Group Nearest Neighbor Queries

Dimitris Papadias[†] Qiongmao Shen[†] Yufei Tao[§] Kyriakos Mouratidis[†]

[†]*Department of Computer Science*
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{dimitris, qmshen, kyriakos}@cs.ust.hk

[§] *Department of Computer Science*
City University of Hong Kong
Tat Chee Avenue, Hong Kong
taoyf@cs.cityu.edu.hk

Abstract

Given two sets of points P and Q, a group nearest neighbor (GNN) query retrieves the point(s) of P with the smallest sum of distances to all points in Q. Consider, for instance, three users at locations q_1, q_2 and q_3 that want to find a meeting point (e.g., a restaurant); the corresponding query returns the data point p that minimizes the sum of Euclidean distances $|pq_i|$ for $1 \leq i \leq 3$. Assuming that Q fits in memory and P is indexed by an R-tree, we propose several algorithms for finding the group nearest neighbors efficiently. As a second step, we extend our techniques for situations where Q cannot fit in memory, covering both indexed and non-indexed query points. An experimental evaluation identifies the best alternative based on the data and query properties.

1. Introduction

Nearest neighbor (NN) search is one of the oldest problems in computer science. Several algorithms and theoretical performance bounds have been devised for exact and approximate processing in main memory [S91, AMN+98]. Furthermore, the application of NN search to content-based and similarity retrieval has led to the development of numerous cost models [PM97, WSB98, BGRS99, B00] and indexing techniques [SYUK00, YOTJ01] for high-dimensional versions of the problem. In spatial databases most of the work has focused on the *point NN query* that retrieves the k (≥ 1) objects from a dataset P that are closest (usually according to Euclidean distance) to a query point q. The existing algorithms (reviewed in Section 2) assume that P is indexed by a spatial access method and utilize some pruning bounds to restrict the search space. Shahabi et al. [SKS02] and Papadias et al. [PZMT03] deal with nearest neighbor queries in spatial network databases, where the distance between two points is defined as the length of the shortest path connecting them in the network.

In addition to conventional (i.e., point) NN queries, recently there has been an increasing interest in alternative forms of spatial and spatio-temporal NN search. Ferhatosmanoglu et al. [FSAA01] discover the NN in a constrained area of the data space. Korn and Muthukrishnan [KM00] discuss *reverse nearest neighbor queries*, where the goal is to retrieve the data points whose nearest neighbor is a specified query point. Korn et al. [KMS02] study the same problem in the context of data streams. Given a query moving with steady velocity, [SR01, TP02] incrementally maintain the NN (as the query moves), while [BJKS02, TPS02] propose techniques for *continuous NN* processing, where the goal is to return all results up to a future time. Kollios et al. [KGT99] develop various schemes for answering NN queries on 1D moving objects. An overview of existing NN methods for spatial and spatio-temporal databases can be found in [TP03].

In this paper we discuss group nearest neighbor (GNN) queries, a novel form of NN search. The input of the problem consists of a set $P=\{p_1,\ldots,p_N\}$ of static data points in multidimensional space and a group of query points $Q=\{q_1,\ldots,q_n\}$. The output contains the k (≥ 1) data point(s) with the smallest sum of distances to all points in Q. The distance between a data point p and Q is defined as $dist(p,Q)=\sum_{i=1\sim n}|pq_i|$, where $|pq_i|$ is the Euclidean distance between p and query point q_i. As an example consider a database that manages (static) facilities (i.e., dataset P). The query contains a set of user locations $Q=\{q_1,\ldots,q_n\}$ and the result returns the facility that minimizes the total travel distance for all users. In addition to its relevance in geographic information systems and mobile computing applications, GNN search is important in several other domains. For instance, in clustering [JMF99] and outlier detection [AY01], the quality of a solution can be evaluated by the distances between the points and their nearest cluster centroid. Furthermore, the operability and speed of very large circuits depends on the relative distance between the various components in them. GNN can be applied to detect abnormalities and guide relocation of components [NO97].

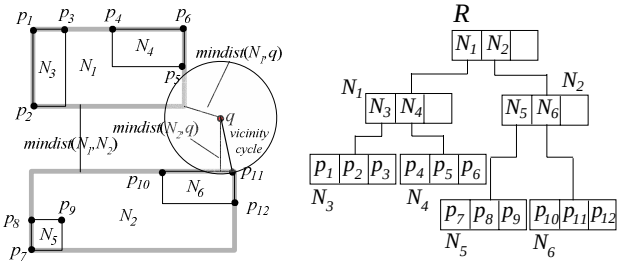
Assuming that Q fits in memory and P is indexed by an R-tree, we first propose three algorithms for solving this problem. Then, we extend our techniques for cases that Q is too large to fit in memory, covering both indexed and non-indexed query points. The rest of the paper is structured as follows. Section 2 outlines the related work on conventional nearest neighbor search and top-k queries. Section 3

describes algorithms for the case that Q fits in memory and Section 4 for the case that Q resides on the disk. Section 5 experimentally evaluates the algorithms and identifies the best one depending on the problem characteristics. Section 6 concludes the paper with directions for future work.

2. Related work

Following most approaches in the relevant literature, we assume 2D data points indexed by an R-tree [G84]. The proposed techniques, however, are applicable to higher dimensions and other data-partition access methods such as A-trees [SYUK00] etc. Figure 2.1 shows an R-tree for point set $P=\{p_1,p_2,\ldots,p_{12}\}$ assuming a capacity of three entries per node. Points that are close in space (e.g., p_1, p_2, p_3) are clustered in the same leaf node (N_3). Nodes are then recursively grouped together with the same principle until the top level, which consists of a single root.

Existing algorithms for point NN queries using R-trees follow the branch-and-bound paradigm, utilizing some metrics to prune the search space. The most common such metric is *mindist*(N,q), which corresponds to the closest possible distance between q and any point in the subtree of node N. Figure 2.1a shows the *mindist* between point q and nodes N_1, N_2. Similarly, *mindist*(N_1,N_2) is the minimum possible distance between any two points that reside in the sub-trees of nodes N_1 and N_2.



(a) Points and node extents (b) The corresponding R-tree

Figure 2.1: Example of an R-tree and a point NN query

The first NN algorithm for R-trees [RKV95] searches the tree in a depth-first (DF) manner. Specifically, starting from the root, it visits the node with the minimum *mindist* from q (e.g., N_1 in Figure 2.1). The process is repeated recursively until the leaf level (node N_4), where the first potential nearest neighbor is found (p_5). During backtracking to the upper level (node N_1), the algorithm only visits entries whose minimum distance is smaller than the distance of the nearest neighbor already retrieved. In the example of Figure 2.1, after discovering p_5, DF will backtrack to the root level (without visiting N_3), and then follow the path N_2,N_6 where the actual NN p_{11} is found.

The DF algorithm is sub-optimal, i.e., it accesses more nodes than necessary. In particular, as proven in [PM97], an optimal algorithm should visit only nodes intersecting the *vicinity circle* that centers at the query point q and has radius equal to the distance between q and its nearest neighbor. In Figure 2.1a, for instance, an optimal algorithm should visit only nodes R, N_1, N_2, and N_6 (whereas DF also visits N_4). The best-first (BF) algorithm of [HS99] achieves the optimal I/O performance by maintaining a heap H with the entries visited so far, sorted by their *mindist*. As with DF, BF starts from the root, and inserts all the entries into H (together with their *mindist*), e.g., in Figure 2.1a, $H=\{<N_1,\ mindist(N_1,q)>,\ <N_2,\ mindist(N_2,q)>\}$. Then, at each step, BF visits the node in H with the smallest *mindist*. Continuing the example, the algorithm retrieves the content of N_1 and inserts all its entries in H, after which $H=\{<N_2,\ mindist(N_2,q)>,\ <N_4,\ mindist(N_4,q)>,\ <N_3,\ mindist(N_3,q)>\}$. Similarly, the next two nodes accessed are N_2 and N_6 (inserted in H after visiting N_2), in which p_{11} is discovered as the current NN. At this time, the algorithm terminates (with p_{11} as the final result) since the next entry (N_4) in H is farther (from q) than p_{11}. Both DF and BF can be easily extended for the retrieval of k>1 nearest neighbors. In addition, BF is also *incremental*. Namely, it reports the nearest neighbors in ascending order of their distance to the query, so that k does not have to be known in advance (allowing different termination conditions to be used).

