Reverse Nearest Neighbor Queries for Dynamic Databases *

Ioana Stanoi, Divyakant Agrawal, Amr El Abbadi
Computer Science Department, University of California at Santa Barbara
{ioana,agrawal,amr@cs.ucsb.edu}

Abstract

In this paper we propose an algorithm for answering reverse nearest neighbor (RNN) queries, a problem formulated only recently. This class of queries is strongly related to that of nearest neighbor (NN) queries, although the two are not necessarily complementary. Unlike nearest neighbor queries, RNN queries find the set of database points that have the query point as the nearest neighbor. There is no other proposal we are aware of, that provides an algorithmic approach to answer RNN queries. The earlier approach for RNN queries ([KM99]) is based on the pre-computation of neighborhood information that is organized in terms of auxiliary data structures. It can be argued that the pre-computation of the RNN information for all points in the database can be too restrictive. In the case of dynamic databases, insert and update operations are expensive and can lead to modifications of large parts of the auxiliary data structures. Also, answers to RNN queries for a set of data points depend on the number of dimensions taken in considerations when initializing the data structures. We propose an algorithmic approach that is flexible enough to support a larger class of RNN queries, and, in order to support them, we also extend the current method of nearest neighbor search to that of *conditional* nearest neighbor.

1 Introduction

The problem of answering Reverse Nearest Neighbor (RNN) queries has been formally defined only recently by [KM99]. It is a complimentary problem to that of finding a Nearest Neighbor (NN) to a query point, studied in the past few years [RKV95, KSF+96, SR99]. While NN queries find the point in the database that is the closest to a given query point according to a specified distance metric, RNN queries find the set of database points that have the query point as the nearest neighbor. The complexity of the RNN problem arises from the fact that the NN/RNN relationship is asymmetric. That is, the fact that a query point q has a data point p as its nearest neighbor, does not imply that p's nearest neighbor is q and therefore does not imply that q's RNN is the data point p. For example, let three points a, b, c be placed on a straight line in this order, such that the nearest neighbor of a is b and the nearest neighbor of b is c. Note that in this case, although the nearest neighbor of a is b, a has no reverse nearest neighbor. In this paper we present an efficient solution for RNN queries for 2-dimensional data, although our approach can be extended to data sets of higher dimensions. For 2-dimensional data, there is a wide range of applications that can benefit from efficiently implementing RNN queries.

Example: Consider a gas vendor company whose marketing decision is to expand by opening a new gas station in the same city. Given several choices for its new location, the company's strategy is to pick the location that can attract customers from other gas stations but, if possible, without negatively impacting its existing gas stations. Having a database that stores the location of all gas stations in the city together with additional information on businesses, an approach is to create a view that selects only the information on gas stations, and query this view for the gas stations that would be geographically closer to the new possible locations. In this case, an RNN query would return the set of gas stations that would be the most affected by a competitor in the given location (the query point). The company can also easily analyze the effects of competing with different companies by looking at the RNN queries of already existing locations.

In this paper we develop an algorithmic solution to the RNN problem that uses a generic index structure, which, in addition to searches for reverse nearest neighbors, can be used to answer any of the standard queries. The presentation is organized as follows. We

*This work was supported in part by NSF under grant numbers CCR-9712108 and EIA-9818320 and IIS98-17432 and IIS99-70700.

first present (Section 2) previously developed methods, compare them with our approach and summarize our goals in Section 3. In Section 3.1 we present the underlying ideas behind the algorithmic solution we propose for the problem of RNN queries. The algorithm is based on existing solutions for nearest neighbor queries, which in Section 3.3 we extend for the purpose of RNN queries to *conditional* nearest neighbor queries. We further present the details of the RNN algorithm (Section 3.4) and we conclude with a summary of possible extensions of our work in Section 4.

2 Background and Motivation

A solution for answering RNN queries for dynamic databases was recently proposed in [KM99]. This solution is based on storing a list of all the data points and their corresponding nearest neighbors in an NN-tree, and the nearest neighborhood distance circles (for 2-dimensional data) in an RNN-tree. RNN queries become point enclosure queries over the RNN-tree. The use of an RNN tree to store information on reverse nearest neighbor may however be too rigid for querying the database. As an example, let a relational database contain tuples with multiple attributes. A tuple would then become a data point in the multidimensional space, but usually not all the available information is necessary to answer a given query. If we revisit the gas station example presented earlier, let the database actually contain tuples with more than the two attributes used to define the point as a geographical location. Assume tuples also include specific data on the type of service they offer. One dimension can therefore store the levels of service as four choices: 0) gas only, 1) mini market, 2) automatic car-wash and 3) hand car-wash. For the purpose of the facility location query this attribute represents information that is not significant and it should be ignored. The data set then is actually projected on only one plane (spanned by the two geographical direction axes). In general, different queries applied to a data set should be able to project the data points on the desired dimensions, and not be restricted to a given set or subset of attributes. So far, there has been no proposal for answering RNN queries in such a flexible manner. Since the set of nearest and reverse nearest neighbors to a query point may differ depending on how many dimensions are considered for the same data set,a data structure constructed specifically for answering RNN queries is too rigid. Consider a query point q and the data set $\{p_1, p_2, p_3\}$ illustrated in Figure 1. If the query RNN(q) is defined over all three dimensions (XYZ), then the reverse nearest neighbor of q is p_1. However, if the queried space is restricted to only two dimensions (XY), then RNN(q) will return data point p_2. We believe that to accommodate RNN queries spanning over a variable subset of dimensions, an algorithmic approach is more appropriate.

