The K Group Nearest-Neighbor Query on Non-indexed RAM-Resident Data

George Roumelis[1], Michael Vassilakopoulos[2], Antonio Corral[3(✉)], and Yannis Manolopoulos[1]

[1] Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece
{groumeli,manolopo}@csd.auth.gr
[2] Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece
mvasilako@inf.uth.gr
[3] Department of Informatics, University of Almeria, Almería, Spain
acorral@ual.es

Abstract. Data sets that are used for answering a single query only once (or just a few times) before they are replaced by new data sets appear frequently in practical applications. The cost of buiding indexes to accelerate query processing would not be repaid for such data sets. We consider an extension of the popular (K) Nearest-Neighbor Query, called the (K) Group Nearest Neighbor Query (GNNQ). This query discovers the (K) nearest neighbor(s) to a group of query points (considering the sum of distances to all the members of the query group) and has been studied during recent years, considering data sets indexed by efficient spatial data structures. We study (K) GNNQs, considering non-indexed RAM-resident data sets and present an existing algorithm adapted to such data sets and two Plane-Sweep algorithms, that apply optimizations emerging from the geometric properties of the problem. By extensive experimentation, using real and synthetic data sets, we highlight the most efficient algorithm.

Keywords: Spatial query processing · Plane-sweep · Group nearest-neighbor query · Algorithms

1 Introduction

Spatial database is a database that offers spatial data types (for example, types for points, line segments, regions, etc.), a query language with spatial predicates, spatial indexing techniques and efficient processing of spatial queries [1]. It has

G. Roumelis, M. Vassilakopoulos, A. Corral and Y. Manolopoulos—Work funded by the GENCENG project (SYNERGASIA 2011 action, supported by the European Regional Development Fund and Greek National Funds); project number 11SYN_8_1213.

A. Corral—Supported by the MINECO research project [TIN2013-41576-R].

C. Grueau and J.G. Rocha (Eds.): GISTAM 2015, CCIS 582, pp. 69–89, 2016.
DOI: 10.1007/978-3-319-29589-3_5

grown in importance in several fields of application such as urban planning, resource management, transportation planning, etc. Together with them come various types of complex queries that need to be answered efficiently.

One of the most representative and studied queries in Spatial Databases is the (K) Nearest-Neighbor Query (NNQ), that discovers the (K) nearest neighbor(s) to a query point. An extension that is important for practical applications is the (K) Group Nearest Neighbor Query (GNNQ), that discovers the (K) nearest neighbor(s) to a group of query points (considering the sum of distances to all the members of the query group). This query has been studied during recent years, considering data sets indexed by efficient spatial data structures. An example of its utility could be when we have a set of meeting points (data set) and a set of user locations (query set), and we want to find the set of one (K) meeting point(s) that minimizes the sum of distances for all user locations, since each user will travel from his/her location to each of the K meeting points. More specifically, user locations may represent residence locations and meeting points may represent points of interest (cultural landmarks). Each of the K points is visited by each user for whole day inspection and the user returns to his/her residence overnight, before visiting the next landmark on the following day. We may interested to solve such a problem for a specific pair of data and query sets only once, but we may face several such problems for different pairs of sets. Building indexes for the data sets would be needed only if several queries would be answered for the these sets, which might evolve gradually in the course of time and not be completely replaced by new ones.

One of the most important techniques in the computational geometry field is the Plane-Sweep (PS) algorithm, which is a type of algorithm that uses a conceptual sweep line to solve various problems in the Euclidean plane, E^2, [2]. The name of PS is derived from the idea of sweeping the plane from left to right with a vertical line (front) stopping at every transaction point of a geometric configuration to update the front. All processing is done with respect to this moving front, without any backtracking, with a look-ahead on only one point each time [3]. For instance, the PS technique has been successfully applied in spatial query processing, mainly for intersection joins [4].

In [5], the problem of processing K Closest Pair Query between RAM-based point sets was studied, using PS algorithms. Two improvements that can be applied to a PS algorithm and a new algorithm that minimizes the number of distance computations, in comparison to the classic PS algorithm, were proposed. By extensive experimentation, using real and synthetic data sets, the most efficient improvement was highlighted and it was shown that the new PS algorithm outperforms the classic one.

In this paper, we study (K) GNNQs, considering non-indexed data sets (a frequent case in practical applications, see the example given previously), unlike previous research presented in Sect. 2 that consider that one or both data sets are indexed by structures of the R-tree family. Our target is to design efficient non-index based algorithms for (K) GNNQs and highlight the most efficient among

them. Thus, we present three (RAM-based) algorithms[1], an existing one adapted to non-indexed data sets and two novel PS ones, that apply optimizations emerging from the geometric properties of the problem. Several experiments have been performed, using real and synthetic data sets, to show the most efficient algorithm.

The paper is organized as follows. In Sect. 2, we review the related literature and motivate the research reported here. In Sect. 3, three new PS algorithms for GNNQs are presented. In Sect. 4, a comparative performance study is reported. Finally, in Sect. 5, conclusions on the contribution of this paper and future work are summarized.

2 Related Work and Motivations

GNN queries are introduced in [7] and it consist in given two sets of points P and Q, a GNN query retrieves the point(s) of P with the smallest sum of distances to all points in Q. GNN queries are also known as aggregate nearest neighbor (ANN) queries [8]. In [7], the authors have developed three different methods, MQM (multiple query method), SPM (single point method) and MBM (minimum bounding method), to evaluate a GNN query that minimizes the total distance from a set of query points to a data point. In [8] these methods have been extended to minimize the minimum and maximum distance in addition to the total distance with respect to a set of query points. All these methods assume that the data points are indexed using an R-tree and can be implemented using both depth-first search and best-first search algorithms.

In general terms, MQM performs an incremental search for the nearest data point of each query point in the set and compute the aggregate distance from all query points for each retrieved data point. The search ends when it is ensured that the aggregate distance of any non-retrieved data point in the database is greater than the current K-th minimum aggregate distance, that is the K GNNs are found. It means MQM is a threshold algorithm, since it computes the nearest neighbor for each query point incrementally, updating different thresholds according to the target of the (K) GNN. The main disadvantage of MQM is that it traverses the R-tree multiple times and it can access the same data point more than once.

The other methods, SPM and MBM, find the K GNNs in a single traversal of the R-tree. SPM approximates the centroid of the query distribution area and continues the searching with respect to the centroid until the current (K) GNNs are determined. During the search, some heuristics based on triangular inequality are used to prune intermediate nodes and determine the real nearest neighbors to Q. MBM regards Q as a whole and uses its MBR M to prune the search space in a single query, in either a depth-first or best-first manner. Moreover, two pruning heuristics involving the distance from an intermediate node to M or query points are proposed and they can be used in either traversal

[1] This paper is a post proceedings enhanced version of [6], where the last two algorithms of the current paper are presented and compared.

policy. Experimental results showed that the performance of MBM is better than SPM and MQM for memory and disk resident query points, since it traverses the R-tree once and takes the query distribution area into account. Moreover, according to the comparison conducted in [7], MBM is better than SPM in terms of node access and CPU cost while MQM is the worst.

In [9], the authors propose two pruning strategies for (K) GNN queries which take into account the distribution of query points. Such methods employ an ellipse to approximate the extent of multiple query points, and then derive a distance or minimum bounding rectangle using that ellipse to prune intermediate nodes in a depth-first search via an R*-tree. These methods are also applicable to the best-first traversal. The experimental results show that the proposed pruning strategies are more efficient than the methods presented in [7].

