Data & Knowledge Engineering 68 (2009) 705–727

Contents lists available at ScienceDirect

Data & Knowledge Engineering

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

On efficient mutual nearest neighbor query processing in spatial databases

Yunjun Gao [a,b,*], Baihua Zheng [a], Gencai Chen [b], Qing Li [c]

[a] School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902, Singapore
[b] College of Computer Science, Zhejiang University, Hangzhou 310027, China
[c] Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong

ARTICLE INFO

Article history:
Received 6 August 2008
Received in revised form 9 April 2009
Accepted 9 April 2009
Available online 3 May 2009

Keywords:
Query processing
Nearest neighbor
Spatial databases
Algorithm

ABSTRACT

This paper studies a new form of nearest neighbor queries in spatial databases, namely, *mutual nearest neighbor* (MNN) search. Given a set D of objects and a query object q, an MNN query returns from D, the set of objects that are among the k_1 ($\geqslant 1$) nearest neighbors (NNs) of q; meanwhile, have q as one of their k_2 ($\geqslant 1$) NNs. Although MNN queries are useful in many applications involving decision making, data mining, and pattern recognition, it cannot be efficiently handled by existing spatial query processing approaches. In this paper, we present the first piece of work for tackling MNN queries efficiently. Our methods utilize a conventional data-partitioning index (e.g., R-tree, etc.) on the dataset, employ the state-of-the-art database techniques including best-first based k nearest neighbor (kNN) retrieval and reverse kNN search with TPL pruning, and make use of the advantages of batch processing and reusing technique. An extensive empirical study, based on experiments performed using both real and synthetic datasets, has been conducted to demonstrate the efficiency and effectiveness of our proposed algorithms under various experimental settings.

1. Introduction

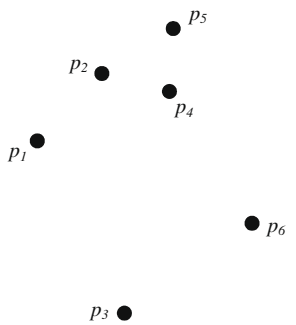
This paper studies a new form of nearest neighbor (NN) queries, namely *mutual nearest neighbor* (MNN) search. Given a dataset D, a query point q, and two parameters k_1 and k_2, an MNN query retrieves those objects $p \in D$ such that $p \in NN_{k1}(q)$[1] and $q \in NN_{k2}(p)$, i.e., it requires each *answer object*[2] to be one of the k_1 nearest neighbors (NNs) to q and meanwhile has q as one of its k_2NNs. Consequently, it considers not only the spatial proximity of the answer objects to q, but also the spatial proximity of q to the answer objects. In other words, the conventional NN query is *asymmetric*, while MNN retrieval is *symmetric*. Although it is well known that asymmetric NN search fits the requirements of lots of applications, there are still many other practical applications that require symmetric NN queries. Some real-life applications are presented as follows.

Resource allocation. Consider that a logistic company A has six branches (labeled as $p_1, p_2, p_3, p_4, p_5, p_6$), as shown in Fig. 1a. In order to guarantee the quality of service, company A assigns each branch *two* nearby branches as backup to provide necessary supports in cases such as running out of cargo; meanwhile, it has to balance the workload of each branch, and thus assigns one branch to *only two* other branches. Suppose that we employ conventional k ($\geqslant 2$) nearest neighbor (kNN) search, and process the branches in the order of p_1, p_2, p_3, p_4, p_5, p_6. In particular, p_1 is the first branch evaluated and it is linked to its

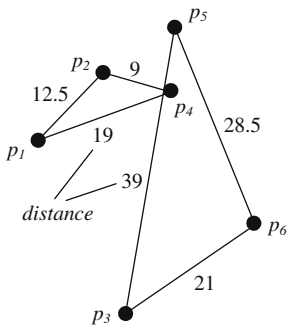
* Corresponding author. Address: School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902, Singapore. Tel.: +65 6828 0277; fax: +65 6828 0919.
E-mail addresses: yjgao@smu.edu.sg, gaoyj@zju.edu.cn (Y. Gao), bhzheng@smu.edu.sg (B. Zheng), chengc@zju.edu.cn (G. Chen), itqli@cityu.edu.hk (Q. Li).

[1] Without loss of generality, notation $NN_k(q)$ represents k nearest neighbors of a specified query point q.
[2] For the rest of this paper, we refer to the data objects in the final query result set as *answer objects*.

0169-023X/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2009.04.004



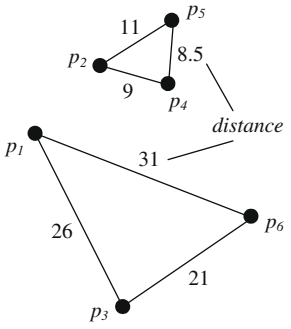
Branch	Backup	Distance
p_1	p_2, p_4	12.5, 19
p_2	p_4, p_1	9, 12.5
p_3	p_6, p_5	21, 39
p_4	p_2, p_1	9, 19
p_5	p_6, p_3	28.5, 39
p_6	p_3, p_5	21, 28.5
Average distance		21.5



(a) Branch placement

(b) Assignment with kNN search

Branch	Backup	Distance
p_1	p_3, p_6	26, 31
p_2	p_4, p_5	9, 11
p_3	p_6, p_1	21, 26
p_4	p_5, p_2	8.5, 9
p_5	p_4, p_2	8.5, 11
p_6	p_3, p_1	21, 31
Average distance		17.75



(c) Assignment with MNN search

Fig. 1. Example of resource allocation application.

2 nearest branches (i.e., p_2 and p_4). Next, p_2 is evaluated. As p_2 is already linked to p_1, we only need to find one branch out of branches p_3, p_4, p_5, and p_6 that is the closest to p_2, i.e., p_4. Then, p_3 is evaluated. Since p_1, p_2, and p_4 are all linked to two other branches, they are out of the consideration. Hence, p_3 is linked to branches p_5 and p_6. The process continues, and Fig. 1b depicts the result of assignment[3]. Although kNN retrieval provides one assignment, it can only ensure that the assigned two branches to p (denoted as p' and p'') are the nearest (or close) to p, but it does not consider whether p' and p'' are more suitable to p than to other branches. On the other hand, MNN search considers symmetric NN relationship. If p is assigned to p' using the MNN retrieval with respect to k_1 and k_2, it means that p is one of k_1NNs to p' and meanwhile p' is one of k_2NNs to p. Continuing the running example, we utilize MNN queries to deal with the assignment of branches. Initially, both k_1 and k_2 are set to one in order to retrieve the objects that are closest to each other. Consequently, p_3 is linked to p_6 and p_4 is linked to p_5 because they are NN objects to each other. Then, we increase either k_1 or k_2 by one (i.e., $k_1 = 1$, $k_2 = 2$ or $k_1 = 2$, $k_2 = 1$). Branch p_4 is linked to p_2 as p_4 is the NN object of p_2 and meanwhile p_2 is one of p_4's 2 NN objects, i.e., p_4 satisfies an MNN query with $k_1 = 1$ and $k_2 = 2$ issued at p_2. Next, we set both k_1 and k_2 to 2, and p_2 is linked to p_5 since p_2 and p_5 are MNN objects to each other with respect to $k_1 = k_2 = 2$. Here, p_2, p_4, and p_5 will be out of the consideration in the subsequent assignment as they are all linked to two other branches, and the only left branch is p_1. Finally, p_1 is linked to branches p_3 and p_6 to finish the assignment. Fig. 1c illustrates the assignment which is generated by MNN queries. Observe that, the assignment derived from MNN queries reduces the average distance between a branch and its backup to 17.75, compared with the average distance 21.5 generated by kNN search.

Matchmaking. A matchmaking service provider B has lots of members and its responsibility is to, for a new member, recommend a set of candidate members that may have interests on the new members. As the popularity of B highly depends on the success probability of the matchmaking, all the matches it recommends have to be *perfect*. For a new member m, 2NN search can identify two members, say m_1 and m_2, that are closest to the conditions (e.g., age, education level, hobbies, location, etc.) specified by m, but it ignores the fact that m might not be appealing to m_1 and m_2. On the other hand, reverse k(=2) NN (i.e., R2NN) retrieval can identify those members, say m_3, m_4, and m_5, who will rank m as their top-2 choices. However, it again ignores the fact that m might have more interest on members other than m_3, m_4, and m_5. In summary, both kNN search and RkNN search are asymmetric, while matchmaking service is symmetric. Thus, MNN search is more suitable. By carefully

[3] Note that the assignment using kNN queries depends on the processing order of the branches.

selecting k_1 and k_2, MNN query can find m' for m such that m' is one of the top-k_1 choices for m and meanwhile m is one of the top-k_2 choices for m', and thus improves the success ratio of the matching.

In addition, MNN search is useful for data analysis operation. Specifically, Gowda and Krishna [15] use the mutual neighborhood value (MNV)[4] [22] for every pair of data points to produce a hierarchical clustering tree, and utilize the notation of mutual nearest neighborhood to obtain a modified condensed training set [16]. Ding and He [12] report that the k-mutual nearest neighbor (kMN) consistency[5] can be used to improve the performance of K-means algorithm, which is the most popular clustering method. In [6,18,34,48], the authors study applications of the k-mutual neighborhood graph[6] [21] (that can be computed with MNN retrieval) for knowledge discovery. In particular, the authors discuss the application of k-mutual neighborhood graph in data mining operations such as clustering and outlier detection tasks. Jin et al. [24] develop an efficient measure of local outliers based on a symmetric neighborhood relationship which takes both the NNs and reverse NNs into consideration, and proposes several mining algorithms to detect top-n outliers [23] efficiently.

More recently, Wong et al. [45] introduce the concept of *bichromatic* MNN that considers two datasets and employ it to deal with the spatial matching problem. Gao et al. [14] explore the MNN query over *moving object trajectories*. In particular, they thoroughly investigate two classes of queries, viz. MNN_P and MNN_T queries, which are MNN search defined with respect to stationary query points and moving query trajectories, respectively. Different from all the above work, this paper focuses on *monochromatic* MNN (that involves a single dataset) query processing for *spatial* (instead of spatiotemporal) objects. Furthermore, the MNN retrieval also differs from the existing k-closest pair query [2,8–10,36], which considers two spatial datasets D_A, D_B, and return k *pairs* of objects (o_a, o_b) such that $o_a \in D_A, o_b \in D_B$, and these k pair-wise distances are the smallest among all possible object pairs in $D_A \times D_B$.

Given an MNN search, a naive solution is to find the set of k_1 NNs of a given query point q, denoted by $NN_{k1}(q)$, and then verify whether each point p in $NN_{k1}(q)$ has q as one of its k_2 NNs; If yes, p is an actual MNN of q with respect to k_1 and k_2. Unfortunately, this method is extremely inefficient because it needs to browse the dataset multiple times, resulting in high I/O overhead and expensive CPU cost, especially for large values of k_1 and k_2.

Motivated by the significance of MNN queries and the lack of efficient algorithms, in this paper, we propose four novel and efficient MNN query processing algorithms, namely, *two-step algorithm* (TS), *reuse two-heap algorithm* (RTH), *algorithm using NN search with pruning*(NNP), and *algorithm using RNN search with pruning* (RNNP). Our approaches (i) utilize a data-partitioning index (i.e., R*-tree [3]) on the dataset, (ii) employ the state-of-the-art database techniques including best-first based kNN retrieval [19] and reverse kNN (RkNN) search with TPL pruning [41], and (iii) make use of the advantages of batch processing and reusing technique. To the best of our knowledge, this paper is the first piece of work aiming at efficiently tackling monochromatic MNN queries in spatial databases. The efficiency and effectiveness of our proposed algorithms are demonstrated through extensive experiments using both real and synthetic datasets.

The rest of this paper is organized as follows. Section 2 reviews related work, including NN and RNN queries. Section 3 formalizes the MNN query and analyzes its characteristics, followed by a baseline algorithm. Four improved algorithms (i.e., TS, RTH, NNP, and RNNP) for MNN queries and their corresponding efficiency are elaborated in Section 4. Section 5 presents extensive experimental results and reports our findings. Finally, Section 6 concludes the paper with some directions for future work.

2. Related work

In this section, we briefly review algorithms for NN/kNN retrieval, and that for RNN/RkNN search. Although there are multiple indexes available, our work adopts R-tree, one of the most well-known spatial indexes, and hence the related work surveyed in this section is based on R-tree and its variants.

2.1. NN/kNN query algorithms

Following the common methodology in the relevant literature, we assume that the dataset is indexed by an R-tree due to its efficiency and popularity. Our solutions, however, are applicable to other access methods (e.g., X-tree [5], etc.). The R-tree [17] and its variants (most notably the R*-tree [3]) are generalizations of B-trees in a multi-dimensional space. Fig. 2 shows a set of data points $D = \{a,b,c,d,e,f,g,h,i,j\}$ and a corresponding R-tree that indexes D, assuming that the node capacity is three. Points close in space (e.g., a,b,c) are clustered in the same leaf node (e.g., N_3). Nodes are then recursively grouped together with the same principle until the top level, which consists of a single root node denoted by *Root*.

The algorithms for NN/kNN search on R-trees follow the branch-and-bound paradigm and utilize some metrics to prune the search space: (i) $mindist(q,N)$, (ii) $maxdist(q,N)$, and (iii) $minmaxdist(q,N)$, where q is a query point and N is the minimum bounding rectangle (MBR) associated with a node. The $mindist(q,N)$ and $maxdist(q,N)$ give the lower and upper bounds of the distance from q to any point in the subtree of N. The $minmaxdist(q,N)$ defines an upper bound of the distance between q and

[4] Let x_j be the mth NN of x_i and x_i be the nth NN of x_j. Then, the MNV between x_i and x_j, denoted by $\text{MNV}(x_i,x_j)$, is defined as $m+n$, i.e., $\text{MNV}(x_i,x_j) = m+n$, where $m,n = 1,2,\ldots,K$ and K is a given neighborhood depth.

[5] If x_i is $(k+1)$-MN consistent with respect to cluster C_p, then x_i must be kMN consistent with respect to C_p (see [12] for details).

[6] Generally, each vertex of the k-mutual neighborhood graph represents a data item. For each pair of data items, only if both of them are among the k-most similar data items of each other, can there be an edge between the two corresponding vertices.

