Data & Knowledge Engineering 70 (2011) 955–983

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

On multi-type reverse nearest neighbor search

Xiaobin Ma [a,*], Chengyang Zhang [b], Shashi Shekhar [c], Yan Huang [b], Hui Xiong [d]

[a] 1 Oracle Drive, Nashua, NH 03062, USA
[b] Department of Computer Science and Engineering, University of North Texas, TX, 76207, USA
[c] Department of Computer Science and Engineering, University of Minnesota, 200 Union Steet SE, Minneapolis, MN 55455, USA
[d] Management Science and Information Systems Department, Rutgers University, NJ, 07102, USA

ARTICLE INFO

Article history:
Received 6 July 2010
Received in revised form 3 May 2011
Accepted 16 June 2011
Available online 7 July 2011

Keywords:
Reverse nearest neighbor search
Spatial database
Location-based service

ABSTRACT

This paper presents a study of the Multi-Type Reverse Nearest Neighbor (MTRNN) query problem. Traditionally, a reverse nearest neighbor (RNN) query finds all the objects that have the query point as their nearest neighbor. In contrast, an MTRNN query finds all the objects that have the query point in their multi-type nearest neighbors. Existing RNN queries find an *influence set* by considering only one feature type. However, the *influence* from multiple feature types is often critical for strategic decision making in many business scenarios, such as site selection for a new shopping center. To that end, we first formalize the notion of the MTRNN query by considering the *influence* of multiple feature types. We also propose R-tree based algorithms to find the *influence set* for a given query point and multiple feature types. Finally, experimental results are provided to show the strength of the proposed algorithms as well as design decisions related to performance tuning.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

Given a data set P and a query point $f_{q,q}$, a *reverse nearest neighbor* (RNN) query finds all objects in P that have the query point $f_{q,q}$ as their nearest neighbor. When the data set P and the query point $f_{q,q}$ are of the same feature type, the RNN query is said to be monochromatic. When they are from two different feature types, the query is bichromatic. An RNN is said to represent a set of points in P that has been *influenced* by a query point $f_{q,q}$ [1] and for this reason RNN queries have been widely used in Decision Support Systems, Profile-Based Marketing, etc. For example, before deciding to build a new supermarket, a company needs to know how many customers the supermarket may potentially attract. In other words, it needs to know the *influence* of opening a supermarket. An RNN query can be used to find all residential customers that live closer to the new supermarket than any other supermarket. If we assume that the presence alone of a new supermarket will determine whether customers choose to shop at it, results of this query may be used to decide whether to go ahead with the project. However, queries based on such assumptions may not always produce useful results. A decision to shop at the new store may also be influenced by the presence of other types of stores. For example, some customers may want the opportunity to shop for groceries, electronics, and wine. Here, what influences customers' choice of grocery store is the shortest route through one grocery store, one electronics store, and one wine shop rather than the shortest route to the grocery store alone. In this case, the RNN query needs to consider the influence of feature types besides grocery store. To date, both monochromatic and bichromatic RNN queries have considered the influence of only one feature type. As the above example shows, there is a need to also consider the influence of other feature types in addition to that of the given query point.

Fig. 1 illustrates how two feature types affect the results of an RNN query. In the figure, point $f_{q,q}$ is the given query point and point $f_{q,1}$ is another point from the same feature type F_q, which we call the query feature type. Point $f_{1,1}$ is a point from a second

* Corresponding author. Tel.: +1 603 897 3351.
E-mail addresses: xiaobin@cs.umn.edu (X. Ma), chengyang@unt.edu (C. Zhang), shekhar@cs.umn.edu (S. Shekhar), huangyan@unt.edu (Y. Huang), hxiong@rutgers.edu (H. Xiong).

0169-023X/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2011.06.003

feature type F_1. p_1 is a point from the data set P in which the RNNs will be found. The perpendicular bisector $\perp(f_{q,1},f_{q,q})$ or l_1 between points $f_{q,1}$ and $f_{q,q}$ in Fig. 1a divides the space into two half-spaces: $Plane(l_1,f_{q,1})$ containing point $f_{q,1}$ and $Plane(l_1,f_{q,q})$ containing point $f_{q,q}$. Any point falling inside half plane $Plane(l_1,f_{q,1})$ is closer to point $f_{q,1}$ than $f_{q,q}$. Similarly half plane $Plane(l_1,f_{q,q})$ contains all points that are closer to $f_{q,q}$ than $f_{q,1}$. For perpendicular bisector $\perp(f_{q,q},f_{q,1})$ or l_2, these properties also hold.

In Fig. 1a, if the influence of point $f_{1,1}$ is not considered, p_1 is an RNN of the given query point $f_{q,q}$ since $f_{q,q}$ is the nearest neighbor of p_1. However, when the influence of point $f_{1,1}$ is considered, p_1 is not an RNN of $f_{q,q}$ because the distance of the route $R(p_1,f_{q,1},f_{1,1})$ is shorter than the distance of the route $R(p_1,f_{q,q},f_{1,1})$. As an opposite example, in Fig. 1b, when the influence of point $f_{1,1}$ is not considered, p_1 is not an RNN of the given query point $f_{q,q}$ since $f_{q,1}$ is the nearest neighbor of p_1. However, when the influence of point $f_{1,1}$ is considered, p_1 is an RNN of $f_{q,q}$ because the distance of the route $R(p_1,f_{q,q},f_{1,1})$ is shorter than the distance of the route $R(p_1,f_{q,1},f_{1,1})$. This example shows that an RNN query may give different results depending on the number of feature types accounted for. Implicit in the multi-type RNN search that we describe is a query to find the shortest distance from point p_1 through one instance of given feature types F_q and F_1. We call this query a multi-type nearest neighbor (MTNN) query in [2].

Fig. 2 illustrates another business query problem. Assume that a business rule states "Build a new grocery store if the store will be a nearest neighbor of more than 30 residences". In the figure, assume that point $f_{q,i}$ is a grocery store from feature type F_q, grocery store. The new grocery store to be built by our grocery store chain is represented by the point $f_{q,q}$, which is the query point. Every grocery store is represented by a circle. Point $f_{1,i}$ is a wine shop from feature type F_1, wine shop. Every wine shop is represented by a triangle. The solid points represent residences. The lines l_1, l_2, l_3, l_4 and l_5 are perpendicular bisectors between point $f_{q,q}$ and $f_{q,1}$, $f_{q,q}$ and $f_{q,2}$, $f_{q,q}$ and $f_{q,3}$, $f_{q,q}$ and $f_{q,4}$, and $f_{q,q}$ and $f_{q,5}$ respectively. A classical RNN query considers the influence of the grocery store only and finds that all residences located inside the region formed by lines l_1, l_2, l_3, l_4 and l_5 have the point $f_{q,q}$ as their nearest neighbor. Thus the new grocery store $f_{q,q}$ is the nearest neighbor of these residences, and the residences are the reverse nearest neighbors of $f_{q,q}$. In this business context, shoppers from the residences are considered likely to visit the new grocery store $f_{q,q}$ because their route to the new store is shorter than to any other grocery store. Further, the number of residences totals 37 in this case, which is enough to justify going ahead with the project to build the new grocery store $f_{q,q}$.

However, this classical RNN query result does not account for shoppers who want the opportunity to buy not only groceries but also wine. (Some municipalities do not permit the sale of wine in grocery stores.) Such shoppers are interested in finding the shortest path through not one, but two types of stores, a grocery store and a wine shop. A reverse nearest neighbor search that considers the influence of the additional feature type wine shop may yield more useful results as shown in Fig. 2. Indeed, after applying the same distance calculation method, i.e., the MTNN query defined in [2] and used previously in Fig. 1, the shortest route from some of the residences (the blue or gray points in the figure) through one grocery store and one wine shop does not contain the new grocery store $f_{q,q}$. This means that these residences are no longer RNNs of the new store and not likely to visit it. More specifically, when two feature types F_q grocery store and F_1 wine shop are considered in the query, only 21 residences are found to be RNNs of $f_{q,q}$, which does not meet the threshold required for building a new store.

As this example demonstrates, an RNN query that considers the influence of more than one feature type can produce quite different results from an RNN query based on a single feature type. The differences can be two-fold. In our example above, the multi-type query returned a different *number* of RNNs. In other cases, the *specific* RNNs returned may differ. For example, a grocery chain may be interested in knowing not only the number of potential shoppers but also the average household income of potential shoppers at a new grocery store. A query that considers the influence of the grocery store alone will generate one answer (e.g., average household income = $27,000), while a query that considers the influence of additional nearby businesses (e.g., wine shop and electronics store) may generate another (e.g. average household income = $42,000, reflecting the fact that a different set of customers is attracted to these shops). Whether it is a difference in the number of RNNs returned or the RNNs themselves, multi-type RNN queries have considerable potential to impact decision-making in business applications.

1.1. Our contributions

We formalize the Multi-Type Reverse Nearest Neighbor (MTRNN) query problem to consider the influence of other feature types in addition to the feature type of the given query point. We propose R-tree based algorithms to prune the search space by filtering R-tree nodes and points that are definitely not the MTRNN of the given query point. Then refinement approaches are used to remove the false hit points.

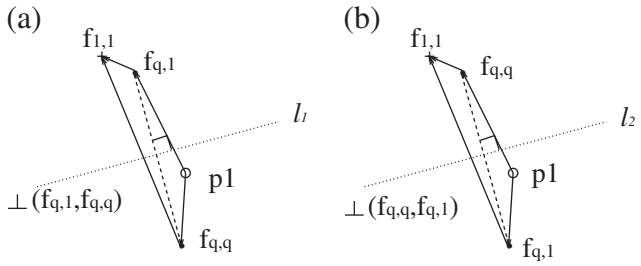


Fig. 1. Influence of two feature types.

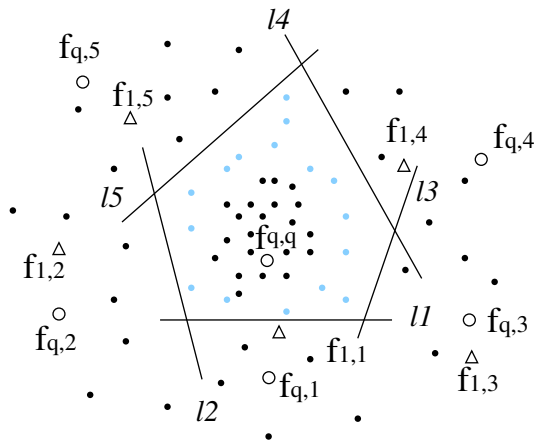


Fig. 2. A use case.

Our experiments on both synthetic and real data sets demonstrate that typical MTRNN queries can be answered by our algorithms within reasonable time. The design decisions related to performance tuning are provided as well. We also include experimental results that vividly highlight the degree to which MTRNN query results can differ from results of traditional RNN queries.

1.2. Overview

The remainder of this paper is organized as follows. Section 2 formalizes the MTRNN problem and presents a brute force algorithm as a baseline algorithm. In Section 3 we propose two filtering methods to prune the search space and three refinement approaches to remove the false hit points. Section 4 analyzes the complexity of the algorithms and Section 5 gives the experimental setup, results and a discussion. Related work is presented in Section 6. Finally, in Section 7, we conclude and suggest future work.

2. Preliminaries

In this section, we introduce some basic concepts, explain some symbols used in the remainder of the paper and give a formal statement of the MTRNN query problem. We also present a one-step brute force algorithm for the MTRNN query as a baseline algorithm that is directly based on the formulation of the MTRNN query problem. Table 1 summarizes the symbols to be used in the rest of this paper.

2.1. Problem formulation

Let p_i represent any point in the queried data set P, $\{p_1, p_2, \ldots, p_n\}$ be a point set, F_i be a feature type, and $f_{i,i'}$ be a point $f_{i'}$ from feature type F_i.

Table 1
Summary of symbols.

Notation	Description
P	Queried data set
p_i	A point from queried data set P
F_q	Feature type q that the query point $f_{q,q}$ belongs to
F_i	Feature type i
$f_{q,q}$	A query point
$f_{i,j}$	A point j from feature type F_i
s_i	A point i in space
$s_i s_j$	Line segment $s_i s_j$
$len(s_i s_j)$	Length of line segment $s_i s_j$
$R(\ldots,\ldots)$	A route through some given points
$d(R(\ldots,\ldots))$	Distance of a route through some given points
R_i	An *R-tree index node*
fr	A *feature route*
S_{fr}	A set of *feature route*
S_{frps}	A set of *feature route point set*
S_c	A candidate MTRNN set

Definition 1. Partial route

A *Partial Route* $R(p_i f_{1,1'} f_{2,2'},\ldots f_{l,l'})$ is a route from point p_i through points from different feature types $F_1, F_2, \ldots, F_l$. $d(R(p_i f_{1,1'} f_{2,2'},\ldots f_{l,l'}))$ is the distance of the partial route.

Definition 2. Multi-Type Nearest Neighbor (MTNN)

An MTNN of a given point p_i is defined to be the ordered point sequence $<f_{1,1'} f_{2,2'},\ldots f_{k,k'}>$ such that $d(R(p_i f_{1,1'} f_{2,2'},\ldots,f_{k,k'}))$ is minimum among all possible routes from p_i through one instance of feature types $F_1, F_2, \ldots, F_k$.

$R(p_i f_{1,1'} f_{2,2'},\ldots,f_{k,k'})$ is called the MTNN route for the given point p_i. Please note although an MTRNN route resulted from an MTRNN query is an ordered sequence, the order of feature types is not specified in an MTNN query.

Definition 3. Multi-Type Reverse Nearest Neighbor (MTRNN)

An MTRNN of the given point $f_{q,q}$ is defined to be a point p_i in the queried data set P such that the MTNN of the point p_i contains the query point $f_{q,q}$.

For a point p_i in set P and feature types $F_1,\ldots,F_q,\ldots F_k$, find the MTNN route $R(p_i f_{1,1'} f_{2,2'},\ldots,f_{k,k'})$. If the given query point $f_{q,q}$ is on the MTNN route, the point p_i is an MTRNN of the query point $f_{q,q}$.

An MTRNN query is a query finding all MTRNNs for the given query point $f_{q,q}$, k feature types, and the queried data set P. As in the MTNN problem, the order of the given k feature types is not specified because an MTRNN query is trying to find the influence of given feature types. The features have influence no matter what order they are in. Therefore, we don't force any order constraint on the feature types for an MTRNN query.

In our problem formulation, we are querying against a data set P whose feature type differs from the given k feature types. We use Euclidean distance as the distance metric. The query point $f_{q,q}$ is from feature type F_q of the given feature types. Each data set of different feature types has its own separate R-tree index that will be used to find nearest neighbor in the algorithm Fig. 4 and in the adapted MTNN algorithm Fig. 14 of the refinement step. The query finds all MTRNNs in the queried data set P in terms of k different feature types for the given query point $f_{q,q}$. That is, for every point in the queried data set P, first find its MTNN. If the given query point $f_{q,q}$ is on the MTNN of this query point, the queried point is considered to be an MTRNN of the given query point $f_{q,q}$.

The following is the formal definition of the MTRNN query problem.

2.1.1. Problem: The MTRNN query

Given:

- A data set P to be queried against
- Distance metric: Euclidean distance
- k sets of points, each set coming from one of k different feature types
- A query point $f_{q,q}$, coming from feature F_q of the k feature types
- Seperate R-tree index for each data set including queried data set and feature data set

Find:

- Multi-type reverse nearest neighbors (MTRNNs) in the queried data set P for the given k different feature types and the query point $f_{q,q}$ such that $f_{q,q}$ is on the MTNN of each MTRNN point

Objective:

- Minimize the length of the route starting from an MTRNN covering the query point $f_{q,q}$ of F_q and one instance of every other feature type excluding feature type F_q

In our problem formulation, the objective function follows the definition used in [2–5]. The objective function, however, in these studies is very general and does not consider the possible influence difference of different feature types. In other words, every feature type has the same influence on a point. It also does not consider the route returning to the query point, which may be useful in some applications. The objective function defined here incorporates these considerations. Thus, our objective function, represented as $d(R(p_i f_{1,1'} f_{2,2'},\ldots,f_{k,k'}))$, is the distance of the route from a point p_i through one instance of every feature of the given k feature types. If the shortest route found from point p_i contains the query point $f_{q,q}$, then this point p_i is an MTRNN of the query point $f_{q,q}$.