A new method to evaluate a (K) GNN query for non-indexed data points using projection-based pruning strategies was presented in [10]. Two points projecting-based ANNQ algorithms were proposed, which can efficiently prune the data points without indexing. This new method projects the query points into a special line, on which their distribution is analysed, for pruning the search space.

In [11], a new property in vector space was proposed and, based on it some efficient bound estimations were developed for two most popular types of ANN queries (sum and maximum). Taking into account these bounds, indexed and non-index ANN algorithms were designed. The proposed algorithms showed interesting results, especially for high dimensional queries.

Other related contributions in this research line have been proposed in the literature. In [12] an efficient algorithm for (K) GNN query considering privacy preserving was proposed, and the existing (K) GNN algorithms [8] for point locations were extended to regions in order to preserve user privacy. In [13], the (K) GNN query in road networks based on network voronoi diagram was solved. In [14], the reverse top-K group nearest neighbor search is presented. In [15], the KNN and (K) GNN queries are extended to get a new type of query, so-called K Nearest Group (KNG) query. It retrieves closest elements from multiple data sources, and finds K groups of elements that are closest to a given query point, with each group containing one object from each data source. And recently, for uncertain databases, probabilistic (K) GNN query was studied by [16,17].

Therefore, the (K) GNN is an active research line nowadays and most of the contributions have used indexes (of the R-tree family) for their solutions. The main motivation of this paper is to exmanine the use of the Plane-Sweep technique to solve the problem proposed in [7], when neither of the inputs are indexed. Due to not using indexes, the algorithms proposed in this paper are completely different to previous solutions. To the best of our knowledge, there are not any existing solutions for the (K) GNNQ without indexes. The unnecessity of indexes is not infrequent in practical applications, when the data sets change at a very rapid rate, or the data sets are not reusable for subsequent queries (see the example in Sect. 1).

3 RAM-Based Algorithms for GNNQ

In this section we introduce three RAM-based algorithms for processing GNNQ. The input of this query consists of a set $P = \{p_0, p_1, \cdots, p_{N-1}\}$ of static data points in the Euclidean plane, E^2, and a group of query points $Q = \{q_0, q_1, \cdots, q_{M-1}\}$. 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 $p \in P$ and Q is defined as $sumdist(p, Q) = \sum_{i=0}^{M-1} dist(p, q_i)$, where $dist(p, q_i)$ is the Euclidean distance between $p \in P$ and a query point $q_i \in Q$. In the following, $dx\_dist(p, q)$ represents the dx-distance ($\Delta x(p, q)$) between two points p and q over the X-axis and $dy\_dist(p, q)$ represents the dy-distance ($\Delta y(p, q)$) over the Y-axis. The sum of distances (dx-distances) between one given point $p \in P$ and all query points of Q ($q_i \in Q$) is defined as $sumdist(p, Q) = \sum_{i=0}^{M-1} dist(p, q_i)$ ($sumdx(p, Q) = \sum_{i=0}^{M-1} dx\_dist(p, q_i)$).

The **first algorithm** that we present is called Single Point Method over Non-Indexed Data ($SPMNI$) and is a non-indexed data extension/reformation of the SPM algorithm proposed in [7] (this is the most efficient algorithm of [7] that can be adapted to non-indexed data, since MBM is based on the MBR concept used in tree and other indexes). Instead of the sorted list used in the SPM algorithm, we used a max binary heap (keyed by *sumdist* and called $MaxKHeap$) to keep the K data points with the smallest sum of distances to the query points found so far (the *sumdist* of the root of the $MaxKHeap$ is denoted by δ). In order to sort the points of the P data set acccording to their distance to the centroid (c) of query points, the $SPMNI$ algorithm uses an array of $|P|$ length named Centroid Nearest Neighbor List ($cNNl$). Every element of $cNNl$ is a pair of type $< i, dist(p, c) >$ where i is the index of the point p of P set and $dist(p, c)$ is the distance between the point p and the centroid c.

The algorithm works as follows. First, the algorithm, calculates the coordinates of the centroid, computes the sum of distances of the centroid to the query points ($sumdistCQ$), and after creates and sorts the $cNNl$ (preparation stage). For these, $SPMNI$ calls the functions $Calculate\_Centroid\_Coord(Q)$ (line 1), $Create\_CentroidNN\_List(P, c)$ (line 5), and $Sort\_CentroidNN\_List$ (line 6).

The search process of the $KGNN$ starts and until $MaxKHeap$ is full the steps bellow are repeated. The index i of the point p of the P set which is the next NN to the centroid c and the distance to the centroid $dist(p, c)$ is retrieved from the $cNNl$ list using the value of the index of the $cNNl$ list, j. Next, the $sumdist(p, Q)$ is calculated and the $< p, sumdist(p, Q) >$ pair is inserted into the heap. The index j is incremented in order to point to the next NN to the centroid c, and the iteration is repeated, unless the $MaxKHeap$ has become full.

When the $MaxKHeap$ is full, the same steps are repeated with a few differences. In [7] it was proved that for every data point p with $|Q| \cdot dist(p, c) \geq \delta + sumdist(c, Q)$, p can be ignored, without calculating any distance to the query points. Since the left part of the previous inequality grows for every sub-

Algorithm 1. SPMNI.

Input: Two X-sorted arrays of points $p[0, 1, \cdots, N-1]$, $q[0, 1, \cdots, M-1]$, and $MaxKHeap$.

Output: $MaxKHeap$ storing the K NNs having smallest sums of distances to all query points.

1: $c(x, y) = Calculate\_Centroid\_Coord(Q)$ ▷ calculate the coordinates of the Centroid
2: $sumdistCQ = 0.0$
3: **for** $k = 0; k < M; k++$ **do** ▷ for each query point q
4: $sumdistCQ+ = dist(c, q[k])$
5: $cNNl$=Create_CentroidNN_List(P,c) ▷ create the list of NN to the centroid
6: Sort_CentroidNN_List($cNNl$) ▷ sort entries of the list $cNNl$ according to their $dist$
7: $j = 0$
8: **while** $MaxKHeap$ *is not* $full$ **do**
9: $p = P[cNNl[j].i]$ ▷ retrieve the point p as current NN of the Centroid c
10: **for** $k = 0; k < M; k++$ **do** $sumdist+ = dist(p, q[k])$ ▷ $\forall$ q, add dist to current point
11: $MaxKHeap.insert(p, sumdist)$
12: $j = j + 1$ ▷ increment index j to the next NN
13: **while** $j < N$ **do**
14: $p = P[cNNl[j].i]$ ▷ retrieve the point p as current NN of the Centroid c
15: $dpc = cNNl[j].dist$ ▷ retrieve the distance dist(p,c) from the list $cNNl$
16: **if** $M \cdot dpc \geq MaxKHeap.root.dist + sumdistCQ$ **then** ▷ termination condition
17: **break** ▷ exit j, all other NNs have larger sum of distances
18: **for** $k = 0; k < M; k++$ **do** $sumdist+ = dist(p, q[k])$ ▷ $\forall$ q, add dist to current point
19: **if** $sumdist < MaxKHeap.root.dist$ **then**
20: $MaxKHeap.insertFull(p, sumdist)$
21: $j = j + 1$ ▷ increment index j to the next NN

sequent NN to the centroid that is retrieved, all the NNs after the one that makes this condition true can be ignored. Thus, this condition can be not only one pruning condition, but termination condition of the process of $KGNNQ$. This termination condition is checked in the beginning of every iteration in this second part of the algorithm. Moreover, for a data point p retrieved from the $cNNl$ list and after the $sumdist(p, Q)$ has been calculated, this sum of distances will be compared with the δ value of the $MaxKHeap.root.dist$. There are 2 cases:

1. *Case 1*: If $sumdist(p, Q)$ is larger than or equal to δ, then p will be not inserted in the heap.
2. *Case 2*: If the $sumdist(p, Q)$ is smaller than δ, then p will be inserted in the heap, after $MaxKHeap.root$ has been deleted (**rule 1**).