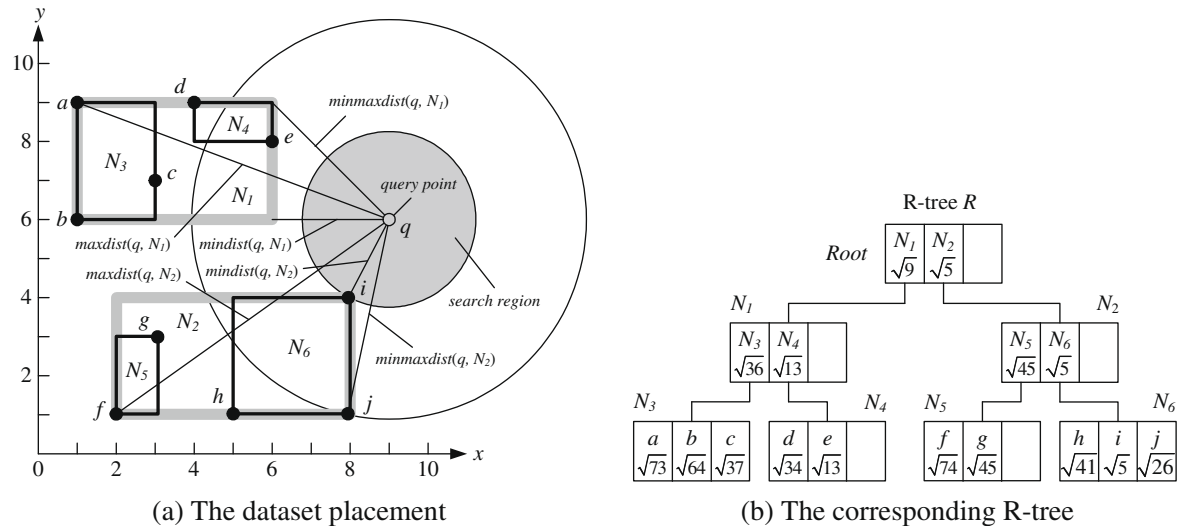


Fig. 2. Example of an R-tree and a 3NN query.

Table 1
The trace of BF algorithm for 3NN search.

Action	Heap content	NNs
visit *Root*	$\{N_2, N_1\}$	$\emptyset$
follow N_2	$\{N_6, N_1, N_5\}$	$\emptyset$
follow N_6	$\{i, N_1, j, h, N_5\}$	$\emptyset$
remove i	$\{N_1, j, h, N_5\}$	$\{i\}$
follow N_1	$\{N_4, j, N_3, h, N_5\}$	$\{i\}$
follow N_4	$\{e, j, d, N_3, h, N_5\}$	$\{i\}$
remove e	$\{j, d, N_3, h, N_5\}$	$\{i, e\}$
remove j	$\{d, N_3, h, N_5\}$	$\{i, e, j\}$

its NN in N. In other words, there is at least one point located inside N whose distance to q does not exceed $minmaxdist(q,N)$. Fig. 2a illustrates these pruning metrics between q and nodes N_1, N_2.

Existing NN/kNN query methods are based on either *best-first* (BF) or *depth-first* (DF) traversal. The DF algorithms [7,35] retrieve the NN(s) by traversing the R-trees in the depth-first fashion. As demonstrated in [33], the DF algorithm is suboptimal, i.e., it accesses more I/O than necessary. Nevertheless, it requires only bounded memory and at most a single tree path resides in memory during search.

The BF algorithm proposed in [19] achieves the optimal I/O performance, meaning that it visits only the qualified entries that may contain/be the NN(s) of q, e.g., the entries (including *Root*, N_2, and N_6) covered by the so-called *search region* (the shaded circle) in Fig. 2a. BF maintains a priority queue (e.g., a heap H used in this paper) with the entries visited so far, sorted in ascending order of their *mindist*. Starting from the root of the tree, BF inserts all the root entries into H together with their *mindist*. Then, the top entry e with the smallest *mindist* is de-heaped from H and evaluated. There are two cases: (i) e is a leaf entry and the corresponding data object is reported as an actual NN of q; or (ii) e is an intermediate (i.e., non-leaf) entry and the child entries of e are inserted into H. BF proceeds to evaluate the top entry de-heaped from H in the same manner until k ($\geqslant 1$) NN(s) of q are retrieved.

As an example, consider the R-tree R depicted in Fig. 2b, where the number in each entry e refers to the $mindist(q,e)$. Note that when e refers to a point, $mindist(q,e) = dist(q,e)$,[7] and these numbers are derived *on-the-fly* during query processing. Now suppose a 3NN query is issued at point q. The detailed steps of BF algorithm are illustrated in Table 1 where, for simplicity, we omit associated distances to q for node MBRs and data points.

Recently, many variants of NN search have been studied. Ferhatosmanoglu et al. [13] discuss *constrained* NN search that discovers the NN(s) in a constrained area of the data space. Song and Roussopoulos [38] and Tao et al. [43] investigate *continuous* NN search independently, in which the goal is to handle the NN retrieval in the setting of moving query objects and static dataset. Papadias et al. [31,32] explore *group* NN and *aggregate* NN queries. Zhang et al. [47] introduce *all* NN queries where, given two datasets D_1 and D_2, the goal is to retrieve for each point $p_1 \in D_1$ its NN $p_2 \in D_2$. Aghbari [1] proposes a *plug* &

[7] Without loss of generality, $dist(p_i,p_j)$ is a function to compute the Euclidean distance between any two points p_i, p_j, although any distance metric can be used in general.

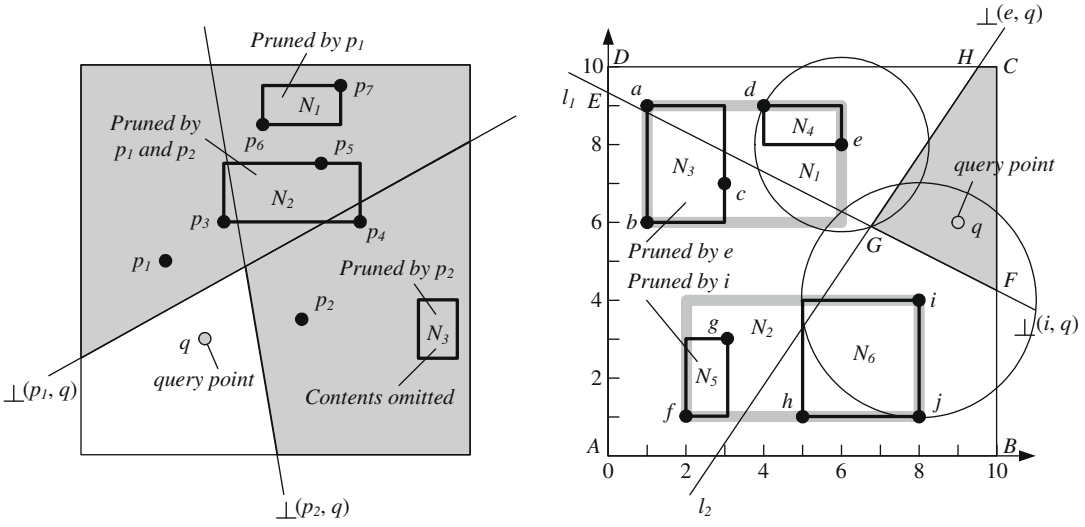
search approach to significantly speed up the *k*NN search of existing data partitioning methods. Deng et al. [11] consider *surface k*NN search, where the distance is calculated from the *shortest path* along a terrain surface. Hu and Lee [20] study the *range* NN query that retrieves the NN(s) for every point in a range. However, to the best knowledge of the authors, none existing work has solved the MNN query.

2.2. RNN/RkNN query algorithms

A *reverse* NN (RNN) query finds all the points in a data set D that take a specified query point q as their nearest neighbor. *Reverse k*NN (R*k*NN) search generalizes RNN to retrieve the points in D that have q as one of their *k*NNs. Early algorithms [26,29,30,46] are based on pre-computation. For each point p, it pre-computes the distance between p and its NN p', and forms a vicinity circle $cir(p,p')$ that is centered at p and has $dist(p,p')$ as the radius. Then, it checks a given query point q against all the vicinity circles $cir(p,p')$ for $p \in D$, and returns those having their vicinity circles enclosing q as the answer objects. To facilitate the examination, all the vicinity circles are indexed with RNN-tree [26] or R*d*NN-tree [46]. This method has two major shortcomings: (i) the index construction cost and update overhead is very expensive; and (ii) although the approach can be extended to deal with the R*k*NN query (if the corresponding *k*NN information for each point is available), it is limited to answer R*k*NN retrieval for a fixed k.

Inspired by the defects of pre-computation based approaches, several alternative RNN/R*k*NN search methods without pre-computation have been proposed. First, Stanoi et al. [39] develop a query algorithm based on *filter-refinement* framework. It partitions a 2D space around q into six equal regions, and guarantees that only the nearest object to q in each sub-region may become the result. Consequently, the candidate set only contains 6 objects, retrieved by constraint NN search [13] at each sub-region. It then validates each candidate by checking whether q is its nearest neighbor. The efficiency of this algorithm is assured by the small cardinality of the candidate set, whose size increases exponentially with the dimensionality. In order to break the curse of dimensionality, Singh et al. [37] present a multi-step algorithm for the RNN query in a high-dimensional space. Nevertheless, the algorithm may incur *false misses*, that is, it cannot guarantee that all the answer objects are returned. To address the deficiencies of the above algorithms, Tao et al. [41] propose a novel solution to R*k*NN search, called TPL. TPL is very efficient in a low-dimensional space. Since our proposed algorithms utilize R*k*NN retrieval with TPL pruning, in the following we describe the TPL algorithm via an illustrative example.

To explain the rationale of TPL, we consider the dataset shown in Fig. 3a. Let $\perp(p,q)$ be the perpendicular bisector of the segment connecting point p and point q. The bisector $\perp(p,q)$ divides the data space into two half-planes: $HP_q(p,q)$ that contains q and $HP_p(p,q)$ that contains p. Any point p' or MBR N falling inside $HP_p(p,q)$ must have p closer to it than q. Thus, p'/N cannot be/contain an RNN of q and can be safely pruned away. As illustrated in Fig. 3a, the bisector $\perp(p_1,q)$ partitions the data space into two half-planes, i.e., $HP_q(p_1,q)$ and $HP_{p1}(p_1,q)$. As points p_6, p_7 (contained in N_1) fall inside the half-plane $HP_{p1}(p_1,q)$, they are closer to p_1 than to q, and hence they for sure are not answer objects. Similarly, N_3 can also be discarded because it falls into the half-plane $HP_{p2}(p_2,q)$. It is important to note that the pruning of an MBR may require multiple half-planes in some cases. In Fig. 3a, for example, N_2 can be pruned since it lies entirely in $HP_{p1}(p_1,q) \cup HP_{p2}(p_2,q)$ (the shaded area). In addition, the number of half-planes $HP_p(p,q)$ that a given point p' falls in represents the number of data points that



(a) TPL pruning (b) RNN search with TPL pruning

Fig. 3. Example of TPL algorithm.

are closer to p' than q. Consequently, if a data point is inside at least k $HP_p(p,q)$ half-planes, it cannot be an RkNN candidate, and thus can be safely pruned away.

TPL follows a filter-refinement framework. In the filter step, TPL continuously prunes the search space based on the bisector(s) between q and its NN(s), until all the objects located inside the search space are evaluated. This idea is depicted in Fig. 3b. TPL assumes an R-tree on the dataset, and uses the BF based NN query algorithm to retrieve the points. In this example, the first point (i.e., the first NN of q) evaluated is i, which is added to a candidate set S_c. Then, TPL obtains the bisector $\perp(i,q)$ (i.e., line l_1), and shrinks the search space from the entire space (i.e., *ABCD*) to a trapezoid *EFCD*. Therefore, points h, j in N_6 can be pruned, which are maintained by a refinement set S_{rfn}. Similarly, node N_5 does not need to be accessed and is added to S_{rfn}, as it falls fully in *ABFE*.

Next, among the objects enclosed in *EFCD*, TPL identifies the point e (i.e., the second NN of q) and inserts it into S_c. Here TPL captures another bisector $\perp(e,q)$ (i.e., line l_2) and further shrinks the search region (from *EFCD*) to quadrilateral *GFCH*. Obviously, both point d and node N_3 can be pruned. At this time, the filter step of the TPL algorithm terminates because there is no any data object left inside *GFCH*. Similar to Table 1, Table 2 lists the executive processes of TPL during the filter stage.

After the termination of the filter step, TPL has a candidate set S_c (=$\{i,e\}$), and a set S_{rfn} (=$\{j,d,N_3,h,N_5\}$). In the refinement step, TPL eliminates *false hits* by reusing the pruned points/MBRs maintained in S_{rfn}. For instance, continue the running example, point $e \in S_c$ is a false hit since it is closer to $d \in S_{rfn}$ than to q; and point $i \in S_c$ is the final RNN of q.

In addition to conventional RNN/RkNN search, various variants of RNN/RkNN queries have been well-studied in the database literature, e.g., Benetis et al. [4] and Tao et al. [42] investigate RkNN search over linearly moving objects with fixed velocities; Stanoi et al. [40] discuss *bichromatic* RNN search; Korn et al. [27] examine *aggregate* RNN retrieval on data streams; Xia and Zhang [44] and Kang et al. [25] study *continuous* RNN *monitoring*; Lee et al. [28] explore *ranked* RkNN search, and so forth.

3. Preliminaries

In this section, we formally define the MNN query, and then reveal some important characteristics of MNN. Subsequently, we present a baseline algorithm for MNN search, and analyze its performance. Table 3 summarizes the symbols to be used in the rest of this paper.

Table 2
The trace of the filter step in the RNN search using TPL algorithm.

Action	Heap content	S_c	S_{rfn}
visit *Root*	$\{N_2,N_1\}$	$\emptyset$	$\emptyset$
visit N_2	$\{N_6,N_1,N_5\}$	$\emptyset$	$\emptyset$
visit N_6	$\{i,N_1,j,h,N_5\}$	$\emptyset$	$\emptyset$
process i	$\{N_1,j,h,N_5\}$	$\{i\}$	$\emptyset$
visit N_1	$\{N_4,j,N_3,h,N_5\}$	$\{i\}$	$\emptyset$
visit N_4	$\{e,j,d,N_3,h,N_5\}$	$\{i\}$	$\emptyset$
process e	$\{j,d,N_3,h,N_5\}$	$\{i,e\}$	$\emptyset$
process j	$\{d,N_3,h,N_5\}$	$\{i,e\}$	$\{j\}$
process d	$\{N_3,h,N_5\}$	$\{i,e\}$	$\{j,d\}$
process N_3	$\{h,N_5\}$	$\{i,e\}$	$\{j,d,N_3\}$
process h	$\{N_5\}$	$\{i,e\}$	$\{j,d,N_3,h\}$
process N_5	$\emptyset$	$\{i,e\}$	$\{j,d,N_3,h,N_5\}$

Table 3
Summary of symbols.