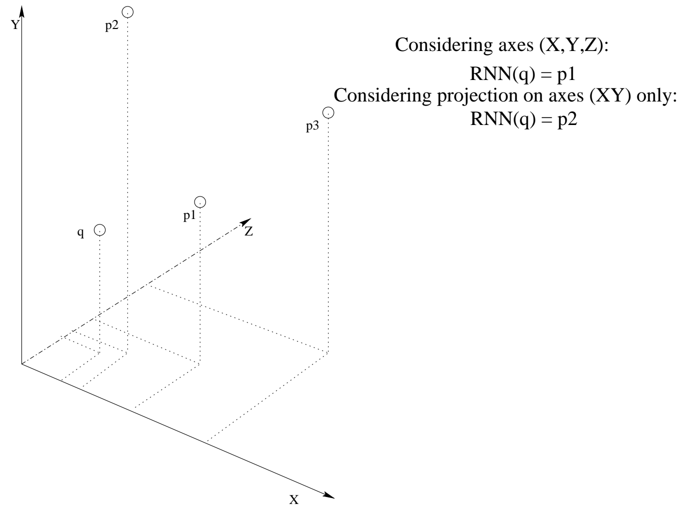


Figure 1: Example of varying nearest neighbor set depending on number of dimensions

In this paper we consider dynamic databases, that undergo inserts and deletes of data points due to user updates. In [KM99] the update and query of dynamic databases is much more expensive than in the static case since insert and delete operations over the database lead to the computation of changes and to the modification of the NN-tree and RNN-tree indexing structures. In contrast, we are interested in RNN queries in the context of data warehouses used for decision support, where updates are frequent and I/O performance is important. As underlying data sources get updated frequently, the views in a data warehouse should reflect these changes and answer to user queries based on data that must be as up-to-date as possible. The update method therefore must be both simple and efficient. In this case, an algorithm based on continuously maintaining a list of the data points and their nearest neighbors is inefficient, since both inserting and deleting an element lead to the partial recomputation of the list. For the type of decision support queries we are interested in, combining the information on the state of the database with auxiliary information on pre-computed partial answers overloads the maintenance task of the view itself.

3 Our Solution

In this paper, we present our solution for answering RNN queries in a two-dimensional space. First, we will introduce the observations that lead to the proposed technique, then give a simple algorithm for finding the reverse nearest neighbor RNN(q) to a query point q. As we will show, our method is based on already

existing algorithms for answering nearest neighbor queries. Applications that use our method can benefit from the following advantages:

- Our goal is to integrate RNN queries in the framework of already existing database and access structures. The approach we are using is therefore algorithmic and independent of data structures constructed specifically for a set of such queries.
- Updates to the database only impact performance in the sense that they affect the size of the R-tree used for indexing the data set for range and spatial join queries. In the case of insert and delete operations over the database, no computational or storage cost will be imposed specifically to facilitate RNN queries. However, we assume that NN queries are supported in such systems.
- No additional data structures are necessary, and therefore the space requirement does not increase. The algorithm we propose is based on using the underlying indexing data structure (R-tree), also necessary to answer NN queries. We assume that a spatial data structure is necessary for many applications that require the ability to answer range queries, topological queries or nearest neighbor queries.
- The solution we present facilitates "what if" queries, based on RNN(q) queries where q is not constrained to be one of the points in the data set. This class of queries are especially useful for decision making applications such as those involving summarization and interpretation of data in data warehouse views.

3.1 Preliminaries

In this paper we assume that the distance between two points $p_1 = (p_{1_1}, p_{1_2}, \cdots p_{1_n})$ and $p_2 = (p_{2_1}, p_{2_2}, \cdots p_{2_n})$ is the Euclidean distance: $dist(p_1 p_2)^2 = \sum_{i=n}^{i=1} |p_{1_i} - p_{2_i}|^2$. The algorithm we propose however is not dependent on this definition, and other distance metrics (such as L_∞) can also be used.