2.2. One step baseline algorithm for the MTRNN query

We developed a naive approach directly based on the concept of the MTRNN query problem and the relationship between MTNN and MTRNN queries. For the given query point $f_{q,q}$, this one-step algorithm simply scans all the points in the queried data set P. More specifically, for every point p_i in P, it finds the MTNN of this point in the given data sets of k different feature types using algorithm described in [5] for one permutation of feature types and then applying the algorithm for all permutations of feature

types as did in [2]. If the query point $f_{q,q}$ is on the MTNN of a data point p_i, the data point p_i is an MTRNN of $f_{q,q}$. This brute force algorithm does not prune any data points so it is very time-consuming and not scalable. It is presented here as a baseline algorithm because it is directly based on the formulation of the MTRNN query problem and is useful for evaluating the correctness of our MTRNN algorithm.

The scalability of the MTRNN algorithm depends on the efficiency of the underlying MTNN search of all the points in the queried data set. Due to the complexity of the current MTNN algorithm [2,5], achieving a scalable MTRNN algorithm requires designing an efficient filtering method to prune most of the queried data before the MTNN algorithm is applied. In the following section, we present an MTRNN algorithm with an efficient filtering step.

3. Multi-type reverse nearest neighbor algorithms

Our MTRNN algorithm is an on-line algorithm consisting of the three major steps, preparatory, filtering, and refinement. The output of the MTRNN query that take into account the influence of multiple feature types is a set of reverse nearest neighbors. The preparatory step in Section 3.1 finds *feature routes* for the filtering step by applying a greedy algorithm that uses R-tree indexes from all feature types during searching. Thus, normally only a small portion of data will be examined. The filtering step in Section 3.2 eliminates R-tree nodes that cannot contain an MTRNN by utilizing *feature routes* and then retrieves all remaining points that are potential MTRNNs to form a candidate MTRNN point set. In this section, we describe two pruning techniques, called closed region pruning and open region pruning, to eliminate all R-tree nodes and points that cannot possibly be MTRNN points. We also prove that both closed and open region pruning techniques do not introduce any false misses in Lemma 1 and Lemma 2 respectively. The refinement step in Section 3.3 removes all the false hit points by three refinement approaches among which the final approach is to search the multi-type nearest neighbor (MTNN) of each candidate point. If the query point is not one of the points in the MTNN of a candidate point, the candidate point is a false hit and can be eliminated. Otherwise, the candidate point is an MTRNN of the given query point. We prove that the refinement step does not cause any false miss along with the description of the algorithm.

Fig. 3 presents the overall flow of the MTRNN algorithm and its preparatory, filtering and refinement steps. We will discuss these three steps in detail in the following sections. Because *feature route* is a crucial component in both filtering and refinement, in the following we first discuss the algorithms to find *feature routes* in the preparatory step.

3.1. Preparatory step: finding feature routes

A *feature route* plays an important role in the MTRNN algorithm. We first define *feature route* and related concepts and then describe our approach of finding the *feature routes*.

Definition 4. Multi-type route (MTR)

Given k different feature types, a *multi-type route* is a route that goes through one instance of every feature type. Assume there are four feature types F_1, F_2, F_3 and F_4. An MTR could be $R(f_{1,1},f_{2,1},f_{3,1},f_{4,1})$.

Definition 5. Feature route

Given k different feature types, a *feature route* is a multi-type route such that the distance from the fixed starting point through all other points in the MTR route is shortest.

Algorithm MTRNN(R-trees, R-tree, $f_{q,q}$, F_q)
Input : R-trees for each feature, R-tree index root of queried data set, query point $f_{q,q}$
the query feature type F_q
Output : MTRNN set S_c

```
1.  //1.Preparatory Step
2.  Partition Space and put subspace into set S_sub
3.  //Find initial greedy route
4.  S_frps = Greedy(R-trees, f_q,q, ∅)
5.  Find feature route set S_fr from S_frps
6.  S'_fr = FindFeatureRoutes(R-trees, f_q,q, S_fr, S_sub)
7.  S_fr = S'_fr ∪ § _fr
8.  //2.Filtering Step
9.  S_c = Filtering(R-trees,R-tree,S_fr,f_q,q,NIL)
10. //3.Refinement Step
11. S_c = Refinement(R-trees,R-tree,f_q,q,S_c,S_fr,F_q)
12. return S_c
```

Fig. 3. MTRNN algorithm.

From the MTR $R(f_{1,1},f_{2,1},f_{3,1},f_{4,1})$ illustrated above, we can get four *feature routes*. Fixing point $f_{1,1}$ and finding the shortest distance from point $f_{1,1}$ through the three other points we get one route. This route, assuming $R(f_{1,1},f_{4,1},f_{3,1},f_{2,1})$ starting from point $f_{1,1}$ with the shortest distance, is a *feature route*. Starting from each of other three points respectively and finding the route with shortest distance yields four *feature routes*.

Definition 6. I-distance

Given a *feature route*, the (shortest) distance of this route is called an *I-distance*.

Definition 7. Feature route point set

Given a *feature route*, a *feature route point set* consists of all points in the *feature route*.

A *feature route* can be identified by a given *feature route point set* and a fixed starting point. Given a *feature route point set* containing k points from k different feature types there are k *feature routes* and k corresponding *I-distances* starting from each point of the given *feature route point set*.

The position of a *feature route* on the search space will affect its filtering ability. Therefore, it is preferred that different *feature routes* be found for different subspaces of the entire search space. In our algorithms we divide the space into several subspaces by straight lines intersected at the query point with the same angles between two neighbor lines and find *feature routes* for each of these subspaces respectively.

Next, we show how to find initial *feature route*. Fig. 4 displays the pseudo-code for the greedy MTR finding procedure. This greedy algorithm uses a heuristic method by assuming a fixed order of feature types represented as $<F_q,F_1,\ldots,F_{q-1},F_{q+1},\ldots,F_k>$ among which F_i is feature type and greedily find the MTR. A greedy approach is necessary because finding a *feature route* using the MTNN algorithm is very time-consuming. A greedy approach is also sufficient because the route it finds is used only for pruning purposes. We find the greedy MTR by greedily finding a route for a specified ordered list of features starting from the given query point $f_{q,q}$. A greedy MTR route in terms of k feature types for the given query point $f_{q,q}$ of feature type F_q is found by finding in feature type F_q the nearest neighbor $f_{q,1'}$ of $f_{q,q}$ inside a subspace, and then in feature type F_1, finding the nearest neighbor $f_{1,1'}$ of the point $f_{q,1'}$. This procedure continues until all feature types have been visited. During this procedure, the R-tree index of each feature type is used in the nearest neighbor search algorithm based on work in [6]. All points on the greedy MTR route form a *feature route point set*. When using a greedy approach to find an MTR we should avoid generating the same MTR more than one time. This is done by making sure that the nearest neighbor of the starting point $f_{q,q}$ is not in the existing feature route set.

Generating one greedy MTR route, thus one *feature route point set*, for each of a few subspaces may not create large enough pruning regions. For an MTRNN query to generate pruning regions larger enough to filter as many as R-tree nodes and points as possible, it is required to generate enough *feature routes*. However, there is a tradeoff. The greater the number of *feature routes* the more expensive the filtering cost due to the greater number of pruning regions generated for each feature route. In our experiments, we show how many *feature routes* are enough for our filtering algorithm.

There are two approaches to generate the *feature routes*. The first is to generate m different *feature route point sets* by finding m NNs of the query point $f_{q,q}$ in feature type F_q and from each of these m NNs greedily find the MTRs to get m greedy MTR routes. This would likely enlarge the pruning regions, and thus increase the filtering ability and reduce the refining cost. The other approach to generate more *feature routes*, and thus larger pruning regions, is to partition the space into more subspaces. Because one greedy MTR is found for one subspace, more subspaces means that more greedy MTR are generated. So, more *feature routes* are then generated. Both approaches have similar effect to increase the filtering ability and reduce cost. In our experiments, we apply the second approach to generate more subspaces and thus *feature routes*.

Algorithm Greedy(R-trees, $f_{q,q}$, S_{fr})
Input : R-trees for each feature, query point $f_{q,q}$, existing feature route set S_{fr}
Output : A feature route point set S_{frps}

```
1.  q = f_{q,q}, S_frps = ∅
2.  For next feature in feature list with predefined order
3.      Remove head from the feature list
4.      Find NN of q in current feature
5.      //Avoid finding the same S_frps twice
6.      If q is f_{q,q} and NN is in S_fr
7.          return ∅
8.      put the NN into S_frps
9.      q = NN
10. return S_frps
```

Fig. 4. Find greedy MTR.

Algorithm FindFeatureRoutes(R-trees, $f_{q,q}$, S_{fr}, S_{sub})
Input : R-trees for each feature, query point $f_{q,q}$, existing feature route set S_{fr}, subspace set S_{sub}
Output : new feature route set S'_{fr}

```
1.  S'_fr = ∅
2.  For each subspace in S_sub
3.      S_frps = Greedy(R-trees, f_q,q, S_fr)
4.      For each point in S_frps
5.          Fix this point as starting point;
6.          Find I-distance and corresponding feature route;
7.          Put the feature route into feature route set S'_fr
8.  return S'_fr
```

Fig. 5. Find feature routes.

Fig. 5 describes the algorithm for finding *feature routes* and corresponding *I-distances*. Since an *I-distance* is shortest, the route that has length *I-distance* is a *Hamilton Path*. We use the existing *Hamilton Path-finding* algorithm to find the *I-distance*. As it is known, *Hamilton Path-finding* problem is NP-complete. For large number of feature types, it is very time-consuming. However, it is acceptable to use the existing algorithm to find exact shortest path for the *Hamilton Path-finding* in our case since our MTRNN algorithm is only used when the number of feature type is small due to its complexity. As will be discussed later, the shorter the *I-distance*, the better it is for filtering efficiency.

In the *feature route* finding algorithm described in Fig. 5, only the first point of a greedy MTR route is required to be inside a specified subspace because the *feature routes* generated this way could possibly be shorter. If we require all points in a *feature route point set* to be inside the same subspace, the *feature routes* may be longer, which will generate pruning regions with smaller size, thus decreasing the filtering ability. Although a *feature route* generated with this strategy may fall into different subspaces, this should not change its filtering ability much statistically, considering that part of any *feature route* could fall outside the specified subspace and all *feature routes* are used to generate pruning regions in an R-tree node filtering. It is worth noting that for every greedy MTR route, one *feature route point set* and *k* different *feature routes* are created.

To summarize the concepts of space partitioning and feature routes, Fig. 6 illustrates a specific space partitioning of six subspaces and some feature routes. Three lines l_1, l_2 and l_3 intersect at the query point $f_{q,q}$, and partition the space around it into six subspaces $S_1, S_2, \ldots, S_6$. For convenience, one of the lines is parallel to the *x* axis, and the angle between the lines is $360^\circ/6 = 60^\circ$. Please note that the space can be divided into any number of subspaces. Although this specific partitioning scheme with six subspaces in Fig. 6 is the same as in [7], the MTRNN algorithm does not use the property used in [7]. Starting from each subspace, a greedy MTR route is found. In the figure, sample *feature routes* of three feature types are given. In this example, not all points on one of the *feature routes* are inside the same subspace, because the *feature route* finding algorithm does not guarantee that all points will be inside the same subspace.

3.2. The filtering step: R-tree node level pruning

After finding *feature routes* we use two filtering approaches, closed region pruning and open region pruning to prune the search space. In both approaches, *feature routes* are used to generate pruning regions such that any point inside these regions cannot be MTRNNs, and thus can be filtered without causing any false miss. We begin with the discussion of closed region pruning.

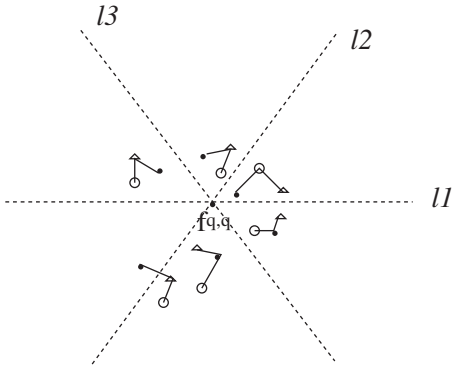


Fig. 6. Feature routes on the divided space.

3.2.1. Closed region pruning

Definition 8. Closed pruning region

A closed pruning region is a circle centered at the starting point $f_{i,j}$ in the *feature route* with radius = *MINDIST* from the query point $f_{q,q}$ to an R-tree node of the queried data − *I-distance* of the feature route starting at the point $f_{i,j}$.

Closed region pruning generates closed regions for each feature route and one R-tree node and uses these regions to prune the R-tree node. Fig. 7 illustrates three pruning scenarios. In the figure, $f_{q,q}$ is the query point and R_1, R_2 and R_3 are R-tree nodes that could be either leaf nodes or internal nodes in the R-tree index of the queried data. There are three points inside a *feature route point set* from three different feature types F_q, F_1 and F_2.

In Fig. 7, r_i is the length of a *feature route* and r_i is the length between the MINDIST from a query point to an R-tree node and r_i. In Fig. 7a the length of *feature route* $R(f_{2,1},f_{1,1},f_{q,1})$ is r_1 which is the *I-distance* starting from point $f_{2,1}$ through point $f_{1,1}$ and $f_{q,1}$. During the pruning procedure as shown in Fig. 7a, first the minimum distance MINDIST from the query point $f_{q,q}$ to the R-tree node R_1 has been calculated. Next the *I-distance* r_1 of the *feature route* $R(f_{2,1},f_{1,1},f_{q,1})$ was retrieved from the pre-calculated results in preparatory step. The MINDIST from $f_{q,q}$ to R_1 is r_1+r_1 because r_1 = *MINDIST* from $f_{q,q}$ to R_1-r_1. In the following, a circle C_1 is drawn, centered at the starting point $f_{2,1}$ of the *feature route* $R(f_{2,1},f_{1,1},f_{q,1})$. It can be seen that circle C_1 does not intersect the R-tree node R_1. Therefore, the length of the route from a point p_1 inside R_1 to the query point $f_{q,q}$ could be shorter than the distance from the same point through route $R(f_{2,1},f_{1,1},f_{q,1})$, i.e., $d(R(p_1,f_{q,q}))$ could be shorter than $d(R(p_1,f_{2,1},f_{1,1},f_{q,1}))$, which means that from a point inside R_1 it is possible to find an MTNN that contains the query point $f_{q,q}$. Thus, all points inside R_1 should be evaluated to find whether their MTNNs contain the query point $f_{q,q}$ or not.

In Fig. 7b, r_2 is the *I-distance* of route $R(f_{q,2},f_{1,2},f_{2,2})$. r_2 = *MINDIST* from $f_{q,q}$ to R_2-r_2 so the MINDIST from $f_{q,q}$ to R_2 is r_2+r_2. Similarly a circle C_2 centered as $f_{q,2}$ is drawn. At this time it covers the R-tree node R_2 completely. Therefore the length of the route from a point p_2 inside R_2 through $R(f_{q,2},f_{1,2},f_{2,2})$ could not be longer than the distance from the same point to the query point $f_{q,q}$, i.e., $d(R(p_2,f_{q,2},f_{1,2},f_{2,2}))<d(R(p_2,f_{q,q}))$, which means from any point inside R_2 we could not find an MTNN that contains the query point $f_{q,q}$. Thus, the entire node R_2 can be pruned.

In Fig. 7c, r_3 is the *I-distance* of route $R(f_{2,3},f_{1,3},f_{q,3})$. r_3 = *MINDIST* from $f_{q,q}$ to R_3-r_3 so the MINDIST from $f_{q,q}$ to R_3 is r_3+r_3. Another circle C_3 centered at $f_{2,3}$ is drawn and intersects the R-tree node R_3. Therefore the length of the route from a point p_3 inside the intersection of R_3 and circle C_3 through route $R(f_{2,3},f_{1,3},f_{q,3})$ could not be longer than the distance from the same point to the query point $f_{q,q}$, i.e., $d(R(p_3,f_{2,3},f_{1,3},f_{q,3}))<d(R(p_3,f_{q,q}))$. However, the route from a point p'_3 of R_3 outside the intersection through route $R(f_{2,3},f_{1,3},f_{q,3})$ could be longer than $d(R(p'_3,f_{q,q}))$. That means from any point inside the intersection we could not find an MTNN that contains the query point $f_{q,q}$ but from a point outside the intersection an MTNN could be found containing the query point. Thus, the part of page R_3 inside the circle C_3 could be pruned.

From the pruning procedure illustrated in Fig. 7, we know that the closed pruning region cannot contain any MTRNN. Thus, pruning the closed region won't cause any false miss. The following Lemma 1 formally proves this point.

