On efficient mutual nearest neighbor query processing in spatial databases

Yunjun Gao\textsuperscript{a,b,*}, Baihua Zheng\textsuperscript{a}, Gencai Chen\textsuperscript{b}, Qing Li\textsuperscript{c}

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

\textbf{Abstract}

This paper studies a new form of nearest neighbor queries in spatial databases, namely, \textit{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 $k_1$ nearest neighbors (NNs) of $q$; meanwhile, have $q$ as one of their $k_2$ 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 ($k$NN) retrieval and reverse $k$NN 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.

\textbf{1. Introduction}

This paper studies a new form of nearest neighbor (NN) queries, namely \textit{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 $p \in \text{NN}_{k_1}(q)$\textsuperscript{1} and $q \in \text{NN}_{k_2}(p)$, i.e., it requires each \textit{answer object}\textsuperscript{2} to be one of the $k_1$ nearest neighbors (NNs) to $q$ and meanwhile has $q$ as one of its $k_2$NNs. 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 \textit{asymmetric}, while MNN retrieval is \textit{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.

\textbf{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 \textit{only} two other branches. Suppose that we employ conventional $k$ ($\geq 2$) nearest neighbor ($k$NN) 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...
search can identify two members, say the success probability of the matchmaking, all the matches it recommends have to be RNN (i.e., R²NN) retrieval can identify those members, say p at NN objects to each other. Then, we increase either order to retrieve the objects that are closest to each other. Consequently, k branches to kNN retrieval with respect to p. As the popularity of p, branches, they are out of the consideration. Hence, p is linked to branches p and p. The process continues, and Fig. 1b depicts the result of assignment. 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 k1 and k2, it means that p is one of k1NNs to p’ and meanwhile p’ is one of k2NNs to p. Continuing the running example, we utilize MNN queries to deal with the assignment of branches. Initially, both k1 and k2 are set to one in order to retrieve the objects that are closest to each other. Consequently, p is linked to p and p is linked to p because they are NN objects to each other. Then, we increase either k1 or k2 by one (i.e., k1 = 1, k2 = 2 or k1 = 2, k2 = 1). Branch p is linked to p2 as p is the NN object of p and meanwhile p2 is one of p’s 2 NN objects, i.e., p satisfies an MNN query with k1 = 1 and k2 = 2 issued at p. Next, we set both k1 and k2 to 2, and p is linked to p since p and p are MNN objects to each other with respect to k1 = k2 = 2. Here, p, p, and p 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. Finally, p is linked to branches p and p 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 and m, 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 and m. On the other hand, reverse k(=2) NN (i.e., R²NN) retrieval can identify those members, say m, m, and m, 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, m, and m. In summary, both kNN search and RkNN search are asymmetric, while matchmaking service is symmetric. Thus, MNN search is more suitable. By carefully

---

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 \( \text{top}-k_1 \) choices for \( m \) and meanwhile \( m \) is one of the \( \text{top}-k_2 \) 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 (kMNN) 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\(_q\) 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 \((a_o, o_b)\) 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 \( \text{NN}_k(q) \), and then verify whether each point \( p \) in \( \text{NN}_k(q) \) 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 \( k \)NN retrieval [19] and reverse \( k \)NN (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 \( \text{Root} \).

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) \( \text{mindist}(q,N) \), (ii) \( \text{maxdist}(q,N) \), and (iii) \( \text{minmaxdist}(q,N) \), where \( q \) is a query point and \( N \) is the minimum bounding rectangle (MBR) associated with a node. The \( \text{mindist}(q,N) \) and \( \text{maxdist}(q,N) \) give the lower and upper bounds of the distance from \( q \) to any point in the subtree of \( N \). The \( \text{minmaxdist}(q,N) \) defines an upper bound of the distance between \( q \) and \( a \).