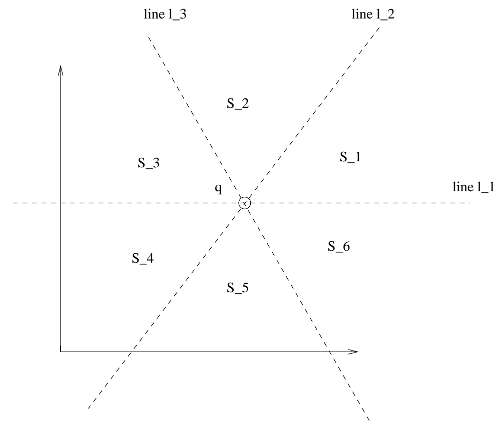
Although the following results refer to data points in a 2-Dimensional space, they can easily be adapted for higher dimensions. In the following, we use the notation RNN(q) to denote the answer to reverse nearest neighbor queries for point q. NN(q) returns the set of points that satisfy the nearest neighbor condition with respect to q. Note that both RNN(q) and NN(q) are sets of points in the database, while query point q may or may not correspond to an actual data point in the data base.

Let query point $q = (x_q, y_q)$ be defined as having value x_q on the x-coordinate and y_q on the y-coordinate. Moreover, let three lines l_1, l_2, and l_3 intersect at q, and divide the space around it into six regions $S_1, S_2, \cdots S_6$ (Figure 2(a)). We fix l_1 to be parallel to the x axis, and the angle between l_1 and l_2, l_2 and l_3, l_3 and l_1 to be exactly 60°. For the rest of the paper, we will refer to l_1, l_2 and l_3 as space dividing lines.

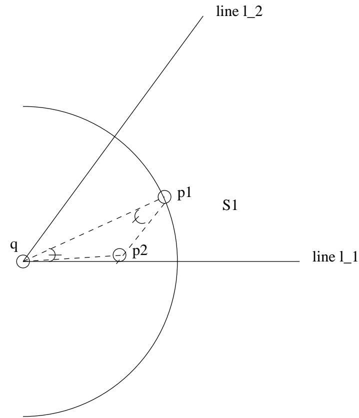
Proposition 1 *: For a given 2-dimensional dataset, any query point q, RNN(q) will return at most six data points [Smi97, KM99].*

In higher dimensions the number of maximum data points that satisfy RNN(q) is still a constant, whose value depends on the number of dimensions considered.

The result stated by Proposition 1 is based on the observation that the upper limit for the number of data points in RNN(q) is given by the scenario where all these points are on a circle with the center at q. The distance between two consecutive points on the circle must be the same as the radius of the circle, i.e., the distance between a point in RNN(q) and q.



(a) Dividing the space around query point q.



(b) Example for Proposition 3.

Figure 2:

A consequence of this observation is the following:

Proposition 2 *: Let the space around a query point q be divided into six equal regions $S_i (1 \leq i \leq 6)$ by*

straight lines intersecting q. S_i therefore is the space between two space dividing lines. Then:

1. *There exist at most two RNN points in each region S_i*
2. *if there exist exactly two RNN points in a region S_i, then each point must be on one of the space dividing lines through q delimiting S_i.*

Proposition 3 *: Let $p = NN(q)$ in S_i. If p is not on a space dividing line, then either $NN(p) = q$ (and then RNN(q)=p), or $\nexists$ $RRN(q) \in S_i$.*

Proof: To show that Prop. 3 holds, assume w.l.o.g. that there are two points in a region S_1 delimited by space dividing lines l_1 and l_2. Let point p_1 be an NN(q), $p_1 \in S_i$ (Figure 2(b)). Let p_2 be another point in S_i such that p_2 is RNN(q). We will show that either $dist(q, p_1) = dist(q, p_2)$ and p_1 on l_1, p_2 on l_2 (or vice-versa), or p_1 and p_2 are the same data point. Assume $p_1 \neq p_2$. Considering the triangle $\triangle qp_1p_2$, it must be the case that p_2 is closer to q then to p_1. That is, $dist(q, p_2) \leq dist(p_2, p_1) \Longrightarrow (\widehat{p_1qp_2}) \leq (\widehat{qp_1p_2}) \leq 60^o$, and therefore angle $(\widehat{qp_2p_1}) \geq 60^o$. Since both p_1 and p_2 are in S_1, if the angle is 60°, it is the case that p_1 is on l_1 or l_2, while, by construction of the space dividing lines, p_2 must be on the opposite line delimiting S_i (l_2 orl_1). Eliminating this case, we look now at the scenario where the angle between lines qp_1 and qp_2 is less than 60° and $(\widehat{qp_2p_1}) > 60^o$. Because now $(\widehat{qp_2p_1}) > (\widehat{qp_1p_2})$ then it must be the case that $dist(qp_1) > dist(qp_2)$, which implies that NN(q)=p_2 and ¬(NN(q)=p_1). However, this consequence contradicts the initial hypothesis that NN(q)=p_1.
□