Lemma 1. *For any point contained inside an R-tree node of the queried data set, if it is also contained inside a closed pruning region it cannot be an MTRNN, and thus can be pruned without causing any false miss.*

Proof. Assume p_i is contained in an R-tree node and inside a closed pruning region. Because p_i is contained in the R-tree node, the distance from p_i to the query point is longer than or equals the MINDIST from the query point to the R-tree node. Since the radius of the closed region circle equals the MINDIST from the query point to the R-tree node − the *I-distance* of the *feature route*, the distance from p_i that is inside the circle representing the closed pruning region to the starting point of the *feature route* + *I-distance* is shorter than MINDIST from the query point to the R-tree node. Therefore, the distance from p_i through the *feature route* is shorter than the distance from p_i to the query point. As we know that the distance of the MTNN route starting at p_i is shortest among all

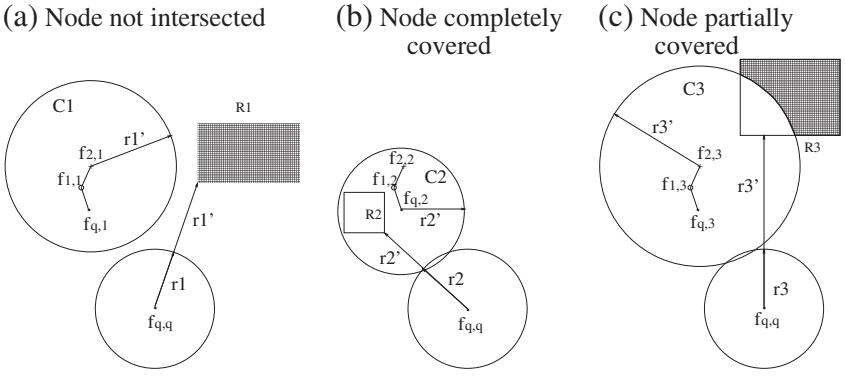


Fig. 7. Three pruning scenarios.

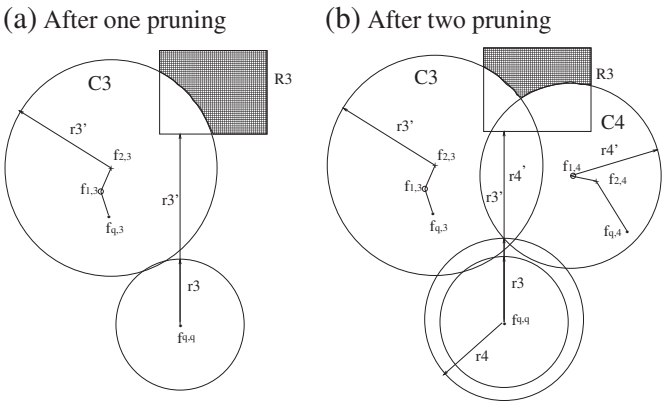


Fig. 8. Pruning one node.

routes starting from p_i through one instance of each of k feature types, any route from p_i through the query point cannot be the MTNN route, which means the MTNN route from p_i does not contain the query point. So p_i is not an MTRNN of the query point.

Please note that the set of points mentioned in Lemma 1 does not contain any point on the circle of the closed region circle.

Fig. 8 illustrates the pruning process on one R-tree node with two feature routes. Fig. 8a that is Fig. 7c shows the pruning result with feature route $R(f_{2,3},f_{1,3},f_{q,3})$. The intersected part of R_3 and circle C_3 has been pruned. However, the shaded part may still contain potential MTRNNs as discussed for scenario three in Fig. 7c. In Fig. 8b, another feature route $R(f_{1,4},f_{2,4},f_{q,4})$ joins in the pruning process. Similar to scenario three in Fig. 7c, the length of the circle C_4 centered at $f_{1,4}$ is the difference r_4 between the MINDIST from the query point $f_{q,q}$ to R_3 and the *I-distance* of feature route $R(f_{1,4},f_{2,4},f_{q,4})$. The closed pruning region represented by circle C_4 also prunes part of R-tree node R_3 as shown in Fig. 8b. For better understanding, we draw two circles centered at $f_{q,q}$ with radius r_3 and r_4 among which r_3 is the *I-distance* of feature route $R(f_{2,3},f_{1,3},f_{q,3})$ and r_4 is the *I-distance* of feature route $R(f_{1,4},f_{2,4},f_{q,4})$. The shaded part of R_3 in Fig. 8b contains potential MTRNNs. The other part of R_3 cannot contain any MTRNN so it can be pruned without causing any false miss.

3.2.1.1. The filtering ability of a feature route in the closed region pruning. As we noted earlier, the filtering ability of a feature route depends on its position relative to the position of the R-tree node to be pruned. The radius of the circle centered at a point on a *feature route* equals the MINDIST from the query point to an R-tree node — the *I-distance* of the feature route. If the MINDIST from the query point to the R-tree node is less than the *I-distance* of a *feature route*, this feature route will not be used to generate a closed pruning region for this R-tree node and it will not prune any data point inside this R-tree node. The more the circle covers an R-tree, the better the filtering ability. So, in order to increase the pruning ability of a *feature route*, it is necessary to calculate its *I-distance*, which is a *Hamilton Path* problem. Because the number of feature types is not big and the MTNN finding algorithm is complicated, it is worth calculating the *I-distance* and having it serve as the length of the *feature route*. After calculating the radius of the circle, both the circle's size and location are determined on the space. The closer the R-tree node is to the center of a circle, the higher the probability that the circle covers more of the R-tree, and thus the better the filtering ability. Therefore, we can incrementally add more *feature routes* that are close to an R-tree node into the *feature route set* and use them to filter the remaining part of the R-tree node and other R-tree nodes later.

3.2.2. Open region pruning

Our second pruning approach prunes an open region solely based on each *feature route*, thus pruning all points and R-tree nodes inside this region. This pruning approach is especially effective when pruning an R-tree node far away from the query point.

There are multiple ways to generate an open pruning region. A theoretic maximum open region that is generated from a *feature route* and can be pruned is represented as a half plane separated by a curve as illustrated in Fig. 9a. In the figure, $f_{q,q}(x_1,y_1)$ is the query point from feature F_q, $f_{q,1}(x_2,y_2)$ is any point from feature F_q, $f_{1,1}$ is a point from feature F_1 and $f_{2,1}$ is a point from feature F_2. The space has been divided by curve l into two planes. Plane(l,$f_{q,1}$) represents the plane separated by curve l and containing point $f_{q,1}$ and Plane(l,$f_{q,q}$) represents the plane separated by curve l and containing the query point $f_{q,q}$.

Next, we will define the Plane(l,$f_{q,1}$) such that the distance from any point p inside this plane to $f_{q,1}$ plus the *I-distance* of *feature route* $R(f_{q,1},f_{1,1},f_{2,1})$ is shorter than the distance from point p to the query point $f_{q,q}$. Therefore, it is impossible that the query point $f_{q,q}$ is on the MTNN of the point p, which means that p is not MTRNN of the query point $f_{q,q}$. Because point p is any point inside Plane (l,$f_{q,1}$), the whole Plane(l,$f_{q,1}$) can be pruned without incurring any false miss.

In the following, we describe how to find curve l so that it can divide the space into Plane(l,$f_{q,1}$) and Plane(l,$f_{q,q}$). In Fig. 9a curve l and straight line $f_{q,1}f_{q,q}$ intersect at point s_h. $R(f_{q,1},f_{1,1},f_{2,1})$ is a *feature route* and the distance from point s_h to starting point $f_{q,1}$ of the *feature route* $R(f_{q,1},f_{1,1},f_{2,1})$ plus the *I-distance* of this *feature route* equals the distance from point s_h to the query point $f_{q,q}$, i.e., $d(R(s_h,f_{q,1},f_{1,1},f_{2,1})) = d(R(s_h,f_{q,q}))$. In other words, point s_h divides the line segment $f_{q,1}f_{q,q}$ into two parts so that $d(R(s_h,f_{q,q})) - d(R$

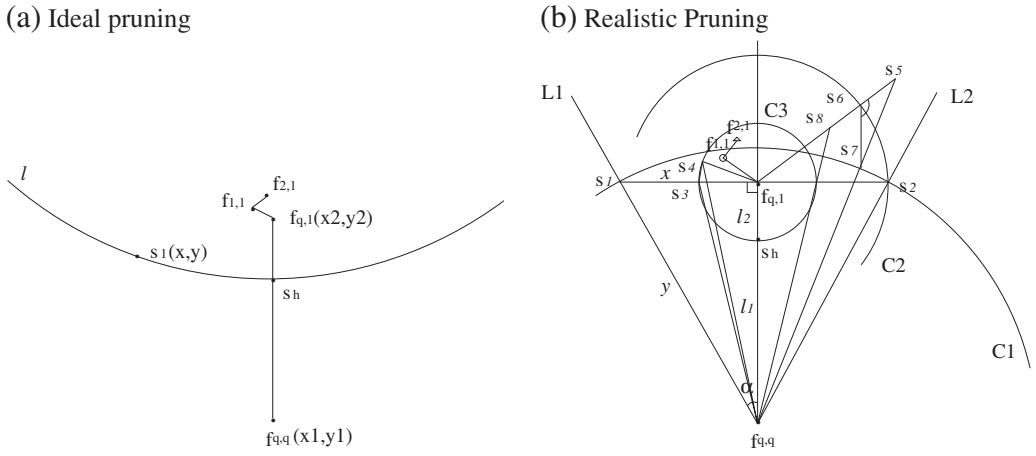


Fig. 9. Open region pruning.

$(s_h,f_{q,1}))=d(R(f_{q,1},f_{1,1},f_{2,1}))$. In the following discussion, we will use h to represent the *I-distance* of the *feature route* $R(f_{q,1},f_{1,1},f_{2,1})$ in Fig. 9a.

In order to guarantee that a point inside Plane$(l,f_{q,1})$ is not an MTRNN of the query point $f_{q,q}$, any point $p(x_p,y_p)$ inside Plane $(l,f_{q,1})$ should satisfy the equation $\sqrt{(x_p-x_1)^2+(y_p-y_1)^2}\geq\sqrt{(x_p-x_2)^2+(y_p-y_2)^2}+d(R(f_{q,1},f_{1,1},f_{2,1}))$ among which $d(R(f_{q,1},f_{1,1},f_{2,1}))=h$. Thus, a point $s_1(x,y)$ on curve l should satisfy the equation $\sqrt{(x-x_1)^2+(y-y_1)^2}=\sqrt{(x-x_2)^2+(y-y_2)^2}+h$, which can actually be transformed into a quartic (4-th degree) equation. Because positions of point $f_{q,q}(x_1,y_1)$ and $f_{q,1}(x_2,y_2)$ and *I-distance* h are known, the curve l is known and divides the space into two planes.

Although curve l in Fig. 9a can be used to maximally prune R-tree nodes and points, it is not easy to check on which side of the curve l an R-tree node or a point falls. From a practical point of view, a simple representation of an open pruning region should be used in order to prune points and R-tree nodes efficiently. To simplify the point check process, we propose a simpler open region pruning approach based on a simpler region description. In Fig. 9b, the *feature route* is $R(f_{q,1},f_{1,1},f_{2,1})$, starting at point $f_{q,1}$ with *I-distance* h. The point s_h divides the line segment $f_{q,q}f_{q,1}$ into two parts with length l_1 and l_2 such that $l_1-l_2=h$. Since the positions of points $f_{q,q}$ and $f_{q,1}$ and *I-distance* h are known, the position of s_h is known, thus the lengths of l_1 and l_2 being also known.

We construct the simple pruning region as follows. In Fig. 9b we draw a straight line s_1s_2 passing through point $f_{q,1}$ and perpendicular to line $f_{q,q}f_{q,1}$. We then draw line $f_{q,q}s_1$ as line L_1 and line $f_{q,q}s_2$ as line L_2. We take $y-x=h$ in which y is the length of line $f_{q,q}s_1$ and x is the length of line $f_{q,1}s_1$. Since $(l_1+l_2)^2+x^2=y^2$, we can calculate $x=\frac{2l_1l_2}{l_1-l_2}$ and $y=\frac{l_1^2+l_2^2}{l_1-l_2}$. Because l_1 and l_2 are known, x and y are known. Therefore, the positions s_1 and s_2 are also known.

In formulas $x=\frac{2l_1l_2}{l_1-l_2}$ and $y=\frac{l_1^2+l_2^2}{l_1-l_2}$, l_2 is between 0 and $\frac{len(f_{q,1}f_{q,q})}{2}$. When l_2 is 0, which means the *I-distance* of the *feature route* $R(f_{q,1},f_{1,1},f_{2,1})$ is equal to or longer than len$(f_{q,1}f_{q,q})$, no point can be pruned by using the open region generated based on this *feature route*. When l_2 is $\frac{len(f_{q,1}f_{q,q})}{2}$, which means the *I-distance* h of this *feature route* is 0, the MTRNN problem reduces to the classic RNN problem for this *feature route*. Then the perpendicular bisector $\perp(f_{q,1},f_{q,q})$ divides the data space into two half planes: one that contains $f_{q,1}$ (Plane$(\perp(f_{q,1},f_{q,q}),f_{q,1})$), and one that contains $f_{q,q}$ (Plane$(\perp(f_{q,1},f_{q,q}),f_{q,q})$). No point in Plane$(\perp(f_{q,1},f_{q,q}),f_{q,1})$ can be an RNN or an MTRNN of $f_{q,q}$ and thus all R-tree nodes and points in Plane$(\perp(f_{q,1},f_{q,q}),f_{q,1})$ can be pruned.

Next we prove that all points in the open pruning region formed by lines L_1 and L_2 excluding the triangle $f_{q,q}s_1s_2$ in Fig. 9b can be pruned. That is, the distance from any point inside this region to the query point $f_{q,q}$ is longer than the distance of the point to point $f_{q,1}$ plus the *I-distance* h of the feature route $R(f_{q,1},f_{1,1},f_{2,1})$ starting at point $f_{q,1}$.

Lemma 2. *No point inside an open pruning region defined by a feature route and the query point can be an MTRNN, and thus can be pruned.*

Proof. The open pruning region containing point $f_{q,1}$ is formed by lines L_1 and L_2, excluding triangle $f_{q,q}s_1s_2$. It is divided into three parts as shown in Fig. 9b. The first part (part 1) is the open pruning region outside circle C_2. The second part (part 2) is the intersection of circle C_2 and open pruning region, excluding the circle C_1. The third part (part 3) is the intersection of circle C_1 and the open pruning region. We prove the lemma by demonstrating that any point in any part of the open pruning region can be pruned without causing any false miss.

In Fig. 9b, s_1 and s_2 are positioned on the circle C_1 centered at $f_{q,q}$ with radius y or $len(f_{q,q}s_1)$ and also on the circle C_2 centered at $f_{q,1}$ with radius x or $len(f_{q,1}s_1)$. The radius of the smallest circle C_3 centered at $f_{q,1}$ is len$(f_{q,1},s_4)$ in which s_4 is any point inside part 3.

First, assume a point s_5 in part 1 is outside of the circle C_2. We need to prove $d(R(s_5,f_{q,1},f_{1,1},f_{2,1}))<d(R(s_5,f_{q,q}))$. From the figure, it can be seen that $d(R(s_5,f_{q,1},f_{1,1},f_{2,1}))=d(R(s_5,s_6))+d(R(s_6,f_{q,1}))+d(R(f_{q,1},f_{1,1},f_{2,1}))$ and $d(R(s_5,f_{q,q})=d(R(s_5,s_7)+d(R(s_7,f_{q,q}))$. Since len

$(f_{q,q}s_7) = \text{len}(f_{q,q}s_2)$, $\text{len}(f_{q,1}s_6) = \text{len}(f_{q,1}s_2)$ and $\text{len}(f_{q,q}s_2) - \text{len}(f_{q,1}s_6) = h$, $\text{len}(f_{q,q}s_7) - \text{len}(f_{q,1}s_6) = d(R(f_{q,1},f_{1,1},f_{2,1}))$. Therefore, we only need to prove $\text{len}(s_5s_7) \geq \text{len}(s_5s_6)$. It is easy to prove angle $\angle s_7s_6s_5 \geq 90^\circ$ so $\text{len}(s_5s_7) \geq \text{len}(s_5s_6)$. So, s_5 can be pruned.