The next two algorithms that we developped are novel Plane-Sweep algorithms that make use of the *median* point of the query set Q. A simple application of Plane-Sweep, assuming that both data sets are sorted in ascending order of their X-values, would compute the sum of distances of each data point to all the query points, by examining the data points from left to right, along the sweeping axis (e.g. X-axis). Let p with $sumdx(p, Q) \geq \delta$, then, for every p' with $p'.x \geq p.x$, $sumdx(p', Q) \geq sumdx(p, Q)$. Moreover, $sumdist(p', Q) \geq sumdx(p', Q)$. Thus, $sumdist(p', Q) \geq \delta$ and we do not need to calculate any distance for p'. Note that, while the sweep line approaches (moves away from) the median point(s), $sumdx$ will be decreasing (increasing). This is proved in the Appendix. In the next two algorithms, we find a data point $p_i \in P$ that is X-closest to the *median* point of the query set Q (in case that the query set contains an even number of points, we choose the right of the two median points). This data point is found by binary search. The sweep line is located on p_{i-1} and *moves to left* until a data point p with $sumdx(p, Q) \geq \delta$ is found (**termination condition 1**). Then, the sweep line is located on p_i and *moves to right* until a data point p with $sumdx(p, Q) \geq \delta$ (**termination condition 2**). At this stage, $MaxKHeap$ will contain the K data points of P with the smallest sum of distances to the query points.

As we mensioned above, in [7] it was proved that for every data point p with $|Q| \cdot dist(p, c) \geq \delta + sumdist(c, Q)$, p can be ignored, without calculating any distance. In the third algorithm that we have developed, the centroid c of the query points is also used and the above condition is a pruning condition for points that saves a significant number of calculations. Moreover, in the third algorithm, when the sweep line is outside of the area of query points, then for the current data point p, $sumdx(p, Q) = |Q| \cdot |p.x - c.x|$. Using this condition, we save numerous calculations.

In the Appendix, we prove that the sum of dx-distances between one given point $p(x, y) \in P$ and all points of the query set Q ($sumdx(p, Q)$):

A Is minimized at the median point $q[m]$ (where $q[m]$ is the array notation of q_m),
B For all $p.x \geq q[m].x$, $sumdx$ is constant or increasing with the increment of x, and
C For all $p.x < q[m].x$, $sumdx$ is increasing while x decreases.

The **second algorithm** (that is only based on *median*) is called $GNNPS$ and it uses the helper algorithm *calc_sum_dist* and the function $find\_closest\_point$. Firstly, it calculates the initial position of the sweeping line (preparation state). For this, the algorithm must find the first point $p[i] \in P$ which is on the right of the median of query set $q[m]$ ($p[i].x > q[m].x$), by calling the function $find\_closest\_point$ (line 1). After this, the algorithm sets the sweeping line at the point $p[i-1]$ (line 3) and continues scanning the points of P set decreasing the index i until the *termination condition 1* will be true or the points of P set will have finished (lines 3–5). Lastly, the algorithm sets the sweeping line at the point $p[i]$ and continues scanning the points of P set

Algorithm 2. GNNPS.

Input: Two X-sorted arrays of points $p[0, 1, \cdots, N-1]$, $q[0, 1, \cdots, M-1]$, and $MaxKHeap$.

Output: $MaxKHeap$ storing the K NNs having smallest sums of distances to all query points.

1: $i = find\_closest\_point(0, P, q[m])$ ▷ STEP 1 : Preperation. $q[m]$ is the median query set Q
2: $j = i - 1$
3: **while** $j > -1$ **do** ▷ STEP 2 : Search in the range $p[j].x \leq q[m].x$, descending j (move to left)
4: **if** $calc\_sum\_dist(p[j--], Q, MaxKHeap) == err\_code\_dx$ **then** ▷ Termination cond. 1
5: **break**
6: **while** $i < N$ **do** ▷ STEP 3 : Search in the range $p[i].x > q[m].x$, ascending i (move to right)
7: **if** $calc\_sum\_dist(p[i++], Q, MaxKHeap) == err\_code\_dx$ **then** ▷ Termination cond. 2
8: **break**

increasing the index i until the *termination condition 2* will be true or the points of the P set will have finished (lines 6–8).

The **third algorithm** (that is based on *median* and *centroid*) is called $GNNPSC$ and it uses the helper algorithms *calc_sum_dist_in* and *calc_sum_dist_out* and the function $find\_closest\_point$. Firstly, the algorithm calculates the initial position of the sweeping line and the coordinates of the centroid (preparation state). For these, the algorithm calls the functions $find\_closest\_point$ (line 1) and $Calculate\_Centroid\_Coord(Q)$ (line 3). In the next step, it continues scanning the points of P set decreasing the index j until the *termination condition 1* will be true or the X-coordinate of the current point of P set is smaller than or equal to the X-coordinate of the first query point $q[0]$ ($p[j].x \leq q[0]$). In this state, $GNNPSC$ calls the function *calc_sum_dist_in* to calculate the sum of distances. After exiting the previous loop and if the *termination condition 1* has not arisen (line 12), the algorithm continues decreasing j until the *termination condition 1* will be true or the points of P set will have finished (lines 13–15). Lastly, the algorithm sets the sweeping line at the point $p[i]$ and continues scanning the points of P set increasing the index i just like in the previous step (lines 17–20 inside query set Q and lines 21–24 outside query set Q). Note that the *calc_sum_dist_in* function is the same as *calc_sum_dist*, adding two new parameters (the centroid of Q (c) and its sum of distances to all query points ($sumdistCQ$)) and the following statements just after the line 9.

9 : $distpc = dist(p, c)$
10 : **if** $M \cdot distpc \geq MaxKHeap.root.dist + sumdistCQ$ **then**
11 : **return** $err\_code\_dist\_centroid$

And the remaining statements of *calc_sum_dist_in* from line 12 (12–22) are the same as *calc_sum_dist*.

Algorithm 3. *calc_sum_dist.*

Input: One point p, the sorted array of query points $q[0, 1, \cdots, M-1]$, and $MaxKHeap$.