Notation	Description
D	A data set
q	A query point
T_D	The R-tree on D
$\|T_D\|$	The size of R-tree TD
e	An entry (data object or node MBR) in an R-tree
H	A heap
S_{rslt}	A query result set
S_c	A candidate set
$\|S_c\|$	The cardinality of set S_c
$NN_k(q)$	Result set of a k nearest neighbor query issued at point q
$RNN_k(q)$	Result set of a reverse k nearest neighbor query issued at point q
$MNN_{k1,k2}(q)$	Result set of a mutual nearest neighbor query with respect to k_1 and k_2 issued at point q

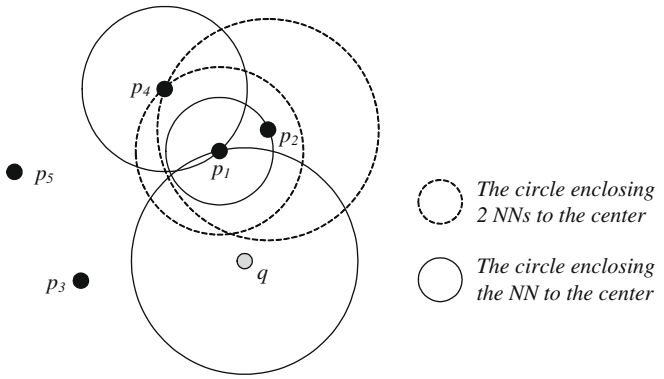


Fig. 4. Illustration of MNN queries.

3.1. Problem definition

Definition 1 (*Mutual nearest neighbor query*). Given a dataset D, a query point q, and two parameters k_1, k_2, a *mutual nearest neighbor* (MNN) query retrieves the set of objects $S \subseteq D$, such that (i) $\forall p \in S$, $p \in NN_{k1}(q)$; and (ii) $\forall p \in S$, $q \in NN_{k2}(p)$. Formally, $MNN_{k1,k2}(q) = \{p \in S | p \in NN_{k1}(q) \wedge q \in NN_{k2}(p)\}$.[8]

As defined in Definition 1, an MNN query returns all the objects in D that are among the k_1 NNs of q and meanwhile have q as one of their k_2NNs. It has two important properties, which make it different from conventional NN search.

Property 1. *MNN is symmetric. That is to say for any two given objects $o_i, o_j \in D$, and fixed k_1, k_2, if object $o_i \in MNN_{k1,k2}(o_j)$, then $o_j \in MNN_{k2,k1}(o_i)$.*

*k*NN search is asymmetric. As shown in Fig. 4, point p_4 has p_1 as its NN, while p_1 has p_2 but not p_4, as its NN. However, MNN retrieval is symmetric. For example, $MNN_{1,2}(p_1) = \{p_2\}$ indicating that $p_2 \in NN_1(p_1)$ and $p_1 \in NN_2(p_2)$. Hence, according to the definition of the MNN query, $p_1 \in MNN_{2,1}(p_2)$ satisfies as well.

Property 2. *Given a query point q, the cardinality of q's mutual nearest neighbors (MNNs), denoted by $|MNN_{k1,k2}(q)|$, varies as the data distribution of D changes.*

*k*NN search consistently returns k answer objects, whereas MNN queries with the same k_1 and k_2 issued at different query points might return result sets with different cardinalities. For instance, $|MNN_{1,2}(q)| = |\emptyset| = 0$, $|MNN_{1,2}(p_1)| = |\{p_2\}| = 1$, as depicted in Fig. 4.

Taking the definitions of MNN, *k*NN, and R*k*NN queries into consideration, we elicit the following lemma and theorems, which constitute the basis of our proposed algorithms for MNN search. Some straightforward proofs are omitted for space saving.

Lemma 1. *If the distance from each data object in D to q is unique, then the cardinality of $MNN_{k1,k2}(q)$ varies in the range of [0, k_1].*[9]

Proof. According to Definition 1, $MNN_{k1,k2}(q) = \{p \in S | p \in NN_{k1}(q) \wedge q \in NN_{k2}(p)\}$. Thus, $|MNN_{k1,k2}(q)| \leqslant |NN_{k1}(q)|$ ($=k_1$). □

Theorem 1. *$MNN_{k1,k2}(q)$ is a subset of $NN_{k1}(q)$, i.e., $MNN_{k1,k2}(q) \subseteq NN_{k1}(q)$.*

Theorem 2. *$MNN_{k1,k2}(q)$ is a subset of $RNN_{k2}(q)$, i.e., $MNN_{k1,k2}(q) \subseteq RNN_{k2}(q)$.*

3.2. Baseline algorithm for MNN search

Based on the definition of MNN query, a naive approach, called *simple processing algorithm* (SP) is proposed. It adopts a *filtering-verification* framework, i.e., first conducting a *k*NN search to retrieve the candidate set $S_c = NN_{k1}(q)$ and then verifying each candidate $c \in S_c$. The verification of a candidate c can be conducted again via a *k*NN search to check whether $q \in NN_{k2}(c)$. If yes, it means that q is among the k_2NNs of c and hence c is returned as an answer object. Otherwise, c is discarded, i.e., it is a false hit. Fig. 5 shows the pseudo-code of SP algorithm. Note that SP invokes the BF-*k*NN function, a BF algorithm for *k*NN

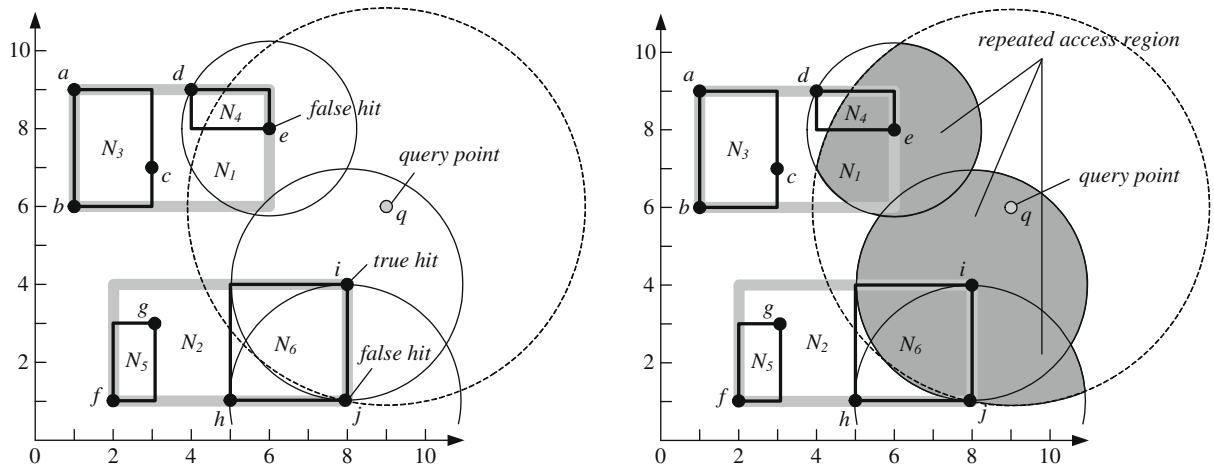
[8] The MNN search can also be formulized as $MNN_{k1,k2}(q) = \{p \in S | p \in NN_{k1}(q) \wedge p \in RNN_{k2}(q)\}$.

[9] Note that the distance between each data object in D and q might not be unique, i.e., there may have multiple data objects with the same distances to q, and thus q could have more than k_1 MNNs.

Algorithm SP (q, k_1, k_2, S_{rslt})
Input: q: a query point; k_1: the number of NNs; k_2: the number of NNs
Output: S_{rslt}: a query result set

1. perform BF-kNN (q, k_1, S_c) using min-heap H_1 // BF algorithm for kNN search proposed in [19]
2. **for** each point $c \in S_c$ **do**
3. perform BF-kNN (c, k_2, S_t) using min-heap H_2
4. **if** $q \in S_t$ **then**
5. $S_{rslt} = S_{rslt} \cup \{c\}$ // c is an actual MNN of q w.r.t. k_1 and k_2
6. **return** S_{rslt}

Fig. 5. Simple processing algorithm (SP).



(a) An $MNN_{3,1}$ query (b) The corresponding RAR in the $MNN_{3,1}$ query

Fig. 6. Example of SP algorithm.

search [19] (described in Section 2.1), to retrieve kNNs of a specified query point; and employs an *in-memory* heap to facilitate the *best-first* traversal paradigm.

Consider an $MNN_{3,1}$ query q shown in Fig. 6a, where we use the same data set as Fig. 2. The SP first calls BF-kNN to retrieve the k_1 (=3) NNs of q, i.e., $NN_3(q) = \{i,e,j\}$ (enclosed in a dotted circle). Then, SP examines each point in $NN_3(q)$. Finally, SP confirms that $i \in NN_3(q)$ is indeed an MNN of q and it is returned as an answer object; whereas points e and j in $NN_3(q)$ are false hits as $NN_1(e) = \{d\}$ and $NN_1(j) = \{i\}$.

The correctness of SP is obvious. Observe that the candidate set returned by BF-kNN is always a superset (i.e., $NN_{k1}(q)$) of the final result set (i.e., $MNN_{k1,k2}(q)$), that is, it does not incur *false misses* because $MNN_{k1,k2}(q) \subseteq NN_{k1}(q)$ according to Theorem 1. Every false hit $p \in NN_{k1}(q)$ is subsequently eliminated during the verification step by verifying whether q is among $NN_{k2}(p)$. Consequently, SP can return the *exact* set of MNNs.

Lemma 2. *The SP algorithm loads some entries (node MBRs or data objects) of the R-tree T_D from the disk multiple times.*

Proof. Since the SP algorithm requires traversing the R-tree T_D repeatedly for filtering and verifying the MNN candidates, it loads/accesses some index entries (e.g., the root of T_D) multiple times. □

As an example, Fig. 6b illustrates the *repeated access region* (RAR) (shaded area) in the filtering and verification steps of the $MNN_{3,1}$ query depicted in Fig. 6a. As seen from this diagram, SP visits entries N_1, N_2, N_6, i, j thrice, and visits entries N_4, e twice.

For the SP algorithm, let $|H_1|$ be the size of heap H_1, and $|H_2|$ be the size of heap H_2. The time and space complexities of SP algorithm are analyzed in Theorem 3.

Theorem 3. *The time and space complexities of the SP algorithm are $O((|S_c| + 1) \times \log|T_D|)$ and $O(|H_1| + |H_2|)$, respectively.*

Proof. The SP algorithm follows the filtering-verification framework. In the filtering step, SP takes $O(\log|T_D|)$ for obtaining candidate set S_c; in the verification step, SP incurs $O(|S_c| \times \log|T_D|)$ in order to check whether each candidate in S_c is an actual MNN of q. Thus, the time complexity of the algorithm is $O((|S_c| + 1) \times \log|T_D|)$. The storage of the SP algorithm is dominated

by heap H_1 (used in the filtering step) and heap H_2 (utilized in the verification step). Hence, the space complexity of the algorithm is $O(|H_1| + |H_2|)$. □

SP is very inefficient in terms of I/O overhead and CPU cost, especially for large values of k_1 and k_2, as also demonstrated by our experimental results to be presented in Section 5. To overcome this deficiency, we propose four algorithms to improve the performance of MNN query processing via different optimization techniques.

4. Optimizations

In this section, we focus on the optimizations of MNN query. Our objective is to reduce the number of node accesses (i.e., I/O cost) and speed up search performance accordingly. For this purpose, several enhanced algorithms for MNN search, namely *two-step algorithm* (TS), *reuse two-heap algorithm* (RTH), *algorithm using NN search with pruning* (NNP), and *algorithm using RNN search with pruning* (RNNP), are developed.

4.1. Two-step algorithm

The SP algorithm has to verify every single object included in the candidate set S_c, and thus the verification process may need access the dataset $|S_c|$ times. Since the only objective of the verification step is to validate whether each candidate in S_c has a specified query point q as one of its k_2NNs, we can issue an Rk_2NN query at point q to find out the set of objects that have q as one of their k_2NNs, which is guaranteed to be a superset of the final result set of MNN retrieval, i.e., $MNN_{k1,k2}(q) \subseteq RNN_{k2}(q)$, as stated in Theorem 2. In view of this, we propose *two-step algorithm* (TS), and the pseudo-code of TS algorithm is depicted in Fig. 7.

Based on Definition 1, $MNN_{k1,k2}(q) = \{p \in S | p \in NN_{k1}(q) \wedge p \in RNN_{k2}(q)\}$. Consequently, the correctness of TS algorithm is evident. On the other hand, TS has to scan the dataset *twice*, one for *k*NN search and the other for R*k*NN search. Compared with SP algorithm which needs access the dataset ($|S_c| + 1$) times, TS can decrease the I/O overhead, especially when k_1 is very large and k_2 is very small, i.e., $k_1 \gg k_2$.

4.2. Reuse two-heap algorithm

Our second algorithm tries to improve the performance of SP using different optimization techniques. The SP algorithm employs two heaps: (i) heap H_1 used by the function BF-k_1NN for retrieving the candidate set S_c of MNNs (Line 1 of SP); and (ii) heap H_2 utilized by the function BF-k_2NN for verifying each candidate c in S_c (Line 3 of SP). As mentioned earlier, SP has to visit some nodes (e.g., *Root* node of T_D) *multiple times*. Motivated by this observation, an algorithm, namely *reuse two-heap algorithm* (RTH), is proposed, which (i) attempts to fully use *locally available nodes* (e.g., those nodes in H_1 and H_2) in order to reduce the redundant node accesses, and (ii) develops an *early termination condition* so that the verification process of a candidate $c \in S_c$ may be terminated earlier without finding all the k_2NNs of c. Fig. 8 presents the pseudo-code of RTH algorithm.

Algorithm TS (q, k_1, k_2, S_{rslt})
Input: q: a query point; k_1: the number of NNs; k_2: the number of NNs
Output: S_{rslt}: a query result set

1. perform BF-*k*NN (q, k_1, S_c) using min-heap H_1
2. perform TPL-R*k*NN (q, k_2, S_t) using min-heap H_2 // TPL algorithm for R*k*NN search proposed in [41]
3. **return** $S_{rslt} = S_c \cap S_t$

Fig. 7. Two-step algorithm (TS).

Algorithm RTH (q, k_1, k_2, S_{rslt})
Input: q: a query point; k_1: the number of NNs; k_2: the number of NNs
Output: S_{rslt}: a query result set

1. initialize a min-heap H accepting entries of the form (e, *key*) and $S_{rslt} = \emptyset$
2. perform BF-*k*NN (q, k_1, S_c) using H
3. $S_{temp} = S_c$ // for reusing the data objects in S_c that have been accessed in the filtering step
4. **for** each point $c \in S_c$ **do**
5. Verify (c, k_2, S_{temp}, H, S_{rslt}) // see Figure 9
6. **return** S_{rslt}

Fig. 8. Reuse two-heap algorithm (RTH).