When a point s_8 in part 2 is inside the circle C_2 but outside the circle C_1, $f_{q,q}s_8$ is longer than $f_{q,1}s_2$ but $f_{q,1}s_8$ is shorter than $f_{q,1}s_2$. Therefore $\text{d}(\text{R}(s_8,f_{q,1}f_{1,1},f_{2,1})) < \text{d}(\text{R}(s_8,f_{q,q}))$ and the point s_8 can be pruned.

Finally, assume a point s_4 in part 3 is inside the circle C_1. Since $\angle s_3s_4f_{q,1} > \angle s_3s_4f_{q,q}$ and $\angle s_4s_3f_{q,1} = \angle s_3s_4f_{q,1}$ we have $\angle s_4s_3f_{q,1} > \angle s_3s_4f_{q,q}$. Thus, $\angle s_4s_3f_{q,q} > \angle s_3s_4f_{q,q}$. Therefore we derive $\text{len}(s_4f_{q,q}) > \text{len}(s_3f_{q,q})$. Since we know $\text{len}(s_3f_{q,q}) - \text{len}(s_3f_{q,1}) > h$, then $\text{len}(s_4f_{q,q}) - \text{len}(s_4f_{q,1}) > h$. So, the distance from point s_4 to $f_{q,1}$ plus *I-distance* starting at $f_{q,1}$ is shorter than the length of $s_4f_{q,q}$, i.e., $\text{d}(\text{R}(s_4,f_{q,1}f_{1,1},f_{2,1})) < \text{d}(\text{R}(s_4,f_{q,q}))$. Therefore, point s_4 can be pruned.

Since Lemma 2 proves that no point inside an open region can be an MTRNN, pruning the whole open region won't introduce any false miss.

Fig. 10 presents the pseudo-code for the algorithm applying both closed region and open region pruning. If an R-tree node is entirely contained inside pruning regions, the R-tree node can be filtered "safely" and the output set is empty, meaning the MTNN from a point in the R-tree node cannot contain the given query point and can safely be filtered without causing a false miss. Otherwise, if the R-tree node is an internal node output the R-tree node itself. The filtering algorithm will then visit all child nodes of this internal R-tree node later. If the R-tree node is a leaf node and not all of it is covered by pruning regions, use the center point of the node as a query point to find in specified subspaces a greedy MTR route whose points form a new *feature route point set*. Next, find *feature routes* and use these new *feature routes* to generate new pruning regions and try to prune the R-tree node. Because a center point of an R-tree node can be anywhere either close to or far away from the query point, we only find new greedy MTR routes whose starting point is inside the subspace containing the center point in order to avoid generating too many *feature routes*.

3.2.3. Filtering algorithm

The complete filtering step should search all candidate points in the data set against which the query is issued. Since R-trees are widely used in spatial databases, we assume an R-tree index is available for each feature type and the queried data set. Our filtering algorithm utilizes R-tree indexes to generate closed and open pruning regions that are used to filter as many as R-tree nodes and points in the queried data set. The example in Fig. 11 illustrates how the filtering algorithm works.

In this example, the R-tree nodes at the first level contain R_1, R_2, R_3 and R_4. Fig. 11a gives the R-tree index of the queried data set. The query point $f_{q,q}$ is of feature type F_q. For simplicity, we don't draw R-tree nodes of feature data sets and only illustrate the filtering process in two subspaces sub_1 and sub_2.

Initially, as shown in Fig. 11b we use 2 NN strategy and find in feature type F_q two nearest neighbors $f_{q,1}$ and $f_{q,2}$ of the query point $f_{q,q}$ inside the subspace sub_1. Then we find nearest neighbor $f_{1,1}$ of $f_{q,1}$ and nearest neighbor $f_{1,2}$ of $f_{q,2}$ in feature type F_1. Finally we find nearest neighbor $f_{2,1}$ of $f_{1,1}$ and nearest neighbor $f_{2,2}$ of $f_{1,2}$ in feature type F_2. So far, we get two *feature route point sets* $\{f_{q,1}f_{1,1}f_{2,1}\}$ and $\{f_{q,2}f_{1,2}f_{2,2}\}$. For each *feature route point set*, we calculate the *I-distances* of the *feature routes* starting from each point in the *feature route point set*. For example, the *I-distance* of the *feature route* starting at point $f_{q,1}$ in the *feature route point set* $\{f_{q,1}f_{1,1}f_{2,1}\}$ is $d(R(f_{q,1},f_{1,1},f_{2,1}))$. Similarly we find *feature route point sets* $\{f_{q,4}f_{1,3}f_{2,3}\}$ and $\{f_{q,5}f_{1,3}f_{2,3}\}$ in subspace sub_2. Because either *I-distance* of any feature route from sets $\{f_{q,4}f_{1,3}f_{2,3}\}$ and $\{f_{q,5}f_{1,3}f_{2,3}\}$ is longer than MINDIST from $f_{q,q}$ to an R-tree

Algorithm OneNodePrune(R-trees,R-tree,S_{fr},$f_{q,q}$)
Input : R-trees for all feature types, an R-tree node to be pruned, *feature route* set S_{fr},
query point $f_{q,q}$
Output : Empty set, R-tree node or candidate MTRNNs

```
1.  Calculate mindist = the MINDIST from the query point f_{q,q} to R-tree node;
2.  S'_fr = S_fr, NoCenterCreated = true;
3.  For each feature route in S'_fr
4.      Calculate the difference r between mindist and I-distance of the feature route;
5.      Form a closed pruning region;
6.      Form an open pruning region
7.  If R-tree node is entirely contained inside pruning regions
8.  Then return empty set
9.  Else If R-tree node is an internal node
10.     Then return R-tree node
11.     Else If NoCenterCreated
12.         //Prune same R-tree node with newly generated feature route sets
13.         Then NoCenterCreated = false;
14.             Find center s_c of the R-tree node and put subspace containing s_c into S_sub;
15.             S'_fr = FindFeatureRoutes(Rtrees,s_c,S_fr,S_sub);
16.             S_fr = S'_fr ∪ S_fr;
17.             Goto 3
18.         Else return All points inside the R-tree node but outside all the pruning regions
```

Fig. 10. One node pruning algorithm.

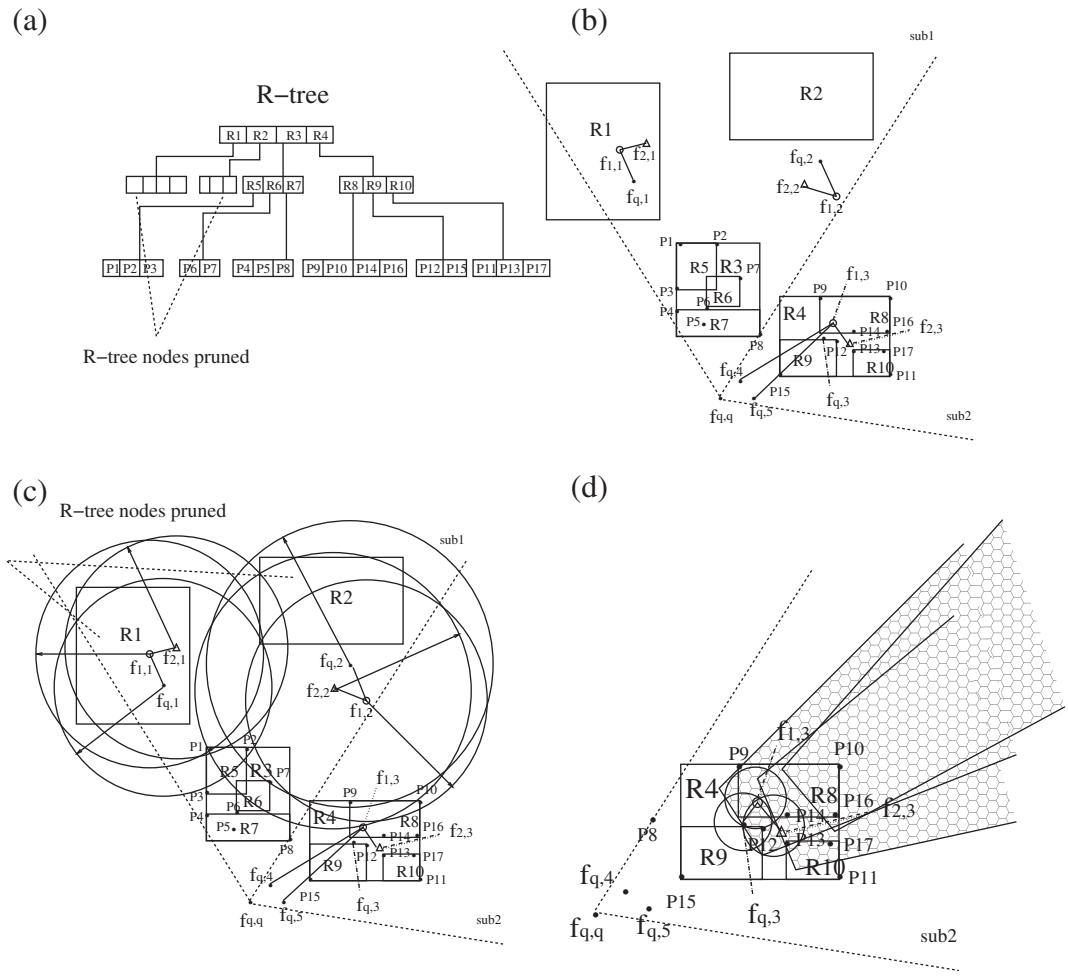


Fig. 11. A filtering example.

node (R_3 or R_4) or R-tree nodes (R_1 and R_2) have already been pruned by other pruning regions, sets $\{f_{q,4}f_{1,3}f_{2,3}\}$ and $\{f_{q,5}f_{1,3}f_{2,3}\}$ are not used to prune any R-tree node. For simplicity we ignore them and do not draw pruning regions generated from them.

The next step illustrated in Fig. 11c shows the closed region pruning with *feature routes* found so far. It starts with calculating the MINDIST from the query point $f_{q,q}$ and R-tree node R_1. Following this step, we calculate the difference between the MINDIST from the query point $f_{q,q}$ and R-tree node R_1 and the *I-distance* of $R(f_{q,1},f_{1,1},f_{2,1})$ and draw a circle centered at $f_{q,1}$ with this difference. We repeat these steps for all points in the *feature point route set* $\{f_{q,1},f_{1,1},f_{2,1}\}$ and get three *feature routes* and circles. The R-tree node R_1 is completely covered by these circles so it cannot contain any MTRNN and can be pruned. Similarly R-tree node R_2 can be pruned completely. However, R-tree node R_3 is only partially covered by the circles. Since from the center point of R_3 no new *feature route* set is found in the subspace sub_1, we don't try to prune R-tree node R_3 again. Thus, it traverses down node R_3 to visit R_5, R_6 and R_7. At R_5, points p_1 and p_2 are inside the circles and can be pruned but point p_3 is left as a potential MTRNN. Similarly point p_6 in node R_6 and points p_4, p_5 and p_8 in node R_7 are potential MTRNNs. Since R-tree nodes R_1 and R_2 were pruned earlier and R-tree node R_3 is not pruned by the open pruning regions, the open pruning regions generated from feature point set $\{f_{q,1},f_{1,1},f_{2,1}\}$ and $\{f_{q,2},f_{1,2},f_{2,2}\}$ have not been drawn.

In this example, R-tree node R_4 is not entirely pruned by all existing closed and open pruning regions (not shown in this figure for simplicity) and only points p_9 and p_{10} in node R_8 were pruned. Fig. 11d gives the example about how to prune points inside R-tree node R_4 of region sub_2 by using closed and open region pruning techniques generated from a new query point. For simplicity, only subspace sub_2 is shown in Fig. 11d. We take the center of node R_4 as the new query point and find a greedy route $R(f_{q,3},f_{1,3},f_{2,3})$ from it. As before, three circles are drawn. Point p_{14} in node R_8, point p_{12} in node R_9 and point p_{13} in node R_{10} are then pruned by the new circles. So far, the only potential MTRNNs are point p_{15} in node R_9, point p_{16} in R_8 and points p_{11} and p_{17} in node R_{10}. Now we apply the open region pruning approach. The open pruning region is the region filled with hexagons. At this time, points p_{16} in R_8 and p_{17} in node R_{10} fall into the open pruning regions so they are pruned. Finally only point p_{15} in node R_9 and point p_{11} in node R_{10} are left as candidate MTRNNs.

Fig. 12 gives the pseudo-code of the filtering algorithm. For an internal R-tree node, the Filtering function is called twice. At the first call, center point p_c is empty and the existing *feature route* set is used to prune the R-tree node. At the second call, center point

Algorithm Filtering(R-trees,R-tree,S_{fr},$f_{q,q}$,p_c)
Input : R-trees for all feature types, R-tree for data set being queried,
Feature Routes S_{fr}, Query Point $f_{q,q}$, center point p_c
Output : Empty set, R-tree node or candidate MTRNN set S_c

```
1.  R-tree = OneNodePrune(R-trees,R-tree,S_fr,f_q,q)
2.  If R-tree is empty set
3.  Then return empty set
4.  Else If R-tree is an internal node
5.      Then If p_c is not NIL
6.          Then return R-tree
7.          Else Find center p_c of R-tree node and put subspace containing p_c into S_sub
8.              S'_fr = FindFeatureRoute(R-trees,p_c,S_fr,S_sub);
9.              S_fr = S'_fr ∪ S_fr;
10.             //Prune R-tree node with new feature route set
11.             R-tree = Filtering(R-trees,R-tree,S'_fr,f_q,q,p_c)
12.             If R-tree is empty set
13.             Then Return empty set
14.             //Prune child nodes of the R-tree node
15.             For each child node of R-tree
16.                 Add Filtering(R-trees,child node,S_fr,f_q,q) into PointSet
17.     Else Add R-tree into PointSet
18. return PointSet
```

Fig. 12. Filtering algorithm.

p_c is found and the newly generated *feature route* set is used to prune the R-tree node. If the R-tree node still cannot be pruned completely, each child node is pruned with all *feature routes* including the newly generated ones. After filtering, most of the points in the queried data set are safely pruned without causing any false miss and a candidate point set S_c containing all potential MTRNNs has been generated.

3.3. Refinement step: removing false hit points

The refinement step further eliminates points in the MTRNN candidate set S_c so that only qualified MTRNNs will remain. Three refinement approaches are applied to guarantee all false hits will be eliminated.

Fig. 13 shows the pseudo-code for the complete refinement step.

The first approach uses existing *feature routes* to eliminate false hits from candidate MTRNNs. After filtering, we have a set S_c of candidate MTRNN points and a set S_{fr} of *feature routes*. Since the *I-distances* of *feature routes* have already been calculated, they can be directly used to eliminate false hits that cannot be real MTRNNs. If the minimum distance *mindist*, distance from an MTRNN candidate point p to the starting point of one *feature route* plus the *I-distance* of the *feature route*, is shorter than the distance d_1 from p to the query point $f_{q,q}$, which means the MTNN distance from the point p is shorter than the distance d_1 and the MTNN of point p cannot contain the query point $f_{q,q}$, the point p cannot be an MTRNN of the query point $f_{q,q}$ and can be pruned. In other words, this approach does not introduce any false miss.

In the second approach, a greedy MTR route and corresponding *feature route point set* are calculated if an MTRNN candidate point p cannot be pruned by using the first approach. Note that query point $f_{q,q}$ is considered as a point in the data set of feature type F_q when finding this greedy MTR route. From the new *feature route point set*, a new set of *feature routes* can be found. If the minimum distance *mindist*, distance from an MTRNN candidate point p to the starting point of one new *feature route* plus the *I-distance* of the new *feature route*, is shorter than d_1, this point could be pruned. Similar to the first approach, the second approach does not cause any false miss. Since it is possible there are some points close to p in the candidate set, this newly found *feature route point set* could be useful to prune these points (and other points); thus the new point set from this greedy MTR route is added into the set of *feature route point set* S_{frps} and all *I-distances* for all *feature routes* from this *feature route point set* are saved for future pruning.

If an MTRNN candidate point p cannot be pruned by the first two approaches, an MTNN algorithm that utilizes R-tree index of each feature type is applied to calculate the real MTNN for this point p. As in the second approach, query point $f_{q,q}$ is considered as a point in the dataset of feature type F_q. After finding MTNN of the point p, we get a set of MTNN points and a corresponding MTNN route. If query point $f_{q,q}$ is in the MTNN of the point p, then point p is an MTRNN of this query point. Otherwise, p is eliminated from S_c. This approach does not cause any false miss. From the MTNN algorithm in [2] we know that the MTNN route from the point p is shortest among all possible routes from p going through each point from each different feature types. If the query point $f_{q,q}$ is on this MTNN route or, in other words, in the point set of MTNN, this point p is an MTRNN of the query point $f_{q,q}$ according to the problem definition formalized in Section 2. Thus, this third approach does not introduce false miss.