---

\(^4\) Let \( x_i \) be the mth NN of \( x_j \) and \( x_j \) be the mth NN of \( x_i \). 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 \( k \) \( k \)-NN consistent with respect to cluster \( C_r \), then \( x_i \) must be \( k \)MNN consistent with respect to \( C_r \) (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.
its NN in \( N \). In other words, there is at least one point located inside \( N \) whose distance to \( q \) does not exceed \( \text{minmaxdist}(q,N) \). 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 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 \( \text{mindist} \). Starting from the root of the tree, BF inserts all the root entries into \( H \) together with their \( \text{mindist} \). Then, the top entry \( e \) with the smallest \( \text{mindist} \) 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 \( k \) (\( \geq 1 \)) 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 \( \text{mindist}(q,e) \). Note that when \( e \) refers to a point, \( \text{mindist}(q,e) = \text{dist}(q,e) \), 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 &

---

**Table 1**

<table>
<thead>
<tr>
<th>Action</th>
<th>Heap content</th>
<th>NNs</th>
</tr>
</thead>
<tbody>
<tr>
<td>visit Root</td>
<td>{( N_2 ), ( N_1 )}</td>
<td>( N_1 ), ( N_2 )</td>
</tr>
<tr>
<td>follow ( N_2 )</td>
<td>{( N_3 ), ( N_1 ), ( N_2 )}</td>
<td>( N_1 ), ( N_2 ), ( N_3 )</td>
</tr>
<tr>
<td>follow ( N_6 )</td>
<td>{( i, N_1 ), h, ( N_2 )}</td>
<td>( i ), ( N_1 ), h, ( N_2 )</td>
</tr>
<tr>
<td>remove ( i )</td>
<td>{( N_3 ), i, h, ( N_2 )}</td>
<td>( i ), h, ( N_2 )</td>
</tr>
<tr>
<td>follow ( N_1 )</td>
<td>{( N_4 ), j, ( N_1 ), h, ( N_2 )}</td>
<td>( i ), h, ( N_2 )</td>
</tr>
<tr>
<td>follow ( N_4 )</td>
<td>{( e, j, d, N_1 ), h, ( N_2 )}</td>
<td>( i ), h, ( N_2 )</td>
</tr>
<tr>
<td>remove ( e )</td>
<td>{( j, d, N_1 ), h, ( N_2 )}</td>
<td>( i ), h, ( N_2 )</td>
</tr>
<tr>
<td>remove ( j )</td>
<td>{( d, N_1 ), h, ( N_2 )}</td>
<td>( i ), h, ( N_2 )</td>
</tr>
</tbody>
</table>

---

Without loss of generality, \( \text{dist}(p_1,p_2) \) is a function to compute the Euclidean distance between any two points \( p_1, p_2 \), although any distance metric can be used in general.
TPL pruning, in the following we describe the TPL algorithm via an illustrative example. Search, called TPL. TPL is very efficient in a low-dimensional space. Since our proposed algorithms utilize R search [13] at each sub-region. It then validates each candidate by checking whether a sub-region may become the result. Consequently, the candidate set only contains 6 objects, retrieved by constraint NN search. 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. The efficiency of this algorithm is assured by the small cardinality of the candidate set, whose size increases exponentially with the dimensionality. To facilitate the examination, all the vicinity circles are indexed with RNN-tree [26] or RdNN-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 RkNN query (if the corresponding kNN information for each point is available), it is limited to answer RkNN retrieval for a fixed k.

Inspired by the defects of pre-computation based approaches, several alternative RNN/RkNN 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 RkNN search, called TPL. TPL is very efficient in a low-dimensional space. Since our proposed algorithms utilize RkNN 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 \( HP_q(p, q) \) 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., \( HP_{p_1}(p_1, q) \) and \( HP_{p_2}(p_1, q) \). As points \( p_6, p_7 \) (contained in \( N_i \)) fall inside the half-plane \( HP_{p_1}(p_1, q) \), they are closer to \( p_1 \) than to \( q \), and hence they for sure are not answer objects. Similarly, \( N_2 \) can also be discarded because it falls into the half-plane \( HP_{p_2}(p_2, q) \). 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 \( HP_{p_1}(p_1, q) \). \( HP_{p_2}(p_2, q) \) (the shaded area). In addition, the number of half-planes \( HP_{p_1}(p_1, q) \) that a given point \( p' \) falls in represents the number of data points that

![Fig. 3. Example of TPL algorithm.](image-url)
are closer to \( p \) than \( q \). Consequently, if a data point is inside at least \( k \) \( HP(p,q) \) 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 \( l_i \) (i.e., line \( l_i \)), and shrinks the search space from the entire space (i.e., \( ABCD \)) to a trapezoid \( EFCD \). Therefore, points \( h, j \) in \( N_0 \) can be pruned, which are maintained by a refinement set \( S_{rfn} \). Similarly, node \( N_0 \) 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 \( S_{rfn} = \{j,d,N_5,h,N_3\} \). 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/R\( k \)NN search, various variants of RNN/R\( k \)NN queries have been well-studied in the database literature, e.g., Benetis et al. [4] and Tao et al. [42] investigate R\( k \)NN 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 R\( k \)NN 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.