Algorithm Verify (p, k, S_{temp}, H, S_{rslt})
Input: p: the candidate object that has not been verified so far; k: the number of NNs;
S_{temp}: an auxiliary set; H: a heap; S_{rslt}: a result set

1. $S_{temp} = S_{temp} \cup (\cup_{e \in H} e)$, $H = \emptyset$, $cnt = 0$, and $contflag$ = TRUE
2. **for** each entry $e \in S_{temp}$ **do**
3. insert (e, $mindist(e, p)$) into H
4. $S_{temp} = \emptyset$ // for the next round
5. **while** H is not empty and $contflag$ = TRUE **do**
6. de-heap the top entry (e, $mindist(e, p)$) from H
7. **if** e is a data object o **then**
8. $S_{temp} = S_{temp} \cup \{o\}$
9. **if** $o \neq p$ **then**
10. $cnt = cnt + 1$
11. **if** $dist(o, p) \geq dist(q, p)$ **then**
12. $contflag$ = FALSE and $S_{rslt} = S_{rslt} \cup \{p\}$ // p is an actual MNN of q
13. **if** $cnt = k$ **then**
14. $contflag$ = FALSE // p is not an actual MNN of q
15. **else** // e is an intermediate entry
16. **for** each entry $e_i \in e$ **do**
17. insert (e_i, $mindist(e_i, p)$) into H
18. **if** H is empty and $contflag$ = TRUE **then**
19. $S_{rslt} = S_{rslt} \cup \{p\}$ // p is an actual MNN of q

Fig. 9. The Verify algorithm.

Similar as SP algorithm, RTH utilizes BF-kNN to retrieve the candidate set S_c (Line 2), and then verifies each candidate in S_c (Lines 4–5) via Verify algorithm (shown in Fig. 9). Instead of scanning the dataset *from scratch* like the verification step of SP does, Verify starts the traversal based on a local view of the dataset (Lines 1–3 of Verify). Initially, only the root node is known. After the processing of BF-kNN algorithm, more knowledge of the data distribution (i.e., nodes of finer granularity) preserved by the heap H and the candidate set S_c is obtained. Consequently, only the accesses to those nodes not locally available are necessary. Since Verify continuously evaluates candidate objects, the access to some nodes will be triggered and thus more and more knowledge of the underlying dataset is accumulated. However, the whole dataset is only visited *once* even in the worst-case scenario. Compared with SP, RTH reduces the traversal of the dataset from ($|S_c| + 1$) times to once. In addition, once RTH encounters a data object o such that $o \in NN_{k2}(c)$ and $dist(o,c) \geqslant dist(q,c)$, candidate $c \in S_c$ is for sure one of the k_2NNs to q, and hence the verification process of c can be terminated without finding all the k_2NNs of c. This early termination condition can improve the search performance further.

It is worth noting that the auxiliary set S_{temp} is reset to $\emptyset$ in the Line 4 of Verify algorithm. This initialization is required in order to avoid storing some unnecessary entries for every verification round. Moreover, it is possible that the number of NNs of a candidate c is smaller than a specified k value, if the cardinality of dataset is smaller than k. This necessitates the operations involved in the Lines 18 and 19 of Verify algorithm.

Fig. 10 presents an example. Assume an $MNN_{3,1}$ query is issued at the point q (as shown in Fig. 6a). After the filtering step, $\{i,e,j\}$ has been identified as the candidate set S_c and H (=$\{d,N_3,h,N_5\}$) represents the current view of the dataset (as illus-

View$_0$: right after the filtering step	
$H =$	$S_{temp} = \{i, e, j\}$
View$_1$: after performing the Lines 1-3 of Verify *algorithm when verifying i*	
$H =$	$S_{temp} = \{i, e, j, d, N_3, h, N_5\}$
View$_2$: after performing the Lines 4-19 of Verify *algorithm when verifying i*	
$H =$	$S_{temp} = \{i, j\}$

Fig. 10. Example of RTH algorithm.

trated in Fig. 10: $View_0$). In the subsequent verification step, candidate objects in S_c are verified one by one. When $i \in S_c$ is verified, RTH first sorts all the locally available objects/nodes in ascending order of their minimal distances to i, and en-heaps H there (as shown in Fig. 10: $View_1$). Thereafter, RTH de-heaps top entry from H for evaluation. As for candidate i, it can be confirmed to be an actual answer object once object j (i.e., the entry in H having the second smallest distance to i) is accessed due to $dist(i,j) > dist(i,q)$. The situation of H and S_{temp} after verifying i is depicted in Fig. 10: $View_2$. Compared with SP algorithm, RTH does not incur any extra node access for validating i. Similarly, RTH can confirm that objects e and j are not the final answer objects based on local knowledge, i.e., $dist(e,q) > dist(e,d)$ and $dist(j,q) > dist(j,i)$.

As RTH shares the same processing method as SP, we ignore the proof of its correctness. However, it significantly reduces the number of nodes accesses, as pointed out in Lemma 3.

Lemma 3. *The RTH algorithm loads any entry (node MBR or data object) in the R-tree T_D from the disk only once.*

Proof. The lemma is correct because RTH stores and reuses all the entries (containing node MBRs and data objects) that have been visited so far during query processing. □

4.3. Algorithm using NN search with pruning

Although TS and RTH reduce the I/O overhead by using batch processing and reusing technology, there is still room for performance improvement based on the following observations. TS, no matter what is the value of k_1, issues an Rk_2NN query to retrieve all the objects that have q as one of their k_2NNs. Nevertheless, only those objects that belong to $NN_{k1}(q)$ will be included in the final result set. When $k_1 \ll k_2$, TS might suffer from expensive RkNN query cost. If RkNN retrieval only considers those objects included in $NN_{k1}(q)$, the search performance may be improved. On the other hand, RTH takes $NN_{k1}(q)$ as the candidate set because $NN_{k1}(q) \supseteq MNN_{k1,k2}(q)$ according to Theorem 1 (presented in Section 3.1). However, $NN_{k1}(q)$ may contain some false hits that cannot become actual answer objects. Therefore, if we can prune away those false hits in the filtering stage, the overall cost of verification step can be decreased. Our third algorithm, namely *algorithm using NN search with pruning* (NNP), is inspired by these two observations. It incorporates several pruning strategies during the search. Fig. 11 depicts the pseudo-code of NNP algorithm.

NNP is similar to RTH, but it employs pruning heuristics at two places to improve the search performance. The first pruning is integrated with kNN search (Line 3), handled by NNP-Finding algorithm; and the second one is a *self-pruning* (Lines 5 and 6). The main target is to remove those candidates that definitely will not belong to $RNN_{k2}(q)$. As the self-pruning is very straightforward, we only explain the NNP-Finding algorithm, which is presented in Fig. 12.

NNP-Finding performs a BF algorithm for kNN ($k = k_1$) search with respect to q and meanwhile it enables TPL pruning techniques via TPL-k-Trim algorithm (proposed in [41]). As presented in [41], TPL-k-Trim takes as input a query point q, a parameter k, a candidate set S_c, and an entry e, and it determines whether the entry e is closer to at least k objects in S_c than to q. If yes, $q \notin NN_k(e)$, i.e., $e \notin RNN_k(q)$, and TPL-k-Trim returns ∞. Otherwise, e cannot be pruned. NNP-Finding visits the nodes/points based on ascending order of their distances to q. If the accessed entry refers to a data object o, it invokes TPL-k-Trim to examine whether o can be discarded, with pruned objects preserved in the refine set S_{rfn} and un-pruned objects preserved in the candidate set S_c (Lines 6–13). Otherwise, the accessed entry must be an intermediate (i.e., a non-leaf) node and its child entries are de-heaped for later examinations (Lines 14–16). Thereafter, NNP-Finding checks its early termination condition, i.e., whether any un-examined object $o' \in NN_{k1}(q)$. The main idea is to find out a node/point e' in H that is the closest to q and meanwhile cannot be pruned by TPL-k-Trim algorithm, and then count the number of objects that are for sure closer to q than e' (Lines 17–25). Suppose set S contains the nodes/points that are closer to q than e', we adopt a conservative approach to estimate $|S|$ based on $\sum_{o_i \in S}(f_{\min}(o_i))^l$ in order to avoid any false miss. Here, f_{min} is the minimum node fanout (e.g., 40% of the node capacity), and l is the level of any object $o_i \in S$ (counting from the leaf level as level 0). If the number exceeds k_1, e' and

Algorithm NNP (q, k_1, k_2, S_{rslt})
Input: q: a query point; k_1: the number of NNs; k_2: the number of NNs
Output: S_{rslt}: a query result set

1. $S_{rslt} = S_c = S_{rfn} = S_{temp} = \varnothing$
2. initialize a min-heap H accepting entries of the form (e, key)
3. perform NNP-Finding (q, k_1, k_2, S_c, S_{rfn}) using H // see Figure 12
4. $S_{temp} = S_c \cup S_{rfn}$ // for reusing all the data objects and nodes that have been accessed in the filtering step
5. **if** $|S_c| > 1$ **then** // conduct self-pruning in $NN_{k1}(q)$ for eliminating false hits
6. prune $\forall\, c \in S_c$ where there exists k_2 candidates $c' \in S_c$ satisfying $dist(c', c) < dist(q, c)$
7. **for** each point $c \in S_c$ **do**
8. Verify (c, k_2, S_{temp}, H, S_{rslt}) // see Figure 9
9. **return** S_{rslt}

Fig. 11. Algorithm using NN search with pruning (NNP).

Algorithm NNP-Finding (q, k_1, k_2, S_c, S_{rfn})
Input: q: a query point; k_1: the number of NNs; k_2: the number of NNs; S_c: the set of candidates that have not been verified; S_{rfn}: the set of points pruned by the TPL pruning technique

1. $cnt = 0$, $PruneDist = \infty$, $S = \emptyset$
2. initialize a min-heap H accepting entries of the form (e, key)
3. insert (R-tree root, 0) into H
4. **while** H is not empty **do**
5. de-heap the top entry (e, $mindist(e, q)$) from H
6. **if** e is a data object o and $o \neq q$ **then**
7. **if** TPL-k-Trim (q, k_2, S_c, o) $\neq \infty$ **then** // TPL-k-trim algorithm proposed in [41]
8. $S_c = S_c \cup \{o\}$ // o is a qualifying candidate
9. **else**
10. $S_{rfn} = S_{rfn} \cup \{o\}$ // o is a non-qualifying candidate
11. $cnt = cnt + 1$
12. **if** $cnt = k_1$ **then**
13. **break** // terminate algorithm
14. **else** // e is an intermediate entry
15. **for** each entry $e_i \in e$ **do**
16. insert (e_i, $mindist(e_i, q)$) into H
17. find the first entry (e', $mindist(e', q)$) in H having TPL-k-Trim (q, k_2, S_c, e') $\neq \infty$
18. **if** e' exists **then** // e' may contain (be) one of the RkNNs ($k = k_2$) of q
19. $PruneDist = mindist(e', q)$
20. **else** // e' does not exist
21. $PruneDist = \infty$
22. find the set $S = \{e'' \in H \mid maxdist(e'', q) < PruneDist\}$ from H
23. **if** $ObjNum(S) \geq k_1$ **then** // $ObjNum(S)$: the number of data objects covered by all entries in S
24. **break** // terminate algorithm
25. $S = \emptyset$ // for the next round

Fig. 12. The NNP-Finding algorithm.

all the remaining entries in H will not become/contain a candidate object because e' is guaranteed not to contribute to $NN_{k1}(q)$. Consequently, NNP-Finding can be terminated.

In order to facilitate the understanding of NNP-Finding, we take an $MNN_{3,1}$ query issued at q, as depicted in Fig. 6a, as an illustrative example. The trace is shown in Fig. 13, where the distances maintained in H are omitted for simplicity. Initially, NNP-Finding accesses the root node and inserts its entries N_1, N_2 into heap H, sorted in ascending order of their *mindist* to q. Thereafter, it continuously de-heaps the top entry from H for evaluation until the termination condition is satisfied: (i) the cardinality of the candidate set S_c reaches k_1, i.e., $|S_c| = 3$; or (ii) no remaining entry in H can contain/be the object that belongs to $NN_3(q)$.

The first de-heaped entry is N_2, and its child nodes are inserted into $H = \{N_6, N_1, N_5\}$. As current $S_c = \emptyset$ and thus the next entry N_6 might contain the objects that can contribute to $NN_3(q)$, the evaluation continues. N_6 is evaluated and its child entries (i.e., objects h, i, j) are en-heaped H. The algorithm then discovers the first data point i, and keeps it in the candidate set $S_c = \{i\}$. It proceeds to the expansion of nodes N_1, N_4, and then encounters data point e. As e cannot be pruned, it is added to the candidate set with $S_c = \{i, e\}$. Since all the remaining nodes/points in $H = \{j, d, N_3, h, N_5\}$ are closer to either i or e than to q,

Action	H	e'	*PruneDist*	S_c	S_{rfn}	S	*ObjNum*(S)	*cnt*
visit root	N_2 N_1	$\emptyset$	∞	$\emptyset$	$\emptyset$	$\emptyset$	0	0
visit N_2	N_6 N_1 N_5	N_6	$\sqrt{5}$	$\emptyset$	$\emptyset$	$\emptyset$	0	0
visit N_6	i N_1 j h N_5	i	$\sqrt{5}$	$\emptyset$	$\emptyset$	$\emptyset$	0	0
visit i	N_1 j h N_5	N_1	$\sqrt{9}$	$\{i\}$	$\emptyset$	$\emptyset$	0	1
visit N_1	N_4 j N_3 h N_5	N_4	$\sqrt{13}$	$\{i\}$	$\emptyset$	$\emptyset$	0	0
visit N_4	e j d N_3 h N_5	e	$\sqrt{13}$	$\{i\}$	$\emptyset$	$\emptyset$	0	0
visit e	j d N_3 h N_5	*not found*	∞	$\{i, e\}$	$\emptyset$	$\{j, d, N_3, h, N_5\}$	5	2

Return $S_{cnd} = \{i, e\}$, $S_{rfn} = \emptyset$, and terminate when $ObjSum\ (S) \geq k_1\ (= 3)$

Fig. 13. The trace of NNP-Finding algorithm.

none of them can become/contain the candidate object, i.e., $PruneDist = \infty$ and $S = H$. In other words, $ObjNum(S) = 5 \geqslant k_1$ (=3), and hence the algorithm can be terminated, after which $S_c = \{i, e\}$ and $S_{rfn} = \emptyset$.

Similar as RTH, NNP reuses all the locally available entries during query processing. Therefore, NNP loads any entry from the disk at most *once*. Compared with RTH, NNP incorporates a pruning technique into the filtering step. It is guaranteed that the early termination does not miss any real answer object, as demonstrated in Theorem 4. Consequently, NNP can also ensure the correctness of the search.

