $k > 1$, respectively, which explains the cost difference between the two cases. Furthermore, the pruning power of Rule 2 is independent of k, which explains the stable performance of RkNN-Filter for all $k > 1$. The overhead of RkNN-Refine, however, continuously increases with k because a higher k results in a larger number of candidates and hence, more expensive refinement cost.

As expected, the baseline algorithm is worse than our solution by several orders of magnitude. Therefore, it is omitted in the following experiments. Figs. 14b, 14c, and 14d demonstrate similar results for the other data sets.

Fig. 15 shows the CPU time (measured as the number of distance computations) in the experiments of Fig. 14. The CPU overhead of both RkNN-Filter and RkNN-Refine escalates with k. Interestingly, while usually the filter step incurs higher I/O cost, the refinement phase requires more distance computations.

In Fig. 16, we plot the query response time as a function of k, where each result is broken into two components, capturing the overall cost of the filter and refinement steps, respectively. The value on top of each column is the percentage that the filter step accounts for in the total execution time. In most cases, RkNN-Filter dominates (up to 90 percent of) the overall overhead. The only exception is at $k = 16$ in Fig. 16d, which is caused by the characteristics of data set *Signature*. Specifically, the average kNN distance of the objects in *Signature* increases very fast with k, which, in turn, renders the refinement cost to grow rapidly with k (remember that each refinement performs several adapted kNN queries, each of which is more expensive if the average kNN distance is large). As a result, for large k, the total refinement cost exceeds that of filtering.

6.3 Performance of the Statistics-Aided Algorithms

Let us call the algorithm studied in the previous section the *basic* solution. Next, we evaluate the efficiency of the improved algorithm in Section 5 that utilizes statistics. The following experiments focus on the filter step (Stat-Filter in Fig. 11), because the refinement step is identical to that of the basic solution. Furthermore, we concentrate on the I/O performance, since this is the target of optimization in Stat-Filter.

As Stat-Filter aims at optimizing RkNN queries with different k at the same time, each workload in the following experiments contains an (approximately) equal number of RiNN queries, for every $i \in [2, k_{max}]$, where k_{max} is a parameter. Note that a workload does not contain any RNN query, for which Stat-Filter has the same performance as the basic solution.

The efficiency of Stat-Filter depends on the size M of available memory. In Section 5, M equals the number of values that can be stored in memory. Since it is reasonable to allocate more memory for a larger data set, we relate M to the data set cardinality, and represent it as a percentage. For example, for data set *TS* with cardinality 76 k, $M = 2\ \mathrm{percent}$ means that the memory can accommodate 1,520 values. Note that this is not equivalent to the space occupied by 2 percent of the data set. In fact, since *TS* consists of 5D points, 1,520 values are sufficient to represent only 304 points, or 0.4 percent of the data set.

In fact, when a certain amount of memory is available, I/O performance can also be improved by simply using the memory as a buffer. Hence, we compare Stat-Filter against the following BUFFER approach. BUFFER is the basic