Fig. 14 shows the pseudo-code of the MTNN algorithm, which is adapted from the algorithm described in [5] and [2]. The initial greedy distance *dis* is the minimum distance of all routes from point p through all *feature routes*. The major adaptation occurs during partial route growing to the feature type that the query point $f_{q,q}$ belongs to. If none of the current partial routes for a specific permutation contains the query point $f_{q,q}$, it is safe to stop searching for this permutation. Another enhancement is to mark the

Algorithm Refinement(R-trees,R-tree,$f_{q,q}$,S_c,S_{fr},F_q)
Input : R-trees for each feature type, R-tree for data set being queried, Query Point $f_{q,q}$, a candidate MTRNN set S_c, the *feature route* set S_{fr}, the query feature type F_q
Output : MTRNN set S_c

```
1.  mindist = ∞
2.  For each point p in set S_c
3.      Calculate distance d_1 from point p to the query point f_{q,q};
4.      For each feature route fr in S_fr
5.          Calculate distance d_2 from point p to the starting point of feature route fr;
6.          if mindist > d_2+ I-distance of fr
7.              mindist = d_2+ I-distance of fr
8.              If mindist < d_1
9.                  Eliminate point p from set S_c;
10.                 goto 1
11.     S_frps = Greedy(R-trees, p, S_fr);
12.     Calculate I-distances for all feature routes S'_fr starting from all points in S_frps;
13.     For each feature route fr' in S'_fr
14.         Calculate distance d_3 from point p to the starting point of feature route fr';
15.         if mindist > d_3+ I-distance of fr'
16.             mindist = d_3+ I-distance of fr'
17.             If mindist < d_1
18.                 Eliminate point p from set S_c;
19.                 Put the new feature routes S'_fr into S_fr;
20.                 goto 1
21.     mtnn = MTNN(R-trees, R-tree, mindist, p, f_{q,q}, F_q);
22.     If f_{q,q} is not in mtnn
23.         Eliminate point p from set S_c;
24.         Calculate I-distances for all feature routes starting from all points in mtnn;
25.         Put the new feature routes into S_fr;
26. return MTRNN set S_c
```

Fig. 13. Refinement algorithm.

Algorithm MTNN(R-trees,R-tree,dis,q,$f_{q,q}$,F_q)
Input : R-trees for each feature type, R-tree index root of queried data set, potential MTRNN point q, distance dis, Query Point $f_{q,q}$, feature type F_q of query point
Output : MTNN

```
1.  MTNN = ∅
2.  Prune all R-tree nodes that are not intersected by the circle centered at q with radius dis
3.  For each permutation of all features
4.  //For simplicity assume permuation is (1, 2, ..., k)
5.      dis_1 = dis
6.      CurFT = k
7.      For each point p in data set of feature type CurFT
8.          If (d(R(p, q)) < dis_1)
9.              Put R(p) into partial route set S
10.     For i = k − 1 to 1
11.         If CurFT is F_q and f_{q,q} is not in S
12.             return empty
13.         CurFT = i
14.         For each point p' in CurFT
15.             Grow each partial route of S by adding p' to the head
16.             if(Length of new partial route +d(R(p', q)) < dis1)
17.                 put new partial route into partial route set S_1;
18.             Put the partial route with shortest length in S_1 into partial route set S_2;
19.         S = S_2;
20.         Find route in S with shortest distance dis_2;
21.         dis_1 = dis − dis_2
22.     dis =the distance of current shortest route
23. Find MTNN in route of S with shortest length
24. return MTNN
```

Fig. 14. Adapted MTNN algorithm.

partial route ending with query point $f_{q,q}$ after growing the partial route to the feature type. Later, if this marked partial route is not used to grow any further partial routes, the searching for this permutation can be safely stopped. After all features in a permutation are visited, a potential MTNN is generated. After all permutations of all feature types are searched, a real MTNN is generated.

It is worth discussing when the filtering and refinement algorithms fail pruning any point thus work as the naive baseline algorithm. Normally when all queried data points are far away from the query point $f_{q,q}$ and all feature data points that are clustered, our filtering and refinement algorithms may fail or have limited ability to prune points. It means that it is likely the distance from a queried point p to the query point $f_{q,q}$ is longer than the distance of a *feature route* plus the distance from the start point of the *feature route* to point p. If this happens, point p cannot be pruned.

4. Complexity analysis

In this section we study the complexity of the baseline algorithm and our proposed MTRNN algorithm. Our analysis is based on the cost model for nearest neighbor search in low and medium dimensional spaces devised in the work [8] by Tao et al. In the following we compute the expected time complexity in terms of distance calculations required to answer an MTRNN query.

Assume that the points of a queried data set and each feature data set are uniformly distributed in a unit square universe. The number of queried data points is N and each feature data set contains M data points. Similarly to [5], we derive formulas for the following distances

1. The expected distance δ between any pair of points each from a different feature
2. The expected *feature route* distance E_{fr}

Because the cardinality of a feature data set is M and data are uniformly distributed in the unit square universe, it is expected to have $\sqrt{M}$ points along a direction of x or y axis. For two data sets from two different feature types, there are expected $\sqrt{2M}$ points. Because we assume the data are in the unit square universe, the expected distance between any pair of points each from different features is $\delta = \frac{1}{\sqrt{2M}}$.

If there are k points each from a different feature type in a *feature route*, the expected *feature route* distance of the *feature route* is $E_{fr} = (k-1)\delta = \frac{k-1}{\sqrt{2M}}$.

4.1. Cost of baseline algorithm

Since we use the algorithm R-LORD in [5] to find MTRNN for one permutation of feature types, for example, $<F_1, F_2, \ldots, F_k>$, the cost of the baseline algorithm on one queried data point is just the sum of the R-LORD algorithm cost for all permutations.

Cost of R-LORD algorithm consists of two components, the distance calculation of R-tree node access and distance calculation of point search within the range of each iteration. In the following, we discuss these two components derived by Sharifzadeh et al. [5].

As stated in [5] "for each accessed node, R-LORD performs an *O(1) MINDIST* computation. Therefore, the complexity of each R-tree traversal is the same as the number of node accesses during the traversal." Thus, the expected number of R-tree nodes, *NA*, is used to represent the distance calculation of R-tree node access. Following the cost mode proposed in [8], the expected number of node accesses is given as

$$NA = \sum_{i=1}^{h-1} \left(n_i \times P_{NA_i} \right) \tag{1}$$

In this formula, h is the height of R-tree, P_{NA_i} is the probability of accessing a node at level i, and n_i is the total number of nodes at level i. Given total number of points in a data set, the capacity of R-tree node and the average fan-out of R-tree node, h and n_i can be easily derived [8]. For the P_{NA_i} estimation, it is needed to identify the search region. As derived in [5], the expected ranges for iteration 1 is $k \times \delta$ and for all other following iterations are $(k - i + 2) \times \delta$. Therefore, P_{NA_i} can be easily derived [8].

As LORD algorithm, R-LORD performs the same set of distance calculation for the point search for the chosen point set, so the second part of the R-LORD cost is $C_{lord} - kM$ in [5]. kM is removed from the C_{lord} because the algorithm of finding whether points are within the range in LORD is replaced by R-tree node pruning in R-LORD.

The components that should be considered when deriving cost formula for C_{lord} are the expected number of partial routes, the current search range T_v (*dis1* in Fig. 14) and the expected number of points $\pi T_v^2 M$ [8] in a feature type that are closer to the starting point than current search range T_v and will be examined in an iteration.

For the initialization step, the current search range T_v is the length of greedy route $T_c = k\delta$ (*dis* in Fig. 14), the expected number of partial routes is $\pi k^2 \delta^2 M$ and the expected number of points to be examined is M. For the first iteration, the current search range T_v decreases to $(k-1)\delta$, the expected number of partial routes is updated to $\pi(k-1)\delta^2 M$ and the expected number of points to be examined is $\pi(k-1)^2\delta^2 M$. For each iteration, all the parameters are derived and summarized in Table 4 in work [5]. Finally the cost formula of LORD is derived as follows

$$C_{lord} = O\left(kM + k^5\right) \tag{2}$$

Therefore, the expected cost of R-LORD can be given as

$$C_{rlord} = \sum_{i=1}^{k} NA(i) + (C_{lord} - kM) = \sum_{i=1}^{k} NA(i) + O\left(k^5\right) \tag{3}$$

where $NA(i)$ is distance calculation of R-tree node access, that is, the expected number of nodes, accessed in iteration i in R-LORD algorithm [5].

Since the expected length of T_c [5] used in the R-LORD algorithm is the same for all permutations under our assumption and the total number of permutations is $k!$, the cost for one queried point is:

$$C_{1-point} = k! \times \left(\sum_{i=1}^{k} NA(i) + O\left(k^5\right)\right) \tag{4}$$

Therefore, the total cost of the baseline algorithm for all queried points is:

$$C_{baseline} = k! \times O(N) \times \left(\sum_{i=1}^{k} NA(i) + O\left(k^5\right)\right) \tag{5}$$

4.2. Cost of MTRNN algorithm

The efficiency of the MTRNN algorithm is primarily based on the filtering ratio, i.e., the number of candidate MTRNNs after filtering. A good filtering algorithm should dramatically reduce the number of candidate MTRNN points and thus the overall cost of the algorithm. On the other hand, the filtering cost is immaterial when the number of feature types increases, which means the cost of the refinement step is the dominant cost in the MTRNN algorithm. Therefore, we analyze the cost of the refinement step and use it as the total cost of the MTRNN algorithm.

We assume that the *feature routes* are distributed uniformly in every direction from the query point $f_{q,q}$. In order to find the filtering ratio, we should calculate the area of the closed and open pruning regions and then derive the expected number of candidate MTRNN points that fall outside the pruning regions, based on the assumption of uniform data distribution.

For both of the closed and open region pruning approaches discussed in Section 3, a *feature route* starting inside a circle, say C_1, with radius r_1 of length E_{fr}, doesn't have any pruning ability. In other words, no queried data points inside circle C_1 whose radius is r_1 of length E_{fr} can be pruned, so these points are included as the minimum set of points in candidate MTRNN set S_c. Our filtering algorithm cannot prune any point in this minimum set.

Assume that the number of *feature route* is l and that these *feature routes* start outside circle C_1 but inside another circle, say C_2, with radius r_2. The area outside C_1 but inside C_2 is $\pi r_2^2 - \pi r_1^2$. Since all points from all feature types inside this region are expected to be starting points of *feature routes* we have $\frac{1}{\pi r_2^2 - \pi r_1^2} = \frac{kM}{l}$, which gives $r_2 = \sqrt{\frac{2l + \pi k(k-1)^2}{2\pi kM}}$. Assume that the expected distance from the query point $f_{q,q}$ to the starting point of a *feature route* is r. We have $\pi r_2^2 - \pi r^2 = \pi r^2 - \pi r_1^2$ so the expected distance $r = \sqrt{\frac{r_2^2 + r_1^2}{2}} = \sqrt{\frac{l + \pi k(k-1)^2}{2\pi kM}}$.

Next we discuss the area covered by an open pruning region. We first calculate the area of triangle $f_{q,q}S_1S_2$ in Fig. 9b. In the figure, $f_{q,q}F_{q,1}$ is just the expected distance r from the query point $f_{q,q}$ to the starting point of a feature route, so $h = r - r_1 = \frac{\sqrt{2\left(l + \pi(k-1)^2\right)} - \sqrt{\pi(k-1)^2}}{\sqrt{2\pi kM}}$. From Fig. 9b, we have $y - x = h$ and $r^2 + x^2 = y^2$ so $x = \frac{r^2 - h^2}{2h}$. Since r and h are known values, x is also known.

Therefore, inside a region formed by $f_{q,q}L_1$ and $f_{q,q}L_2$, the area of the triangle region that is not covered by this open pruning region is xr. We assume that there are enough *feature routes* such that the covered areas of the open pruning regions touch each other. Therefore, the regions not covered by open pruning regions are of area xrl.

For one *feature route*, a closed region can cover a region with expected area $\pi(r - r_1)^2$. However, half of the region was previously covered by an open pruning region, so one closed region covering a region that was not covered by the pruning regions has area $\frac{\pi(r-r_1)^2}{2}$. Therefore, the total area covered by all the closed pruning regions is $\frac{\pi(r-r_1)^2}{2} l$ for l *feature route*.

So far, we can calculate that the total area that was not covered by open and closed regions is $A_{NL} = xrl - \frac{\pi(r-r_1)^2}{2} l$. This area is the lower bound of the area that was not covered when we assume the open pruning regions are touching. As more and more *feature routes* are added, the open pruning regions will overlap each other and closed pruning regions will cover more area outside circle C_1. Finally only a region with area $A_{NU} = \pi r_1^2$ is not covered. This is the upper bound of the area that was not covered.

When the open regions are not touching each other, the regions that can be pruned are just the sum of all individual open and closed pruning regions. We ignore the formula for this situation.

After deriving the area of regions A_N that were not covered we can easily derive the number of points N_{sc} in the candidate MTRNN set S_c by applying formula $\frac{1}{A_N} = \frac{N}{N_{sc}}$. So, the $N_{sc} = A_{NU}N$ for the upper bound and $N_{sc} = A_{NL}N$ for the lower bound and the total computation for MTRNN algorithm is $N_{sc}C_{rlord}$. Since the number of *feature route* does not increase with the number of data points in different features and queried data set, it is considered as constant in the complexity analysis and ignored in the cost model. Our experiment results in Section 5.3.2 and in Section 5.3.6 also confirm that this feature route number is constant. Because all the components for the derivation of the asymptotic upper bound are given above, we ignore details for simplicity. It is easy to derive that the MTRNN algorithm cost for both lower bound covered area and upper bound covered area in terms of asymptotic upper bound is

$$C_{MTRNN} = O\left(\frac{NK^2}{M}\right) \times C_{1-point} = k! \times O\left(\frac{NK^2}{M}\right) \times \left(\sum_{i=1}^{k} NA(i) + O\left(k^5\right)\right) \tag{6}$$

Although C_{MTRNN} is a factorial function in terms of k, $k!$ is not very big when k is not big, which is the case that MTRNN algorithm applies to.

5. Experimental evaluations

We had two overall goals for our experiments: 1) to evaluate the performance and scalability of the MTRNN algorithm for the MTRNN query and 2) to evaluate the impact of our multi-type approach compared to traditional RNN query methods in terms of number and identity of RNNs returned.

5.1. Settings

5.1.1. Data sets description

We evaluated the performance of the MTRNN algorithm with both synthetic and real datasets.

- Synthetic data sets: all synthetically generated data points were distributed over a 1000×1000 plane. To evaluate the effects of spatial distribution, one queried data set was generated with random distribution (denoted as RAN) and the second with clustered distribution (denoted as CLU). The data points for different feature types were generated separately, resulting in different distributions in space for each type. The CLU dataset comprised 50 to 100 clusters of data points from all the multiple feature types as well as the queried data.
- Real data sets: we used two real data sets in our experiments, denoted as CA and NC. CA was converted from a California Road Network and POI spatial data set [4]. The queried data and data of all feature types were selected from the Road Network nodes and POIs respectively. For the NC data set, the queried data and features were converted from the North Carolina (NC) Master Address Database Project [9] and a GPS POI data set [10]. Both real datasets contained multiple different feature types, and therefore different distributions of data per feature type in our experiments. For each of these real data sets, there are multiple different feature types.

5.1.2. Parameter selection

There were three data parameters in our experimental setup.

- Feature Type (FT): number of feature types used to show the scalability of the algorithm.
- Cardinality of Feature Type (CF): number of data points in each feature type.
- Cardinality of Queried Data (CQ): number of data points in the queried data set.

Table 2 lists the characteristics of each data set and their parameter settings unless specified otherwise.

Unless noted otherwise, we also chose the following parameters for the MTRNN algorithm based on empirical evaluation.

- R-Tree capacity (CR): the capacity of the R-Tree for each feature and queried data set was set to 36.
- Number of Subspaces (NS): the number of subspaces for generating feature routes was set to 30.

Experiment design Fig. 15 gives an overview of the experimental setup. The query processing engine takes the spatial data sets, the parameters to be applied on the data sets, and the baseline, the new 3-step MTRNN, and the classic RNN algorithms as input.

Table 2
Data set description.