The above proposition simply states that for a query point q and region S_i either the nearest neighbor is also the reverse nearest neighbor, or there is no RNN(q) in S_i (if NN(q) is not on the space dividing lines). This is a very important result, since it allows us to have a criterion for limiting the choice of RNN(q) to one or two points in each of the six regions S_i. An algorithm for answering an RNN(q) query then first finds the candidate set of data points: one or two points in each region S_i that satisfy the nearest neighbor condition NN(q). In the second step, these candidate points are tested for the condition that their nearest neighbor is q. The points that satisfy this condition are returned as the answer to the RNN(q) query.

3.2 Algorithm Development

In the past years several papers developed index structures based on R-trees, and discussed possible optimizations ([SR87, Gut84, BKSS90, KF93, KF94, BKK96, PF99, EKK00]. [RKV95] proposed a branch-and-bound method to find the nearest neighbor to a query point, based on the traversal and pruning of the R-tree indexing structure. The intuition behind it is that points in a data space can be grouped in clusters delimited by minimum bounding rectangles (MBR), which in turn can be part of larger clusters. An MBR is the smallest rectangle completely enclosing a set of points. Every face on the MBR then contains at least one data point. Since an MBR is rectangular in shape, it can be described by the coordinates of two of its opposite corners. A node in the R-tree then stores these coordinates, and the levels of the R-tree reflect the granularity of the clusters considered (Figure 3). An algorithm to find NN(q) of a query point q searches down the R-tree, by comparing the nodes at each level. Only the nodes that can lead to nearest neighbor points are further considered, while the rest of the tree is ignored. In [RKV95] the proximity comparison is based on "mindist", the shortest distance from the query point to the rectangular cluster as well as the "minmaxdist" (the minimum over all dimensions, of the distance from the query point to the furthest point on the closest face of the MBR). "Mindist" guarantees that all points in the MBR have at least "mindist" distance from the query point. "Minmaxdist" ensures that there exists at least one data point in the MBR whose distance to the query point is at most "minmaxdist". More details are in Section 3.3. Also, [RKV95] proposes to use the square of the Euclidean distance as a distance measure, because of the reduced computation cost.

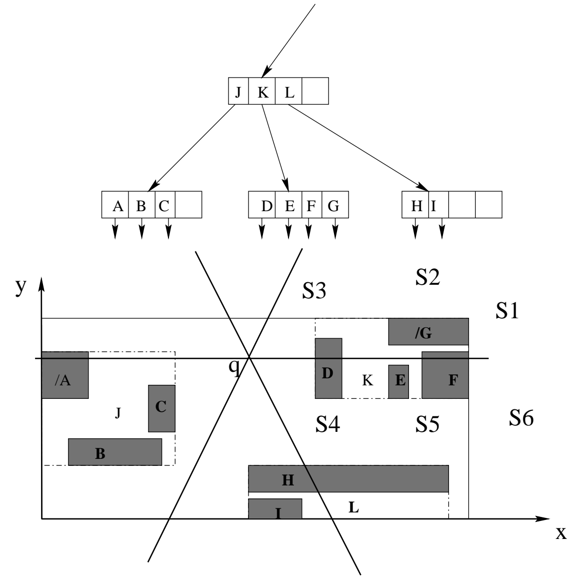


Figure 3: A section of an R-tree and the corresponding regions S_i and minimum bounding rectangles

Our approach for efficiently answering RNN queries builds upon the previous work on NN queries. Since we believe that NN queries are frequent and related to

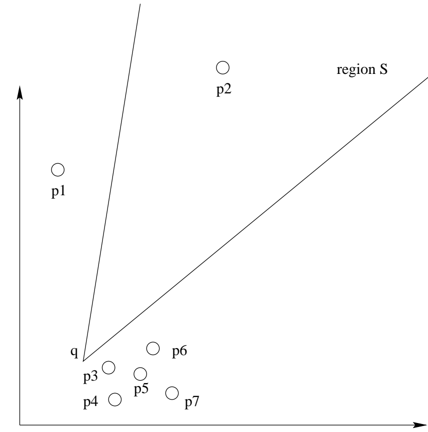
RNN queries, we can safely assume that a method for answering NN queries is already implemented in most systems. The solution described in this paper follows the algorithm described in [RKV95], but it can be easily modified to make use of other techniques developed for NN queries.