<table>
<thead>
<tr>
<th>Action</th>
<th>Heap content</th>
<th>( S_c )</th>
<th>( S_{rfn} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>visit Root</td>
<td>{( N_2,N_1 )}</td>
<td># | # | #</td>
<td></td>
</tr>
<tr>
<td>visit ( N_2 )</td>
<td>{( N_{h2},N_1,N_3 )}</td>
<td># | # | #</td>
<td></td>
</tr>
<tr>
<td>visit ( N_6 )</td>
<td>{( i,N_1,N_1 )}</td>
<td># | # | #</td>
<td></td>
</tr>
<tr>
<td>process ( i )</td>
<td>{( N_2,h,N_3 )}</td>
<td># | # | #</td>
<td></td>
</tr>
<tr>
<td>visit ( N_1 )</td>
<td>{( N_2,h,N_3 )}</td>
<td># | # | #</td>
<td></td>
</tr>
<tr>
<td>visit ( N_4 )</td>
<td>{( e,d,N_1,h,N_1 )}</td>
<td># | # | #</td>
<td></td>
</tr>
<tr>
<td>process ( e )</td>
<td>{( j,d,N_1,h,N_1 )}</td>
<td># | # | #</td>
<td></td>
</tr>
<tr>
<td>process ( j )</td>
<td>{( d,N_1,h,N_1 )}</td>
<td># | # | #</td>
<td></td>
</tr>
<tr>
<td>process ( d )</td>
<td>{( N_1,h,N_1 )}</td>
<td># | # | #</td>
<td></td>
</tr>
<tr>
<td>process ( N_3 )</td>
<td>{( h,N_3 )}</td>
<td># | # | #</td>
<td></td>
</tr>
<tr>
<td>process ( h )</td>
<td>{( N_5 )}</td>
<td># | # | #</td>
<td></td>
</tr>
<tr>
<td>process ( N_5 )</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
</tbody>
</table>

### Table 3
Summary of symbols.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D )</td>
<td>A data set</td>
</tr>
<tr>
<td>( q )</td>
<td>A query point</td>
</tr>
<tr>
<td>( TD )</td>
<td>The R-tree on ( D )</td>
</tr>
<tr>
<td>(</td>
<td>TD</td>
</tr>
<tr>
<td>( e )</td>
<td>An entry (data object or node MBR) in an R-tree</td>
</tr>
<tr>
<td>( H )</td>
<td>A heap</td>
</tr>
<tr>
<td>( S_{res} )</td>
<td>A query result set</td>
</tr>
<tr>
<td>( S )</td>
<td>A candidate set</td>
</tr>
<tr>
<td>(</td>
<td>S</td>
</tr>
<tr>
<td>( NN(q) )</td>
<td>Result set of a ( k ) nearest neighbor query issued at point ( q )</td>
</tr>
<tr>
<td>( RRNN(q) )</td>
<td>Result set of a reverse ( k ) nearest neighbor query issued at point ( q )</td>
</tr>
<tr>
<td>( MNN_{k1,k2}(q) )</td>
<td>Result set of a mutual nearest neighbor query with respect to ( k_1 ) and ( k_2 ) issued at point ( q )</td>
</tr>
</tbody>
</table>
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) $\forall p \in S$, $p \in \text{NN}_{k_1}(q)$; and (ii) $\forall p \in S$, $q \in \text{NN}_{k_2}(p)$.

Formally, $\text{MNN}_{k_1,k_2}(q) = \{ p \in S | p \in \text{NN}_{k_1}(q) \land q \in \text{NN}_{k_2}(p) \}$.

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 $k_2$ NNs. 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 $o_i \in \text{MNN}_{k_1,k_2}(o_j)$, then $o_j \in \text{MNN}_{k_2,k_1}(o_i)$.