The branch-and-bound framework also applies to *closest pair queries* that find the pair of objects from two datasets, such that their distance is the minimum among all pairs. [HS98, CMTV00] propose various algorithms based on the concepts of DF and BF traversal. The difference from NN is that the algorithms access two index structures (one for each data set) simultaneously. If the *mindist* of two intermediate nodes N_i and N_j (one from each R-tree) is already greater than the distance of the closest pair of objects found so far, the sub-trees of N_i and N_j cannot contain a closest pair (thus, the pair is pruned).

As shown in the next section, a processing technique for GNN queries applies multiple conventional NN queries (one for each query point) and then combines their results. Some related work on this topic has appeared in the literature of top-k (or *ranked*) queries over multiple data repositories (see [FLN01, BCG02, F02] for representative papers). As an example, consider that a user wants to find the k images that are most similar to a query image, where similarity is defined according to n features, e.g., color histogram, object arrangement, texture, shape etc. The query is submitted to n retrieval engines that return the best matches for particular features together with their similarity scores, i.e., the first engine will output a set of matches according to color, the second according to arrangement and so on. The problem is to combine the multiple inputs in order to determine the top-k results in terms of their overall similarity.

The main idea behind all techniques is to minimize the extent and cost of search performed on each retrieval engine in order to compute the final result. The *threshold algorithm* [FLN01] works as follows (assuming retrieval of

the single best match): the first query is submitted to the first search engine, which returns the closest image p_1 according to the first feature. The similarity between p_1 and the query image with respect to the other features is computed. Then, the second query is submitted to the second search engine, which returns p_2 (best match according to the second feature). The overall similarity of p_2 is also computed, and the best of p_1 and p_2 becomes the current result. The process is repeated in a round-robin fashion, i.e., after the last search engine is queried, the second match is retrieved with respect to the first feature and so on. The algorithm will terminate when the similarity of the current result is higher than the similarity that can be achieved by any subsequent solution. In the next section we adapt this approach to GNN processing.

3. Algorithms for memory-resident queries

Assuming that the set Q of query points fits in memory and that the data points are indexed by an R-tree, we present three algorithms for processing GNN queries. For each algorithm we first illustrate retrieval of a single nearest neighbor, and then show the extension to k>1. Table 3.1 contains the primary symbols used in our description (some have not appeared yet, but will be clarified shortly).

Symbol	Description
Q	set of query points
Q_i	a group of queries that fits in memory
n (n_i)	number of queries in Q (Q_i)
M (M_i)	MBR of Q (Q_i)
q	centroid of Q
$dist(p,Q)$	sum of distances between point p and query points in Q
$mindist(N,q)$	minimum distance between MBR of node N and centroid q
$mindist(p,M)$	minimum distance between data point p and query MBR M
$\sum n_i \cdot mindist(N,M_i)$	*weighted mindist* of node N with respect to all query groups

Table 3.1: Frequently used symbols

3.1 Multiple query method

The *multiple query method* (MQM) utilizes the main idea of the *threshold algorithm*, i.e., it performs incremental NN queries for each point in Q and combines their results. For instance, in Figure 3.1 (where $Q=\{q_1,q_2\}$), MQM retrieves the first NN of q_1 (point p_{10} with $|p_{10}q_1|$=2) and computes the distance $|p_{10}q_2|$ (=5). Similarly, it finds the first NN of q_2 (point p_{11} with $|p_{11}q_2|$=3) and computes $|p_{11}q_1|$(=3). The point (p_{11}) with the minimum sum of distances ($|p_{11}q_1|+|p_{11}q_2|$=6) to all query points becomes the current GNN of Q.

For each query point q_i, MQM stores a threshold t_i, which is the distance of the current NN, i.e., $t_1=|p_{10}q_1|$=2 and $t_2=|p_{11}q_2|$=3. The total threshold T is defined as the sum of all thresholds (=5). Continuing the example, since T < $dist(p_{11},Q)$, it is possible that there exists a point in P whose distance to Q is smaller than $dist(p_{11},Q)$. So MQM retrieves the second NN of q_1 (p_{11}, which has already been encountered by q_2) and updates the threshold t_1 to $|p_{11}q_1|$ (=3). Since T (=6) now equals the summed distance between the best neighbor found so far and the points of Q, MQM terminates with p_{11} as the final result. In other words, every non-encountered point has distance greater or equal to T (=6), and therefore it cannot be closer to Q (in the global sense) than p_{11}.

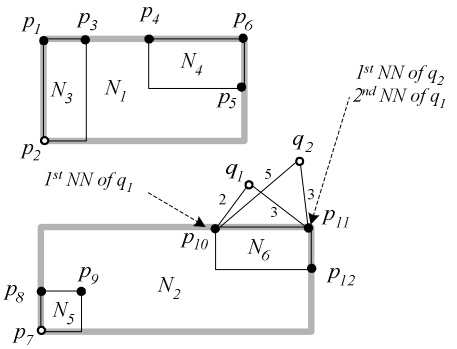


Figure 3.1: Example of a GNN query

Figure 3.2 shows the pseudo code for MQM (1NN), where *best_dist* (initially ∞) is the distance of the *best_NN* found so far. In order to achieve locality of the node accesses for individual queries, we sort the points in Q according to their Hilbert value; thus, two subsequent queries are likely to correspond to nearby points and access similar R-tree nodes. The algorithm for computing nearest neighbors of query points should be incremental (e.g., best-first search discussed in Section 2) because the termination condition is not known in advance. The extension for the retrieval of k (>1) nearest neighbors is straightforward. The k neighbors with the minimum overall distances are inserted in a list of k pairs <p, $dist(p,Q)$> (sorted on $dist(p,Q)$) and *best_dist* equals the distance of the k-th NN. Then, MQM proceeds in the same way as in Figure 3.2, except that whenever a better neighbor is found, it is inserted in *best_NN* and the last element of the list is removed.

```
MQM(Q: group of query points)
/* T : threshold ; best_dist distance of the current NN*/
sort points in Q according to Hilbert value;
for each query point: t_i=0;
T=0; best_dist=∞; best_NN=null; //Initialization
while (T < best_dist)
  get the next nearest neighbor p_j of the next query point q_i;
  t_i = |p_j q_i|; update T;
  if dist(p_j,Q)<best_dist
        best_NN =p_j; //Update current GNN of Q
        best_dist = dist(p_j,Q) ;
 end of while;
return best_NN;
```

Figure 3.2: The MQM algorithm

3.2 Single point method

MQM may incur multiple accesses to the same node (and retrieve the same data point, e.g., p_{11}) through different queries. To avoid this problem, the *single point method* (SPM) processes GNN queries by a single traversal. First, SPM computes the *centroid* q of Q, which is a point in space with a small value of $dist(q,Q)$ (ideally, q is the point with the minimum $dist(q,Q)$). The intuition behind this approach is that the nearest neighbor is a point of P "near" q. It remains to derive (i) the computation of q, and (ii) the range around q in which we should look for points of P, before we conclude that no better NN can be found. Towards the first goal, let (x,y) be the coordinates of centroid q and (x_i,y_i) be the coordinates of query point q_i. The centroid q minimizes the distance function:

$$dist(q,Q)=\sum_{i=1}^{n}\sqrt{(x-x_i)^2+(y-y_i)^2}$$

Since the partial derivatives of function $dist(q,Q)$ with respect to its independent variables x and y are zero at the centroid q, we have the following equations:

$$\begin{cases}\dfrac{\partial\, dist(q,Q)}{\partial\, x}=\sum_{i=1}^{n}\dfrac{x-x_i}{\sqrt{(x-x_i)^2+(y-y_i)^2}}=0\\ \dfrac{\partial\, dist(q,Q)}{\partial\, y}=\sum_{i=1}^{n}\dfrac{y-y_i}{\sqrt{(x-x_i)^2+(y-y_i)^2}}=0\end{cases}$$

Unfortunately, the above equations cannot be solved into closed form for $n>2$, or in other words, they must be evaluated numerically, which implies that the centroid is approximate. In our implementation, we use the *gradient descent* [HYC01] method to quickly obtain a good approximation. Specifically, starting with some arbitrary initial coordinates, e.g. $x=(1/n)\sum_{i=1\sim n}x_i$ and, $y=(1/n)\sum_{i=1\sim n}y_i$, the method modifies the coordinates as follows:

$$x=x-\eta\frac{\partial\, dist(q,Q)}{\partial\, x}\ \text{ and }\ y=y-\eta\frac{\partial\, dist(q,Q)}{\partial\, y},$$

where η is a step size. The process is repeated until the distance function $dist(q,Q)$ converges to a minimum value. Although the resulting point q is only an approximation of the ideal centroid, it suffices for the purposes of SPM. Next we show how q can be used to prune the search space based on the following lemma.