The method we propose finds for each region S_i, $1 \leq i \leq 6$, (see Section 3.1) the points $p = NN(q)$ that satisfy the condition $p \in S_i$. The set of data points $p = NN(q)$ is the result of the algorithm for nearest neighbor search applied to the data set, but considering only the projections on the desired dimensions. Due to Prop.3 (see Section 3.1), in a 2-Dimensional space only one or two such points can satisfy the reverse nearest neighbor condition. We modify the algorithm used to find nearest neighbor points by introducing new restrictions during the pruning of the indexing tree. While traversing the R-tree, a node is first checked for inclusion in S_i, then it is checked for proximity to query point q. The subtrees rooted at nodes that do not satisfy both of these conditions are then pruned from the node set of interest. Following the traversal of the R-tree for each region S_i in the data space, a set of candidate RNN(q) points are returned. Not all candidate points p are indeed reverse nearest neighbors of q, i.e., satisfy the condition that q=NN(p). Since the query point q is not necessarily one of the existing data points, we test that q=NN(p) by comparing the distance $dist(q, p)$ with the distance between p and its nearest neighbor $dist(p, NN(p))$. It is important to note that, if a data point falls on one of the space dividing lines, then it belongs to two of the S_i's. The same points $p = RNN(q)$ can therefore be returned by different iterations of the tree traversal, and a filtering step is needed to exclude such duplicates.

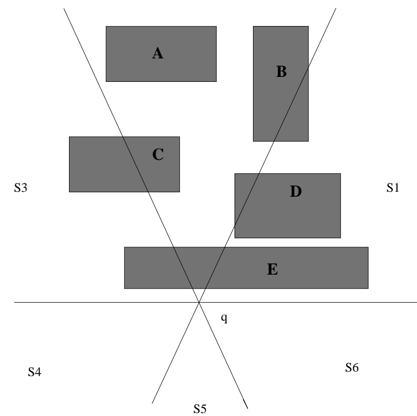
To summarize the above modifications to an algorithm designed for NN queries, the following is an overview. Let a query point be $q = (x_q, y_q)$

1. Construct the space dividing lines l_1, l_2 and l_3 such that:
   (a) l_1, l_2 and l_3 intersect at the query point (x_q, y_q)
   (b) $angle(l_1, l_2) = angle(l_2, l_3) = angle(l_3, l_1) = 60^o$
2. For each S_i (see Section 3.1) do
   (a) traverse R-tree using the conditional NN algorithm (Section 3.3. That is, find p=NN(q) such that $p \in S_i$
   (b) if NN(p) = p' where $dist(pp') \geq dist(pq)$, add p to the list of RNN(q)
3. Eliminate duplicates in RNN(q)

Step $2(a)$ is not straight-forward, since it entails conditions ($p \in S_i$) on the algorithm that finds p = NN(q). In the next section we present the modifications imposed on the NN algorithm in order to support *conditional* NN searches that find the nearest neighbor data point restricted to a region of the space.



(a) Find NN in region S



(b) Cases for MBR containment in a region S2

Figure 4:

3.3 Conditional NN Queries

In this section we analyze nearest neighbor queries constrained to a certain region. Although it is generally an important problem to solve for different types of constraints, for the purpose of the RNN algorithm we consider only linear constraints. We are interested in the special case of finding a nearest neighbor in each of the six regions divided by space dividing lines through a query point. A brute-force algorithm that finds all the nearest neighbors until there is one in the queried region S, can be very inefficient in scenarios like the one described in Figure 4(a). In this case, since data point p_2 in S is actually the furthest from the query point q, a brute-force algorithm would first consider and then discard all other points before finding p_2.

A cluster or an equivalent MBR in the context of R-trees, belongs to a region either fully or partially. In Figure 4(b) we show the five different cases where an MBR (minimum bounding rectangle) is considered to

overlap with a region S_2:

1. MBR A: fully contained in S_2
2. MBR B: three vertices are in S_2. Since B is a minimum bounding rectangle, then there must be some data points in B contained also in S_2
3. MBR C: two of C's vertices are in S_2. This means that an entire edge e of the MBR is in S_2, which by definition guarantees that at least one data point in C is on e and therefore in S_2.
4. MBR D: only one vertex of D is in S_2. It is important to note that in this case there is no implication on the existence of data points contained both in D and in S_2.
5. MBR E: although no vertices are in S_2, part of the E minimum bounding rectangle overlaps with the queried region S. Again, as in the previous case, we cannot make any assumption that any of the data points contained in E are also contained in S.

The approach we use to find a nearest neighbor in a region S_i is to traverse the indexing tree and prune out sections of the tree that cannot lead to an answer either because the MBRs do not belong to the queried region, or because there is a guarantee that other points in S_i are closer to the query point.