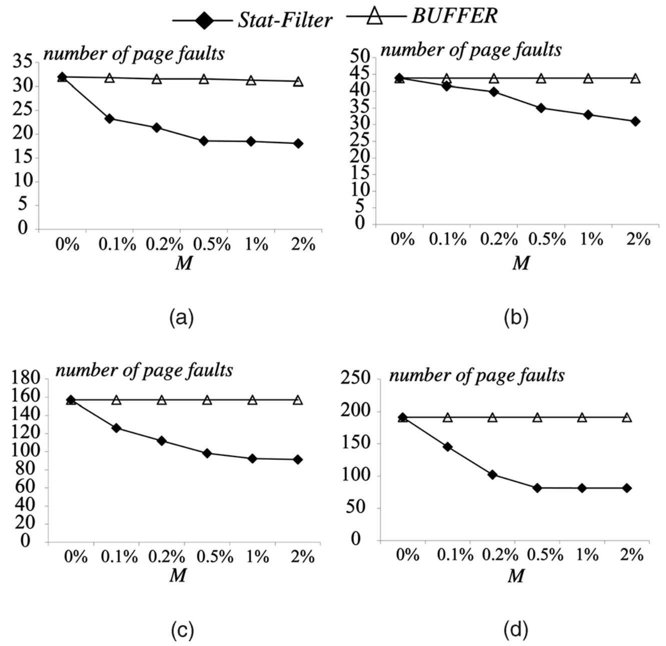


Fig. 17. I/O cost of Stat-Filter versus M ($M =$ number of values kept in memory / data set cardinality, $k_max = 8$). (a) *SF*. (b) *TS*. (c) *Color*. (d) *Signature*.

RkNN-Filter algorithm, coupled with an adapted LRU cache replacement strategy. Specifically, when the cache overflows, the strategy first evicts pages corresponding to nodes at a lower level of the tree. Among pages at the same level, the one least frequently visited is evicted first. Such a strategy aims at pinning the first few levels of the tree in the memory. For a fair comparison, we assign the same amount of memory for both Stat-Filter and BUFFER.

Fixing the parameter k_{max} to 8, Fig. 17a demonstrates the I/O cost, measured as the number of page faults, as a function of M for all data sets. The result at $M = 0$ is essentially the performance of the basic algorithm that is not aided by statistics. The efficiency of Stat-Filter improves significantly as M increases. In particular, when M equals 2 percent, Stat-Filter is faster than the basic algorithm by a factor up to 2. On the other hand, BUFFER receives little improvement. As mentioned earlier, even for the largest M, the memory is sufficient for storing only 0.4 percent of data set *TS*. The corresponding percentages for *SF*, *Color*, and *Signature* equal 1 percent, 0.5 percent, and 0.2 percent, respectively. Buffering has almost no effect with such small amounts of memory. Since BUFFER has the same behavior in all our experiments, it is omitted from the following discussion.

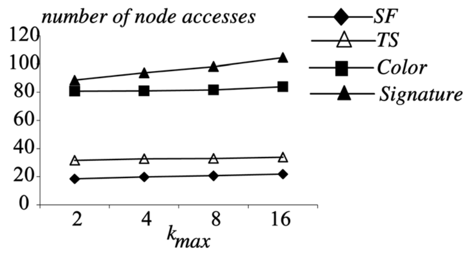


Fig. 18. I/O cost of Stat-Filter versus k_{max} ($M = 0.5$ percent).

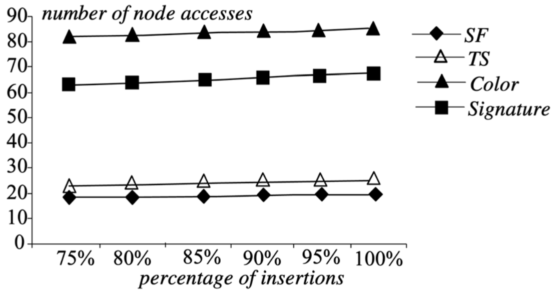


Fig. 19. Performance degradation of Stat-Filter with updates ($k_{max} = 8$, $M = 2$ percent).

In Fig. 18, we set M to 0.5 percent and measure the I/O cost of Stat-Filter on all data sets by varying k_{max} from 2 to 16. Stat-Filter performs more I/Os for a workload with larger k_{max}, which is consistent with the results in Fig. 14.