Lemma 1: Let $Q=\{q_1,\ldots,q_n\}$ be a group of query points and q an arbitrary point in space. The following inequality holds for any point p: $dist(p,Q)\geq n\cdot|p\,q|-dist(q,Q)$, where $|pq|$ denotes the Euclidean distance between p and q.

Proof: Due to the triangular inequality, for each query point q_i we have that: $|pq_i|+|q_iq|\geq|pq|$. By summing up the n inequalities:

$$\sum_{q_i\in Q}|pq_i|+\sum_{q_i\in Q}|q_iq|\geq n\cdot|pq|\Rightarrow dist(p,Q)\geq n\cdot|pq|-dist(q,Q)$$

Lemma 1 provides a threshold for the termination of SPM. In particular, by applying an incremental point NN query at q, we stop when we find the first point p such that: $n\cdot|pq|-dist(q,Q)\geq dist(best\_NN,Q)$. By Lemma 1, $dist(p,Q)\geq n\cdot|pq|-dist(q,Q)$ and, therefore, $dist(p,Q)\geq dist(best\_NN,Q)$. The same idea can be used for pruning intermediate nodes, as summarized by the following heuristic.

Heuristic 1: Let q be the centroid of Q and *best_dist* be the distance of the best GNN found so far. Node N can be pruned if:

$$mindist(N,q)\geq\frac{best\_dist+dist(q,Q)}{n}$$

where $mindist(N,q)$ is the minimum distance between the MBR of N and the centroid q. An example of the heuristic is shown in Figure 3.3, where the *best_dist* = 5+4. Since, $dist(q,Q)$=1+2, the right part of the inequality equals 6, meaning that both nodes in the figure will be pruned.

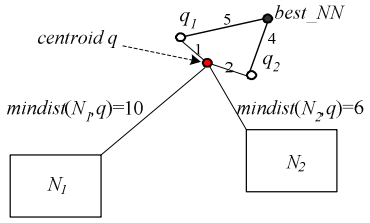


Figure 3.3: Pruning of nodes in SPM

Based on the above observations, it is straightforward to implement SPM using the depth-first or best-first paradigms. Figure 3.4 shows the pseudo-code of DF SPM. Starting from the root of the R-tree (for P), entries are sorted in a *list* according to their *mindist* from the query centroid q and are visited (recursively) in this order. Once the first entry with $mindist(N_j,q)\geq(best\_dist+dist(q,Q))/n$ has been found, the subsequent ones in the *list* are pruned. The extension to k (>1) GNN queries is the same as conventional (point) NN algorithms.

```
SPM(Node: R-tree node, Q: group of query points)
/* q: the centroid of Q*/
if Node is an intermediate node
 sort entries N_j in Node according to mindist(N_j,q) in list;
 repeat
   get_next entry N_j from list;
   if mindist(N_j,q)< (best_dist+dist(q,Q))/n; /* Heuristic 1
         SPM(N_j,Q); /* recursion*/
 until mindist(N_j,q) ≥ (best_dist+dist(q,Q))/n or end of list;
else if Node is a leaf node
 sort points p_j in Node according to mindist(p_j,q) in list;
 repeat
   get_next entry p_j from list;
   if |p_jq|<(best_dist+dist(q,Q))/n; /* Heuristic 1 for points
         if dist(p_j,Q)< best_dist
               best_NN =p_j; //Update current GNN
               best_dist = dist(p_j,Q) ;
 until |p_jq|≥ (best_dist+dist(q,Q))/n or end of list;
return best_NN;
```

Figure 3.4: The SPM algorithm

3.3 Minimum bounding method

Like SPM, the *minimum bounding method* (MBM) performs a single query, but uses the minimum bounding rectangle M of Q (instead of the centroid q) to prune the search space. Specifically, starting from the root of the R-tree for dataset P, MBM visits only nodes that may contain candidate points. In the sequel, we discuss heuristics for identifying such qualifying nodes.

Heuristic 2: Let M be the MBR of Q, and *best_dist* be the distance of the best GNN found so far. A node N cannot contain qualifying points, if:

$$mindist(N,M) \geq \frac{best\_dist}{n}$$

where *mindist*(N,M) is the minimum distance between M and N, and n is the cardinality of Q. Figure 3.5 shows a group of query points $Q=\{q_1,q_2\}$ and the *best_NN* with *best_dist*=5. Since *mindist*(N_1,M) = 3 > *best_dist*/2 = 2.5, N_1 can be pruned without being visited. In other words, even if there is a data point p at the upper-right corner of N_1 and all the query points were at the lower right corner of Q, it would still be the case that *dist*(p,Q)> *best_dist*. The concept of heuristic 2 also applies to the leaf entries. When a point p is encountered, we first compute *mindist*(p,M) from p to the MBR of Q. If *mindist*(p,M) ≥ *best_dist*/n, p is discarded since it cannot be closer than the *best_NN*. In this way we avoid performing the distance computations between p and the points of Q.

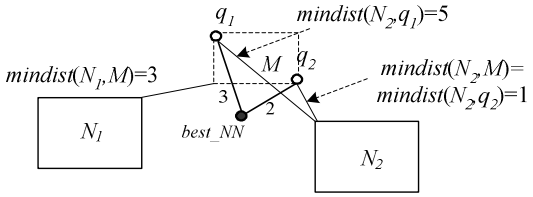


Figure 3.5: Example of heuristic 2

The heuristic incurs minimum overhead, since for every node it requires a single distance computation. However, it is not very tight, i.e., it leads to unnecessary node accesses. For instance, node N_2 (in Figure 3.5) passes heuristic 2 (and should be visited), although it cannot contain qualifying points. Heuristic 3 presents a tighter bound for avoiding such visits.

Heuristic 3: Let *best_dist* be the distance of the best GNN found so far. A node N can be safely pruned if:

$$\sum_{q_i \in Q} mindist(N,q_i) \geq best\_dist$$

where *mindist*(N,q_i) is the minimum distance between N and query point $q_i \in Q$. In Figure 3.5, since *mindist*(N_2, q_1) + *mindist*(N_2, q_2) = 6 > *best_dist* = 5, N_2 is pruned.

Because heuristic 3 requires multiple distance computations (one for each query point) it is applied only for nodes that pass heuristic 2. Note that (like heuristic 2) heuristic 3 does represent the tightest condition for successful node visits; i.e., it is possible for a node to satisfy the heuristic and still not contain qualifying points. Consider, for instance, Figure 3.6, which includes 3 query points. The current *best_dist* is 7, and node N_3 passes heuristic 3, since *mindist*(N_3,q_1) + *mindist*(N_3,q_2) + *mindist*(N_3,q_3) = 5. Nevertheless, N_3 should not be visited, because the minimum distance that can be achieved by any point in N_3 is greater than 7. The dotted lines in Figure 3.6 correspond to the distance between the best possible point p' (not necessarily a data point) in N_3 and the three query points.

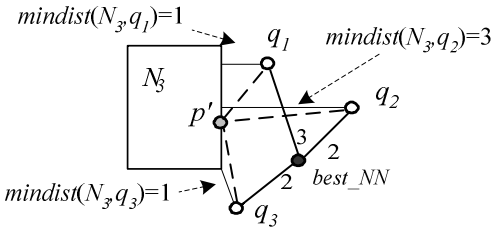


Figure 3.6: Example of a hypothetical optimal heuristic

Assuming that we can identify the best point p' in the node, we can obtain a tight heuristic a follows: if the distance of p' is smaller than *best_dist* visit the node; otherwise, reject it. The combination of the best-first approach with this heuristic would lead to an I/O optimal method (such as the algorithm of [HS99] for conventional NN queries). Finding point p', however, is similar to the problem of locating the query centroid (but this time in a region constrained by the node MBR), which, as discussed in Section 3.2, can only be solved numerically (i.e., approximately). Although an approximation suffices for SPM, for the correctness of *best_dist* it is necessary to have the precise solution (in order to avoid false misses). As a result, this hypothetical heuristic cannot be applied for exact GNN retrieval.