Output: Value $successful\_insertion$ or err_code_dx or err_code_dist and $MaxKHeap$ updated with p if rule 2 was true.

```
1: function calc_sum_dist(p, Q, MaxKHeap)
2:     sumdist = 0.0, sumdx = 0.0
3:     if MaxKHeap is not full then
4:         for k = 0; k < M; k++ do                          ▷ for each query point q
5:             sumdist+ = dist(p, q[k])      ▷ dist(): the Euclidean distance between p and q[k]
6:         MaxKHeap.insert(p, sumdist)
7:         return sucessful_insertion
8:     else
9:         for k = 0; k < M; k++ do                          ▷ for each query point q
10:            sumdx+ = dx_dist(p, q[k])   ▷ dx_dist(): the dx-distance between p and q[k]
11:        if sumdx ≥ MaxKHeap.root.dist then                              ▷ Rule 1
12:            return err_code_dx          ▷ exit k, all other points have longer distance
13:        for k = 0; k < M; k++ do                          ▷ for each query point q
14:            sumdist+ = dist(p, q[k])    ▷ add the distance (dist) from the current point
15:        if sumdist < MaxKHeap.root.dist then                            ▷ Rule 2
16:            MaxKHeap.insertFull(p, sumdist)
17:            return sucessful_insertion
18:        else
19:            return err_code_dist        ▷ not inserted because of sum of distances (sumdist)
```

The following examples illustrate the execution of the algorithms. The point data P set is defined as $P = \{p_0(1,7);\ p_1(2,4);\ p_2(3,1);\ p_3(3,13);\ p_4(8,2);\ p_5(8,18);\ p_6(9,10);\ p_7(10,19);\ p_8(12,12);\ p_9(13,4);\ p_{10}(14,12);\ p_{11}(16,6);\ p_{12}(19,8);\ p_{13}(19,17);\ p_{14}(20,3);\ p_{15}(22,7)\}$, and the point query set Q is defined as $Q = \{q_0(9,7);\ q_1(10,11);\ q_2(12,4);\ q_3(17,7);\ q_4(19,11)\}$. In Fig. 1, P and Q (they are sorted in ascending order of their X-values), the centroid and the median of the query points and the initial position of the sweep line are drawn.

$SPMNI$ starts the preparation stage by calculating the coordinates of centroid point $c(x, y) = (13.4, 8)$ and then calculates the sum of distances between the centroid and the query points $sumdist(c, Q) = 23.374$. Next, $SPMNI$ creates the $cNNl$ list of pairs of type $< i, dist(p, c) >$ for all points of P set. This list is sorted in ascending order for each point in the P set with respect to $dist(p, c)$. Thus, the final form of the sorted list is: $\{< 11, 3.280 >, < 9, 4.020 >, < 10, 4.045 >, < 8, 4.238 >, < 6, 4.833 >, < 12, 5.600 >, < 4, 8.072 >, < 14, 8.280 >, < 15, 8.658 >, < 13, 10.600 >, < 5, 11.365 >, < 7, 11.513 >, < 3, 11.539 >, < 1, 12.081 >, < 0, 12.440 >, < 2, 12.536 >\}$. In other words, we have one complete list of Nearest Neighbors to the Centroid beginning from the closest one.

Algorithm 4. GNNPSC .

Input: Two X-sorted arrays of points $p[0, 1, \cdots, N-1]$, $q[0, 1, \cdots, M-1]$, and $MaxKHeap$.

Output: $MaxKHeap$ storing the K NNs having smallest sums of distances to all query points.

1: $i = find\_closest\_point(0, P, q[m])$ ▷ STEP 1 : Preperation. $q[m]$ is the median of query set Q
2: $j = i - 1$
3: $c(x, y) = Calculate\_Centroid\_Coord(Q)$ ▷ calculate the coordinates of the Centroid
4: $sumdistCQ = 0.0$
5: **for** $k = 0; k < M; k++$ **do** ▷ for each query point q
6: $sumdistCQ+ = dist(c, q[k])$
▷ STEP 2 : Search in the range $p[j].x \leq q[m].x$, descending j (move to left)
7: $cont\_search = true$ ▷ initialize the flag
8: **while** $j > -1$ **and** $p[j].x > q[0].x$ **do** ▷ $\forall$ $p[j]$ inside the query MBR in sweeping axis
9: **if** $calc\_sum\_dist\_in(p[j--], Q, c, sumdistCQ, MaxKHeap) == err\_code\_dx$ **then**
10: ▷ Termination condition 1
11: $cont\_search = false$
12: **break**
13: **if** $cont\_search = true$ **then**
14: **while** $j > -1$ **do** ▷ for each point $p[j]$ on the left of the query MBR in sweeping axis
15: **if** $calc\_sum\_dist\_out(p[j--], Q, c, sumdistCQ, MaxKHeap) == err\_code\_dx$ **then**
16: ▷ Termination condition 1
17: **break**
▷ STEP 3 : Search in the range $p[i].x > q[m].x$, ascending i (move to right)
18: $cont\_search = true$
19: **while** $i < N$ **and** $p[i].x < q[M-1].x$ **do** ▷ $\forall$ $p[i]$ inside the query MBR in sweeping axis
20: **if** $calc\_sum\_dist\_in(p[i++], Q, c, sumdistCQ, MaxKHeap) == err\_code\_dx$ **then**
21: ▷ Termination condition 2
22: $cont\_search = false$
23: **break**
24: **if** $cont\_search = true$ **then**
25: **while** $i < N$ **do** ▷ for each point $p[i]$ on the left of the query MBR in sweeping axis
26: **if** $calc\_sum\_dist\_out(p[i++], Q, c, sumdistCQ, MaxKHeap) == err\_code\_dx$ **then**
27: ▷ Termination condition 2
28: **break**

Index j is initially set to 0, that is, to the first element of the $cNNl$ list and $SPMNI$ continues with main stage. The search process of the $KGNNQ$ starts

Algorithm 5. *calc_sum_dist_in.*

Input: One point p, set of query points Q, centroid c, its sum of distances to all query points $sumdistCQ$ and $MaxKHeap$.

Output: Value $successful\_insertion$ or err_code_dx or err_code_dist and $MaxKHeap$ updated with p if rule 2 was true.

1: **function** *calc_sum_dist_in*(p, Q, c, $sumdistCQ$, $MaxKHeap$)
2: $sumdist = 0.0$, $sumdx = 0.0$
3: **if** $MaxKHeap$ *is not full* **then**
4: **for** $k = 0; k < M; k++$ **do** ▷ for each query point q
5: $sumdist += dist(p, q[k])$ ▷ $dist()$: the dx-distance between p and $q[k]$
6: $MaxKHeap.insert(p, sumdist)$
7: **return** $sucessful\_insertion$
8: **else**
9: $dpc = dist(p, c)$ ▷ $dist()$: the distance between p and c
10: **if** $M \cdot dpc \geq MaxKHeap.root.dist + sumdistCQ$ **then** ▷ prune p w/o computing dists
11: **return** $err\_code\_dist\_centroid$; ▷ not inserted because of sum of distances
12: **for** $k = 0; k < M; k++$ **do** ▷ for each query point q
13: $sumdx += dx\_dist(p, q[k])$ ▷ $dx\_dist()$: the dx-distance between p and $q[k]$
14: **if** $sumdx \geq MaxKHeap.root.dist$ **then** ▷ Rule 1
15: **return** err_code_dx ▷ exit k, all other points have longer distance
16: **for** $k = 0; k < M; k++$ **do** ▷ for each query point q
17: $sumdist += dist(p, q[k])$ ▷ add the distance ($dist$) from the current point
18: **if** $sumdist < MaxKHeap.root.dist$ **then** ▷ Rule 2
19: $MaxKHeap.insertFull(p, sumdist)$
20: **return** $sucessful\_insertion$
21: **else**
22: **return** err_code_dist ▷ not inserted because of sum of distances ($sumdist$)

with empty $MaxKHeap$. The first NN is p_{11}. $sumdist(p_{11}, Q) = 26.599$ is calculated and the pair $< p_{11}, 26.599 >$ is inserted in the $MaxKHeap$ as the first one (lines 8–11). Index j is incremented and the second and third NN are retreived sequentially from the $cNNl$ list, while $MaxKHeap$ is not full. The pairs $< p_9, 27.835 >$ and $< p_{10}, 30.370 >$ are inserted in the $MaxKHeap$. At the end of the rhird iteration the $MaxKHeap$ is full and $MaxKHeap.root.dist$ has value 30.370. The second part of the main stage is started executing lines 13–21. The forth NN is p_8 with $dist(p_8, c) = 4.238$. The terminal condition $|Q| \cdot dist(p, c) \geq \delta + sumdist(c, Q)$ is tested (line 16). The condition $5 \cdot 4.238 \geq 30.370 + 23.374$ is false. Therefore, $SPMNI$ continues calculating $sumdist(p_8, Q)$ (line 18). The variable $sumdist$ is set to the value 30.209. The condition $sumdist(p_8, Q) < MaxKHeap.root.dist$ is true and the previous $MaxKHea.root$ is deleted, because the pair $< p_8, 30.209 >$ must be inserted in

Algorithm 6. *calc_sum_dist_out.*

Input: One point p, set of query points Q, centroid c, its sum of distances to all query points $sumdistCQ$ and $MaxKHeap$.