Property 2. Given a query point $q$, the cardinality of $q$’s mutual nearest neighbors (MNNs), denoted by $|\text{MNN}_{k_1,k_2}(q)|$, 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, $|\text{MNN}_{1,2}(q)| = |\emptyset| = 0$, $|\text{MNN}_{1,2}(p_1)| = |\{p_2\}| = 1$, as depicted in Fig. 4.

Taking the definitions of MNN, $k$NN, and $Rk$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 $\text{MNN}_{k_1,k_2}(q)$ varies in the range of $[0, k_1]$.

Proof. According to Definition 1, $\text{MNN}_{k_1,k_2}(q) = \{ p \in S | p \in \text{NN}_{k_1}(q) \land q \in \text{NN}_{k_2}(p) \}$. Thus, $|\text{MNN}_{k_1,k_2}(q)| \leq |\text{NN}_{k_1}(q)| (=k_1)$. □

Theorem 1. $\text{MNN}_{k_1,k_2}(q)$ is a subset of $\text{NN}_{k_1}(q)$, i.e., $\text{MNN}_{k_1,k_2}(q) \subseteq \text{NN}_{k_1}(q)$.

Theorem 2. $\text{MNN}_{k_1,k_2}(q)$ is a subset of $\text{RNN}_{k_2}(q)$, i.e., $\text{MNN}_{k_1,k_2}(q) \subseteq \text{RNN}_{k_2}(q)$.

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 $S_c = \text{NN}_{k_1}(q)$ 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 $q \in \text{NN}_{k_2}(c)$.

If yes, it means that $q$ is among the $k_2$NNs 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 search.
Algorithm SP \((q, k_1, k_2, S_{in})\)

**Input:**  
- \(q\): a query point; \(k_1\): the number of NNs; \(k_2\): the number of NNs  
**Output:**  
- \(S_{out}\): 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 \((c, k_2, S_1)\) using min-heap \(H_2\)  
4. if \(q \in S_1\) then  
5. \(S_{out} = S_{out} \cup \{c\}\) // \(c\) is an actual MNN of \(q\) w.r.t. \(k_1\) and \(k_2\)
6. return \(S_{out}\)

**Fig. 5.** Simple processing algorithm (SP).

**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 MNN3,1 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 \(NN_3(e) = \{d\}\) and \(NN_3(j) = \{i\}\).

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., \(NN_{k1,2}(q)\)), that is, it does not incur false misses because \(NN_{k1,2}(q) \subseteq NN_{k1}(q)\) 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 MNN3,1 query depicted in Fig. 6a. As seen from this diagram, SP visits entries \(N_1, N_2, N_i, i, j\) 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 $k_2$NNs, we can issue an $Rk_2$NN query at point $q$ to find out the set of objects that have $q$ as one of their $k_2$NNs, which is guaranteed to be a superset of the final result set of MNN retrieval, i.e., $MNN_{k_1,k_2}(q) \subseteq RNN_{k_2}(q)$, 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, $MNN_{k_1,k_2}(q) = \{ p \in S | p \in NN_{k_1}(q) \wedge p \in RNN_{k_2}(q) \}$. 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 $Rk$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 $BF-k_1$NN for retrieving the candidate set $S_c$ of MNNs (Line 1 of SP); and (ii) heap $H_2$ utilized by the function $BF-k_2$NN 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 TD) 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 $k_2$NNs of $c$. Fig. 8 presents the pseudo-code of RTH algorithm.

---

Algorithm TS $(q, k_1, k_2, S_{mb})$

Input: $q$: a query point; $k_1$: the number of NNs; $k_2$: the number of NNs

Output: $S_{mb}$: a query result set

1. perform $BF-k_1$NN $(q, k_1, S_c)$ using min-heap $H_1$
2. perform $TPL-Rk_2$NN $(q, k_2, S_c)$ using min-heap $H_2$ // TPL algorithm for R$k$NN search proposed in [41]
3. return $S_{mb} = S_c \cap S_i$

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

Algorithm RTH $(q, k_1, k_2, S_{mb})$

Input: $q$: a query point; $k_1$: the number of NNs; $k_2$: the number of NNs

Output: $S_{mb}$: a query result set

1. initialize a min-heap $H$ accepting entries of the form $(e, key)$ and $S_{mb} = \emptyset$
2. perform $BF-k_1$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 $(c, k_2, S_{temp}, H, S_{mb})$ // see Figure 9
6. return $S_{mb}$