Heuristics 2 and 3 can be used with both the depth-first and best-first traversal paradigms. For simplicity, we discuss MBM based on depth-fist traversal using the example of Figure 3.7. The root of the R-tree is retrieved and its entries are sorted by their *mindist* to M. Then, the node (N_1) with the minimum *mindist* is visited, inside which the entry of N_4 has the smallest *mindist*. Points p_5, p_6, p_4 (in N_4) are processed according to the value of *mindist*(p_j,M) and p_5 becomes the current GNN of Q (*best_dist*=11). Points p_6 and p_4 have larger distances and are discarded. When backtracking to N_1, the subtree of N_3 is pruned by heuristic 2. Thus, MBM backtracks again to the root and visits nodes N_2 and N_6, inside which p_{10} has the smallest *mindist* to M and is processed first, replacing p_5 as the GNN (*best_dist*=7). Then, p_{11} becomes the best NN (*best_dist*=6). Finally, N_5 is pruned by heuristic 2, and the algorithm terminates with p_{11} as the final GNN. The extension to retrieval of *k*NN and the best-first implementation are straightforward.

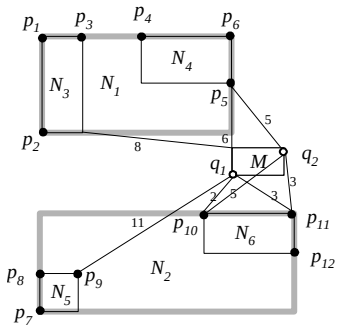


Figure 3.7: Query processing of MBM

4. Algorithms for disk-resident queries

We now discuss the situation that the query set does not fit in main memory. Section 4.1 considers that Q is indexed by an R-tree, and shows how to adapt the R-tree closest pair (CP) algorithm [HS98, CMTV00] for GNN queries with additional pruning rules. We argue, however, that the R-tree on Q offers limited benefits towards reducing the query time. Motivated by this, in Sections 4.2 and 4.3 we develop two alternative methods, based on MQM and MBM, which do not require any index on Q. Again, for simplicity, we describe the algorithms for single NN retrieval before discussing k>1.

4.1 Group closest pairs method

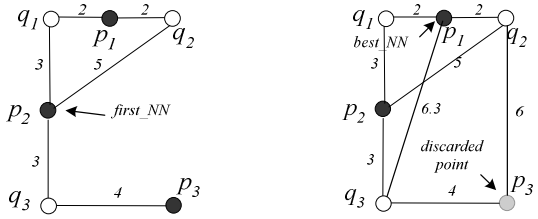
Assume an incremental CP algorithm that outputs closest pairs $<p_i,q_j>$ ($p_i \in P$, $q_j \in Q$) in ascending order of their distance. Consider that we keep the $count(p_i)$ of pairs in which p_i has appeared, as well as, the accumulated distance ($curr\_dist(p_i)$) of p_i in all these pairs. When the count of p_i equals the cardinality n of Q, the global distance of p_i, with respect to all query points, has been computed. If this distance is smaller than the best global distance (*best_dist*) found so far, p_i becomes the current NN.

Two questions remain to be answered: (i) which are the *qualifying* data points that can lead to a better solution? (ii) when can the algorithm terminate? Regarding the first question, clearly all points encountered before the first complete NN is found, are qualifying. Every such point p_i is kept in a list $< p_i, count(p_i), curr\_dist(p_i)>$. On the other hand, if we already have a complete NN, every data point that is encountered for the first time can be discarded since it cannot lead to a better solution. In general, the list of qualifying points keeps increasing until a complete NN is found. Then, non-qualifying points can be gradually removed from the list based on the following heuristic:

Heuristic 4: Assume that the current output of the CP algorithm is $<p_i,q_j>$. We can immediately discard all points p such that:

$$(n\text{-counter}(p)) \cdot dist(p_i,q_j) + curr\_dist(p) \geq best\_dist$$

In other words, p cannot yield a global distance smaller than *best_dist*, even if all its un-computed distances are equal to $dist(p_i,q_j)$. Heuristic 4 is applied in two cases: (i) for each output pair $<p_i,q_j>$, on the data point p_i and (ii) when the global NN changes, on all qualifying points. Every point p that fails the heuristic is deleted from the qualifying list. If p is encountered again in a subsequent pair, it will be considered as a new point and pruned. Figure 4.1a shows an example where the closest pairs are found incrementally according to their distance i.e., ($<p_1,q_1>$, 2), ($< p_1,q_2>$, 2), ($< p_2,q_1>$, 3), ($< p_2,q_3>$, 3), ($< p_3,q_3>$, 4), ($<p_2,q_2>$, 5). After pair $<p_2,q_2>$ is output, we have a complete NN, p_2 with global distance 11. Heuristic 4 is applied to all qualifying points and p_3 is discarded; even if its (non yet discovered) distances to q_1 and q_2 equal 5, its global distance will be 14 (i.e., greater than *best_dist*).



(a) Discovery of 1st NN (b) Termination

Figure 4.1: Example of GCP

For each remaining qualifying point p_i, we compute a *threshold* t_i as: $t_i=(best\_dist-curr\_dist(p_i)) / (n\text{-counter}(p_i))$. In the general case, that multiple qualifying points exist, the global threshold T is the maximum of individual thresholds t_i, i.e., T is the largest distance of the output closest pair that can lead to a better solution than the existing one. In Figure 4.1a, for instance, $T=t_1=7$, meaning that when the output pair has distance ≥ 7, the algorithm can terminate. Every application of heuristic 4 also modifies the corresponding thresholds, so that the value of T is always up to date. Based on these observations we are now ready to establish the termination condition, i.e., GCP terminates when (i) at least a GNN has been found ($best\_dist<\infty$) and (ii) the qualifying list is empty, or the distance of the current pair becomes larger than the global threshold T. Figure 4.1b continues the example of Figure 4.1a. In this case the algorithm terminates after the pair ($< p_1,q_3>$, 6.3) is found, which establishes p_1 as the best NN (and the list becomes empty).

The pseudo-code of the GCP is shown in Figure 4.2. We store the qualifying list as an in-memory hash table on point ids to facilitate the retrieval of information (i.e., $counter(p_i)$, $curr\_dist(p_i)$) about particular points (p_i). If the size of the list exceeds the available memory, part of the table is stored to the disk[1]. In case of kNN queries, *best_dist* equals the global distance of the k-th complete neighbor found so far (i.e., pruning in the qualifying list can occur only after k complete neighbors are retrieved).

[1] In the worst case, the list may contain an entry for each point of P.