Output: Value $successful\_insertion$ or err_code_dx or err_code_dist and $MaxKHeap$ updated with p if rule 2 was true.

```
1: function calc_sum_dist_out(p, Q, c, sumdistCQ, MaxKHeap)
2:     sumdist = 0.0, sumdx = 0.0
3:     if MaxKHeap is not full then
4:         for k = 0; k < M; k++ do                          ▷ for each query point q
5:             sumdist+ = dist(p, q[k])    ▷ dist(): the dx-distance between p and q[k]
6:         MaxKHeap.insert(p, sumdist)
7:         return sucessful_insertion
8:     else
9:         dx = dx_dist(p, c)   ▷ dx_dist(): the dx-distance between p and c (Δx(p, c))
10:        if M · dx ≥ MaxKHeap.root.dist then                          ▷ Rule 1
11:            return err_code_dx;       ▷ exit k, all other points have longer distance
12:        dy = dy_dist(p, c)              ▷ dy_dist(): the dy-distance between p and c
13:        distpc = √(dx² + dy²)
14:        if M · distpc ≥ MaxKHeap.root.dist + sumdistCQ then
15:            return err_code_dist_centroid;
16:        for k = 0; k < M; k++ do                          ▷ for each query point q
17:            sumdist+ = dist(p, q[k])
18:        if sumdist < MaxKHeap.root.dist then                         ▷ Rule 2
19:            MaxKHeap.insertFull(p, sumdist)
20:            return sucessful_insertion
21:        else
22:            return err_code_dist          ▷ not inserted because of sum of distances
   (sumdist)
```

the full $MaxKHeap$ as new root (line 20). So, $MaxKHeap.root.dist = 30.209$ and the index j is incremented (line 21). The fifth NN is p_6 with $dist(p_6, c) = 4.833$. The terminal condition is tested as above (line 16). The condition $5 \cdot 4.833 \geq 30.209 + 23.374$ is false. Therefore, $SPMNI$ continues calculating the $sumdist(p_6, Q)$ (line 18). The variable $sumdist$ is set to the value 29.716. The condition $sumdist(p_6, Q) < MaxKHeap.root.dist$ is true and the previous $MaxKHea.root$ is deleted, because the pair $< p_6, 29.716 >$ must be inserted in the full $MaxKHeap$. The pair $< p_9, 27.835 >$ becomes new root (line 20). So, $MaxKHeap.root.dist = 27.835$. From the sixth to the tenth NNs (p_{12} with $dist(p_{12}, c) = 5.6$, p_4 with $dist(p_4, c) = 8.072$. p_{14} with $dist(p_{14}, c) = 8.280$, p_{15} with $dist(p_{15}, c) = 8.658$ and p_{13} with $dist(p_{13}, c) = 10.6$), the terminal condition is false (line 16). Therefore, $SPMNI$ continues calculating the variable $sumdist$ for each point (line 18). The condition $sumdist(p, Q) < MaxKHeap.root.dist$ is false and the pairs $\{< p_{12}, 32.835 >, < p_4, 43.299 >, < p_{14}, 45.635 >, < p_{15}, 46.089 >, < p_{13}, 55.922 >\}$ must not be inserted in the full $MaxKHeap$. The eleventh and final NN is p_5 with $dist(p_5, c) = 11.365$. The terminal condition is tested as above (line 16) and $5 \cdot 11.365 \geq 29.716 + 23.374$ is true. Therefore,

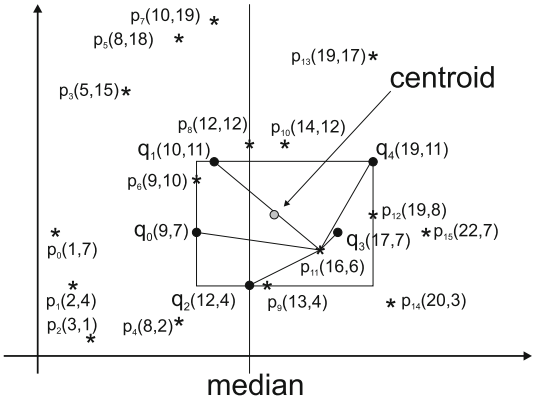

Fig. 1. The points of P and Q, the centroid, the median of the query points and the initial position of the sweep line.

$SPMNI$ is terminated (line 17) by breaking the while ... do loop. While executing this algorithm, we made 71 complete point-point distance calculations, 142 point-point dx-distance calculations, 5 points with their sum of distances were inserted in the $MaxKHeap$ and 10 of 16 points of P set were fully examined and their sumdist distances to the query points were calculated. One point of 16 (the last one) has been partially examined. $dist(p, c)$ has been calculated for all (16) points and all the P set members have been sorted.

In $GNNPS$, firstly (in Step 1) the algorithm searches for the point of the P set which is on the right of the median q_2(12,4) query point (line 1). That is p_9(13,4) point. In Step 2 (lines 3–5) it starts calculating the sum of distances between point p_8(12,12) and all query points. The result is $sumdist(p_8, Q) =$ 30.209 and the point p_8 is inserted in the $MaxKHeap$ (*calc_sum_dist*:lines 2–7). In the next iteration the point p_7(10,19) is examined. The $MaxKHeap$ is full and the second part of the *calc_sum_dist* function (lines 9–19) is executed. The sum of distances is $sumdist(p_7, Q) = 61.108$ larger than the $MaxKHeap.root.dist =$ 30.209 (condition in the *calc_sum_dist*:line 15 is false), so the point is rejected (*calc_sum_dist*:line 19). In the third iteration the point p_6(9,10) is examined and the sum of distances is $sumdist(p_6, Q) = 29.716$ which is smaller (condition of *calc_sum_dist*:line 15 is true) than the $MaxKHeap.root.dist$ therefore the point p_6 is inserted in the $MaxKHeap$ (*calc_sum_dist*:lines 16,17) by replacing the previous root (p_8). In the fourth and fifth iterations for the points p_5 and p_4 the sum of distances are $sumdist(p_5, Q) = 60.317$ and $sumdist(p_4, Q) = 43.299$, respectively; both larger than the $MaxKHeap.root.dist$ and the points are rejected. In the sixth iteration, the point p_3 has $sumdx(p_3.x, Q) = 52$ (condition in *calc_sum_dist*:line 11) which is larger than the $MaxKHeap.root.dist$ and the process (scanning the P set on the left) ends (*calc_sum_dist*:line 12) because it is impossible to find other points of P set on the left of p_3 having sum of distances smaller than 52. The algorithm continues scanning the

points of P set to the right of the median q_2, starting from the p_9 point. Its $sumdist(p_9, Q) = 27.835$ is smaller than the $MaxKHeap.root.dist = 29.716$ so it replaces the existing point in the root of $MaxKHeap$. The next point p_{10} has $sumdist(p_{10}, Q) = 30.370$ and it is rejected. The next iteration will try the point p_{11} which has $sumdist(p_{11}, Q) = 26.599$ the smallest sum of distances and this point (p_{11}) is inserted in the $MaxKHeap$ replacing the previous root p_9. In the last iteration the algorithm examines the point p_{12} which has $sumdx(p_{12}, Q) = 28$ larger than the $MaxKHeap.root.dist = 26.599$ and the process is finally finished. While executing this algorithm we made 46 complete point-point distance calculations, 84 point-point dx-distance calculations, 4 points with their sum of distances were inserted in the $MaxKHeap$ and 10 of the 16 points of P set were examined.