Theorem 4. *The* NNP-Finding *algorithm does not miss any answer object of an MNN query.*

Proof. NNP-Finding adopts a normal BF-*k*NN search (i.e., terminates the search when the cardinality of the candidate set S_c reaches k_1, i.e., $|S_c| = k_1$), but it meanwhile enables an early termination when no remaining entry in H can contain/be the object that belongs to $NN_{k1}(q)$. Now we need to validate that the early termination will not miss any actual answer object. Without loss of generality, we assume the algorithm early terminates after entry e is de-heaped. Consequently, there must be an entry $e' \in H$ that cannot be pruned by TPL-*k*-Trim and meanwhile has $ObjNum(S) \geqslant k_1$ with $S = \{e'' \in H | maxdist(e'', q) < mindist(e', q)\}$. We assume the early termination condition might cause some false miss, and assume an answer object $o \in e_o \in H$ is missed. Since e' is the first entry in H that cannot be pruned away, $maxdist(s, q) < mindist(e', q) \leqslant mindist(e_o, q) \leqslant dist(o, q)$ holds for $\forall s \in S$. As $ObjNum(S) \geqslant k_1$, it is confirmed that at least k_1 objects are closer to q than o, and hence $o \notin NN_{k1}(q)$. This finding contradicts our assumption, and thus the proof completes. □

4.4. Algorithm using RNN search with pruning

As defined in Definition 1, $MNN_{k1,k2}(q) = \{p \in S | p \in NN_{k1}(q) \wedge p \in RNN_{k2}(q)\}$. Our previous algorithms form the candidate set S_c based on $NN_{k1}(q)$ and then verify each candidate c in S_c based on the fact that whether $c \in RNN_{k2}(q)$. However, when $|RNN_{k2}(q)| < |NN_{k1}(q)|$, it is more beneficial to constitute the candidate set based on $RNN_{k2}(q)$ but not $NN_{k1}(q)$, especially when $k_1 \gg k_2$. Our fourth algorithm, namely *algorithm using RNN search with pruning* (RNNP), is motivated by this observation. Fig. 14 shows the pseudo-code of RNNP algorithm. The logic is very similar as NNP algorithm, but it calls RNNP-Finding algorithm to form the candidate set (Line 3). Subsequently, it checks whether each candidate object $c \in NN_{k1}(q)$ for verification, by invoking NN-Verify algorithm (Line 5).

Fig. 15 depicts the pseudo-code for RNNP-Finding algorithm. The basic idea is to retrieve those objects p whose $NN_{k2}(p)$ includes q, i.e., $RNN_{k2}(q)$, based on TPL-*k*-Trim and TPL-***k***-Refinement algorithms as in [41]. Nevertheless, different from conventional R*k*NN search, it takes the characteristic of MNN search into consideration and tries to exclude $p \notin NN_{k1}(q)$ from the candidate set, i.e., $\exists S$ such that $|S| \geqslant k_1$ and $\forall s \in S, dist(s, q) < dist(p, q)$. As RNNP-Finding algorithm accesses the objects according to ascending order of their distances to q, the search can be safely terminated once a de-heaped object o is found to be closer to k_1 other objects than q, i.e., $o \notin NN_{k1}(q)$ (Lines 10–12). The reason behind is that all the remaining objects in H (i.e., those unvisited objects) are for sure not included in $NN_{k1}(q)$ (as proved in Theorem 5).

Note that in RNNP-Finding, TPL-*k*-Trim is applied twice for each node e: (i) when e is expanded (Line 5), and (ii) when e is de-heaped from heap H (Line 16). The second test is necessary, since e may be pruned by some candidate that was discovered after the insertion of e into H. In addition, as with RTH and NNP, RNNP reuses all the entries (involving nodes and data objects) that have been accessed during query processing. Therefore, RNNP also loads any entry from the disk at most *once*.

Theorem 5. *The early termination condition of* RNNP-Finding *algorithm (shown in Lines 10–12) does not miss any actual answer object of MNN search.*

Proof. Without loss of generality, we assume at least one answer object $o' \in MNN_{k1,k2}(q)$ is missed due to the early termination condition of RNNP-Finding algorithm. In other words, $o' \in NN_{k1}(q) \wedge q \in NN_{k2}(o')$ according to Definition 1. We further assume that the early termination condition of RNNP-Finding algorithm is satisfied when an object $o$is evaluated. As o' is missed, o' should not have been visited, i.e., o' is still in the heap H with $mindist(o', q) > mindist(o, q)$. As for $\forall s \in S$, *max-*

Algorithm RNNP (q, k_1, k_2, S_{rslt})
Input: q: a query point; k_1: the number of NNs; k_2: the number of NNs
Output: S_{rslt}: a query result set

1. $S_{rslt} = S_c = S_{rfn} = S_{temp} = \emptyset$
2. initialize a min-heap H accepting entries of the form (e, key)
3. perform RNNP-Finding (q, k_1, k_2, S_c, S_{rfn}) using H // see Figure 15
4. $S_{temp} = S_c \cup S_{rfn}$ // for reusing all the data objects and nodes that have been accessed in the filtering step
5. NN-Verify (S_c, q, k_1, S_{temp}, H, S_{rslt}) // see Figure 16
6. **return** S_{rslt}

Fig. 14. Algorithm using RNN search with pruning (RNNP).

Algorithm RNNP-Finding (q, k_1, k_2, S_c, S_{rfn}, S_{temp})
Input: q: a query point; k_1: the number of NNs; k_2: the number of NNs; S_c: the set of candidates that have not been verified; S_{rfn}: the set of points pruned by the TPL pruning technique; S_{temp}: an auxiliary set

1. $S = \emptyset$ and initialize a min-heap H accepting entries of the form (e, key)
2. insert (R-tree root, 0) into H
3. **while** H is not empty **do**
4. de-heap the top entry (e, $mindist(e, q)$) from H
5. **if** TPL-k-Trim (q, k_2, S_c, e) = ∞ **then**
6. $S_{rfn} = S_{rfn} \cup \{e\}$
7. **else** // e may contain/be a candidate
8. **if** e is a data object o and $o \neq q$ **then**
9. $S_c = S_c \cup \{o\}$
10. $S = \{e' \in (S_c \cup S_{rfn} \cup H) \mid maxdist(e', q) < mindist(o, q)\}$
11. **if** $ObjNum(S) \geq k_1$ **then**
12. **break** // terminate the filter step of RkNN-Pruning algorithm
13. $S = \emptyset$ // for the next round
14. **else** // e is an intermediate entry
15. **for** each entry $e_i \in e$ **do**
16. $mindist(e_i^{resM}, q)$ = TPL-k-Trim (q, k_2, S_c, e_i)
17. **if** $mindist(e_i^{resM}, q) = \infty$ **then**
18. $S_{rfn} = S_{rfn} \cup \{e_i\}$
19. **else**
20. insert (e_i, $mindist(e_i^{resM}, q)$) into H
21. TPL-k-Refinement (q, k_2, S_c, S_{rfn}) // TPL-k-Refinement algorithm proposed in [41]

Fig. 15. The RNNP-Finding algorithm.

$dist(s,q) < mindist(o,q)$, and hence it is obvious that for $\forall s \in S$, $maxdist(s,q) < mindist(o',q)$. Since $ObjNum(S) \geqslant k_1$, at least k_1 objects in S are closer to q than o', and o' for sure does not belong to $NN_{k1}(q)$, i.e., $o' \notin NN_{k1}(q)$, which contradicts our assumption and completes the proof. □

Fig. 16 presents the pseudo-code for NN-Verify algorithm. The basic idea is to conduct a kNN search at the query point q. Whenever a data object o that is included in the candidate set S_c is retrieved, it is added to the result set S_{rslt} due to the fact that $o \in NN_{k1}(q)$ and $o \in RNN_{k2}(q)$ (Lines 8 and 9). NN-Verify can be safely terminated when (i) the kNNs of q have been retrieved (Lines 10 and 11); or (ii) all the data objects contained in S_c have been verified (Lines 10 and 11); or (iii) heap H be-

Algorithm NN-Verify (S_c, q, k, S_{temp}, H, S_{rslt})
Input: S_c: the candidate set containing all the data objects that have not been verified so far; q: a query point; k: the number of NNs; S_{temp}: an auxiliary set; H: a heap; S_{rslt}: a result set

1. $H = \emptyset$, $cnt = 0$
2. **for** each entry $e \in S_{temp}$ **do**
3. insert (e, $mindist(e, q)$) into H
4. **while** H is not empty **do**
5. de-heap the top entry (e, $mindist(e, q)$) from H
6. **if** e is a data object o **then**
7. $cnt = cnt + 1$
8. **if** $o \in S_c$ **then**
9. $S_{rslt} = S_{rslt} \cup \{o\}$ // o is indeed an MNN of q
10. **if** $cnt = k$ **or** $|S_{rslt}| = |S_c|$ **then**
11. **return** // terminate algorithm
12. **else** // e is an intermediate entry
13. **for** each entry $e_i \in e$ **do**
14. insert (e_i, $mindist(e_i, q)$) into H
15. **if** H is empty **then**
16. **return** // terminate algorithm

Fig. 16. The NN-Verify algorithm.

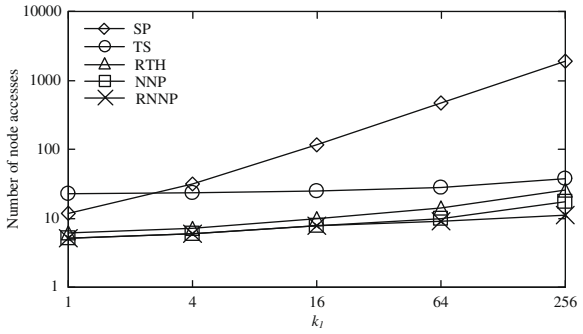
comes empty (Lines 15 and 16). Notice that NN-Verify verifies every candidate in S_c by reusing all the entries (including all the candidate objects in S_c and all the pruned objects/nodes maintained in S_{rfn}) that have been accessed in the filtering step of RNNP (Lines 2 and 3).

5. Experimental evaluation

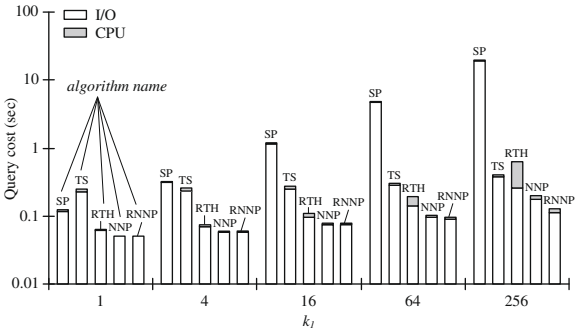
In this section, the efficiency and effectiveness of our proposed MNN query processing algorithms (including SP, TS, RTH, NNP, and RNNP) are evaluated through extensive experiments. All the algorithms are implemented in C++, and the experiments are conducted on a PC with Pentium IV 3.0 GHz CPU and 2GB main memory, running Microsoft Windows XP Professional Edition. We first describe the experimental settings in Section 5.1, and then present the experimental results and our findings in Section 5.2.

5.1. Experimental setup

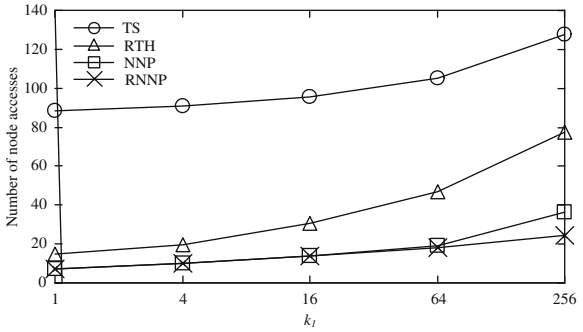
We utilize both real and synthetic data sets in the experiments. Three real datasets are deployed. Specifically, *LB* contains 2D points representing 123,593 geometric locations in Long Beach County; *Wave* includes 3D points representing 60,000



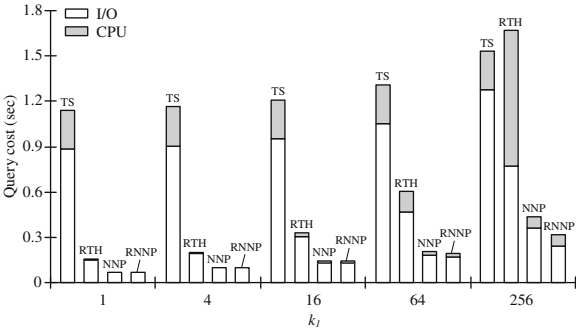
(a) I/O cost on *LB*



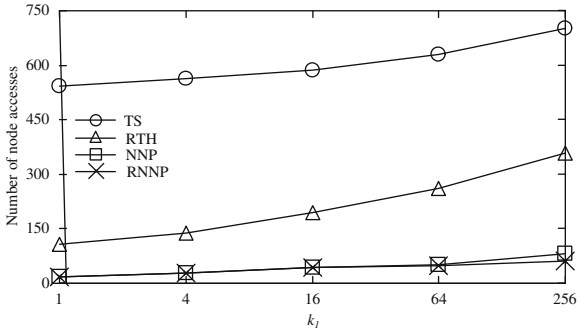
(b) Query cost on *LB*



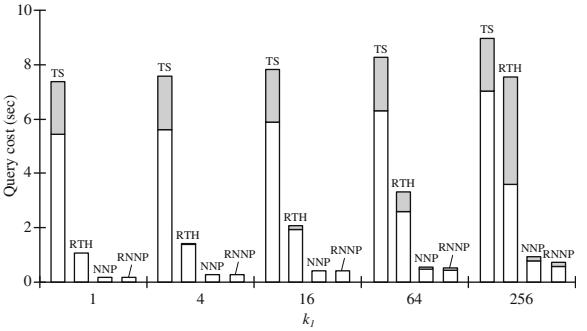
(c) I/O cost on *Wave*



(d) Query cost on *Wave*



(e) I/O cost on *Color*



(f) Query cost on *Color*

Fig. 17. Performance vs. k_1 ($k_2 = 16$).