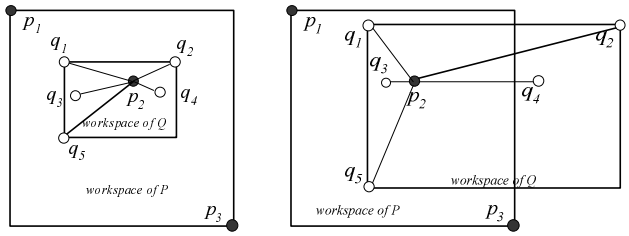
GCP

best_NN = NULL; best_dist = ∞; / initialization*
repeat
 output next closest pair <p_i,q_j> and *dist*(p_i,q_j)
  if p_i is not in *list*
    if *best_dist* < ∞ continue*; /* discard p_i and process next pair*
    else add < p_i, 1, *dist*(p_i,q_j)> in *list*;
  else /* p_i *has been encountered before and still resides in list*
    counter(p_i)++; *curr_dist*(p_i)= *curr_dist*(p_i)+ *dist*(p_i,q_j);
    if *counter*(p_i)= *n*
      if *curr_dist*(p_i)< *best_dist*
        best_NN = p_i; *//Update current GNN*
        best_dist = *curr_dist*(p_i); *T*=0*;*
        for each candidate point *p* in *list*
          if (*n-counter*(*p*))· *dist*(p_i,q_j)+*curr_dist*(*p*) ≥ *best_dist*
            remove *p* from *list*; */* pruned by heuristic 6*
          else */* p not pruned by heuristic 6*
            t= (*best_dist-curr_dist*(*p*)) / (*n-counter*(*p*));
            if *t* > *T* then *T* = *t*; */* update threshold*
      else remove p_i from *list*;
    else /* *counter*(p_i)< *n*
      if *best_dist* < ∞ */* a NN has been found already*
        if (*n-counter*(p_i))· *dist*(p_i,q_j)+*curr_dist*(p_i) ≥ *best_dist*
          remove p_i from *list*; */* pruned by heuristic 6*
        else */*not pruned by heuristic 6*
          t_i= (*best_dist-curr_dist*(p_i)) / (*n-counter*(p_i));
          if t_i > *T* then *T* = t_i; */* update threshold*
until (*best_dist* < ∞) and (*dist*(p_i,q_j) ≥ *T* or list is *empty*);
return *best_NN*;

Figure 4.2: The GCP algorithm

When the workspace (i.e., MBR) of *Q* is small and contained in the workspace of *P*, GCP can terminate after outputting a small percentage of the total number of closest pairs. Consider, for instance, Figure 4.3a, where there exist some points of *P* (e.g., p_2) that are near all query points. The number of closest pairs that must be considered depends only on the distance between p_2 and its farthest neighbor (q_5) in *Q*. Data point p_3, for example, will not participate in any output closest pair since its nearest distance to any query point is larger than $|p_2q_5|$.

On the other hand, if the MBR of *Q* is large or partially overlaps (or is disjoint) with the workspace of *P*, GCP must output many closest-pairs before it terminates. Figure 4.3b, shows such an example, where the distance between the *best_NN* (p_2) and its farthest query point (q_2) is high. In addition to the computational overhead of GCP in this case, another disadvantage is its large heap requirements. Recall that GCP applies an incremental CP algorithm that must keep all closest pairs in the heap until the first NN is found. The number of such pairs in the worst case equals the cardinality of the Cartesian product of the datasets[2]. To alleviate the problem, Hjaltason and Samet [HS99] proposed a heap management technique (included in our implementation), according to which, part of the heap migrates to the disk when its size exceeds the available memory space. Nevertheless, as shown in Section 5, the cost of GCP is often very high, which motivates the subsequent algorithms.



(a) High pruning (b) Low pruning

Figure 4.3: Observations about the performance of GCP

4.2 F-MQM

MQM can be applied directly for disk-resident, non-indexed *Q*, with however, very high cost due to the large number of individual queries that must be performed (as shown in Section 5, its cost increases fast with the cardinality of *Q*). In order to overcome this problem, we propose F-MQM (*file-multiple query method*), which splits *Q* into blocks {Q_1, .., Q_m} that fit in memory. For each block, it computes the GNN using one of the main memory algorithms (we apply MBM due to its superior performance - see Section 5), and finally it combines their results using MQM. The complication is that once a NN of a group has been retrieved, we cannot effectively compute its global distance (i.e., with respect to all data points) immediately. Instead, we follow a *lazy* approach: first we find the GNN p_1 of the first group Q_1; then, we load in memory the second group Q_2 and retrieve its NN p_2. At the same time, we also compute the distance between p_1 and Q_2, whose current distance becomes *curr_dist*(p_1) = *dist*(p_1,Q_1) + *dist*(p_1,Q_2). Similarly, when we load Q_3, we update the current distances of p_1 and p_2 taking into account the objects of the third group. After the end of the first round, we only have one data point (p_1), whose *global* distance with respect to all query points has been computed. This point becomes the current NN.

The process is repeated in a round robin fashion and at each step a new global distance is derived. For instance, when we read again the first group (to retrieve its second NN), the distance of p_2 (first NN of Q_2) is completed with respect to all groups. Between p_1 and p_2, the point with the minimum global distance becomes the current NN. As in the case of MQM, the threshold t_j for each group Q_j equals *dist*(p_j,Q_j), where p_j is the last retrieved neighbor of Q_j. The global threshold *T* is the sum of all thresholds. F-MQM terminates when *T* becomes equal or larger than the global distance of the best NN found so far.

[2] This may happen if there is a data point (on the corner of the workspace) such that (i) its distance to most query points is very small (so that the point cannot be pruned) and (ii) its distance to a query point (located on the opposite corner of the workspace) is the largest possible.

The algorithm is illustrated in Figure 4.4. In order to achieve locality, we first sort (externally) the points of Q according to their Hilbert value. Then, each group is obtained by taking a number of consecutive pages that fit in memory. The extension for the retrieval of k (>1) GNNs is similar to main-memory MQM. In particular, *best_NN* is now a list of k pairs $<p, dist(p,Q)>$ (sorted by the global $dist(p,Q)$) and *best_dist* equals the distance of the k-th NN. Then, it proceeds in the same way as in Figure 4.4.

F-MQM(Q: group of query points)
best_NN = NULL; best_dist = ∞; T=0; /* *initialization*
sort points of Q according to Hilbert value and split them into groups $\{Q_1, .., Q_m\}$ so that each group fits in memory;
while (T < *best_dist*)
  read next group Q_j;
  get the next nearest neighbor p_j of group Q_j ;
  $curr\_dist(p_j) = dist(p_j,Q_j)$;
  t_j = dist(p_j,Q_j); update T;
  if it is the first pass of the algorithm
    for each cur. neighbor p_i of Q_i ($1 \le i < j$) /*update other NN
      $curr\_dist(p_i) = curr\_dist(p_i) + dist(p_i,Q_j)$;
  else /*local NN have been computed for all m groups
    for each cur. neighbor p_i of Q_i ($1 \le i \le m, i \ne j$) /*update other NN
      $curr\_dist(p_i) = curr\_dist(p_i) + dist(p_i,Q_j)$;
    next=(j+1) modulo m; /*group whose global dist. is complete
    if $curr\_dist(p_{next}) < best\_dist$
      $best\_NN = p_{next}$; /*update current GNN of Q
      $best\_dist = curr\_dist(p_{next})$;
  next=(j+1) modulo m; /*next group to process
end while;
return *best_NN*;

Figure 4.4: The F-MQM algorithm

F-MQM is expected to perform well if the number of query groups is relatively small, minimizing the number of applications of the main memory algorithm. On the other hand, if there are numerous groups, the combination of the individual results may be expensive. Furthermore, as in the case of (main-memory) MQM, the algorithm may perform redundant computations, if it encounters the same data point as a nearest neighbor of different query groups. A possible optimization is to keep each NN in memory, together with its distances to all groups, so that we avoid these computations if the same point is encountered later through another group. This however, may not be possible if the main memory size is limited.

4.3 F-MBM

We can extend both SPM and MBM for the case that Q does not fit in memory. Since, as shown in the experiments, MBM is more efficient, here we describe F-MBM, an adaptation of the minimum bounding method. First, the points of Q are sorted by their Hilbert value and are inserted in pages according to this order. A page Q_i contains n_i points (it is possible that the number of points differs, e.g., the last page may be half-full). For each group Q_i, we keep in memory its MBR M_i and n_i (but not its contents). F-MBM descends the R-tree of P (in DF or BF traversal), only following nodes that may contain qualifying points. Given that we have the values of M_i and n_i for each query group in memory, we can quickly identify qualifying nodes as follows.

Heuristic 5: Let *best_dist* be the distance of the best GNN found so far and M_i be the MBR of group Q_i. A node N can be safely pruned if:

$$\sum_{Q_i \in Q} n_i \cdot mindist(N,M_i) \ge best\_dist$$

We refer to the left part of the inequality as the *weighted mindist* of N. Figure 4.5 shows an example, where 5 query points are split into two groups with MBRs M_1, M_2 and *best_dist* = 20. According to heuristic 5, N can be pruned because its *weighted mindist* ($2 \cdot mindist(N,M_1)$ + $3 \cdot mindist(N,M_2)$) is 20, and it cannot contain a better NN.