$GNNPSC$ starts (Step 1) by finding the first point of P set which is on the right of the median point of query set Q. That is the point p_9. Afterwards it calculates the coordinates of centroid point $c(x, y) = (13, 8)$ and then calculates the sum of distances between the centroid and the query points $sumdist(c, Q) = 23.374$. $GNNPSC$ continues with Step 2. In that step, the points of P set are scanned on the left of the p_9 in two particular steps. First from p_8 up to p_7 which have X-coordinate larger than $q_0.x = 9$ by calling the *calc_sum_dist_in* function. There is $sumdist(p_8, Q) = 30.209$ and this point is inserted in the $MaxKHeap$ as the first point while the $MaxKHeap$ is empty (*calc_sum_dist_in*:lines 3–7). The point p_7 is examined next and it is rejected without a need to calculate $sumdist(p_7, Q)$ because the condition of the function *calc_sum_dist_in*:line 10 is true. Step 2 continues scanning the points of P set which are on the left (outside) of the q_0 query point by calling the function *calc_sum_dist_out*. The point p_6 with $sumdist(p_6, Q) = 29.716$ is inserted (*calc_sum_dist_in*:lines 9–20), while points p_5 and p_4 are rejected with $sumdist(p_5, Q) = 60.137$ and $sumdist(p_4, Q) = 43.299$ respectively, both larger than the $MaxKHeap.root.dist = 29.716$ with the point p_6. The next point p_3 is the last point to be examined because it has $sumdx(p_3, Q) = 52$ larger than the current $MaxKHeap.root.dist$. The algorithm continues by executing Step 3, scanning the points of P set on the right of the median query point q_2. The algorithm continues scanning the points of P set to the right starting from the p_9 point. Its $sumdist(p_9, Q) = 27.835$ is smaller than the $MaxKHeap.root.dist = 29.716$ so it replaces the existing point in the root of $MaxKHeap$. The next point p_{10} has $sumdist(p_{10}, Q) = 30.370$ and it is rejected. The next iteration will try the point p_{11} which has $sumdist(p_{11}, Q) = 26.599$ the smallest sum of distances and this point is inserted in the $MaxKHeap$ replacing the previous root p_9. In the last iteration we examine the point p_{12} which has $sumdx(p_{12}, Q) = 28$ larger than the $MaxKHeap.root.dist = 26.599$ and the process is finally finished. While executing this algorithm we made 42 complete point-point distance calculations, 38 point-point dx-distance calculations, 4 points with their sum of distances were inserted in the $MaxKHeap$ and 10 of 16 points of P set were examined.

4 Experimentation

In order to evaluate the behaviour of the proposed algorithms, we have used 6 real spatial data sets of North America, representing cultural landmarks (CL with 9203 points) and populated places (PP with 24493 points), roads (RD with 569120 line-segments) and railroads (RR with 191637 line-segments). To create sets of points, we have transformed the MBRs of line-segments from RD and RR into points by taking the center of each MBR (i.e., $|RD| = 569120$ points, $|RR| = 191637$ points). Moreover, in order to get the double amount of points from RR and RD, we chose the two points with min and max coordinates of the MBR of each line-segment (i.e. $|RDD| = 1138240$ points and $|RRD| = 383274$ points). The data of these 6 files were normalized in the range $[0, 1]^2$. The real data sets we used are geographical. In order to test the performance of our algorithms with data appearing in Science, we have created synthetic clustered data sets of 125000 ($125K$), 250000 ($250K$), 500000 ($500K$) and 1000000 ($1000K$) points, with 125 clusters in each data set (uniformly distributed in the range $[0, 1]^2$), where for a set having N points, $N/125$ points were gathered around the center of each cluster, according to Gaussian distribution (this distibution is common for natural properties of systems within Science). The first real data set (CL) was used to make the query set (Q) by selecting the appropriate number of points randomly. Then the coordinates of these points were appropriately scaled in order to get the MBR of the query points to get a pre-defined size in comparison to the MBR of the data set (P). The other 9 data sets were used as data sets (P) within which we were looking for the NNs.

All experiments were performed on a Laptop PC with Intel Core i5-3210M (2.5 GHz) CPU with 4 GB of RAM and several GBs of secondary storage, with Ubuntu Linux v. 14.04 64 bit, using the GNU C/C++ compiler (gcc). The performance measurements were: (1) the response time (total query execution time) of processing the (K) GNNQ, not counting reading from disk files to main memory and sorting of the data sets, (2) the number of points involved in calculations, (3) the number of X-axis distance computations (dx-distance) and (4) the number of distance computations.

In every experiment the query set was moved on X-axis in 8 equal size steps from the top left corner of the area of the data set (P) up to the right corner and after this, one step down on the Y-axis and so on. The total execution time, and the other experimentation metrics, for each one experiment, were computed as an average of all (the 64) queries.

In Fig. 2 (left), we depict the effect of the number of query points, M (the cardinality of Q), on execution time of all algorithms for the RD data set (the number of group nearest-neighbors, K, was equal to 8 and the size of query-set MBR was 8 % of the data set space). In analogous diagrams created for dx-distance and $dist$ calculations, in most cases, $GNNPS$ had the worse performance, $SPMNI$ was next and $GNNPSC$ had the best performance. It is obvious that the increase of M leads to an increase of the execution time, but with a smaller rate of increase. $GNNPSC$ needs less time than $GNNPS$, because of the use of centroid (the computation of the distance between the centroid and the

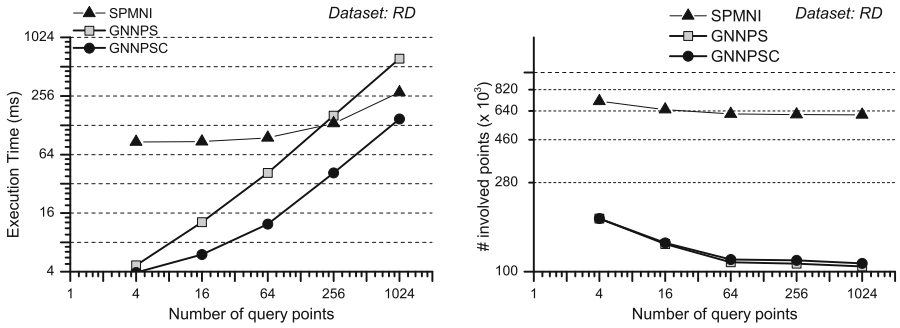

Fig. 2. (Left) Execution time of the algorithms as a function of M (RD data set). # (Right) Points involved in *sumdist* calculations of the algorithms as a function of M (RD data set).

reference point of P set needs one calculation of distance while the computation of the sum of distances between the reference point and all query points needs M distance calculations). Moreover, $GNNPSC$ needs less time than $SPMNI$, although both algorithms make use of centroid, because $SPMNI$ intially calculates the distance of the whole P set from the centroid and sorts the whole P set, while $GNNPSC$ does not access a big part of the P set, due to the termination condition.