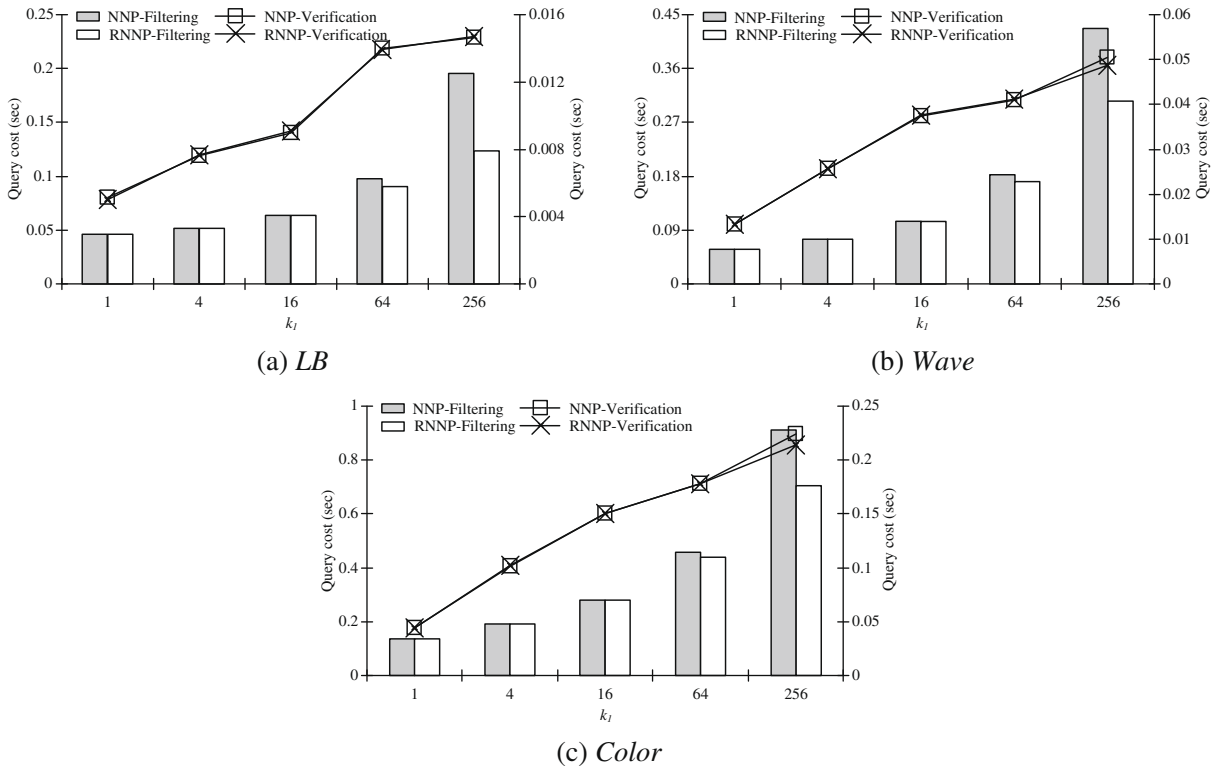


Fig. 18. Filtering and verification step costs of NNP vs. those of RNNP (varying k_1 and $k_2 = 16$).

Table 4
The maximal number of entries in the heap vs. k_1 ($k_2 = 16$) on *LB*, *Wave*, and *Color* datasets respectively.

k_1	*LB*					*Wave*					*Color*				
	SP	TS	RTH	NNP	RNNP	SP	TS	RTH	NNP	RNNP	SP	TS	RTH	NNP	RNNP
1	175	163	177	139	139	344	316	347	140	140	1888	1687	1944	294	294
4	198	182	206	162	162	414	364	459	188	188	2212	2035	2528	475	475
16	247	219	296	207	207	541	473	733	244	244	2632	2485	3565	695	695
64	308	292	445	240	224	679	666	1130	342	319	3229	3224	4802	835	784
256	438	438	827	346	232	1026	1026	1868	573	369	4398	4398	6653	1228	819

measurements of wave directions at the National Buoy Center; and *Color* involves 4D vectors representing the color histograms of 65,000 images.[10] We also create several synthetic datasets with dimensionality varying from two to five and cardinality changing between 128 K and 2048 K, following uniform and zipf (with skew coefficient $\alpha = 0.8$) distributions. For all the datasets, each dimension of the data space is normalized to range [0,10,000], and we assume a point's coordinates on various dimensions are mutually independent.

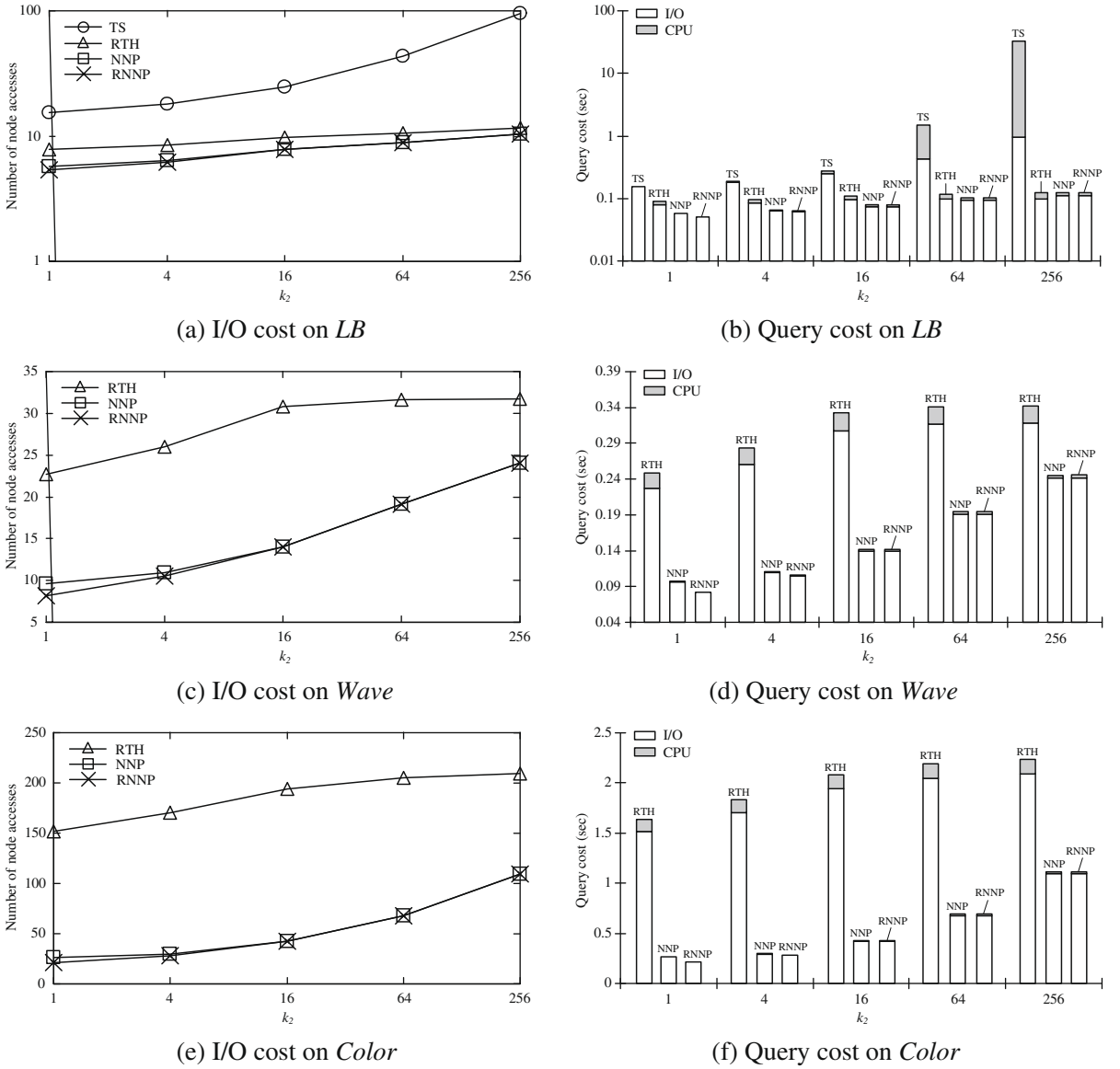
All the datasets are indexed by R*-trees [3] with page size of 1 K bytes (we choose a smaller page size to simulate practical scenarios where the dataset cardinality is much larger, as [41]). The experiments investigate the influence of different factors, including (i) value of k_1, (ii) value of k_2, (iii) dimensionality, (iv) dataset cardinality, and (v) buffer size. The performance metrics are the number of node/page accesses (i.e., I/O overhead), query cost (i.e., the sum of the I/O time and CPU time, where the I/O time is computed by charging 10 ms for each page access, as in [41]), and the maximum number of entries in the heap (as the heap storages dominate the space complexities of our proposed algorithms). Each reported value in the following diagrams is the average performance of 200 queries. The query points are randomly chosen from the set of data points, so that the queries follow the underlying dataset distribution. Unless specifically stated, the size of LRU buffer is 0 in the experiments, i.e., the I/O cost is determined by the number of nodes accessed.

[10] The *LB*, *Wave*, and *Color* datasets are available at the following sites: http://www.census.gov/geo/www/tiger/, http://www.ndbc.noaa.gov, and http://www.cs.cityu.edu.hk/~taoyf/ds.html, respectively.

5.2. Performance study

The first set of experiments studies the effect of k_1 on the efficiency of the algorithms using the real datasets. We fix k_2 to 16 and vary k_1 between 1 and 256 to measure the performance, with the number of node accesses and query cost (in seconds) depicted in Fig. 17. Here, query costs for SP, RTH, NNP, and RNNP are broken into two components, corresponding to the I/O cost and the CPU cost, respectively. The cost increases with k_1 since the number of MNN candidates escalates as k_1 grows. We observe that only the SP algorithm suffers from an exponential performance downgrade and it performs the worst in all the cases. This is because it requires multiple traversals of the R-tree. Thus, SP is omitted from the remaining experiments.

An interesting observation is that NNP and RNNP have the similar performance in most cases, but RNNP is slightly better than NNP when $k_1 \gg k_2$ (e.g., $k_1 = 256$ and $k_2 = 16$), as also demonstrated in the subsequent experiments. The reason behind is that NNP takes $NN_{k1}(q)$ as the candidate set, while RNNP forms the candidate set based on $RNN_{k2}(q)$. When $k_2 \ll k_1$, it is expected that $|RNN_{k2}(q)| < |NN_{k1}(q)|$. In order to further understand the difference between the NNP and RNNP, we divide the query cost into the two parts, corresponding to the filtering step and the verification step, respectively. The experimental results are plotted in Fig. 18, where NNP-Filtering (RNNP-Filtering) and NNP-Verification (RNNP-Verification) represent



(a) I/O cost on *LB*

(b) Query cost on *LB*

(c) I/O cost on *Wave*

(d) Query cost on *Wave*

(e) I/O cost on *Color*

(f) Query cost on *Color*

Fig. 19. Performance vs. k_2 ($k_1 = 16$).

the filtering step and verification step of NNP (RNNP), respectively. Notice that, when $k_1 \gg k_2$ (e.g., $k_1 = 256$ and $k_2 = 16$), the filtering step cost of RNNP is lower than that of NNP for all datasets as $|RNN_{k2}(q)| < |NN_{k1}(q)|$ holds.

However, both NNP and RNNP outperform the other algorithms (including SP, TS, and RTH) by several orders of magnitude in all cases, especially when k_1 is large. For example, in Fig. 17b RNNP improves the query cost by 153/3/5 times, compared against SP/TS/RTH for *LB* dataset with $k_1 = 256$ and $k_2 = 16$. This is because, both NNP and RNNP enable TPL pruning techniques to discard unnecessary entries, and hence accelerate the search. On the other hand, although TS and RTH perform not as good as NNP and RNNP, they are still much better than SP, owing to the reuse technique that significantly reduces the number of nodes accessed. It is worth noting that when $k_1 = 256$ and $k_2 = 16$, TS performs better than RTH in terms of query cost under *LB* and *Wave* datasets, as shown in Fig. 17b and d, respectively. This occasional case is due to the fact that $|RNN_{k2}(q)| \ll |NN_{k1}(q)|$. In addition, both NNP and RNNP consistently outperform TS in Fig. 17, because they integrate the filtering step and the verification step seamlessly, and try to prune away unqualified candidates as soon as possible to further improve the search performance.

Table 4 shows the maximum number of entries in the heap (denoted as n) of different algorithms with respect to k_1, which causes the major run-time memory consumption. Let *dim* be the dimensionality. In our experiments, we allocate 4, $4 \times 2 \times dim$, 4, 4, and 4 bytes to items (contained in a heap entry) ID, coordinate, level, key/distance, and pointer, respectively. Thus, the size of each heap entry (denoted by m) equals 32, 40, 48, and 56 bytes for dimensionalities 2, 3, 4, and 5, respectively. It needs to point out that the maximal heap sizes (calculated as $(m \times n)$ bytes) of all the algorithms are almost *negligible* compared with the R-tree size. As an example, for *Color* dataset with $k_1 = 256$ and $k_2 = 16$, RTH algorithm consumes

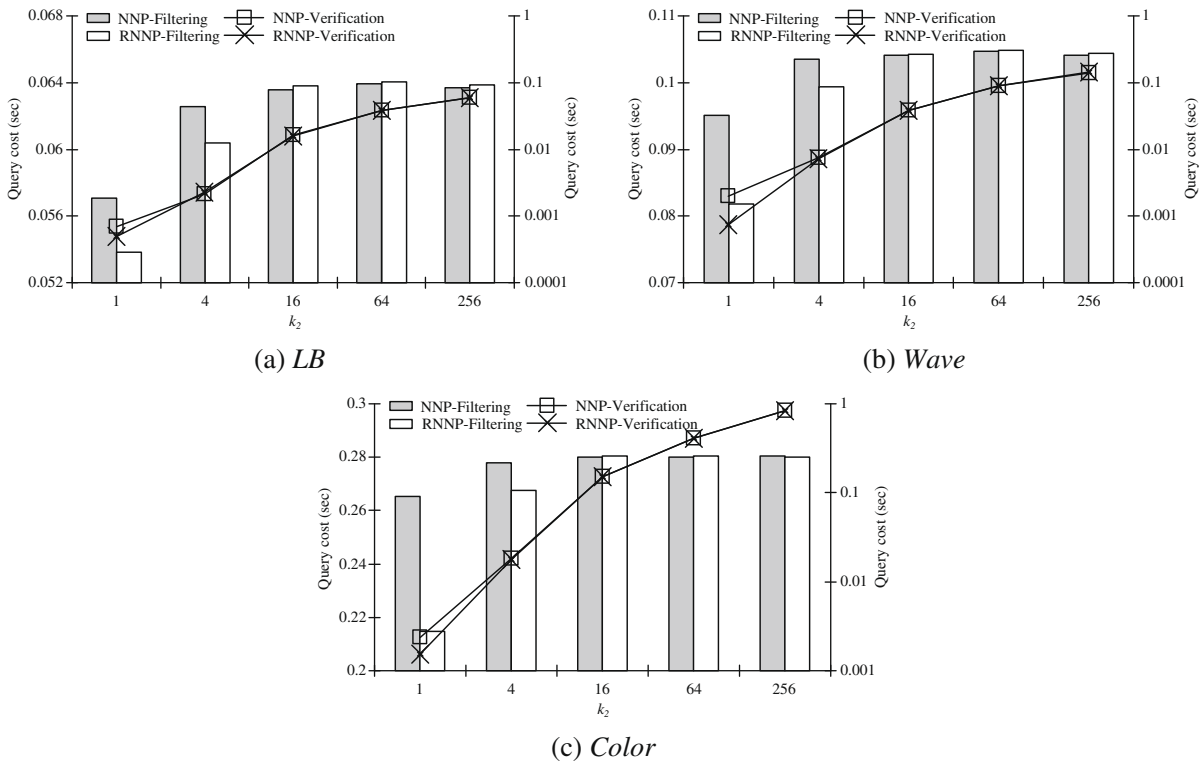
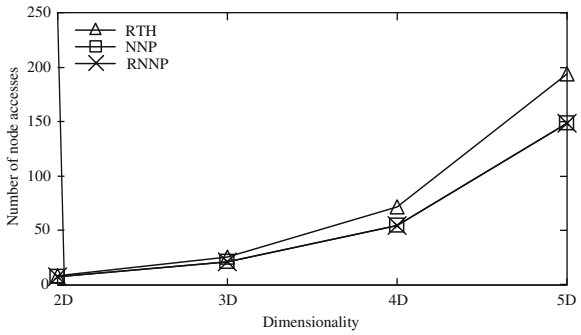


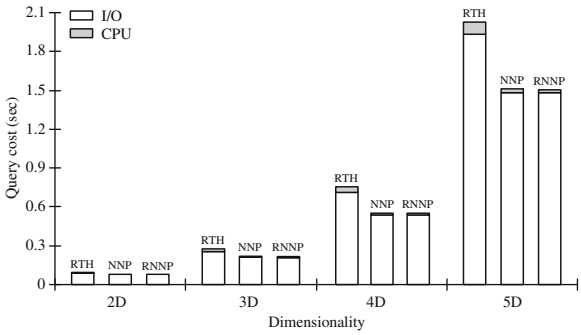
Fig. 20. Filtering and verification step costs of NNP vs. those of RNNP (varying k_2 and $k_1 = 16$).