If the space constraints are eliminated, the problem of conditional nearest neighbor search is extended from a region to the whole space. The traditional NN methods are therefore only a special case of a more general approach. We will show however that, with only minor modifications, we can extend an NN search algorithm [RKV95] to satisfy conditional nearest neighbor queries. The method described in [RKV95] traverses recursively the indexing tree in depth first order. At each non-leaf level it creates a list of the node's children and sorts it according to the distance metric used. For efficiency, the subtrees rooted at nodes in the list will be visited in the sorted order. Once the recursive function on one of the children returns, a candidate for the nearest neighbor is returned and the list of remaining children to be visited is pruned accordingly. The difference between the algorithm proposed by [RKV95] and a *conditional* nearest neighbor algorithm resides only in the metric used to sort and prune the list of candidate nodes. The idea behind the former method is that if we define the $mindist(q, M)$ to be the minimum distance from the query point to an MBR M and $minmaxdist(q, M)$ to be the minimum over all dimensions, of the distance from q to the furthest vertex on the closest plane of M, then it is guaranteed that there exists at least a data point within the [mindist(q,M), minmaxdist(q,M)] range. Then another MBR M' whose $mindist(q, M') > minmaxdist(q, M)$,

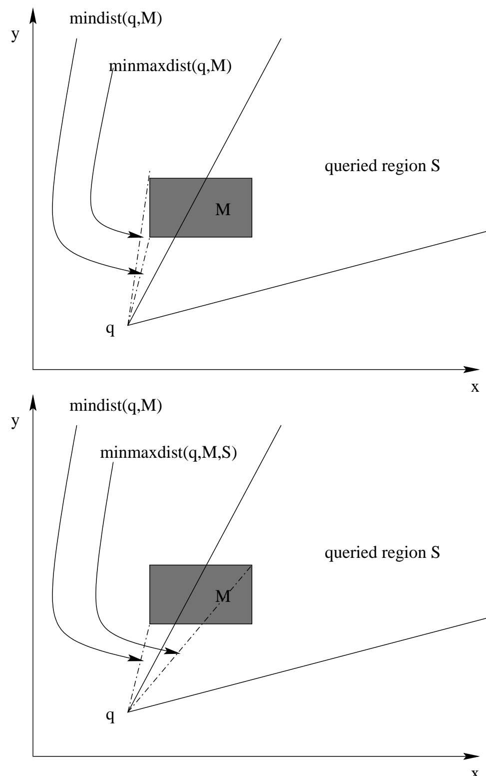


Figure 5: $minmaxdist(q, M)$ is modified to $minmaxdist(q, M, S)$ for conditional NN queries

can be safely discarded. Following the same reasoning, a data point p whose distance to the query point $dist(q, p) > minmaxdist(q, M)$, can also be discarded from the list of candidates to the nearest neighbor. Finally, if there is a point p that satisfies the condition that $dist(q, p) < mindist(q, M)$, then the minimum bounding box M is discarded.

The method described above assumes that the entire volume of a minimum bounding rectangle is contained in the queried region. This is not the case for conditional NN queries, and a few modifications are necessary. In the table below we described five possible positions of MBRs with respect to a region S_2, cases that should be considered when trying to find NN(q) in S_2. Our goal is to modify the notion of $minmaxdist(q, M)$ to have the same implications and therefore be used in the same manner as proposed by [RKV95]. Note that the measure $minmaxdist(q, M)$ may be irrelevant if the MBR M is only partially contained by the the queried region. Since we do accommodate these cases, we redefine $minmaxdist(q, M)$ to be $minmaxdist(q, M, S)$, the distance from the query point to the furthest vertex of M on its closest face included in S.

If an MBR M that intersects region S has one or no vertices in S, then there is no guarantee that

there are data points in both M and S. We then set $minmaxdist = \infty$ to capture this uncertainty. Another case is if M has two or three vertices in S, which implies that at least an edge and therefore at least one data point is in both M and S. This edge is not necessarily the closest to the query point q, and consequently we have to redefine $minmaxdist$ as a function of the distance from q to the furthest vertex of the closest face in region S.

The computation of $mindist(q, M)$ is valid for all cases, since it provides a definite lower bound on the location of data points inside an MBR. For cases where this distance is measured to a point on the MBR that falls outside of region S, the $mindist$ still gives a correct although looser lower bound.

The use of a generic R-tree permits us to easily ignore certain dimensions if the RNN query is not interested in them. In particular, $mindist()$ and $minmaxdist()$ can be defined to simply consider distance to the relevant dimensions only. Hence, depending on the query, alternative dimensions can be considered. This is much more flexible then an approach where a tree is constructed based on the dimensions of interest determined a-priori.

3.4 Algorithm

The RNN algorithm developed in this paper starts by finding one or two nearest neighbors in each of the six regions S_i, and then it tests if these data points indeed satisfy the RNN condition. For clarity, we described how to find the candidate points separately for each region. In reality, the computation corresponding to all the six regions can be done in one traversal of the indexing tree. Separate priority lists should be created for each region, but the information about the minimum bounding rectangles will only be read once. Similarly, the RNN candidates can be verified in one traversal of the indexing tree. We next present the algorithm for finding RNN points.