As mentioned in Section 5.3, Stat-Filter has the best performance right after the statistics are computed but may gradually deteriorate as objects are inserted/deleted to/from the M-tree. The next set of experiments quantifies the degradation rate, using $M = 2$ percent. Toward this, we first create an M-tree on 75 percent of the objects in a data set, after which the statistics required by Stat-Filter are computed. Next, the remaining objects are inserted but each insertion is accompanied by a deletion that removes an existing object from the tree (i.e., the number of objects in the tree remains fixed). The statistics are dynamically maintained during these updates. We measure the I/O cost of Stat-Filter for a workload with $k_{max} = 8$, after 75 percent (i.e., the time of statistics computation), 80 percent, $\ldots$, 100 percent of the data set have been inserted, respectively. The results are presented in Fig. 19. Evidently, the performance of Stat-Filter deteriorates very slowly, such that its I/O cost increases by around only 10 percent after all updates, compared to the cost at 75 percent.

Finally, we study the effectiveness of Stat-Filter on static data, applying the optimization discussed in Section 5.4. Fixing $k_{max} = 8$, Fig. 20 shows the number of node accesses performed by Stat-Filter as a function of M. Clearly, the I/O cost decreases quickly as M increases. The phenomenon is similar to that in Fig. 17, except that here, given the same $M \neq 0$, the improvement over the basic solution (corresponding to $M = 0$) is even more significant.

7 Conclusions and Future Work

In this paper, we presented a careful study of processing RkNN queries in metric spaces. We proposed algorithms

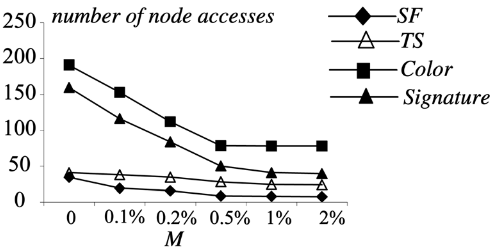


Fig. 20. Performance of Stat-Filter on static data ($k_{max} = 8$).

that do not require representations of the underlying objects, are applicable as long as the similarity between two objects can be evaluated, and satisfy the triangle inequality. Our technique leverages only a metric index and, hence, trivially supports object updates by resorting to the insertion/deletion procedures of the index. Our solutions require small implementation efforts, since they support any types of objects and any similarity metrics in a uniform manner.

This work motivates several directions for future work. First, the efficiency of R*k*NN search may be further enhanced if more powerful pruning heuristics can be discovered. Second, the problem we addressed corresponds to "monochromatic RNN search" defined in [11]. Extending the solutions to the "bichromatic" case [11] is a challenging but exciting topic. Last but not the least, it would be interesting to explore the possibility of using our algorithms for mining the correlation among metric data.

ACKNOWLEDGMENTS

Yufei Tao was supported by Grant CityU 1163/04E from the RGC of the HKSAR government, and SRG Grant 7001843 from the City University of Hong Kong. Man Lung Yiu and Nikos Mamoulis were supported by Grant HKU 7149/03E from the RGC of the HKSAR government.

REFERENCES

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles, " *Proc. SIGMOD Conf.*, pp. 322-331, 1990.

[2] R. Benetis, C.S. Jensen, G. Karciauskas, and S. Saltenis, "Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects," *Proc. Int'l Database Eng. and Applications Symp. (IDEAS),* pp. 44-53, 2002.

[3] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications.* Springer, 2000.

[4] M.M. Breunig, H.-P. Kriegel, R.T. Ng, and J. Sander, "LOF: Identifying Density-Based Local Outliers," *Proc. SIGMOD Conf.*, pp. 93-104, 2000.

[5] P. Ciaccia and M. Patella, "Searching in Metric Spaces with User-Defined and Approximate Distances," *ACM Trans. Database Systems,* vol. 27, no. 4, pp. 398-437, 2002.

[6] P. Ciaccia, M. Patella, and P. Zezula, "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces," *Proc. Very Large Data Bases Conf. (VLDB),* pp. 426-435, 1997.

[7] P. Ciaccia, M. Patella, and P. Zezula, "A Cost Model for Similarity Queries in Metric Spaces," *Proc. Symp. Principles of Database Systems (PODS),* pp. 59-68, 1998.