Table 5
The maximal number of entries in the heap vs. k_2 ($k_1 = 16$) on *LB*, *Wave*, and *Color* datasets respectively.

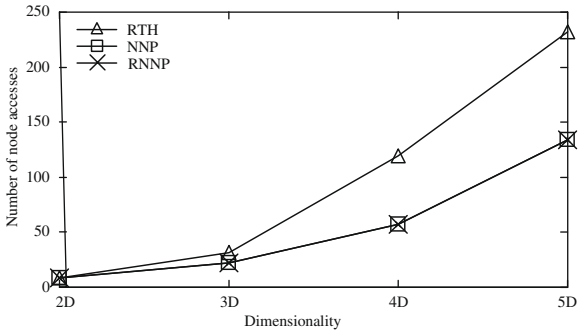
k_1	*LB*					*Wave*					*Color*				
	SP	TS	RTH	NNP	RNNP	SP	TS	RTH	NNP	RNNP	SP	TS	RTH	NNP	RNNP
1	223	219	235	157	144	476	473	540	197	158	2485	2485	2778	444	318
4	231	219	257	175	166	491	473	621	217	204	2492	2485	3132	490	455
16	247	219	296	207	207	541	473	733	244	244	2632	2485	3565	695	695
64	256	219	320	246	246	558	473	753	287	287	2823	2485	3763	992	992
256	263	219	346	273	273	560	473	755	306	306	2879	2485	3844	1243	1243



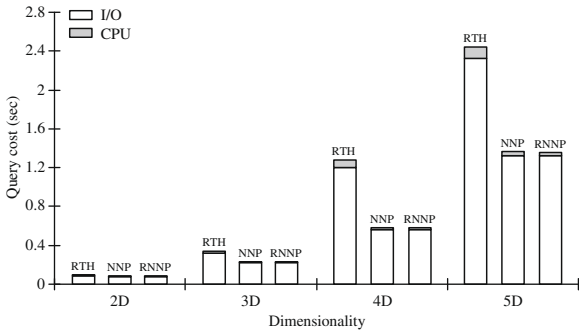
(a) I/O cost on *Uniform*



(b) Query cost on *Uniform*

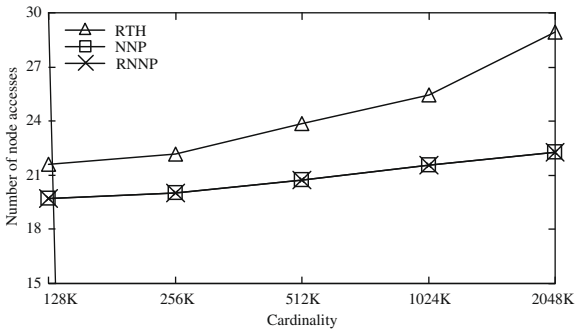


(c) I/O cost on *Zipf*

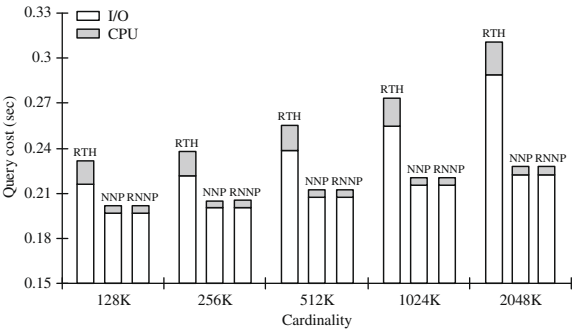


(d) Query cost on *Zipf*

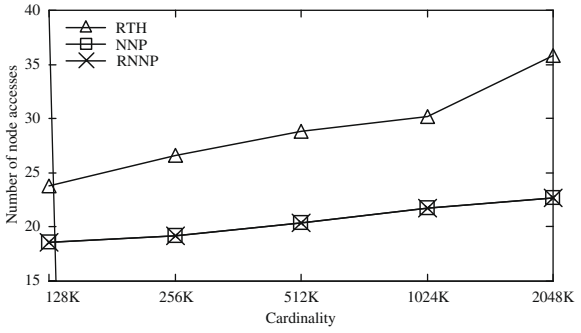
Fig. 21. Performance vs. dimensionality ($k_1 = k_2 = 16$, cardinality = 512 K).



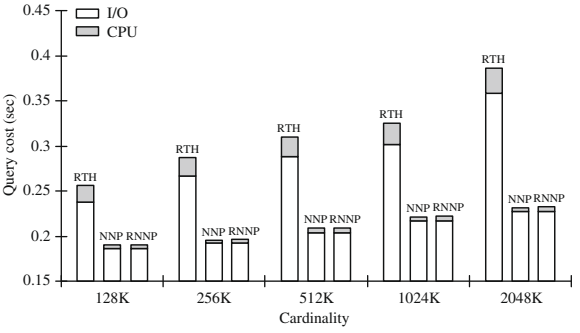
(a) I/O cost on *Uniform*



(b) Query cost on *Uniform*



(c) I/O cost on *Zipf*



(d) Query cost on *Zipf*

Fig. 22. Performance vs. cardinality ($k_1 = k_2 = 16$, dimensionality = 3D).

48 × 6653 bytes ≈312 pages (notice that 6653 is the maximal value in Table 4). In addition, it is observed that RTH is more memory-consuming, compared against the other algorithms. The reason behind is that RTH has to maintain all the entries that have been visited during the query processing in order to reuse them later, whereas NNP and RNNP utilize pruning techniques to discard non-qualifying entries, leading to heap size saving.

Next, we fix k_1 to 16 and vary k_2 between 1 and 256 to evaluate the impact of k_2 on the performance of the algorithms, as shown in Fig. 19. It is observed that the cost of the algorithms increases slightly as k_2 grows, but their ascending trend is not as obvious as that observed from Fig. 17. This is because, as implied by Lemma 1, the maximum number of the final MNNs in the dataset is k_1 (=16), which is fixed. Again, both NNP and RNNP perform the best in all the cases. In particular, the maximum speedup of these two algorithms over RTH is about 7.5 times, occurring under *Color* dataset with $k_1 = 16$ and $k_2 = 1$ (Fig. 19f). The second observation is that RNNP outperforms NNP when $k_1 \gg k_2$ (e.g., $k_1 = 16$ and $k_2 = 1$). The reason behind is that $|RNN_{k2}(q)| \ll |NN_{k1}(q)|$ satisfies when $k_2 \ll k_1$, as also demonstrated in Fig. 20, where the filtering step cost of RNNP is lower than that of NNP when $k_1 = 16$ and $k_2 = 1$.

Table 5 compares the maximal number of entries in the heap as a function of k_2, confirming the observations of Table 4. Notice that even though TS is much better than SP, we ignore TS in Fig. 19c–f, since it is always worse than the other three algorithms (i.e., RTH, NNP, and RNNP), especially for the large values of k_2 (e.g., 256). Similarly, TS is omitted from Figs. 21 and 22 as well.

The third set of experiments explores the influence of the dimensionality on the cost of the algorithms. Due to the low dimensionality of the real dataset, we employ the synthetic datasets with cardinality 512 K and vary dimensionality from 2 to 5. Fig. 21 plots the efficiency of different algorithms in answering $MNN_{16,16}$ queries. The performance of the algorithms degrades as the dimensionality increases. This is because, in general, R-tree becomes less efficient as the dimensionality grows [33] due to the large overlap among the node MBRs at the same level. Moreover, the cost involved in both the filtering and verification steps increases with the growth of dimensionality. However, both NNP and RNNP evidently outperform the other algorithms, and the difference increases with dimensionality. Table 6 lists the maximal number of entries in the heap for the algorithms with respect to dimensionality. As expected, the memory consumption of the algorithms increases as dimensionality ascends. Observe that TS consumes the least memory space for *Uniform* dataset, while for *Zipf* dataset, both NNP and RNNP consume the least in the most of cases, due to different data distributions. Nevertheless, they are consistently better than RTH.

In the sequel, we study the behavior of the algorithms for different dataset cardinalities. The 3D *Uniform* and *Zipf* datasets whose cardinalities range between 128 K and 2048 K are employed. Fig. 22 measures the cost of the algorithms in processing $MNN_{16,16}$ queries as a function of the dataset cardinality. It is observed that the impact of the dataset cardinality is not as obvious as that of dimensionality. This is because given fixed k_1 and k_2, the expansion of all algorithms is roughly the same, which does not depend on the size of the dataset. The step-wise cost growth corresponds to an increase of the tree height. Specifically, for *Uniform* (*Zipf*) dataset, the increase occurs at cardinality 512 K (1024 K). In general, the relative performance of the algorithms remains the same as that of the previous experiments in all cases, namely, both NNP and RNNP perform the best, followed by RTH, TS, and SP is the worst. Table 7 presents the maximal number of entries in the heap (involved in the algorithms) for the dataset cardinality. The phenomena and their explanations are the same as those in Table 6.

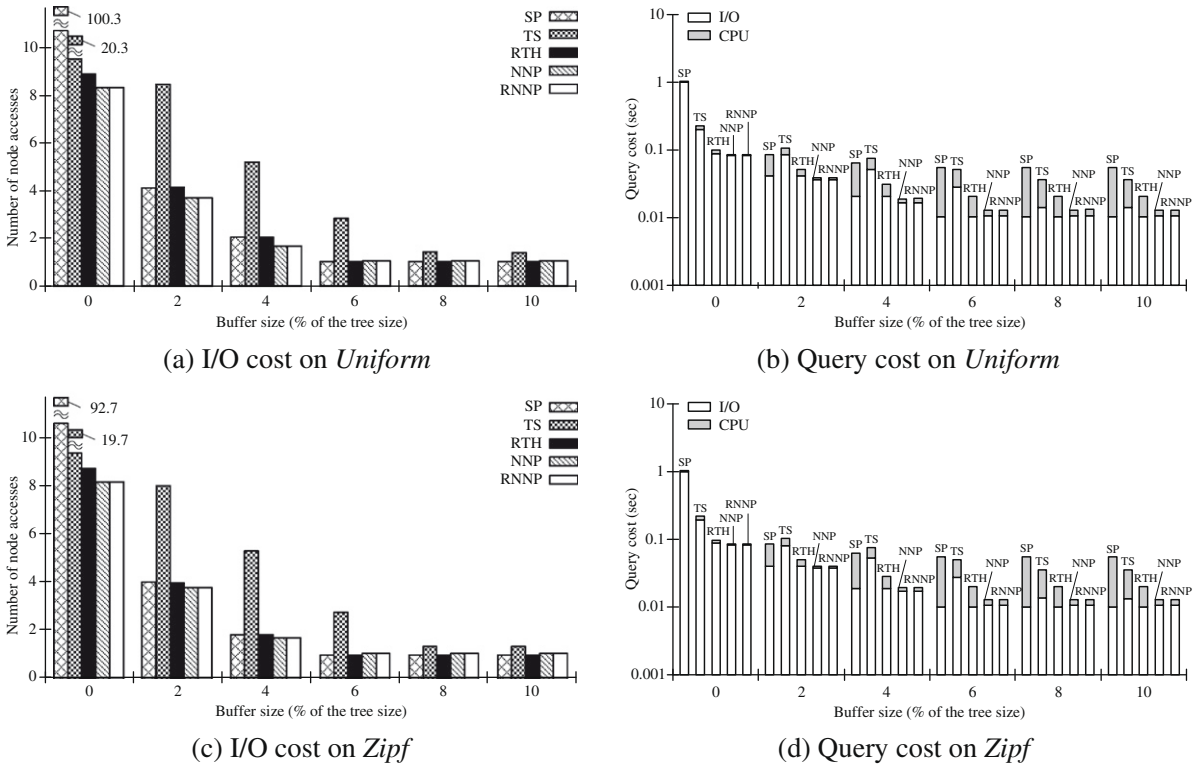
As mentioned at the end of Section 5.1, all the aforementioned experiments are conducted without considering buffers. In the last set of experiments, we examine the performance of the algorithms in the presence of an LRU buffer. Towards this, we perform $MNN_{16,16}$ queries on the 2D synthetic datasets with cardinality = 512 K, varying the buffer size from 0% to 10% of the

Table 6
The maximal number of entries in the heap vs. dimensionality ($k_1 = k_2 = 16$, cardinality = 512 K) on *Uniform* and *Zipf* datasets respectively.

Dimensionality	*Uniform*					*Zipf*				
	SP	TS	RTH	NNP	RNNP	SP	TS	RTH	NNP	RNNP
2D	250	199	279	229	229	233	194	282	231	231
3D	403	340	587	445	445	530	463	730	465	465
4D	747	634	1295	927	927	1478	1337	2182	964	964
5D	1449	1283	2849	2117	2117	2098	1913	3435	1910	1910

Table 7
The maximal number of entries in the heap vs. cardinality ($k_1 = k_2 = 16$, dimensionality = 3D) on *Uniform* and *Zipf* respectively.