RNNsearch(). This is the main function that first creates the structure that holds the information on the query point in the necessary format, and initializes *nearest*, the nearest neighbor list. It then calls $CondNNSearch$ which returns a list of nearest neighbors for each of the regions. The duplicates in this list are eliminated ($EliminateDuplicates()$), and the $NNSearch$ function is used to check if indeed these candidates are RNN points. That is, we test if the query point is a nearest neighbor of the points returned by $CondNNSearch$. For clarity we only describe how the conditional nearest neighbor search is done in parallel. However, in practice we can use $CondNNSearch$ for both conditional and unconditional nearest neighbor searches, and use a flag $condFlag$ to distinguish between the two. In the algorithm below we do not integrate the two functions and we do not show the details for running $NNSearch$ in parallel, but the method is very similar to that of parallelizing $CondNNSearch$. The number of regions, $numSections$, depends on the number of dimensions of the queried space, which is inferred from the format of the query point passed as an argument to the function. Since this information is used to define the dimensions of the query space, the dimensions that are not necessary are ignored during the computation of the distance between two MBRs.

Search for Conditional Nearest Neighbor (CondNNSearch). This is a recursive method for searching the conditional nearest neighbors of a query point, and it is based on [RKV95]. When the function $CondNNSearch$ is passed a pointer to a *node* in the indexing tree, it builds a set of lists *branchList* (branchList[0] ... branchList[numSections-1]) for each region that is searched. The branch lists have pointers to the children of *node*. An additional list, $branchList[numSections]$ of pointers to the children of *node* contains counters of pointers from the other branch lists for each of the children. Children with a higher count are visited first, since this step is required by the search of most regions and the I/O overhead can thus be reduced. The branch lists are first pruned ($CondDownPrune()$) based on the nearest neighbor found so far and also based on the relationship between the MBRs contained by the respective children. In this step, according to the distance metric used, the MBRs that cannot contain nearest neighbors are excluded. $CondNNSearch$ is recursively called on the child with a highest count, i.e., that most of the branch lists point to. The function will return an updated list of nearest neighbors, with more than one element if the respective data points are at the same distance from the query point. The new nearest neighbor candidate will be used again to prune the branch lists and refresh the count list ($CondUpPrune()$). Both branch pruning functions $CondDownPrune$ and $CondUpPrune$ use the distance comparison criterion discussed in Section 3.3. If the branch lists are not null, the next child in decreasing order of the count in $branchList[numSetions]$ will be visited. If *node* is a leaf of the index tree, then $CondNNSearch$ only compares the distance from the children to the nearest points and, if necessary, updates the structure *nearest* that stores the list of nearest neighbors.

4 Conclusion

In the approach proposed by [KM99], an additional tree is constructed to store a set of objects consisting of the data points and their distance to the corresponding nearest neighbors, and an RNN(q) query is

Number of vertices in S_i	$mindist(q, M, S_i)$	$minmaxdist(q, M, S_i)$
0 (no intersection of M with S_i)	∞	∞
0 (M intersects S_i) 1	$mindist(q, M)$	∞
2 3 4	$mindist(q, M)$	$minmaxdist(q, M, S_i)$ = distance to furthest vertex on closest face IN S_i

Table 1: Definition of $mindidst(q, M, S_i)$ and $minmaxdist(q, M, S_i)$

```
------------------------------------------------------------------------------------------------------------

/* ------------------------------------------------------------------------
|RNNsearch: returns a list of reverse nearest neighbors of a quey point $q$
-------------------------------------------------------------------------*/
struct Nearest RNNsearch(q)
struct Rectangle *q;
{
   struct Nearest *nearest, *newnearest, *RNNresult;
   int newRNN; /*number of RNN candidates*/
   Initialize(nearest);
   /*return a list of RNN candidates*/
   NNCondSearch(TreeRoot,q, nearest, numSections, NODECARD,0);
   /*eliminate duplicates from the RNN list and return the count of remaining candidates*/
   numRNN = EliminateDuplicates(nearest);
   for i=0 to numRNN do
      {
      NNSearch(TreeRoot,nearest[i],newnearest);
      /*if the candidate nearest[i] is indeed an RNN, add it to the list*/
      if Rect2Rectdistance(q, &newnearest[i]->branch[0].rect) == Rect2Rectdistance(q,nearest->branch[0].rect)
          Append(RNNresult, nearest[i]);

      }
   return RNNresult;
}
------------------------------------------------------------------------------------------------------------
```