Data set	RAN	CLU	CA	NC
Dist	Random	Clustered	Real	Real
CF	2 k to 10 k	2 k to 10 k	4 k	8 k
CQ	20 k to 100 k	20 k to 100 k	22 k	50 k

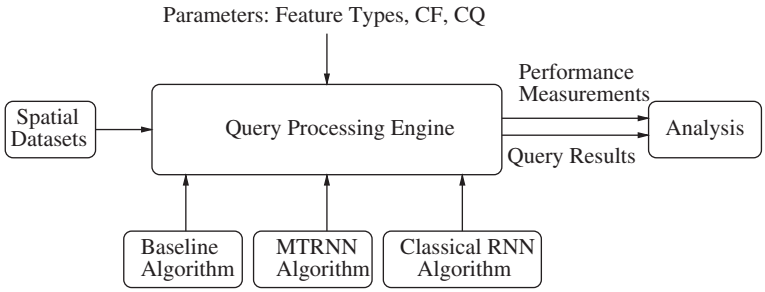


Fig. 15. Experiment setup and design.

The output consists of two categories of data, 1) performance measurements (execution time, number of IOs and filtering ratio) and 2) specific query results (RNNs). The performance measures are used to assess the viability of MTRNN for handling queries of various degrees of complexity. We include a comparison with the baseline algorithm, but this part of the evaluation is necessarily limited because baseline does not do any pruning of queried points, making it too time consuming to test more than a small number of feature types. We do not compare MTRNN with classic RNN on the performance measures listed above because the RNN algorithm is designed to solve classic RNN problems, not MTRNN problems. Since ours is the first formalization of the MTRNN problem, there are no other algorithms available to compare its performance with. Instead, we look to our second category of experimental output and assess the impact of MTRNN on the specific results returned compared to a traditional RNN approach. Note: we use RNNs or RNN points to refer to the query results of both MTRNN and classical RNN queries.

5.2. Evaluation methodology

We evaluated the scalability of the MTRNN algorithm with the following questions:

(1) How do changes in number of feature types affect MTRNN performance?
(2) How do differences in cardinality for each feature type affect performance?

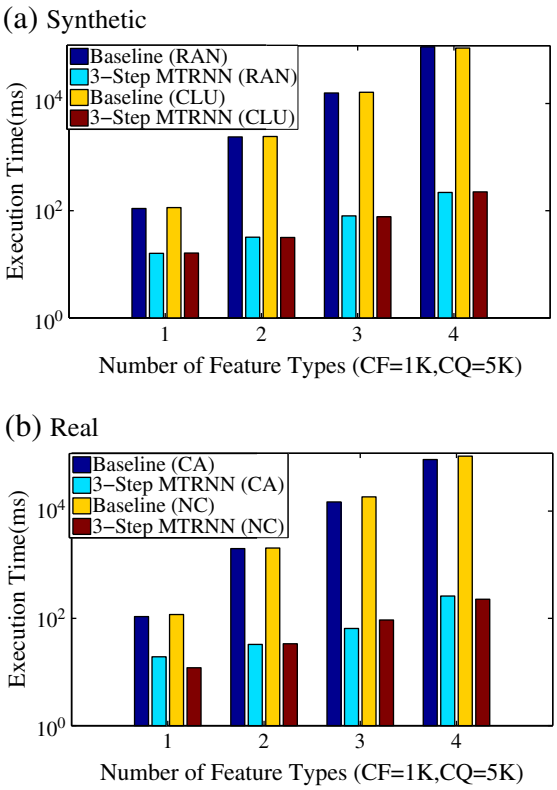


Fig. 16. Performance of baseline and MTRNN algorithms w.r.t. number of feature types.

(3) How do differences in cardinality in the queried data set affect performance?
(4) How do changes in number of feature routes affect the filtering capability of the MTRNN algorithm?

We evaluated the impact of MTRNN on query results compared to classical RNN by asking:

(5) What is the percentage difference in number of RNNs returned for the two queries
(6) What is the percentage difference in specific RNN points returned?

In every experiment for the MTRNN algorithm, we report CPU and IO time for the filtering and refinement steps, IO cost in terms of number of nodes accessed, and percentage of candidates remaining after filtering. We also report specific RNN points returned for both MTRNN and classical RNN algorithms. To reduce the effect of query point bias, we randomly selected 10 query points and averaged the results.

We define the following three metrics for evaluation purposes:
For evaluation of MTRNN algorithm performance:

(1) Filtering ratio *fr* reflects the effectiveness of the filtering step.

$$fr = \frac{Number\ of\ filtered\ points\ before\ refinement}{NumPoints\ in\ queried\ data} \tag{7}$$

In order to directly show how the query results of MTRNN are different from RNN's, we define two metrics instead of use precision and recall.

For evaluation of the impact of MTRNN on query results compared to RNN:

(2) The percentage difference in number of RNN points p_n

$$p_n = \frac{|NumPoints\ in\ MTRNN - NumPoints\ in\ Classical\ RNN|}{NumPoints\ in\ Classical\ RNN} \tag{8}$$

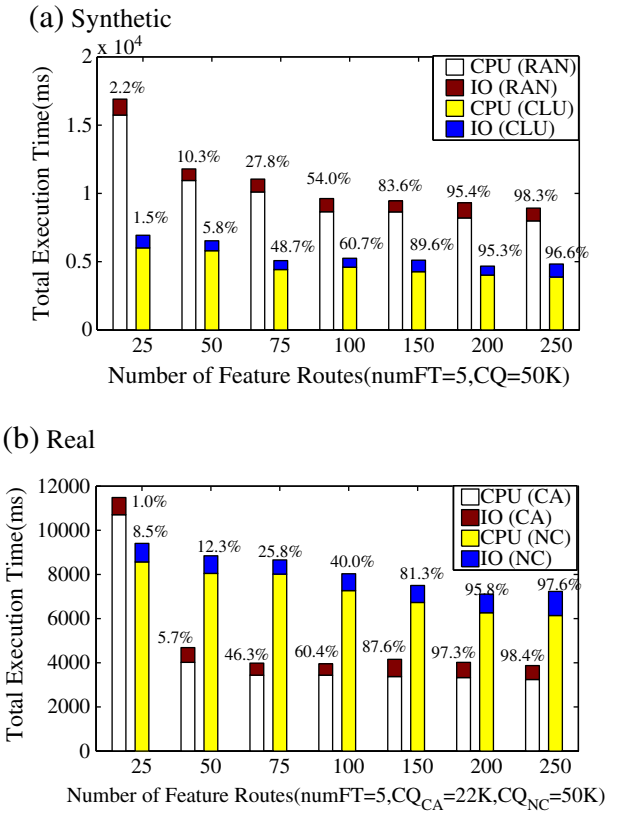


Fig. 17. Performance w.r.t. number of feature routes.

(3) The percentage difference in specific RNN points p_s

$$p_s = \frac{NumPoints\ in\ MTRNN\ but\ not\ in\ ClassicalRNN}{NumPoints\ in\ Classical\ RNN} \tag{9}$$

In other words, p_n shows how much the new MTRNN query affects the answer to the question "how many RNNs are returned?". p_s indicates how significantly the new MTRNN query affects the answer to the question "what RNNs are returned?". p_n and p_s will be meaningless if no RNN is found for the classical RNN query.

Our experiments were performed on a PC with two 2.33 GHz Intel Core 2 Duo CPUs and 3GByte memory running Windows XP SP3 operating system. All algorithms were implemented in Java programming language with Eclipse 3.3.2 as the IDE and JDK 6.0 as the running environment.

5.3. Experimental results

In this section, we present our evaluations of the MTRNN approach on both synthetic and real data. MTRNN performance evaluation includes runtime comparisons with the baseline algorithm using a small number of feature types followed by evaluations of MTRNN performance alone on various other measures. The final set of results presented measures the impact of the MTRNN approach on queries compared to traditional RNN. We end the section with a discussion of some important aspects.

5.3.1. Performance of baseline and MTRNN algorithms with regard to (w.r.t.) number of feature types

We first evaluated the performance of the MTRNN algorithm against the baseline algorithm w.r.t. the number of feature types. Since the baseline algorithm is very time consuming, we only ran the experiments for *numFT* = 1 to 4. We also chose a smaller

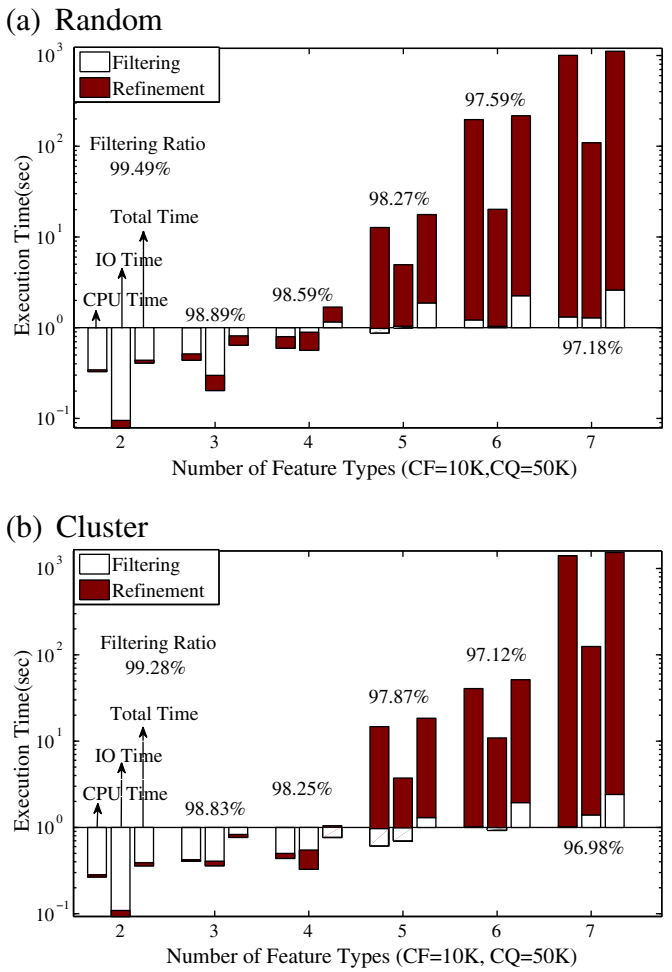


Fig. 18. Scalability of MTRNN w.r.t. number of feature types on synthetic data sets.

subset of the data from both synthetic and real data sets with cardinality of feature data set 1 k and cardinality of queried data set 5 k form comparison purpose because the baseline algorithm for larger data set runs too long. As Fig. 16 shows, the baseline execution time increases much more dramatically than for the 3-Step MTRNN algorithm as the number of feature types increases. On the other hand, our 3-Step MTRNN algorithm is much more scalable with the increment of feature types. When *numFT* = 4, the baseline execution time is more than two orders of magnitude longer. Similar patterns were found with increases in cardinality of feature type data and queried data.

5.3.2. Performance w.r.t. number of feature routes

We then evaluated the performance of the MTRNN algorithm as the number of feature routes was raised from 25 to 250. As discussed in the previous section, the number of feature routes can be increased either by increasing the number of subspaces or mNN search in a subspace. As Fig. 17 shows that both CPU and IO time decrease as the number of feature routes increases. However, when number of feature routes reaches a certain value (200 in the figure), performance gains are small, indicating that the filtering ratio is now increasing slowly. The reason is that pruning regions generated after 200 feature routes mostly overlap existing pruning regions. This result tells us that the number of feature routes in an MTRNN query can be set to a constant value. Our observation is supported by the cost model, where the effect of feature route number was set to be a constant value, too. This is true for both synthetic and real data sets.

5.3.3. Scalability of MTRNN w.r.t. number of feature types

Next we evaluated the scalability of the MTRNN algorithm with larger numbers of feature types than was tested in the first round of experiments. Overall, Fig. 18 and Fig. 19 show that the total time for processing up to 7 types, the maximum tested, is quite acceptable, considering the complexity of the MTRNN algorithm, the size of the data set and the experimental platform.

In addition, we can see the effect of feature type number on CPU and IO time. As shown in Fig. 18a and b (in log scale) for the synthetic data sets, when the feature types number 4 or less, IO and CPU time have similar weights. When there are more than 4

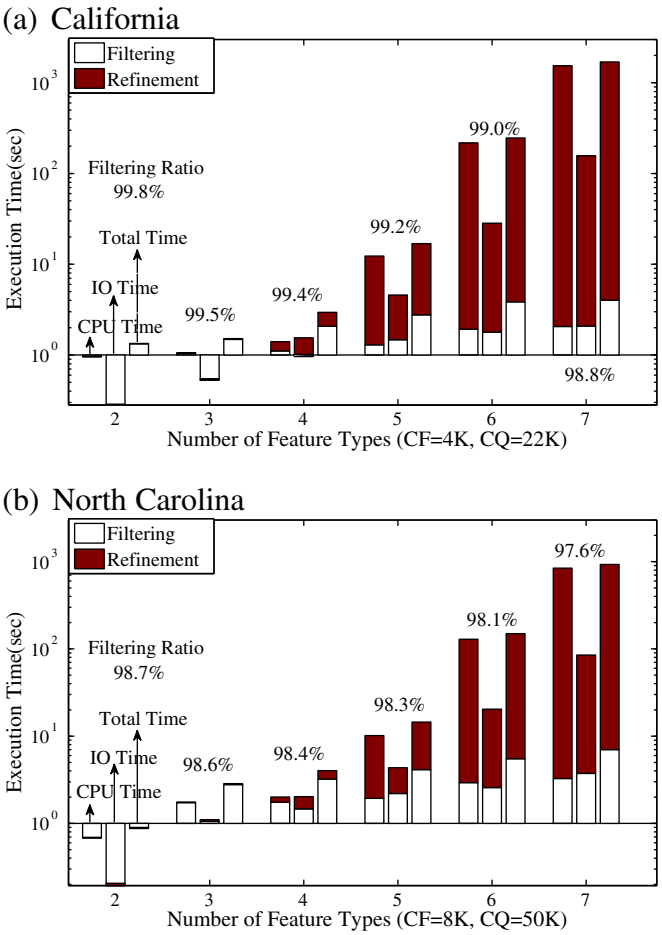


Fig. 19. Scalability of MTRNN w.r.t. number of feature types on real data sets.

feature types, CPU time and the refinement step become dominant ($FT>4$). Meanwhile, the filtering ratio slightly decreases due to the longer *feature routes* and thus the looser distance bound.

The same trends are evident for the real data CA and NC in Fig. 19a and b, again with CPU time and refinement step dominant for $FT>4$. These results indicate that a simpler filtering algorithm works well enough for smaller numbers of feature types.

A closer look at the IO cost shown in Fig. 20 reveals that while both filtering and refinement IO costs grow when FT increases, refinement IO grows faster because the number of permutations involved in the final refinement step increases dramatically as FT increases.

5.3.4. Scalability of MTRNN w.r.t. cardinality of feature types

Next we looked at the scalability of the MTRNN algorithm w.r.t. cardinality of feature types. To better reflect the effects of cardinality, we used the same set of randomly selected query points to reduce the effect of biased queries. Since CPU time is dominant when FT = 6, we only show total time in the figures. In Fig. 21, we can see that for both synthetic and real data the run time of the MTRNN algorithm is less than 300 s for most cases, which is quite good for query problem of this complexity. Furthermore, run time decreases with increases in cardinality of FT. This is mainly due to the improved filtering ratio produced by the contribution of refined feature routes. These results indicate that our filtering algorithm works very well. (Since the maximum cardinality of the real CA data set is 4 k, the experimental results from 5 k to 8 k are only for the real NC data.)

With similar settings, FT = 6 and changing the cardinality of feature types, Fig. 22 shows the similar trend as Fig. 21 for both synthetic and real data sets. In both cases, the queried data sets contain 100 k points and we evaluated the performance for the 10 k to 100 k features data sets. The CA* and NC* data sets were generated with extra random data because the original data sets are not big enough. As the figures show, the run time decreases when increasing the cardinality of feature types. The maximum elapse time is about 150 s even with the large data sets, which indicates the algorithm scales very well.