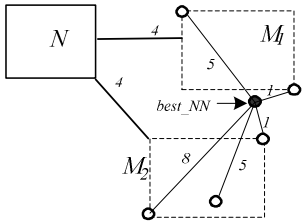


Figure 4.5: Example of heuristic 5

When a leaf node N is reached, we have to compute the global distance of its data points with all groups. Initially the current distance $curr\_dist(p_j)$ of each point $p_j \in N$ is set to 0. Then, for each new group Q_i ($1 \le i \le m$) that is loaded in memory, $curr\_dist(p_j)$ is updated as $curr\_dist(p_j) + dist(p_j,Q_i)$. We can reduce the CPU-overhead of the distance computations based on the following heuristic.

Heuristic 6: Let $curr\_dist(p_j)$ be the accumulated distance of data point p_j with respect to groups $Q_1,.., Q_{i-1}$. Then, p_j can be safely excluded from further consideration if:

$$curr\_dist(p_j) + \sum_{l=i}^{n} n_l \cdot mindist(p_j,M_l) \ge best\_dist$$

Figure 4.6 shows an example of heuristic 6, where the first group Q_1 has been processed and $curr\_dist(p_j) = dist(p_j,Q_1) = 5+3$. Point p_j is not compared with the query points of Q_2, since $8+3 \cdot mindist(p_j,M_2)=20$ is already equal to *best_dist*. Thus, p_j will not be considered for further computations (i.e., when subsequent groups are loaded in memory).

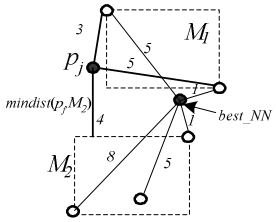


Figure 4.6: Example of heuristic 6

The final clarification regards the order according to which qualifying nodes and query groups are accessed. For nodes we use the *weighted mindist*, based on the intuition that nodes with small values are likely to lead to neighbors with small global distance, so that subsequent visits can be pruned by heuristic 5. When a leaf node N has been reached, each group Q_i is read in memory in descending order of $mindist(N,M_i)$. The motivation is that groups that are far from the node are likely to prune numerous data points (thus, saving the distance computations for these points with respect to other groups). Figure 4.7 shows the pseudo-code of F-MBM based on DF traversal (the BF implementation is similar).

```
F-MBM(Node: R-tree node, Q: group of query points)
/* Q consists of {Q1, .., Qm} that fit in memory
if Node is an intermediate node
 sort entries Nj in Node (according to weighted mindist) in list;
 repeat
   get_next entry Nj from list;
    if weighted mindist(Nj)< best_dist  /*N passes heuristic 5
       F-MBM(Nj, Q) ; /* Recursion
 until weighted mindist(Nj)≥ best_dist  or end of list;
else if Node is a leaf node
 sort points pj in Node (according to weighted mindist) in list;
 for each point pj in list : curr_dist(pj)=0; /* initialization
 sort groups Qi in descending order of mindist(Node, Mi) ;
  repeat
   read next group Qi (1≤i≤m) ;
   for each point pj in list
        if curr_dist(pj)+ Σ_{l=i}^{n} nl · mindist(pj,Ml) ≥ best_dist
          remove pj from list; /* pj fails heuristic 6
        else  /* pj passes heuristic 6
          curr_dist(pj)= curr_dist(pj)+dist(pj,Qi) ;
  until weighted mindist(pj)≥best_dist or end list or end of groups;
  for each point p that remains in list  /*after termination of loops
   if  curr_dist(p)< best_dist
            best_NN =p;   //Update current GNN
             best_dist =  curr_dist(p) ;
return best_NN;
```

Figure 4.7: The F-MBM algorithm

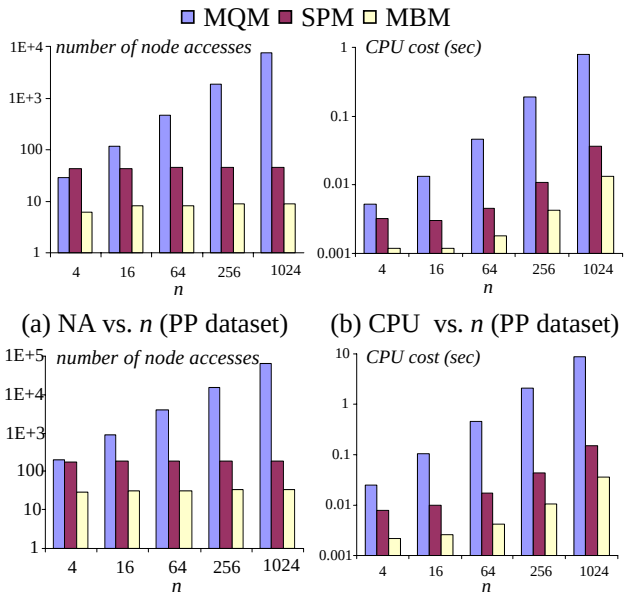
Starting from the root of the R-tree of P, entries are sorted by their *weighted mindist*, and visited (recursively) in this order. Once the first node that fails heuristic 5 is found, all subsequent nodes in the sorted list can also be pruned. For leaf nodes, if a point violates heuristic 6, it is removed from the list and is not compared with subsequent groups. The extension to k NN is straightforward.

5. Experiments

In this section we evaluate the efficiency of the proposed algorithms, using two real datasets: (i) PP [Web1] with 24493 populated places in North America, and (ii) TS [Web2], which contains the centroids of 194971 MBRs representing streams (poly-lines) of Iowa, Kansas, Missouri and Nebraska. For all experiments we use a Pentium 2.4GHz CPU with 1GByte memory. The page size of the R*-trees [BKSS00] is set to 1KByte, resulting in a capacity of 50 entries per node. All implementations are based on the best-first traversal. Both versions of MQM and GCP require BF due to their incremental behavior. SPM and MBM (or F-MBM) could also be used with DF.

5.1 Comparison of algorithms for memory-resident queries

We first compare the methods of Section 3 (MQM, SPM and MBM) for main-memory queries. For this purpose, we use workloads of 100 queries. Each query has a number n of points, distributed uniformly in a MBR of area M, which is randomly generated in the workspace of P. The values of n and M are identical for all queries in the same workload (i.e., the only change between two queries in the same workload is the position of the query MBR). First we study the effect of the cardinality of Q, by fixing M to 8% of the workspace of P and the number k of retrieved group nearest neighbors to 8. Figure 5.1 shows the average number of node accesses (NA) and CPU cost as functions of n for datasets PP and TS.



(a) NA vs. n (PP dataset) (b) CPU vs. n (PP dataset)

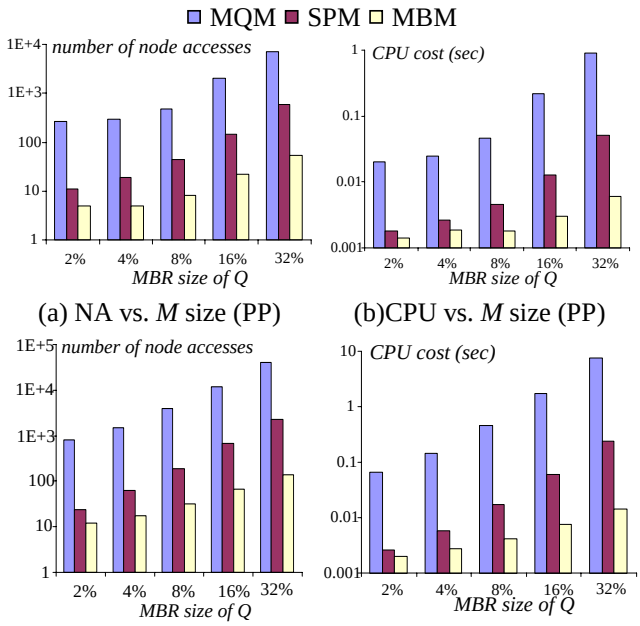
(c) NA vs. n (TS dataset) (d) CPU vs. n (TS dataset)

Figure 5.1: Cost vs. cardinality n of Q (M=8%, k=8)

MQM is, in general, the worst method and its cost increases fast with the query cardinality, because this leads to multiple queries, some of which access the same nodes and retrieve the same points. These redundant computations, affect both the node accesses and the CPU cost significantly (all diagrams are in logarithmic scale). Although most queries access similar paths in the R-tree of P (and, therefore, MQM benefits from the existence of an LRU buffer), its total cost is still prohibitive for large n due to the

high CPU overhead. On the other hand, the cardinality of Q has little effect on the node accesses of SPM and MBM because it does not play an important role in the pruning power of heuristic 1 (for SPM) and heuristics 2, 3 (for MBM). It affects, however, the CPU time, because the distance computations for qualifying data points increase with the number of query points. MBM is better than SPM due to the high pruning power of heuristic 3, as opposed to heuristic 1[3].