For the same parameter settings and data set, in Fig. 2 (right), we depict the effect of M on the number of data set points involved in calculations. We observe that this number of points is reduced as M increases. In $SPMNI$, all points of P set are involved in calculations, since this algorithms intially calculates the distance of the whole P set from the centroid and sorts the whole P set. Regarding the other two algorithms, note that the sums of distances of the points of data P set near the median are enlarged to a smaller extent, compared to the *sumdist* of the points outside the query-set MBR. This enables the termination conditions and makes it possible to get nearest to the median query point. Moreover, we can observe in Fig. 2 that $GNNPSC$ needs more involved points and it is the fastest. This behaviour could be due to that in function *calc_sum_dist_in* we firstly apply the pruning condition of centroid and next the termination condition 1 or 2 is checked. So it is possible that some points may be pruned in $GNNPSC$ rather than being the cause of termination of the scanning.

In Fig. 3 (left), we depict the effect of the size of the query-set MBR, on dx-distance calculations of all algorithms for the $1000K$ data set (the number of group nearest neighbors, K, was equal to 8 and the number of query points was equal to 128).

Analogous diagrams created for distance calculations had similar appearance. In most of these diagrams, $SPMNI$ had the worse execution time, while $GNNPSC$ was always the best. It is obvious that the increase of the size of the query-set MBR leads to an increase of the execution time, but with a smaller

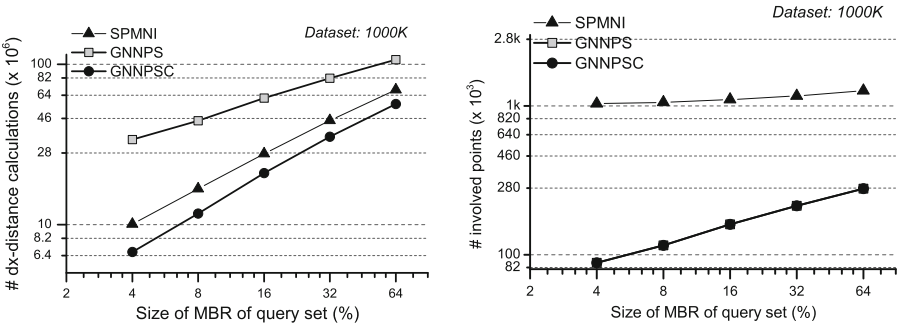

Fig. 3. (Left) # dx-distance calculations of the algorithms as a function of the size of MBR $\mathcal{M}$ ($1000K$ data set). (Right) # Points involved in calculations of the algorithms as a function of the size of MBR $\mathcal{M}$ ($1000K$ data set).

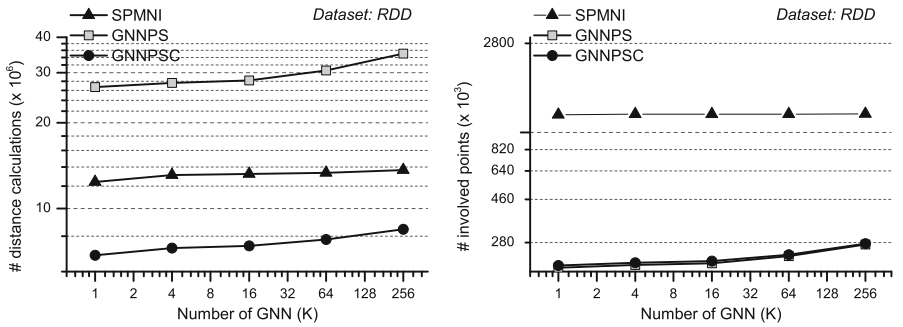

Fig. 4. (Left) # distance calculations of the algorithms as a function of K (RDD data set). (Right) # Points involved in calculations of the algorithms as a function of K (RDD data set).

rate of increase. The size of MBR was increased with a ratio of 4. The execution time, dx-distance and complete distance ($dist$) calculations was increased with ratio in the range 1.2 up to 2 for all data sets of real and synthetic data. For the same parameter settings and data set, in Fig. 3 (right), we depict the effect of the size of the query-set MBR on the number of points involved in calculations. We observe that this numbers of points are increased as the query-set MBR increases with a ratio smaller than 1.4 for $GNNPS$ and $GNNPSC$. We observe in this figure that the number of points involved almost identical and the lines are for $GNNPS$ and $GNNPSC$ overlapped. In $SPMNI$, the number of points involved in calculations is much higher, as explained in the interpretation of Fig. 2 (right).

In Fig. 4 (left), we depict the effect of the number of group nearest-neighbors, K, on distance calculations of all algorithms for the RDD data set (the number of query points, M, was equal to 128 and the size of query-set MBR was 8 % of the data set space). Analogous diagrams created for distance calculations had

similar appearance. Regarding execution time, the comments of Fig. 3 (left) hold for this figure, too. For the same parameter settings and data set, in Fig. 4 (right), we depict the effect of K on the number of points involved in calculations. We observe that this number of points is increased so slowly that it is going to be seen for values of K larger than 64.

From the above experiments, we conclude that:

- The number of data-set points involved in the calculations of $GNNPS$ and $GNNPSC$ algorithms is almost equal. However, the execution time for $GNNPSC$ remains always lower than the execution time of $GNNPS$, due to the pruning condition and the lower dx-distance calculations cost. This number is always signifficantly larger for $SPMNI$, since this algorithms intially calculates the distance of the whole P set from the centroid and sorts the whole P set.
- The main advantages of the Plane-Sweep method are the absence of recalculation, as each point is used in calculations once at most, and the absence of backtracking.
- The number of points involved in calculations is decreased when the number of query points is increased, provided that K and the query-set MBR size remain constant.

5 Conclusions and Future Work

Processing of GNNQs has been based on index structures, so far. In this paper, for the first time, we present new algorithms that can be efficiently applied on RAM-based data for processing the GNNQ. Extending [6], we present a comparison of new PS algorithms that we developed with respect to the best algorithm presented in [7] that can be transformed to work on non-indexed data sets, and we observe the PS algorithms achieve signifficantly better performance. As the experimentation that we performed shows, using synthetic and real data sets, the use of median and centroid in $GNNPSC$, prunes the number of points involved in processing and the number of calculations, in relation to $SPMNI$ and $GNNPS$.

In the future, we plan to compare the best of our algorithms to existing index based solutions. Moreover, the algorithms we present could be transformed/extended to work on high volume, disk resident data that are transferred in RAM in blocks. Additionally, the application of Plane-Sweep to other spatial queries (like Reverse NNQ) could lead to interesting techniques.