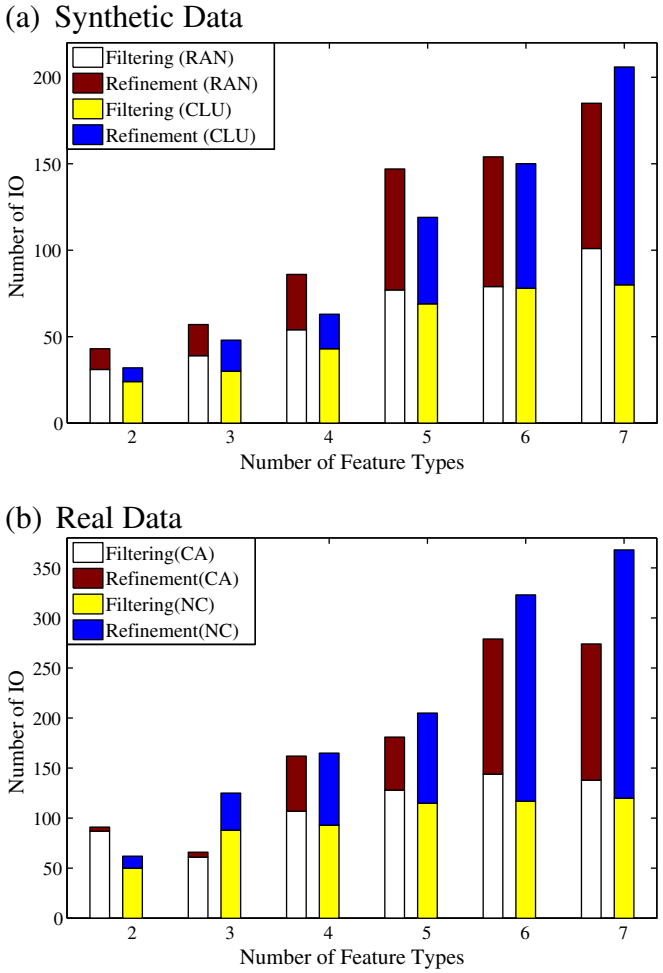


Fig. 20. IO cost of MTRNN w.r.t. number of feature types.

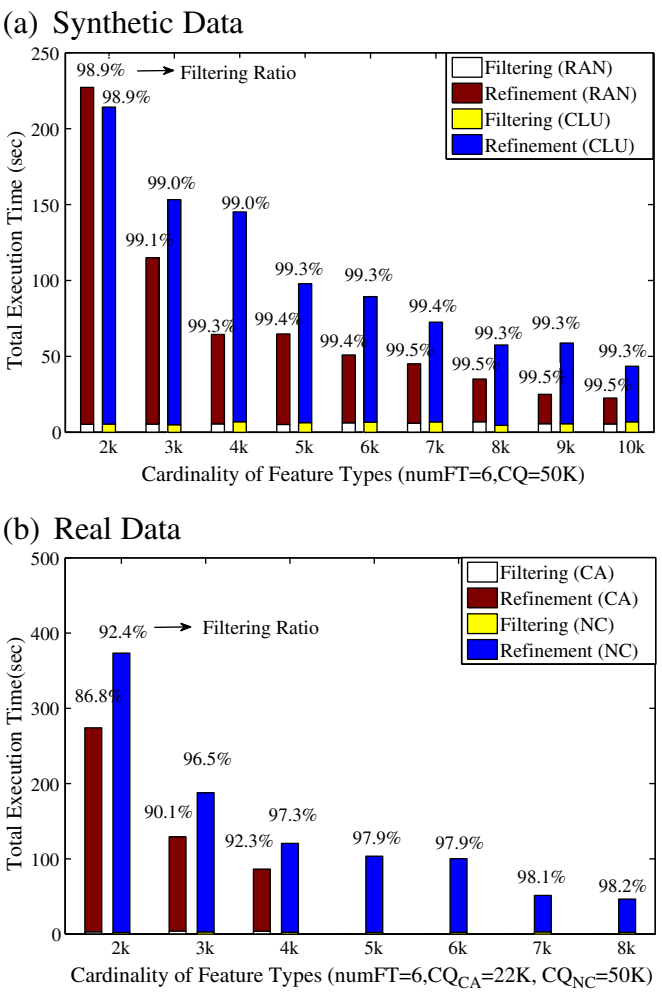


Fig. 21. Scalability of MTRNN w.r.t. cardinality of feature types.

5.3.5. Scalability of MTRNN w.r.t. cardinality of the queried data set

We also examined the scalability of the MTRNN algorithm w.r.t. cardinality of the queried data (Fig. 23). Contrary to the previous experiment, the trend this time is that increasing cardinality of the queried data increases run time (although no run exceeded 10 min even when the data set reached 100 k).

Why does this happen? First, the filtering ratio is large enough and improves much less significantly than the CQ increments on this smaller range of space. Furthermore, because the cardinality of the data sets from the multiple feature types remains the same while the cardinality of the queried data set increases dramatically, the ratio of queried data to feature data also increases dramatically. Thus, more points are left as candidate MTRNNs after the filtering step, resulting in more MTRNN points found and a rise in total run time.

In Fig. 24, the run time increases as the cardinality of queried data sets becomes larger. At this time, the size of queried data sets increase from 20 k to 100 k and the feature data set contains 100 k points for both synthetic and real data sets, the run time changes from about 10 s to 90 s. As in Section 5.3.4, the CA* and NC* data sets were generated with extra random data. This demonstrates that the algorithm can give the query results in a reasonable time range even with quite large queried data sets.

5.3.6. Filtering ratio of MTRNN w.r.t number of feature routes

Next we evaluated how the number of feature routes affects the filtering ratio. As shown in Fig. 25, the filtering ratio is very significant when the number of feature routes exceeds 150, which indicates our filtering algorithm is quite efficient.

Fig. 25 shows the filtering ability for both closed and open region pruning methods. Since closed regions are pruned before open regions, more points are pruned by closed region pruning. As shown in Fig. 25a for synthetic data, the filtering ratio grows as we increase the number of feature routes. However, when feature routes reach a certain number (180 in the figure), the growth of the filtering ratio becomes much less significant. Therefore, to save running time, we limited the number of space divisions to 30 in our experiments (resulting in about 180 feature routes). Fig. 25a also indicates that both closed and open region pruning

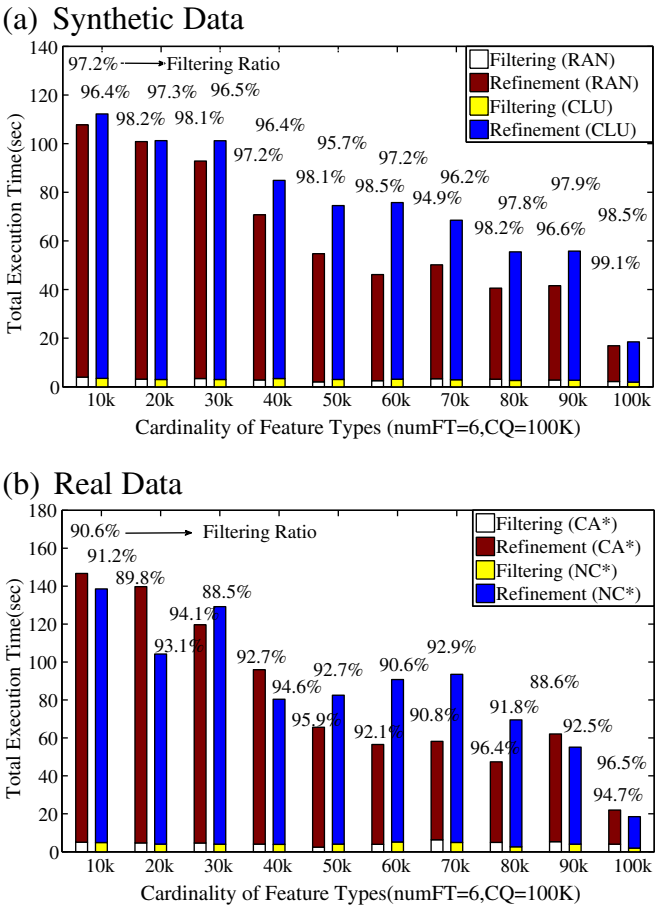


Fig. 22. Scalability of MTRNN w.r.t. cardinality of feature types (large data sets).

contributes to efficiency of the filtering step. The filtering ratio for a closed region pruning is generally higher because it is performed before open region pruning.

Fig. 25b for real data exhibits a similar pattern. That is, when the number of feature routes reaches a certain value, the filtering ratio is more than 90% and becomes steady.

Since we were finding one nearest neighbor in one subspace in this experiment, the results also show how filtering efficiency improves with increasing number of subspaces.

5.3.7. Change in number and specific RNN points returned for MTRNN and classical RNN queries w.r.t. number of feature types

Finally, to highlight the potential impact of the MTRNN query approach, we quantified changes in number of RNNs and specific RNNs w.r.t. FT for MTRNN and classical RNN queries. The results illustrated in Fig. 26 show that p_n and p_s are indeed significant for both synthetic and real data. For synthetic data (Fig. 26a), as the FT increases, the smallest percentage change of p_n with FT = 3 for data set CLU is more than 20% and most of the percentages reached more than 50%. The figure also shows that the MTRNN algorithm tends to find more RNNs when FT is relatively large.

The differences are even more dramatic in RNNs returned for the real datasets Fig. 26b, where the value of p_n and p_s rarely falls below 50% even when the number of features is only 2 or 3. In other words, MTRNN queries consistently generated results that differed by more than 50% from those generated by classical RNN queries.

5.3.8. Discussion

5.3.8.1. Factors that impact MTRNN algorithm performance. The performance of the MTRNN algorithm is usually significantly affected by the filtering ratio for large data sets. The refinement step of the MTRNN algorithm is usually dominant especially with larger feature type number (>4 in our experiment), because the MTNN algorithm applied during this step is very time consuming with feature type number rising. The larger the filtering ratio, the less the MTNN algorithm is called during the refinement step (one MTNN is called for one candidate MTRNN). Therefore, improvement of overall performance depends largely on how many candidates can be pruned during the filtering step. As can be seen from Figs. 21 and 23, the filtering ratio improves as the range of the data increases (cardinality

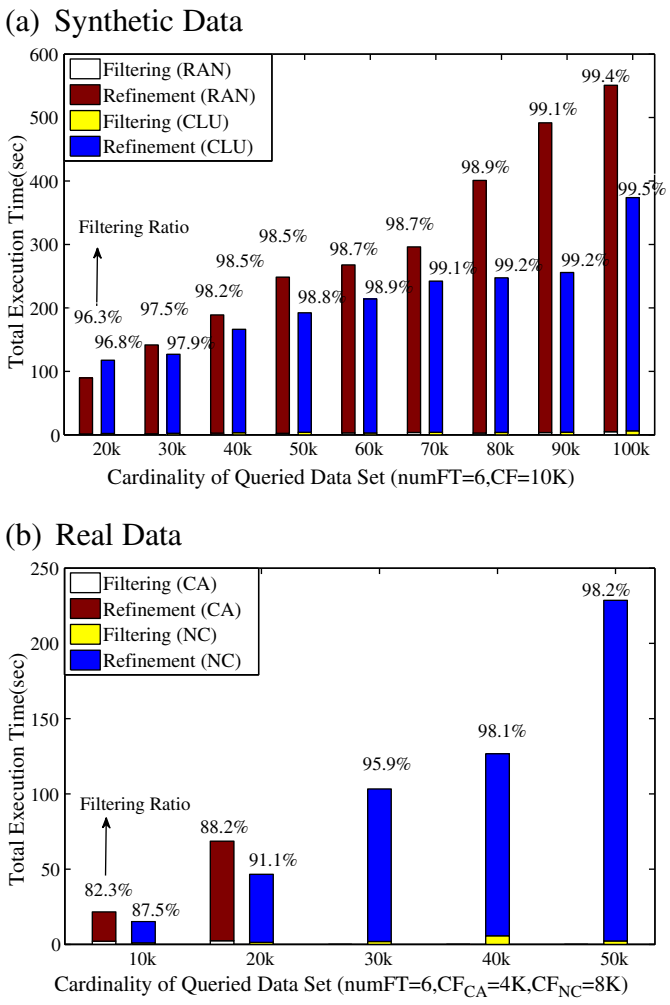


Fig. 23. Scalability of MTRNN w.r.t. cardinality of the queried data sets.

of features and queried data). The reason is that faraway R-Tree nodes are more likely to be pruned. Since the filtering step is less dominant in terms of execution time with large feature type number, we may also increase the number of feature routes to achieve similar effects. Nevertheless, a good filtering algorithm will always help to increase filtering ability.

On the other hand, when number of feature types is small, total execution time is not significant. In this case IO time becomes dominant and performance could be improved by increasing the number of feature routes (thus filtering ratio) and cache size to store more R-Tree nodes in memory.

Impact of Number of MTRNN Query on Results Summarizing the results shown in Fig. 26, it is clear that MTRNN queries can generate dramatically different results from classical RNN queries, both in terms of quantity (How many RNNs are there?) and content (What are the RNNs?). In general, the overlap between the results of MTRNN and RNN queries decreases as the feature type number increases because the other feature types around the queried data set will have an impact, too. Finally, it is important to understand that p_n and p_s could be quite different for the same set of data, with major implications for RNN-based decision-making. For example, in Fig. 26 for the CA data set with three feature types p_n is very small because the number of MTRNN results does not change much. However, the actual results returned are totally different from the classical RNN query results, resulting in a very high value of p_s. Deciding whether to look at p_n, p_s or both depends entirely on the application.

6. Related work

The notion of a reverse nearest neighbor query was first formalized in a pioneering paper by Korn and Muthukrishnan [1]. They proposed algorithms for static and dynamic cases by pre-computing the nearest neighbor for each data point. Following their work, an on-line algorithm that utilized property of 2-dimentional space partitioning was proposed in [7] for dynamic databases and an index structure was devised to answer RNN queries by Yang and Lin in [11]. Since then RNN queries have been extended in a number of dimensions including metric space [12] and [13], high-dimensional space [14], ad-hoc space [15], large graphs [16]

and spatial networks using Voronoi diagram [17]. RNN queries have further been studied in an extended family containing different variations of classic RNN problem. Yao et al. [18] studied reverse furthest neighbors in spatial databases. Tao et al. [19] dealt with monochromatic R*k*NN queries in arbitrary dimensionality for the first time. Wu et al. [20] proposed a new algorithm for both monochromatic and bichromatic R*k*NN queries on 2-dimensional location data. Gao et al. [21] extended RNN to a visible R*k*NN problem, considering presence of obstacles and Vlachou et al. [22] proposed a reverse top-k query problem to support the rank-aware query processing. Wong et al. [23] formalized a maximum bichromatic reverse nearest neighbor (MaxBRNN) problem that finds an optimal region maximizing the size of BRNNs. In this problem the location of the query point is not fixed, which makes it different from existing problems. RNN problem plays an important role in mobile computing. [24,25] researched continuous RNN and [26] studied bichromatic RNN in mobile system. Dellis and Seeger introduced the concept of Reverse Skyline Queries in [27] for the first time. Liang and Chen [28] then extended the skyline queries to uncertain databases. While RNNs have been studied for a large variety of computational problems, a common feature of all these studies is that they only consider the influence of a single feature type, the feature type of the given query point.

Where the effect of multiple feature types has been studied is in nearest neighbor queries. Sharifzadeh et al. [5] proposed an Optimal Sequenced Route (OSR) query problem and provided three optimal solutions, Dijkstra-based, LORD and R-LORD. Essentially, the OSR problem is a special case of the MTNN problem [2] because the order of feature types is fixed for the OSR problem. The OSR work was extended to road networks by using Voronoi diagrams in [29]. In this work, the visiting order of feature types is also fixed, which makes it different from the work in [30]. Recently Kanza et al. [31] formally defined an interactive route search problem in that the route is computed in steps and presented heuristic interactive algorithms for route-search queries in the presence of order constraints. In paper on Trip Planning Queries (TPQ) [4] that find a route through multiple features for all permutations of feature types, a number of fast approximate algorithms were proposed to give sub-optimal solutions. In the work [3] multiple spatial rules represented as a directed graph were imposed on sequenced route query and three algorithms were developed to find sub-optimal travel distance. These studies offer some insight into the handling of multiple feature types. However, if multi-type NN search approaches are to be extended to reverse nearest neighbor searches, they need to be able to handle non-fixed order visits and provide solutions finding optimal routes with shortest distance and then RNNs. Ma et al. [2]

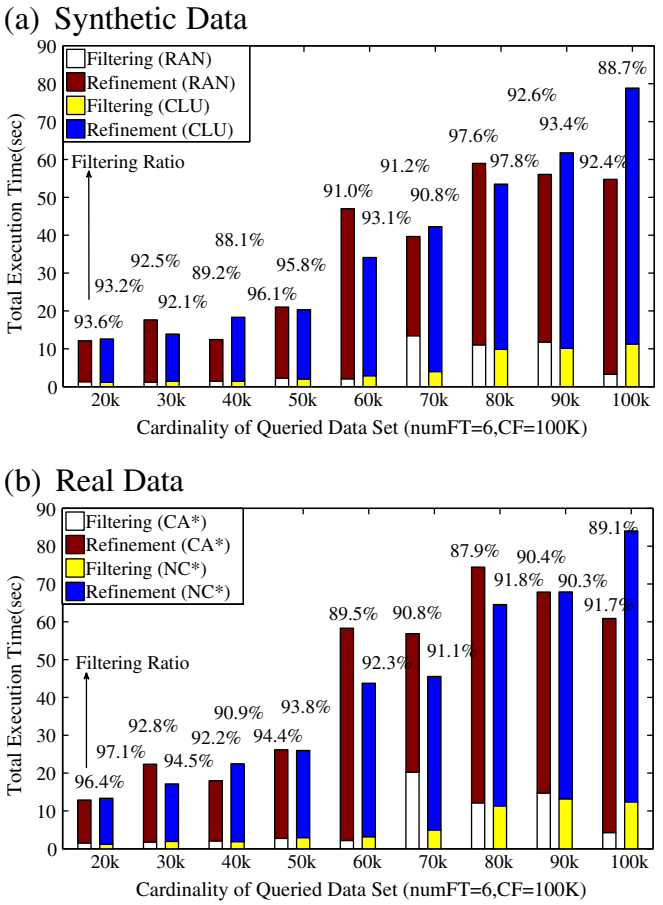


Fig. 24. Scalability of MTRNN w.r.t. Cardinality of the Queried Data Sets (Large Data Sets).