Cardinality	*Uniform*					*Zipf*				
	SP	TS	RTH	NNP	RNNP	SP	TS	RTH	NNP	RNNP
128K	328	272	505	415	415	392	336	558	389	389
256K	353	289	527	421	421	444	388	624	414	414
512K	364	304	540	431	431	476	410	657	415	415
1024K	390	327	583	446	446	504	435	708	466	466
2048K	462	390	676	475	475	602	524	838	486	486



(a) I/O cost on *Uniform*

(b) Query cost on *Uniform*

(c) I/O cost on *Zipf*

(d) Query cost on *Zipf*

Fig. 23. Performance vs. buffer size ($k_1 = k_2 = 16$, dimensionality = 2D, cardinality = 512 K).

R-tree size. To obtain stable statistics, We further assume the first 100 queries are performed to warm up the buffer and only measure the average performance of the last 100 queries. Fig. 23 shows the cost with respect to the buffer size for *Uniform* and *Zipf* datasets. When the buffer size equals 0, every node access incurs a page access. Since the algorithms may retrieve the same entries multiple times during query processing, even a small buffer ensures that some of such entries are loaded from the disk only once, resulting in a dramatic reduction in the I/O cost and the improvement of overall query cost. In Fig. 23a, for instance, when the buffer size changes from 0% to 2%, the number of node accesses (i.e., I/O cost) reduces 24.5, 2.4, 2.1, 2.3, and 2.3 times for algorithms SP, TS, RTH, NNP, and RNNP, respectively. It is also observed that the performance of the algorithms stabilizes once the buffer size reaches 8%, meaning that this buffer size is sufficient for keeping all the entries visited in memory. Both NNP and RNNP again outperform their competitors significantly in all cases.

To summarize, from the above experimental results on both real and synthetic datasets, we can conclude that both NNP and RNNP consistently provide the best performance under all the settings, and RNNP is the best choice if $k_2 \ll k_1$ holds. Although both TS and RTH perform not as efficient as NNP and RNNP, they still outperform SP significantly in all cases. SP is definitely inappropriate for MNN queries as it is always worse than the other four algorithms. In addition, the maximal number of entries in the heap for each algorithm is negligible compared to the dataset size.

6. Conclusions

This paper presents the first piece of work that solves MNN queries in spatial databases. As a new form of NN search, MNN is interesting from a research point of view and has practical relevance to several applications including decision making, data mining, and pattern recognition. In this paper, we provide a formal definition of MNN retrieval and propose a suite of algorithms (containing SP, TS, RTH, NNP, and RNNP) for efficient processing of MNN queries on multi-dimensional datasets. Our methods follow a two step (i.e., filtering-verification) methodology: a filtering step for retrieving a set of candidates, and the subsequent verification step for eliminating the false hits. An extensive experimental study upon real and synthetic datasets confirms that both NNP and RNNP outperform the other three algorithms significantly in terms of I/O overhead and total query cost under all settings. The performance improvement is due to the fact that both NNP and RNNP algorithms reuse all the entries that have been visited during the search and eliminate unnecessary node accesses by effective pruning strategies.

The proposed techniques in this paper only consider the Euclidian space. A promising direction for future work may concern their extension to metric space, such as road network. In this case, the triangular inequality has to be used (instead of bisectors) for pruning the search space. We also intend to investigate efficient algorithms for handling the MNN query with respect to a line segment which contains continuous query points instead of a fixed query point.

References

[1] Z.A. Aghbari, Array-index: a plug & search *k* nearest neighbors method for high-dimensional data, Data and Knowledge Engineering 52 (3) (2005) 333–352.
[2] F. Angiulli, C. Pizzuti, An approximate algorithm for top-*k* closest pairs join query in large high dimensional data, Data and Knowledge Engineering 53 (3) (2005) 263–281.
[3] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R^*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of ACM International Conference on Management of Data (SIGMOD), 1990, pp. 322–331.
[4] R. Benetis, C.S. Jensen, G. Karciauskas, S. Saltenis, Nearest and reverse nearest neighbor queries for moving objects, The International Journal on Very Large Data Bases 15 (3) (2006) 229–250.
[5] S. Berchtold, D.A. Keim, H.-P. Kriegel, The X-tree: an index structure for high-dimensional data, in: Proceedings of the 22nd International Conference on Very Large Data Base (VLDB), 1996, pp. 28–39.
[6] M.R. Brito, E.L. Chavez, A.J. Quiroz, J.E. Yukich, Connectivity of the mutual *k*-nearest-neighbor graph in clustering and outlier detection, Statistics and Probability Letters 35 (1) (1997) 33–42.
[7] K.L. Cheung, A.W.-C. Fu, Enhanced nearest neighbour search on the R-tree, SIGMOD Record 27 (3) (1998) 16–21.
[8] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Closest pair queries in spatial databases, in: Proceedings of ACM International Conference on Management of Data (SIGMOD), 2000, pp. 189–200.
[9] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Algorithms for processing k-closest-pair queries in spatial databases, Data and Knowledge Engineering 49 (1) (2004) 67–104.
[10] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Cost models for distance joins queries using R-trees, Data and Knowledge Engineering 57 (1) (2006) 1–36.
[11] K. Deng, X. Zhou, H. Shen, K. Xu, X. Lin, Surface *k*-NN query processing, in: Proceedings of the 22nd International Conference on Data Engineering (ICDE), 2006, p. 78.
[12] C.H.Q. Ding, X. He, *K*-nearest-neighbor consistency in data clustering: incorporating local information into global optimization, in: Proceedings of ACM Symposium on Applied Computing (SAC), 2004, pp. 584–589.
[13] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, A. Abbadi, Constrained nearest neighbor queries, in: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases (SSTD), 2001, pp. 257–278.
[14] Y. Gao, G. Chen, Q. Li, B. Zheng, C. Li, Processing mutual nearest neighbor queries for moving object trajectories, in: Proceedings of the 9th International Conference on Mobile Data Management (MDM), 2008, pp. 116–123.
[15] K.C. Gowda, G. Krishna, Agglomerative clustering using the concept of mutual nearest neighborhood, Pattern Recognition 10 (2) (1978) 105–112.
[16] K.C. Gowda, G. Krishna, The condensed nearest neighbor rule using the concept of mutual nearest neighborhood, IEEE Transactions on Information Theory 25 (4) (1979) 488–490.
[17] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of ACM International Conference on Management of Data (SIGMOD), 1984, pp. 47–57.
[18] D. Harel, Y. Koren, Clustering spatial data using random walks, in: Proceedings of the 7th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), 2001, pp. 281–286.
[19] G.R. Hjaltason, H. Samet, Distance browsing in spatial databases, ACM Transactions on Database Systems 24 (2) (1999) 265–318.
[20] H. Hu, D.L. Lee, Range nearest-neighbor query, IEEE Transactions on Knowledge and Data Engineering 18 (1) (2006) 78–91.
[21] A.K. Jain, R.C. Dubes, Algorithms for Clustering Data, Prentice-Hall Advanced Reference Series, Prentice-Hall, Englewood Cliffs, New Jersy, 1988.
[22] A.K. Jain, R.P.W. Duin, J. Mao, Statistical pattern recognition: a review, IEEE Transactions on Pattern Analysis and Machine Intelligence 22 (1) (2000) 4–37.
[23] W. Jin, Anthony K.H. Tung, J. Han, Mining top-*n* local outliers in large databases, in: Proceedings of the 7th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), 2001, pp. 293–298.
[24] W. Jin, Anthony K.H. Tung, J. Han, W. Wang, Ranking outliers using symmetric neighborhood relationship, in: Proceedings of the 10th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), 2006, pp. 577–593.
[25] J.M. Kang, M.F. Mokbel, S. Shekhar, T. Xia, D. Zhang, Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors, in: Proceedings of the 23rd International Conference on Data Engineering (ICDE), 2007, pp. 806–815.
[26] F. Korn, S. Muthukrishnan, Influence sets based on reverse nearest neighbor queries, in: Proceedings of ACM International Conference on Management of Data (SIGMOD), 2000, pp. 201–212.
[27] F. Korn, S. Muthukrishnan, D. Srivastava, Reverse nearest neighbor aggregates over data streams, in: Proceedings of the 28th International Conference on Very Large Data Base (VLDB), 2002, pp. 814–825.
[28] C.K. Ken Lee, B. Zheng, W.-C. Lee, Ranked reverse nearest neighbor search, IEEE Transactions on Knowledge and Data Engineering 20 (7) (2008) 894–910.
[29] K.-I. Lin, M. Nolen, C. Yang, Applying bulk insertion techniques for dynamic reverse nearest neighbor problems, in: Proceedings of the 7th International Database Engineering and Applications Symposium (IDEAS), 2003, pp. 290–297.
[30] A. Maheshwari, J. Vahrenhold, N. Zeh, On reverse nearest neighbor queries, in: Proceedings of the 14th Canadian Conference on Computational Geometry (CCCG), 2002, pp. 128–132.
[31] D. Papadias, Q. Shen, Y. Tao, K. Mouratidis, Group nearest neighbor queries, in: Proceedings of the 20th International Conference on Data Engineering (ICDE), 2004, pp. 301–312.
[32] D. Papadias, Y. Tao, K. Mouratidis, K. Hui, Aggregate nearest neighbor queries in spatial databases, ACM Transactions Database Systems 30 (2) (2005) 529–576.
[33] A. Papadopoulos, Y. Manolopoulos, Performance of nearest neighbor queries in R-trees, in: Proceedings of the 6th International Conference on Database Theory (ICDT), 1997, pp. 394–408.
[34] Y. Qian, K. Zhang, Discovering spatial patterns accurately with effective noise removal, in: Proceedings of the 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD), 2004, pp. 43–50.
[35] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, in: Proceedings of ACM International Conference on Management of Data (SIGMOD), 1995, pp. 71–79.
[36] H. Shin, B. Moon, S. Lee, Tie-breaking strategies for fast distance join processing, Data and Knowledge Engineering 41 (1) (2002) 67–83.
[37] A. Singh, H. Ferhatosmanoglu, A. Tosun, High dimensional reverse nearest neighbor queries, in: Proceedings of ACM International Conference on Information and Knowledge Management (CIKM), 2003, pp. 91–98.
[38] Z. Song, N. Roussopoulos, *K*-nearest neighbor search for moving query point, in: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases (SSTD), 2001, pp. 79–96.
[39] I. Stanoi, D. Agrawal, A. El Abbadi, Reverse nearest neighbor queries for dynamic databases, in: Proceedings of ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD), 2000, pp. 44–53.
[40] I. Stanoi, M. Riedewald, D. Agrawal, A. Abbadi, Discovery of influence sets in frequently updated databases, in: Proceedings of the 27th International Conference on Very Large Data Base (VLDB), 2001, pp. 99–108.
[41] Y. Tao, D. Papadias, X. Lian, Reverse *k*NN search in arbitrary dimensionality, in: Proceedings of the 30th International Conference on Very Large Data Base (VLDB), 2004, pp. 744–755.
[42] Y. Tao, D. Papadias, X. Lian, X. Xiao, Multidimensional reverse *k*NN search, The International Journal on Very Large Data Bases 16 (3) (2007) 293–316.

[43] Y. Tao, D. Papadias, Q. Shen, Continuous nearest neighbor search, in: Proceedings of the 28th International Conference on Very Large Data Base (VLDB), 2002, pp. 287–298.
[44] T. Xia, D. Zhang, Continuous reverse nearest neighbor monitoring, in: Proceedings of the 22nd International Conference on Data Engineering (ICDE), 2006, p. 77.
[45] R.C.-W. Wong, Y. Tao, A.W.-C. Fu, X. Xiao, On efficient spatial matching, in: Proceedings of the 33rd International Conference on Very Large Data Base (VLDB), 2007, pp. 579–590.
[46] C. Yang, K.-I. Lin, An index structure for efficient reverse nearest neighbor queries, in: Proceedings of the 17th International Conference on Data Engineering (ICDE), 2001, pp. 485–492.
[47] J. Zhang, N. Mamoulis, D. Papadias, Y. Tao, All-nearest-neighbors queries in spatial databases, in: Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM), 2004, pp. 297–306.
[48] Y. Zhang, C. Zhang, S. Wang, Clustering in knowledge embedded space, in: Proceedings of the 14th European Conference on Machine Learning (ECML), 2003, pp. 480–491.

Yunjun Gao received the Master degree in computer science from Yunnan University, China in 2005, and the Ph.D. degree in computer science from Zhejiang University, China in 2008. He is currently a postdoctoral research fellow in the School of Information Systems, Singapore Management University, Singapore. His research interests include spatial databases, spatio-temporal databases, mobile/pervasive computing, and geographic information systems. He is a member of the ACM, ACM SIGMOD, and IEEE.

Baihua Zheng received the Bachelor degree in computer science from Zhejiang University, China in 1999, and the Ph.D. degree in computer science from the Hong Kong University of Science and Technology, Hong Kong in 2003. She is currently an assistant professor in the School of Information Systems, Singapore Management University, Singapore. Her research interests include mobile/pervasive computing and spatial databases. She is a member of the ACM and the IEEE.

Gencai Chen is a professor in the College of Computer Science, Zhejiang University, China. He was a visiting scholar in the Department of Computer Science, State University of New York at Buffalo, USA, from 1987 to 1988, and the winner of the special allowance, conferred by the State Council of China in 1997. He is currently a vice dean of the College of Computer Science, a director of the Computer Application Engineering Center, and a vice director of the Software Research Institute, Zhejiang University. His research interests include database systems, artificial intelligence, and CSCW.

Qing Li is a professor in the Department of Computer Science, City University of Hong Kong where he joined as a faculty member since September 1998. Before that, he has taught at the Hong Kong Polytechnic University, the Hong Kong University of Science and Technology and the Australian National University (Canberra, Australia). He is a guest professor of the University of Science and Technology of China, a visiting professor at the Institute of Computing Technology (Knowledge Grid), Chinese Academy of Science (Beijing, China), an adjunct professor of the Hunan University (Changsha, China), and a guest professor (software technology) of the Zhejiang University (Hangzhou, China). His research interests include object modeling, multimedia databases, and web services. He is a senior member of IEEE, a member of ACM SIGMOD and IEEE Technical Committee on Data Engineering. He is the chairperson of the Hong Kong Web Society and also served/is serving as an executive committee (EXCO) member of the IEEE-Hong Kong Computer Chapter and the ACM Hong Kong Chapter. In addition, he serves as a councilor of the Database Society of Chinese Computer Federation, a councilor of the Computer Animation and Digital Entertainment Chapter of Chinese Computer Imaging and Graphics Society and is a Steering Committee member of DASFAA, ICWL and the international WISE Society.