Fig. 8. Reuse two-heap algorithm (RTH).
Algorithm Verify \((p, k, S_{\text{temp}}, H, S_{\text{res}})\)

**Input:**
- \(p\): the candidate object that has not been verified so far;
- \(k\): the number of NNs;
- \(S_{\text{temp}}\): an auxiliary set;
- \(H\): a heap;
- \(S_{\text{res}}\): a result set

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

**Fig. 9.** The Verify algorithm.

Similar as SP algorithm, RTH utilizes BF-\(k\)NN 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-\(k\)NN algorithm, more knowledge of the data distribution (i.e., nodes of finer granularity) preserved by the heap \(H\) and the candidate set \(S_{\text{temp}}\) 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 \(o \in \text{NN}_k(c)\) and \(\text{dist}(o, c) \geq \text{dist}(q, c)\), candidate \(c \in S_c\) is for sure one of the \(k\)NNs to \(q\), and hence the verification process of \(c\) can be terminated without finding all the \(k\)NNs of \(c\). This early termination condition can improve the search performance further.

It is worth noting that the auxiliary set \(S_{\text{temp}}\) 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.** Example of RTH algorithm.

<table>
<thead>
<tr>
<th>View0: right after the filtering step</th>
</tr>
</thead>
</table>
| \(H = \begin{array}{cccc}
\text{id} & \text{key} & d & N_1 & h & N_2 & N_3 \\
\end{array}\) |
| \(S_{\text{temp}} = \{i, e, j\}\) |

<table>
<thead>
<tr>
<th>View1: after performing the Lines 1-3 of Verify algorithm when verifying i</th>
</tr>
</thead>
</table>
| \(H = \begin{array}{cccc}
\text{id} & \text{key} & i & j & h & N_1 & N_2 & N_3 & d \\
\end{array}\) |
| \(S_{\text{temp}} = \{i, e, j, d, N_3, h, N_5\}\) |

<table>
<thead>
<tr>
<th>View2: after performing the Lines 4-19 of Verify algorithm when verifying i</th>
</tr>
</thead>
</table>
| \(H = \begin{array}{cccc}
\text{id} & \text{key} & h & e & N_1 & N_2 & N_3 & d \\
\end{array}\) |
| \(S_{\text{res}} = \{i, j\}\) |
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\(_2\)). 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 \( S_{\text{temp}} \) 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 \( Rk_2\)NN query to retrieve all the objects that have \( q \) as one of their \( k_1\)NNs. Nevertheless, only those objects that belong to \( NN_k(q) \) will be included in the final result set. When \( k_1 < k_2 \), TS might suffer from expensive \( RkNN \) query cost. If \( RkNN \) retrieval only considers those objects included in \( NN_k(q) \), the search performance may be improved. On the other hand, RTH takes \( NN_k(q) \) as the candidate set because \( NN_k(q) \geq MNN_{k_2}(q) \) according to Theorem 1 (presented in Section 3.1). However, \( NN_k(q) \) may contain some false hits that cannot become actual answer objects. Therefore, if we can prune away these 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 \( RNN_{k_2}(q) \). 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 \( \infty \). 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 \( S_{\text{ref}} \) 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 \( o' \in NN_{k_1}(q) \). 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 \( \sum_{i,j}(f_{\text{min}}(o_i)) \) in order to avoid any false miss. Here, \( f_{\text{min}} \) 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 \((q, k_1, k_2, S_{\text{init}})\)**

**Input:** \( q \): a query point; \( k_1 \): the number of NNs; \( k_2 \): the number of NNs

**Output:** \( S_{\text{init}} \): a query result set

1. \( S_{\text{ref}} = S_c = S_{\text{temp}} = S_{\text{new}} = \emptyset \)
2. Initialize a min-heap \( H \) accepting entries of the form \((e, \text{key})\)
3. Perform NNP-Finding \((q, k_1, k_2, S_c, S_{\text{ref}})\) using \( H \) \(\text{// see Figure 12}\)
4. \( S_{\text{new}} = S_c \cup S_{\text{ref}} \) \(\text{// for reusing all the data objects and nodes that have been accessed in the filtering step}\)
5. If \( |S_c| > 1 \) then \(\text{// conduct self-pruning in \( NN_k(q) \) for eliminating false hits}\)
6. Prune \( V \) \( c \in S_c \) where there exists \( k_2 \) candidates \( e' \in S_c \) satisfying \( \text{dist}(e', c) < \text{dist}(q, c) \)
7. For each point \( c \in S_c \), do
8. Verify \((c, k_3, S_{\text{temp}}, H, S_{\text{init}})\) \(\text{// see Figure 9}\)
9. Return \( S_{\text{init}} \)