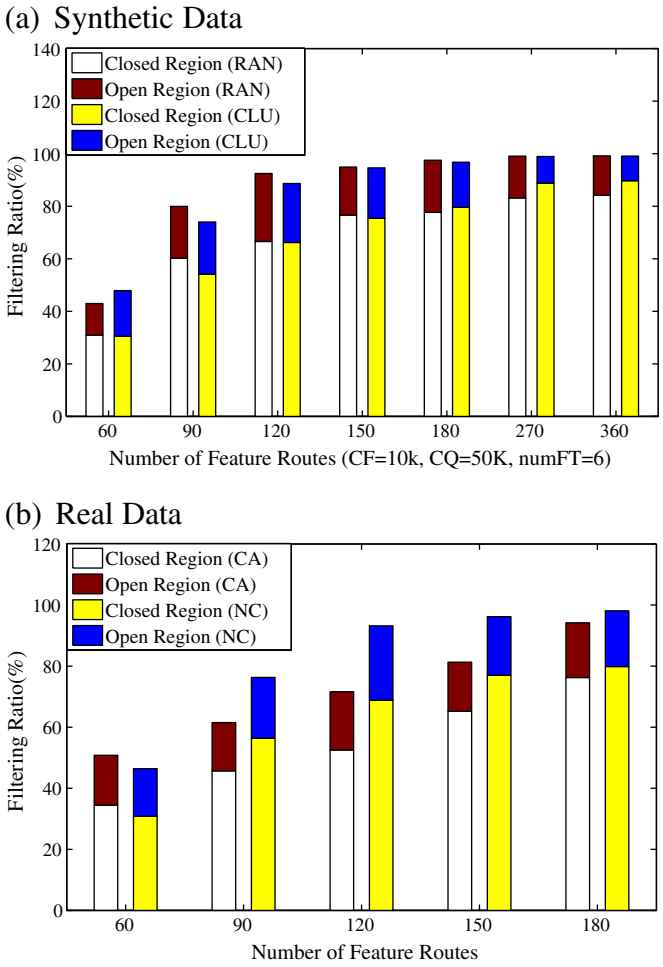


Fig. 25. Filtering ratio of MTRNN w.r.t. number of feature routes.

formalized a Multi-Type NN query problem and proposed a Page Level Upper Bound (PLUB) based algorithm to find an optimal route for the MTNN query without any predefined visiting order of feature types. This work was then extended to spatio-temporal road networks in [30]. In this paper, we build on our study of multi-type reverse NN solutions with non-fixed visits. To our knowledge, ours is the first to consider the influence of multiple features in reverse nearest neighbor problems.

7. Conclusions and future work

We formalized a multi-type reverse nearest neighbor problem (MTRNN) and developed an MTRNN query algorithm by exploiting a two-step R-tree node-level pruning strategy. In the coarse-level pruning step, we generate closed and open pruning regions which can be used to prune an entire R-tree or part of an R-tree for a given query. In the refinement step, the remaining data points are evaluated and the true MTRNN points are identified for the given query. We compared the MTRNN algorithm with a traditional RNN query method in terms of number of feature types, number of points in each feature type and queried data set. The experiment results show that our algorithm not only returns MTRNNs within a reasonable time but that MTRNN results differ significantly from results of traditional RNN queries. This finding has important implications for decision-making algorithms used in business and confirms the value of further investigation of MTRNN query approaches.

As for future work, we plan to introduce different weight factors for different feature types into the objective function. It is important for some business applications since in reality the influence of different feature types may be different for different types of applications. We are then interested in the application of MTRNN queries on road networks using road network distance or other types of spatio-temporal databases. We believe that the computational complexity of MTRNN queries will rise dramatically when applied to such databases. Therefore, we plan to explore off-line techniques for pre-computing of intermediate results of MTRNN queries as well as indexing, data structure, and other methods for efficient storage and retrieval of results. Another direction to extend our MTRNN work is to design some good heuristic approaches to find approximate results as well as design corresponding measurements to evaluate the results that sufficiently capture the problem's complexity.

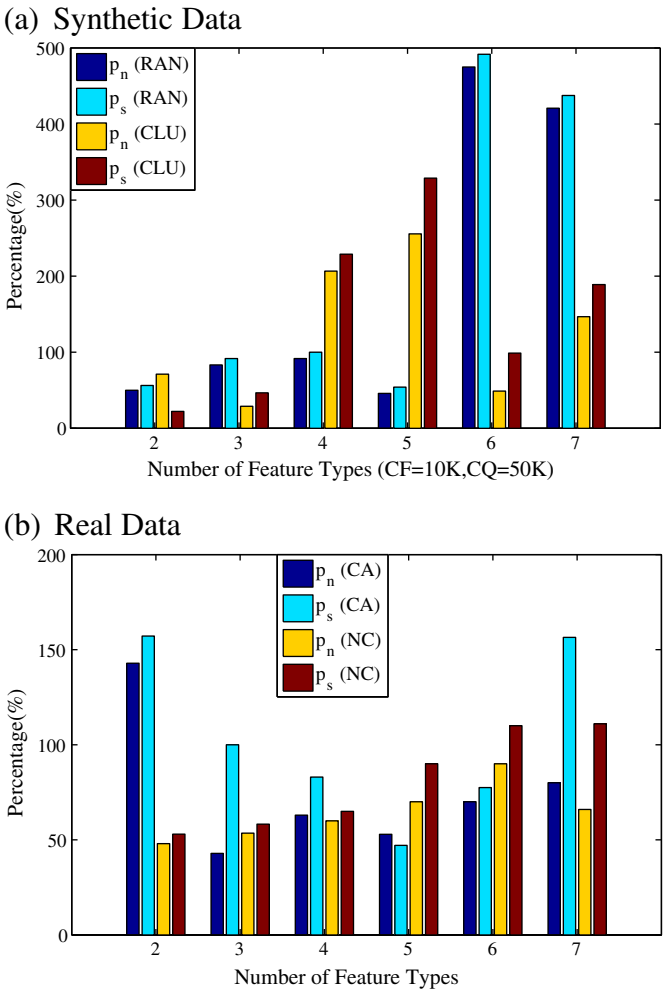
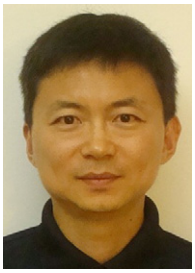


Fig. 26. Change of RNNs w.r.t. number of feature types.

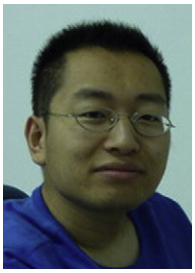
References

[1] F. Korn, S. Muthukrishnan, Influence sets based on reverse nearest neighbor queries, ACM SIGMOD, 2000.
[2] X. Ma, S. Shekhar, H. Xiong, P. Zhang, Exploiting page level upper bound for multi-type nearest neighbor queries, ACM GIS, 2006.
[3] H. Chen, W. Ku, M. Sun, R. Zimmermann, The multi-rule partial sequenced route query, ACM GIS, 2008.
[4] F. Li, D. Chen, M. Hadjieleftherious, On trip planning queries in spatial databases, SSTD, 2005.
[5] M. Sharifzadeh, M. Kolahdouzan, C. Shahabi, The optimal sequenced route query, VLDB Journal, 2006, doi:10.1007/s0078-006-0038-6.
[6] N. Roussopoulos, F.V.S. Kelly, Nearest neighbor queries, ACM SIGMOD, 1995.
[7] I. Stanoi, D. Agrawal, A.E. Abbadi, Reverse nearest neighbor queries for dynamic databases, SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, 2000.
[8] Y. Tao, J. Zhang, D. Papadias, N. Mamoulis, An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces, TKDE, 2004.
[9] North Carolina Center for Geographic Information and Analysis, NC master address dataset project, http://www.cgia.state.nc.us/Services/NCMasterAddress 2009.
[10] GPS Data Team, GPS POI files, http://poidirectory.com/poifiles/ 2010.
[11] C. Yang, K.-I. Lin, An index structure for efficient reverse nearest neighbor queries, ICDE, 2001.
[12] E. Achtert, C. Bahm, P. Krager, P. Kunath, A. Pryakhin, M. Renz, Efficient reverse k-nearest neighbor search in arbitrary metric spaces, ACM SIGMOD, 2006.
[13] Y. Tao, M.L. Yiu, N. Mamoulis, Reverse nearest neighbor search in metric spaces, TKDE, 7(4–5), 2006.
[14] A. Singh, H. Ferhatosmanoglu, A.S. Tosun, High dimensional reverse nearest neighbor queries, CIKM, 2003.
[15] M.L. Yiu, N. Mamoulis, Reverse nearest neighbors search in ad-hoc subspaces, ICDE, 2006.
[16] M.L. Yiu, D. Papadias, N. Mamoulis, Y. Tao, Reverse nearest neighbors search in large graphs, TKDE, 18(4), 2006, pp. 540–553.
[17] M. Safar, D. Ebrahimi, D. Taniar, Voronoi-based reverse nearest neighbour query processing on spatial networks, ACM Multimedia Systems Journal (MMSJ), Volume 15, Issue 5, Springer, 2009, pp. 295–308, published by.
[18] B. Yao, F. Li, P. Kumar, Reverse furthest neighbors in spatial databases, ICDE, 2009.
[19] Y. Tao, D. Papadias, X. Lian, Reverse kNN search in arbitrary dimensionality, VLDB, 2004.
[20] W. Wu, F. Yang, C.-Y. Chan, K.-L. Tan, Finch: evaluating reverse k-nearest-neighbor queries on location data, VLDB, 2008.
[21] Y. Gao, B. Zheng, G. Chen, W.-C. Lee, K. Lee, Q. Li, Visible reverse k-nearest neighbor queries, ICDE, 2009.
[22] A. Vlachou, C. Doulkeridis, Y. Kotidis, K. Norvag, Reverse top-k queries, ICDE, 2010.
[23] R.C.-W. Wong, T. Ozsu, P. Yu, A.W. Fu, Efficient method for maximizing bichromatic reverse nearest neighbor, VLDB, 2009.
[24] J.M. Kang, M. Mokbel, S. Shekhar, T. Xia, D. Zhang, Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors, ICDE, 2007.
[25] T. Xia, D. Zhang, Continuous reverse nearest neighbor monitoring, ICDE, 2006.

[26] Q.T. Tran, D. Taniar, M. Safar, Bichromatic reverse nearest-neighbor search in mobile systems, IEEE Systems Journal Vol. 4, Issue 2 (2010) 230–242.
[27] E. Dellis, B. Seeger, Efficient computation of reverse skyline queries, VLDB, 2007.
[28] X. Lian, L. Chen, Monochromatic and bichromatic reverse skyline search over uncertain databases, ACM SIGMOD, 2008.
[29] M. Sharifzadeh, C. Shahabi, Processing optimal sequenced route queries using Voronoi diagrams, Geoinformatica, 2008, doi:10.1007/s10707-007-0034-z.
[30] X. Ma, S. Shekhar, H. Xiong, Multi-type nearest neighbor queries on road networks with time window constraints, ACM SIG SPATIAL GIS, 2009.
[31] R. Levin, Y. Kanza, E. Safra, Y. Sagiv, Interactive route search in the presence of order constraints, VLDB, 2010.

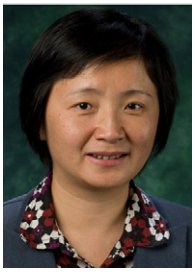


Xiaobin Ma has received his B.S. degree in Computer Science from Tianjin University in 1993 and M.S. in Computer Science from University of Minnesota in 2005. He is currently a software engineer at Oracle Corporation. His research interests include spatial databases, location based services, database query optimization and materialized view processing.



Chengyang Zang has received his B.S. degree in Industrial Automation from University of Science and Technolgy, Beijing in 2000 and Ph.D. degree in Computer Science from University of North Texas, Denton, TX, USA. He is currently a software engineer at Teradata Corporation. His research interests include streaming data management, location based services, spatio-temporal databases and database query optimization.

Shashi Shekhar received the B. Tech degree in Computer Science from the Indian Institute of Technology, Kanpur, India, in 1985, the M.S. degree in Business Administration and the Ph.D. degree in Computer Science from the University of California, Berkeley, CA, USA, in 1989. He is a McKnight Distinguished University Professor the University of Minnesota, Minneapolis, MN, USA. His research interests include spatial databases, spatial data mining, geographic and information systems (GIS), and intelligent transportation systems. He is a co-author of a textbook on Spatial Databases (Prentice Hall, 2003, isbn 0-13-017480-7), co-edited an Encyclopedia of GIS (Springer, 2008, isbn 978-0-387-30858-6) and has published over 260 research papers in peer-reviewed journals, books, and conferences, and workshops. He is serving as a co-Editor-in-Chief for Geo-Informatica: An International Journal on Advances in Computer Sc. for GIS, a series editor for the Springer Briefs in GIS, a general co-chair for Intl. Symposium on Spatial and Temporal Databases (2011) and a program co-chair for Intl. Conf. on Geographic Information Science (2012). He served on the editorial boards of IEEE Transactions on Knowledge and Data Engineering as well as the IEEE-CS Computer Science & Engineering Practice Board. He also served as a program co-chair of the ACM Intl. Workshop on Advances in Geographic Information Systems, 1996. He is serving on the National Academy's Future Work-force for Geospatial Intelligence Committee (2011). He served as a member of the mapping science committee of the National Research Council National Academy of Sciences (2004-9), as well as the Board of Directors of University Consortium on GIS (2003-2004). Dr. Shekhar received the IEEE Technical Achievement Award (2006). He is a Fellow of the IEEE Computer Society, a Fellow of the American Association for Advancement to Science, as well as a member of the ACM.



Yan Huang received her B.S. degree in Computer Science from Beijing University, Beijing, China, in 1997 and Ph.D. degree in, Computer Science from University of Minnesota, Twin-cities, MN, USA, in 2003. She is currently an associate professor at the Computer Science and Engineering department of University of North Texas, Denton, TX, USA. Her research interests include spatio-temporal Databases and mining, geo-stream data processing, and mobile computing.

Hui Xiong is currently an Associate Professor in the Management Science and Information Systems department at Rutgers University. He received the B.E. degree from the University of Science and Technology of China — (USTC), China, the M.S. degree from the National University of Singapore — (NUS), Singapore, and the Ph.D. degree from the University of Minnesota — (UMN), USA. His general area of research is data and knowledge engineering, with a focus on developing effective and efficient data analysis techniques for emerging data intensive applications. He has published over 90 technical papers in peer-reviewed journals and conference proceedings. He is a co-editor of Clustering and Information Retrieval (Kluwer Academic Publishers, 2003) and a co-Editor-in-Chief of Encyclopedia of GIS (Springer, 2008). He is an Associate Editor of the Knowledge and Information Systems journal and has served regularly in the organization committees and the program committees of a number of international conferences and workshops. He is a senior member of the ACM and IEEE.