In order to measure the effect of the MBR size of Q, we set n=64, k=8 and vary M from 2% to 32% of the workspace of P. As shown in Figure 5.2, the cost (average NA and CPU time) of all algorithms increases with the query MBR. For MQM, the termination condition is that the total threshold T (i.e., sum of thresholds for each query point) should exceed *best_dist*, which, however, increases with the MBR size. Therefore, MQM retrieves more NNs for each query point. For SPM (MBM), the reason is the degradation of pruning power of heuristic 1 (heuristic 2 and 3) with the MBR size of Q.

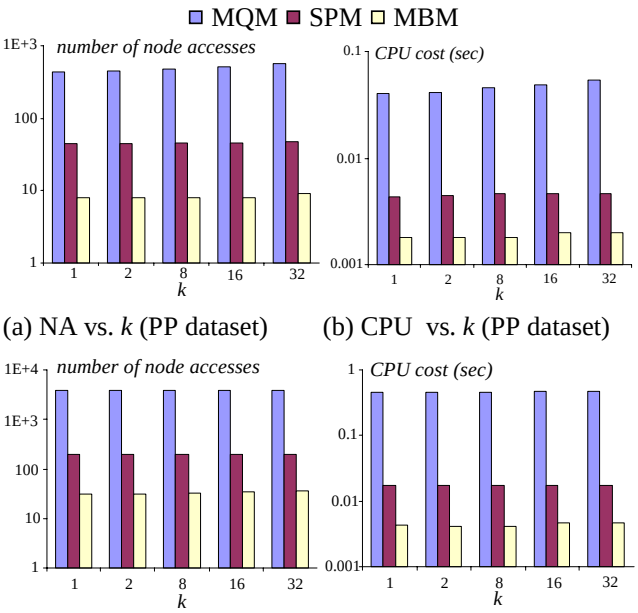


(a) NA vs. M size (PP) (b)CPU vs. M size (PP)

(c) NA vs. M size (TS) (d)CPU vs. M size (TS)

Figure 5.2: Cost vs. size of MBR of Q (n=64, k=8)

Finally, in Figure 5.3, we set n= 64, M=8% and vary the number k of retrieved neighbors from 1 to 32. The value of k does not influence the cost of any method significantly, because in most cases a large number of neighbors are found in the same node with a few extra computations. The relative performance of the algorithms is similar to the previous diagrams: MBM is clearly the most efficient method, followed by SPM.



(a) NA vs. k (PP dataset) (b) CPU vs. k (PP dataset)

(c) NA vs. k (TS dataset) (d) CPU vs. k (TS dataset)

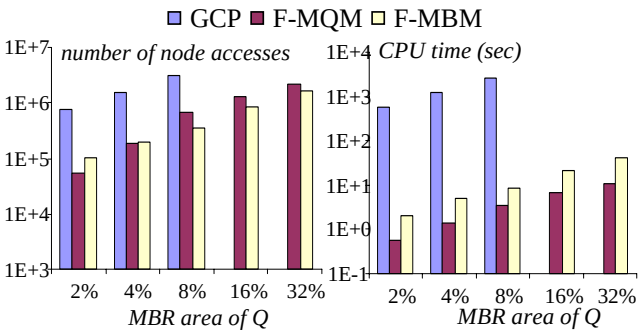
Figure 5.3: Cost vs. num. of retrieved NNs (n=64, M=8%)

5.2 Comparison of algorithms for disk-resident queries

For this set of experiments we use both datasets (PP, TS) alternatively as query and data points. For GCP we assume that both datasets are indexed by R-trees, whereas for F-MQM and F-MBM, the dataset that plays the role of Q is sorted (according to Hilbert values) and split into blocks of 10000 points, that fit in memory. The cost of sorting and building the R-trees is not taken into account. Since now the query cardinality n is fixed to that of the corresponding dataset, we perform experiments by varying the relative workspaces of the two datasets.

First, we assume that the workspaces of P and Q have the same centroid, but the area M (of the MBR of Q) varies between 2% and 32% of the workspace of P (similar to the experiments of Figure 5.2). Figure 5.4 shows NA and CPU time assuming that PP is the query dataset and k=8. GCP has the worst performance and its cost increases fast with M for the reasons discussed in Section 4.1. When M exceeds 8% percent of the workspace of P, GCP does not terminate at all due to the huge heap requirements. The other two algorithms are more than an order of magnitude faster. F-MQM outperforms F-MBM, except for NA in case of large (> 4%) query workspaces. The good performance of F-MQM (compared to the main-memory results) is due to the fact that the query set (PP) contains 24493 data points and, therefore, it generates only 3 query groups. Each query group is processed in memory (by MBM) and their results are combined with relatively small overhead.

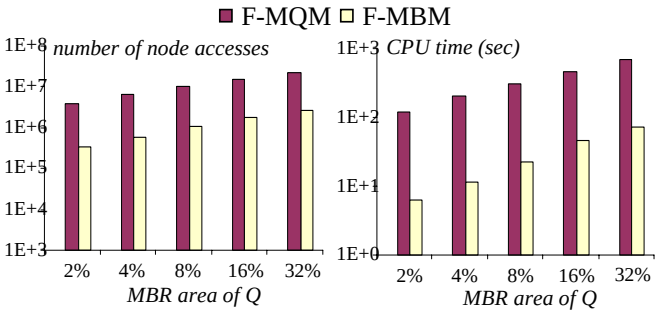
[3] We implemented a version of MBM with only heuristic 2 and we found it inferior to SPM. Nevertheless, heuristic 2 is useful (in conjunction with heuristic 3) because it reduces the CPU time requirements of the algorithm.



(a) NA vs. M size (b) CPU vs. M size

Figure 5.4: Cost vs. size of MBR of Q (k=8, P=TS, Q=PP)

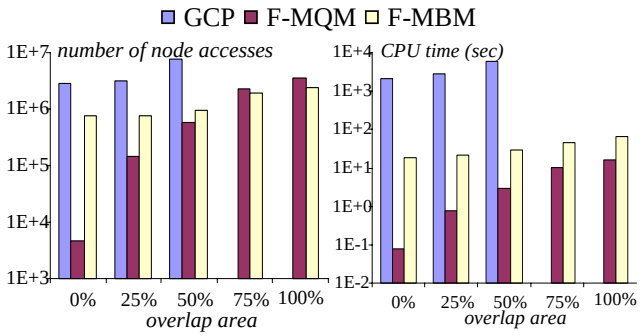
Figure 5.5 illustrates a similar experiment, where PP plays the role of the dataset and TS the role of the query set (recall that the cardinality of TS is almost an order of magnitude higher than that of PP). In this case F-MBM is clearly better, due to the large number (20) of query groups whose results must be combined by F-MQM. Comparing Figure 5.5 with 5.4, we observe that the performance of F-MBM is similar, while F-MQM is significantly worse. This is consistent with the main-memory behavior of MQM (Figure 5.1) where the cost increases fast with the cardinality of the query set. GCP is omitted from the diagrams because it incurs excessively high cost.