Table 2: Algorithm for Reverse Nearest Neighbor search

```
-------------------------------------------------------------------------------------------------------------
/* -----------------------------------------------------------------
|CondNNSearch: recursive conditional nearest neighbor search. Based
|on the query point ("point") and the number of dimensions, divide
|the space into a number numSections of regions and search in parallel
|for the nearest neighbors.
-----------------------------------------------------------------*/
CondNNSearch(node,point,nearest,numSections,NODECARD,condFlag)
Node *node;
Rectangle *point;
struct Nearest nearest;
int numSections,NODECARD; /*NODECARD is the number of a node's children*/
boolean condFlag; /*distinguishes between conditional and unconditional search. We
                  do not explicitly use it in this presentation*/
{
   branch[numSections+1][NODECARD] branchList;
   int i,j,k, dist, nearestDist;
   if (node->level >= 0) /*this is an internal node*/
   {
       /*generate the six branchLists*/
       for i:=0 to NODECARD do
           for k:=0 to numSections do
               Update(branchList[k]);
        /*perform downward pruning*/
        CondDownPrune(point, nearest, branchList);
        /*create branchList[numSections], the ordered counter list*/
        UpdateCount(branchList);
        for j:=0 to NODECARD do
        {
           newnode = branchList[numSections][0]->child;
           /*recursively visit children*/
           CondNNSearch(newnode,point, nearest,numSections,NODECARD,condFlag);
           /*perform upward pruning of the branch lists*/
           CondUpPrune(point, nearest, branchList);
           /*update the ordered counter list, branchList[numSections]*/
           UpdateCount(branchList);
        }
     }
     else /*this is a leaf node*/
        for i:=0 to NODECARD
           if (node->branch[i].child != NULL)
           {
               /*"dist" is the distance between node->branch[i].rect and query point*/
               dist = Rect2Rectdistance(point, &node->branch[i].rect);
               nearestDist = Rect2Rectdistance(point, &nearest[0]->branch[0].rect)
               /*if nearest has not been set yet or it represents a rectangle
               further from the query point than node->branch[i]*/
               if ((nearest[0].rect == NULL)||(dist < nearestDist))
               /*if branch[i].rect is a possible nearest neighbor, add it to the NN list*/
               else if (dist == Rect2Rectdistance(point, &nearest->branch[i].rect))
                   Append(nearest, branch->[i].rect);
           }
}
-------------------------------------------------------------------------------------------------------------
```

Table 3: Algorithm for Conditional Nearest Neighbor search

answered by testing the enclosure of q within these objects. Therefore, only one traversal of the RNN tree is necessary to compute answers to RNN queries. However, the drawbacks to this approach arise if updates are allowed to modify the database. To maintain the advantage of fast query answers, sections of both NN and RNN trees should be modified with every insert and delete operation. We choose to develop an algorithmic method, since we are specifically interested in dynamic databases, where updates are frequent and queries have the flexibility to refer only to a subset of the data space dimensions. Our method does not require the construction and maintenance of any additional data structures and only the underlying indexing tree is updated. We believe that in dynamic databases the disadvantage of a higher cost during query computation is outweighed by the simplicity and efficiency of updates. In our future plans we include a performance comparison with the original solution [KM99]. Also, we are interested in exploring alternative implementations based on R-trees, to concurrently evaluate a wider range of queries.

References

[BKK96] S. Berchtold, D. A. Keim, and H. P. Kriegel. The x-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB)*, pages 28–36, 1996.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and reactangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, May 1990.

[EKK00] M. Ester, J. Kohlhammer, and H. P. Kriegel. The dc-tree: a fully dynamic index structure for data warehouses. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, March 2000.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, June 1984.

[KF93] I. Kamel and C. Faloutsos. On packing r-trees. In *Proceedings of the 2nd International Conference on Information and Knowledge Management (CIKM)*, pages 490–499, 1993.

[KF94] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the International Conference on Very Large Databases*, September 1994.

[KM99] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. Technical report, AT&T Labs Research, http://www.research.att.com/resources/trs/, 1999.

[KSF+96] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest-neighbor search in medical image databases. In *Proceedings of the International Conference on Very Large Databases*, September 1996.

[PF99] G. Proietti and C. Faloutsos. I/o complexity for range queries on region data stored using an r-tree. In *Proceedings of the International Conference on Data Engineering (ICDE)*, March 1999.

[RKV95] N. Roussopoulos, S. Kelly, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 71–79, May 1995.

[Smi97] M. Smid. Closest point problems in computational geometry. In *Handbook on Computationl Geometry, Elsevier Science Publishing*, 1997.

[SR87] T. Sellis and N. Roussopoulos. The r+-tree: A dynamic index for multidimensional objects. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB)*, pages 507–518, May 1987.

[SR99] T. Sellis and N. Roussopoulos. Distance browsing in spatial databases. In *ACM Transactions on Database Systems (TODS), 24(2)*, June 1999.