**Fig. 11.** Algorithm using NN search with pruning (NNP).
all the remaining entries in $H$ will not become/contain a candidate object because $e_0$ is guaranteed not to contribute to $NN_k(q)$. 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 $NN_k(q)$.

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 $NN_k(q)$, the evaluation continues. $N_6$ is evaluated and its child entries (i.e., objects $h, i, j$) are en-heaped. 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 $q$, NNP-Finding can be terminated.

\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
\textbf{Action} & \multicolumn{2}{c|}{\textbf{H}} & \textbf{\textbf{e'}} & \textbf{\textbf{PruneDist}} & \textbf{\textbf{S}} & \textbf{\textbf{\textbf{S}}_{\textbf{rm}}}$ & \textbf{\textbf{ObjNum(S)}} & \textbf{\textbf{cnt}} \\
\hline
visit root & $N_1, N_2, N_3$ & $\emptyset$ & $\infty$ & $\emptyset$ & $\emptyset$ & $\emptyset$ & 0 & 0 \\
visit $N_2$ & $N_6, N_1, N_5$ & $N_6, i, j, h, N_2$ & $\sqrt{5}$ & $\emptyset$ & $\emptyset$ & $\emptyset$ & 0 & 0 \\
visit $N_6$ & $i, N_1, j, h, N_2$ & $i, N_7, j, h, N_2$ & $\sqrt{5}$ & $\emptyset$ & $\emptyset$ & $\emptyset$ & 0 & 0 \\
visit $i$ & $N_1, j, h, N_2$ & $N_7, i, j, h, N_2$ & $\sqrt{5}$ & $\emptyset$ & $\emptyset$ & $\emptyset$ & 0 & 1 \\
visit $N_1$ & $N_2, j, N_2, h, N_1$ & $N_2, j, N_2, h, N_1$ & $\sqrt{13}$ & $\{i\}$ & $\emptyset$ & $\emptyset$ & 0 & 0 \\
visit $N_4$ & $e, j, d, N_3, h, N_5$ & $e, j, d, N_3, h, N_5$ & $\sqrt{13}$ & $\{i\}$ & $\emptyset$ & $\emptyset$ & 0 & 0 \\
visit $e$ & $j, d, N_3, h, N_5$ & not found & $\infty$ & $\{i, e\}$ & $\emptyset$ & $\{j, d, N_3, h, N_5\}$ & 5 & 2 \\
\hline
\end{tabular}

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

Fig. 12. The NNP-Finding algorithm.

Fig. 13. The trace of NNP-Finding algorithm.
none of them can become/contain the candidate object, i.e., \( \text{PruneDist} = \infty \) and \( S = H \). In other words, \( \text{ObjNum}(S) = 5 \geq k_1 (=3) \), and hence the algorithm can be terminated, after which \( S_c = \{i, e\} \) and \( S_{\text{re}} = \emptyset \).

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.

\textbf{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 \( \text{NN}_k(q) \). 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 \( \text{ObjNum}(S) \geq k_1 \) with \( S = \{e' \in H | \text{max-dist}(e', q) < \text{mindist}(e', q)\} \). 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, \( \text{max-dist}(s, q) < \text{mindist}(e', q) \leq \text{mindist}(e_o, q) \leq \text{dist}(o, q) \) holds for \( \forall s \in S \). As \( \text{ObjNum}(S) \geq k_1 \), it is confirmed that at least \( k_1 \) objects are closer to \( q \) than \( o \), and hence \( o \notin \text{NN}_k(q) \). This finding contradicts our assumption, and thus the proof completes. \( \square \)

\[ \text{max-dist}(s, q) < \text{mindist}(e', q) \leq \text{mindist}(e_o, q) \leq \text{dist}(o, q) \]

\[ \forall s \in S \]

\[ k_1 \]

\[ o \notin \text{NN}_k(q) \]

\[ \text{Dist}(o, q) \]

\[ \text{mindist}(e', q) \]

\[ \text{max-dist}(s, q) \]

\[ \text{ObjNum}(S) \geq k_1 \]

\[ \forall s \in S \]