[8] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A.E. Abbadi, "Constrained Nearest Neighbor Queries, *Proc. Symp. Spatial and Temporal Databases (SSTD),* pp. 257-278, 2001.

[9] G.R. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *ACM Trans. Database Systems,* vol. 24, no. 2, pp. 265-318, 1999.

[10] G.R. Hjaltason and H. Samet, "Index-Driven Similarity Search in Metric Spaces," *ACM Trans. Database Systems,* vol. 28, no. 4, pp. 517-580, 2003.

[11] F. Korn and S. Muthukrishnan, "Influence Sets Based on Reverse Nearest Neighbor Queries," *Proc. SIGMOD Conf.*, pp. 201-212, 2000.

[12] K.-I. Lin, M. Nolen, and C. Yang, "Applying Bulk Insertion Techniques for Dynamic Reverse Nearest Neighbor Problems," *Proc. Int'l Database Eng. and Applications Symp. (IDEAS),* pp. 128-132, 2002.

[13] A. Maheshwari, J. Vahrenhold, and N. Zeh, "On Reverse Nearest Neighbor Queries," *Proc. Canadian Conf. Computational Geometry (CCCG),* pp. 128-132, 2002.

[14] A. Nanopoulos, Y. Theodoridis, and Y. Manolopoulos, "C2P: Clustering Based on Closest Pairs," *Proc. Very Large Data Bases Conf. (VLDB),* pp. 331-340, 2001.

[15] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient Algorithms for Mining Outliers from Large Data Sets," *Proc. SIGMOD Conf.,* pp. 427-438, 2000.

[16] N. Roussopoulos, S. Kelley, and F. Vincent, Nearest Neighbor Queries, *Proc. SIGMOD Conf.*, pp. 71-79, 1995.

[17] A. Singh, H. Ferhatosmanoglu, and A.S. Tosun, "High Dimensional Reverse Nearest Neighbor Queries," *Proc. Conf. Information and Knowledge Management (CIKM),* pp. 91-98, 2003.

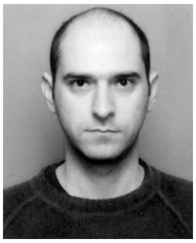
[18] I. Stanoi, D. Agrawal, and A.E. Abbadi, "Reverse Nearest Neighbor Queries for Dynamic Databases," *Proc. ACM SIGMOD Workshop,* pp. 744-755, 2000.

[19] Y. Tao, D. Papadias, and X. Lian, "Reverse *k*NN Search in Arbitrary Dimensionality," *Proc. Very Large Data Bases Conf. (VLDB),* pp. 744-755, 2004.

[20] C. Yang and K.-I. Lin, "An Index Structure for Efficient Reverse Nearest Neighbor Queries," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 485-492, 2001.

Yufei Tao received the PhD degree in computer science from the Hong Kong University of Science and Technology. During 2002-2003, he was a visiting scientist at the Carnegie Mellon University, Pittsburgh. Since September 2003, he has been an assistant professor in the Department of Computer Science at the City University of Hong Kong. He is the winner of the Hong Kong Young Scientist Award 2002, conferred by the Hong Kong Institution of Science.

Man Lung Yiu received the bachelor's degree in computer engineering from the University of Hong Kong, China, in 2002. He is currently a PhD candidate in the Department of Computer Science at the University of Hong Kong. His research interests include databases and data mining.



Nikos Mamoulis received the diploma in computer engineering and informatics in 1995 from the University of Patras, Greece, and the PhD degree in computer science in 2000 from the Hong Kong University of Science and Technology. Since September 2001, he has been an assistant professor in the Department of Computer Science at the University of Hong Kong. In the past, he has worked as a postdoctoral researcher at the Centrum voor Wiskunde en Informatica (CWI), the Netherlands. His research interests include complex data management, data mining, advanced indexing and query processing, and constraint satisfaction problems. He has published more than 60 articles in reputable international conferences and journals and served in the program committees of major database and data mining conferences.