(a) NA vs. M size (b) CPU vs. M size

Figure 5.5: Cost vs. size of MBR of Q (k=8, P=PP, Q=TS)

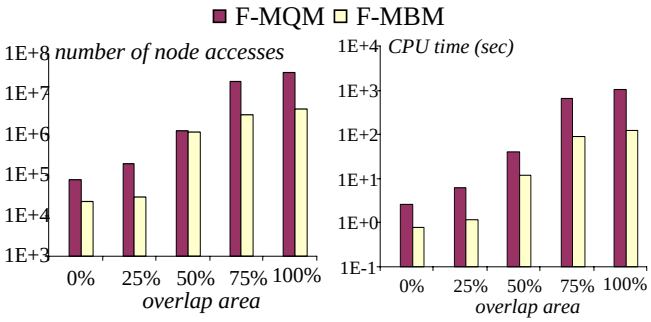
In order to further investigate the effect of the relative workspace positions, for the next set of experiments we assume that both datasets lie in workspaces of the same size, and vary the overlap area between the workspaces from 0% (i.e., P and Q are totally disjoint) to 100% (i.e. on top of each other). Intermediate values are obtained by starting from the 100% case and shifting the query dataset on both axes. Figure 5.6 shows the cost of the algorithms assuming that Q=PP. The cost of all algorithms grows fast with the overlap area because it: (i) increases the number of potential candidates within the threshold of F-MQM (ii) reduces the pruning power of F-MBM heuristics and (iii) increases the number of closest pairs that must be output before the termination of GCP. F-MQM clearly outperforms F-MBM for up to 50% overlap. In order to explain this, let us consider the 0% overlap case assuming that the query workspace starts at the upper-right corner of the data workspace. The nearest neighbors of all query groups must lie near this upper-right corner, since such points minimize the total distance. Therefore, F-MQM can find the best NN relatively fast, and terminate when all the points in or near the corner have been considered. On the other hand, because each query group has a large MBR (recall that it contains 10000 points), numerous nodes satisfy the pruning heuristic of F-MBM and are visited.



(a) NA vs. overlap area (b) CPU vs. overlap area

Figure 5.6: Cost vs. overlap area (k=8, P=TS, Q=PP)

Figure 5.7 repeats the experiment by setting Q=TS. The clear winner is F-MBM, again due to the numerous queries that must be performed by F-MQM. We also performed experiments by varying the number of neighbors retrieved, while keeping the other parameters fixed. As in the case of main-memory queries, k does not have a significant effect on performance (and the diagrams are omitted).



(a) NA vs. overlap area (b) CPU vs. overlap area

Figure 5.7: Cost vs. overlap area (k=8, P=PP, Q=TS)

In summary, the best algorithm for disk-resident queries depends on the number of query groups. F-MQM is usually preferable when the query dataset is partitioned in a small number of groups; otherwise, F-MBM is better. GCP has very poor performance in all cases. We also experimented with an alternative version of MBM that uses an R-tree on Q (instead of Hilbert sorting). The technique, however, did not provide performance benefits because for each qualifying point of P we have to compute its accumulated distance to all query points anyway.

6. Conclusion

Given a dataset *P* and a group of query points *Q*, a group nearest neighbor query retrieves the point of *P* that minimizes the sum of distances to all points in *Q*. In this paper we describe several algorithms for processing such queries, including main-memory and disk-resident *Q*, and experimentally evaluate their performance under a variety of settings. Since the problem is by definition expensive, the performance of different algorithms normally varies up to orders of magnitude, which motivates efficient processing methods.

In the future we intend to explore the application of related techniques to variations of group nearest neighbor search. Consider, for instance, that *Q* represents a set of facilities and the goal is to assign each object of *P* to a single facility so that the sum of distances (of each object to its nearest facility) is minimized. Additional constraints (e.g., a facility may serve at most *k* users) may further complicate the solutions. Similar problems have been studied in the context of clustering and recourse allocation, but the proposed methods are different from the ones presented in this paper. Furthermore, it would be interesting to study other distance metrics (e.g., network distance) that necessitate alternative pruning heuristics and algorithms.

Acknowledgements

This work was supported by grant HKUST 6180/03E from Hong Kong RGC.

References

[AMN+98] Arya, S., Mount, D., Netanyahu, N., Silverman, R., Wu, A. An Optimal Algorithm for Approximate Nearest Neighbor Searching, *Journal of the ACM*, 45(6): 891-923, 1998.

[AY01] Aggrawal, C., Yu, P. Outlier Detection for High Dimensional Data. *SIGMOD*, 2001.

[B00] Bohm, C. A Cost Model for Query Processing in High Dimensional Data Spaces. *TODS*, Vol. 25(2): 129-178, 2000.

[BCG02] Bruno, N., Chaudhuri, S., Gravano, L. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. *TODS* 27(2): 153-187, 2002.

[BGRS99] Beyer, K., Goldstein, J., Ramakrishnan, R., Shaft, U. When Is Nearest Neighbor Meaningful? *ICDT*, 1999.

[BJKS02] Benetis, R., Jensen, C., Karciauskas, G., Saltenis, S. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. *IDEAS*, 2002.

[BKSS90] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[CMTV00] Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M. Closest Pair Queries in Spatial Databases. *SIGMOD*, 2000.

[F02] Fagin, R. Combining Fuzzy Information: an Overview. *SIGMOD Record*, 31 (2): 109-118, 2002.

[FLN01] Fagin, R., Lotem, A., Naor, M. Optimal Aggregation Algorithms for Middleware. *PODS*, 2001.

[FSAA01] Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., Abbadi, A. Constrained Nearest Neighbor Queries. *SSTD*, 2001.

[G84] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD*, 1984.

[JMF99] Jain, A., Murthy, M., Flynn, P., Data Clustering: A Review. *ACM Comp. Surveys*, 31(3): 264-323, 1999.

[HS98] Hjaltason, G., Samet, H. Incremental Distance Join Algorithms for Spatial Databases. *SIGMOD*, 1998.

[HS99] Hjaltason, G., Samet, H. Distance Browsing in Spatial Databases. *TODS*, 24(2), 265-318, 1999.

[HYC01] Hochreiter, S., Younger, A.S., Conwell, P. Learning to Learn Using Gradient Descent. *ICANN*, 2001.

[KGT99] Kollios, G., Gunopulos, D., Tsotras, V. Nearest Neighbor Queries in Mobile Environment. *STDBM*, 1999.

[KM00] Korn, F., Muthukrishnan, S. Influence Sets Based on Reverse Nearest Neighbor Queries. *SIGMOD*, 2000.

[KMS02] Korn, F., Muthukrishnan, S. Srivastava, D. Reverse Nearest Neighbor Aggregates Over Data Streams. *VLDB*, 2002.

[NO97] Nakano, K., Olariu, S. An Optimal Algorithm for the Angle-Restricted All Nearest Neighbor Problem on the Reconfigurable Mesh, with Applications. *IEEE Trans. on Parallel and Distributed Systems* 8(9): 983-990, 1997.

[PM97] Papadopoulos, A., Manolopoulos, Y. Performance of Nearest Neighbor Queries in R-trees. *ICDT*, 1997.

[PZMT03] Papadias, D., Zhang, J., Mamoulis, N., Tao, Y. Query Processing in Spatial Network Databases. *VLDB*, 2003.

[RKV95] Roussopoulos, N., Kelly, S., Vincent, F. Nearest Neighbor Queries. *SIGMOD*, 1995.

[S91] Sproull, R. Refinements to Nearest Neighbor Searching in K-Dimensional Trees. *Algorithmica*, 6(4): 579-589, 1991.

[SKS02] Shahabi, C., Kolahdouzan, M., Sharifzadeh, M. A Road Network Embedding Technique for K-Nearest Neighbor Search in Moving Object Databases. *ACM GIS*, 2002.

[SR01] Song, Z., Roussopoulos, N. K-Nearest Neighbor Search for Moving Query Point. *SSTD*, 2001.

[SYUK00] Sakurai, Y., Yoshikawa, M., Uemura, S., Kojima, H. The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. *VLDB*, 2000.

[TP02] Tao, Y., Papadias, D. Time Parameterized Queries in Spatio-Temporal Databases. *SIGMOD*, 2002.

[TP03] Tao, Y., Papadias, D. Spatial Queries in Dynamic Environments. *ACM TODS*, 28(2): 101-139, 2003.

[TPS02] Tao, Y., Papadias, D., Shen, Q. Continuous Nearest Neighbor Search. *VLDB*, 2002.

[Web1] www.maproom.psu.edu/dcw/

[Web2] dke.cti.gr/People/ytheod/research/datasets/

[WSB98] Weber, R., Schek, H.J., Blott, S. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. *VLDB*, 1998.

[YOTJ01] Yu, C., Ooi, B, Tan, K., Jagadish, H. Indexing the Distance: An Efficient Method to KNN Processing. *VLDB*, 2001.