4.4. Algorithm using RNN search with pruning

As defined in Definition 1, \( \text{MNN}_{k_1, k_2}(q) = \{p \in S | p \in \text{NN}_{k_2}(q) \land p \in \text{NN}_{k_1}(q)\} \). Our previous algorithms form the candidate set \( S_c \) based on \( \text{NN}_{k_2}(q) \) and then verify each candidate \( c \) in \( S_c \) based on the fact that whether \( c \in \text{NN}_{k_1}(q) \). However, when \( |\text{NN}_{k_2}(q)| < |\text{NN}_{k_1}(q)| \), it is more beneficial to constitute the candidate set based on \( \text{NN}_{k_2}(q) \) but not \( \text{NN}_{k_1}(q) \), especially when \( k_1 \geq k_2 \). 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 \( c \in \text{NN}_k(q) \) 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 \( \text{NN}_{k_2}(p) \) includes \( q \), i.e., \( \text{NN}_{k_2}(q) \), based on TPL-\( k \)-Trim and TPL-\( k \)-Refinement algorithms as in [41]. Nevertheless, different from conventional RANN search, it takes the characteristic of MNN search into consideration and tries to exclude those unvisited objects (i.e., \( \text{NN}_{k_1}(q) \)) that cannot be pruned by \( \text{TPL-}k\text{-Trim} \) and \( \text{TPL-}k\text{-Refinement} \) algorithms as in [41]. 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., \( o \notin \text{NN}_{k_1}(q) \) (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 \( \text{NN}_{k_1}(q) \) (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 (including nodes and data objects) that have been accessed during query processing. Therefore, RNNP also loads any entry from the disk at most once.

\textbf{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 \( o' \in \text{MNN}_{k_1, k_2}(q) \) is missed due to the early termination condition of RNNP-Finding algorithm. In other words, \( o' \in \text{NN}_{k_1}(q) \land q \in \text{NN}_{k_2}(o') \) 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 \( \text{mindist}(o', q) > \text{mindist}(o, q) \). As for \( \forall s \in S \), max-
Algorithm RNNP-Finding \((q, k_1, k_2, S, S_{\text{tmp}}, S_{\text{tmp}})\)

**Input:** \(q\): a query point; \(k_1\): the number of NNs; \(k_2\): the number of NNS; \(S\): the set of candidates that have not been verified; \(S_{\text{tmp}}\): the set of points pruned by the TPL pruning technique; \(S_{\text{tmp}}\): an auxiliary set

1. \(S = \emptyset\) and initialize a min-heap \(H\) accepting entries of the form \((e, \text{key})\)
2. insert (R-tree root, 0) into \(H\)
3. while \(H\) is not empty do
4.   delete the top entry \((e, \text{mindist}(e, q))\) from \(H\)
5.     if TPL-\(k\)-Trim \((q, k_1, S, e) = \infty\) then
6.       \(S_{\text{tmp}} = S_{\text{tmp}} \cup \{e\}\)
7.     else // e may contain/be a candidate
8.       if \(e\) is a data object \(o\) and \(o \neq q\) then
9.         \(S_{\text{tmp}} = S_{\text{tmp}} \cup \{o\}\)
10.        \(S = \{e' \in (S_{\text{tmp}} \cup S_{\text{tmp}} \cup H) \mid \maxdist(e', q) < \text{mindist}(o, q)\}\)
11.       if ObjNum(S) \(\geq k_1\) then
12.         break // terminate the filter step of RANN-Pruning algorithm
13.     end if
14.     \(S = \emptyset\) // for the next round
15.   end if
16. end while
17. Fig. 15. The RNNP-Finding algorithm.

Algorithm NN-Verify \((S_c, q, k, S_{\text{tmp}}, H, S_{\text{tmp}})\)

**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; \(S_{\text{tmp}}\): an auxiliary set; \(H\): a heap; \(S_{\text{tmp}}\): a result set

1. \(H = \emptyset\), \(\text{cnt} = 0\)
2. for each entry \(e \in S_{\text{tmp}}\) do
3.   insert \((e, \text{mindist}(e, q))\) into \(H\)
4. while \(H\) is not empty do
5.   delete the top entry \((e, \text{mindist}(e, q))\) from \(H\)
6.   if \(e\) is a data object \(o\) then
7.     \(\text{cnt} = \text{cnt} + 1\)
8.     if \(o \in S\) then
9.       \(S_{\text{tmp}} = S_{\text{tmp}} \cup \{o\}\) // \(o\) is indeed an MNN of \(q\)
10.      if \(\text{cnt} = k\) or \(|S_{\text{tmp}}| = |S_c|\) then
11.         return // terminate algorithm
12.   end if
13.   else // \(e\) is an intermediate entry
14.     for each entry \(e_i \in e\) do
15.       insert \((e_i, \text{mindist}(e_i, q))\) into \(H\)
16.     end for
17.   end if
18. end while
19. if \(H\) is empty then
20.   return // terminate algorithm
21. 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 \( S_{fn} \)) 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

![Graphs and bar charts showing experimental results](image.png)

Fig. 17. Performance vs. \( k_1 (k_2 = 16) \).
measurements of wave directions at the National Buoy Center; and Color involves 4D vectors representing the color histograms of 65,000 images. 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 $a = 0.8$) 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.

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

<table>
<thead>
<tr>
<th>$k_1$</th>
<th>LB</th>
<th>Wave</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SP</td>
<td>TS</td>
<td>RTH</td>
</tr>
<tr>
<td>1</td>
<td>175</td>
<td>163</td>
<td>177</td>
</tr>
<tr>
<td>4</td>
<td>198</td>
<td>182</td>
<td>206</td>
</tr>
<tr>
<td>16</td>
<td>247</td>
<td>219</td>
<td>296</td>
</tr>
<tr>
<td>64</td>
<td>308</td>
<td>292</td>
<td>445</td>
</tr>
<tr>
<td>256</td>
<td>438</td>
<td>438</td>
<td>827</td>
</tr>
</tbody>
</table>

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

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 $NN_{k_1}(q)$ as the candidate set, while RNNP forms the candidate set based on $RNN_{k_2}(q)$. When $k_2 \ll k_1$, it is expected that $|RNN_{k_2}(q)| < |NN_{k_1}(q)|$. 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

![Fig. 17. Performance vs. $k_1$ ($k_2 = 16$).](image-url)
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 \( |RNN_{k_2}(q)| < |NN_{k_1}(q)| \) 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 \( |RNN_{k_2}(q)| < |NN_{k_1}(q)| \). 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 \( \text{dim} \) be the dimensionality. In our experiments, we allocate \( 4 \times 2 \times \text{dim} \), 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. 21. Performance vs. dimensionality ($k_1 = k_2 = 16$, cardinality = 512 K).

Fig. 22. Performance vs. cardinality ($k_1 = k_2 = 16$, dimensionality = 3D).
48 x 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. 19B). 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 \( |NN_{k_2}(q)| \ll |NN_{k_1}(q)| \) satisfies when \( k_2 < k_1 \), 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 Figs. 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>
<thead>
<tr>
<th>Dimensionality</th>
<th>Uniform</th>
<th>Zipf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SP</td>
<td>TS</td>
</tr>
<tr>
<td>2D</td>
<td>250</td>
<td>199</td>
</tr>
<tr>
<td>3D</td>
<td>403</td>
<td>340</td>
</tr>
<tr>
<td>4D</td>
<td>747</td>
<td>634</td>
</tr>
<tr>
<td>5D</td>
<td>1449</td>
<td>1283</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cardinality</th>
<th>Uniform</th>
<th>Zipf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SP</td>
<td>TS</td>
</tr>
<tr>
<td>128K</td>
<td>328</td>
<td>272</td>
</tr>
<tr>
<td>256K</td>
<td>353</td>
<td>289</td>
</tr>
<tr>
<td>512K</td>
<td>364</td>
<td>304</td>
</tr>
<tr>
<td>1024K</td>
<td>390</td>
<td>327</td>
</tr>
<tr>
<td>2048K</td>
<td>462</td>
<td>390</td>
</tr>
</tbody>
</table>
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 $k_2 < k_1$ 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


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.