Appendix

Lemma: The sum of dx-distances between one given point $p(x, y) \in P$ and all points of the query set Q ($sumdx(p, Q)$):

A Is minimized at the median point $q[m]$ (where $q[m]$ is the array notation of q_m),

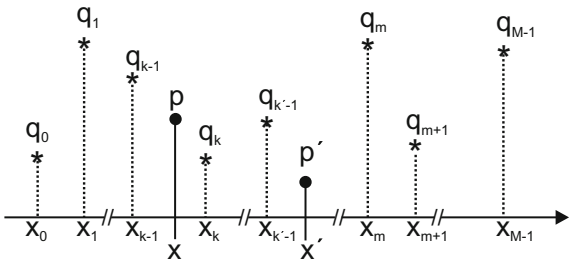

Fig. 5. The point p has K query points on the left and the point p' ($p'.x > p.x$) has K' query points on the left.

B For all $p.x \geq q[m].x$, *sumdx* is constant or increasing with the increment of x, and
C For all $p.x < q[m].x$, *sumdx* is increasing while x decreases.

Proof: Property **A** has been proved in [18]. To prove property **B**, for every point $p \in P$ and $q \in Q$, we use $\Delta x(p,q) = \begin{cases} p.x - q.x \text{if } p.x \geq q.x \\ q.x - p.x \text{if } p.x < q.x \end{cases}$
If the point p has K query points on the left ($p.x < q[K-1].x$) and $M-K$ query points on the right (Fig. 5), then: $sumdx(p,Q) = \sum_{i=0}^{K-1}(p.x - q[i].x) + \sum_{i=K}^{M-1}(q[i].x - p.x) = Kp.x - \sum_{i=0}^{K-1} q[i].x + \sum_{i=K}^{M-1} q[i] - (M-K)p.x = (2K-M)p.x - \sum_{i=0}^{K-1} q[i].x + \sum_{i=K}^{M-1} q[i].x$
For another point $p' \in P$ with $p'.x > p.x$ which has K' query points on the left (Fig. 5) and $M - K'$ query points on the right, it is: $sumdx(p',Q) = (2K' - M)p'.x - \sum_{i=0}^{K'-1} q[i].x + \sum_{i=K'}^{M-1} q[i].x$
The difference between dx-distances of the points p' and p is: $\Delta sumdx = sumdx(p',Q) - sumdx(p,Q) = (2K - M)(p'.x - p.x) + 2\left[(K'-K)p'.x - \sum_{i=K}^{K'-1} q[i].x\right]$. If the set of the query points Q has cardinality M and this is an even number then there are two medians $q[m1]$ and $q[m2]$, while if M is odd then there is only one median point $q[m]$.
B.1 M is even and $q[m1].x \leq p.x < p'.x$ then $M \leq 2K \leq 2K'$ so $(2K-M) \geq 0$, $(p'.x - p.x) \geq 0$ and $(K'-K)p'.x - \sum_{i=K}^{K'-1} q[i].x \geq 0$ because $p'.x \geq q[i].x$, whereas $K \leq i \leq K'$

B.2 All of the above apply to M if it is odd and it is only one median point $q[m].x \leq p.x < p'.x$. It is proven that for all points p on the right of the median query point the sum of dx-distances is increasing.
C For both types of cardinality of the query set Q and for the case $p.x < p'.x < q[m].x$ it is: $\Delta sumdx = (2K - M)(p'.x - p.x) + 2(K' - K)p'.x - 2\sum_{i=K}^{K'-1} q[i].x \leq (2K - M)(p'.x - p.x) + 2(K' - K)p'.x - 2(K' - K)p.x = 2(K - M)(p'.x - p.x) + 2(K' - K)(p'.x - p.x) = (2K - M + 2K' - 2K)(p'.x - p.x) = (2K' - M)(p'.x - p.x) < 0$. It is proven that for all points p on the left of the median query point the sum of dx-distances is strictly decreasing. □

References

1. Rigaux, P., Scholl, M., Voisard, A.: Spatial Databases - with Applications to GIS. Elsevier, San Francisco (2002)
2. Preparata, F.P., Shamos, M.I.: Computational Geometry - An Introduction. Springer, New York (1985)
3. Hinrichs, K., Nievergelt, J., Schorn, P.: Plane-sweep solves the closest pair problem elegantly. Inf. Process. Lett. **26**, 255–261 (1988)
4. Jacox, E.H., Samet, H.: Spatial join techniques. ACM Trans. Database Syst. **32**, 7 (2007)
5. Roumelis, G., Vassilakopoulos, M., Corral, A., Manolopoulos, Y.: A new plane-sweep algorithm for the K-closest-pairs query. In: Geffert, V., Preneel, B., Rovan, B., Štuller, J., Tjoa, A.M. (eds.) SOFSEM 2014. LNCS, vol. 8327, pp. 478–490. Springer, Heidelberg (2014)
6. Roumelis, G., Vassilakopoulos, M., Corral, A., Manolopoulos, Y.: Plane-sweep algorithms for the k group nearest-neighbor query. In: GISTAM Conference, pp. 83–93. Scitepress (2015)
7. Papadias, D., Shen, Q., Tao, Y., Mouratidis, K.: Group nearest neighbor queries. In: ICDE Conference, pp. 301–312. IEEE (2004)
8. Papadias, D., Tao, Y., Mouratidis, K., Hui, C.K.: Aggregate nearest neighbor queries in spatial databases. ACM Trans. Database Syst. **30**, 529–576 (2005)
9. Li, H., Lu, H., Huang, B., Huang, Z.: Two ellipse-based pruning methods for group nearest neighbor queries. In: ACM-GIS Conference, pp. 192–199. ACM (2005)
10. Luo, Y., Chen, H., Furuse, K., Ohbo, N.: Efficient methods in finding aggregate nearest neighbor by projection-based filtering. In: Gervasi, O., Gavrilova, M.L. (eds.) ICCSA 2007, Part III. LNCS, vol. 4707, pp. 821–833. Springer, Heidelberg (2007)
11. Namnandorj, S., Chen, H., Furuse, K., Ohbo, N.: Efficient bounds in finding aggregate nearest neighbors. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) DEXA 2008. LNCS, vol. 5181, pp. 693–700. Springer, Heidelberg (2008)
12. Hashem, T., Kulik, L., Zhang, R.: Privacy preserving group nearest neighbor queries. In: EDBT Conference, pp. 489–500. ACM (2010)
13. Zhu, L., Jing, Y., Sun, W., Mao, D., Liu, P.: Voronoi-based aggregate nearest neighbor query processing in road networks. In: ACM-GIS Conference, pp. 518–521. ACM (2010)

14. Jiang, T., Gao, Y., Zhang, B., Liu, Q., Chen, L.: Reverse top-k group nearest neighbor search. In: Wang, J., Xiong, H., Ishikawa, Y., Xu, J., Zhou, J. (eds.) WAIM 2013. LNCS, vol. 7923, pp. 429–439. Springer, Heidelberg (2013)
15. Zhang, D., Chan, C., Tan, K.: Nearest group queries. In: SSDBM Conference, p. 7. ACM (2013)
16. Lian, X., Chen, L.: Probabilistic group nearest neighbor queries in uncertain databases. IEEE Trans. Knowl. Data Eng. **20**, 809–824 (2008)
17. Li, J., Wang, B., Wang, G., Bi, X.: Efficient processing of probabilistic group nearest neighbor query on uncertain data. In: Bhowmick, S.S., Dyreson, C.E., Jensen, C.S., Lee, M.L., Muliantara, A., Thalheim, B. (eds.) DASFAA 2014, Part I. LNCS, vol. 8421, pp. 436–450. Springer, Heidelberg (2014)
18. Ahn, H.-K., Bae, S.W., Son, W.: Group nearest neighbor queries in the L_1 plane. In: Chan, T.-H.H., Lau, L.C., Trevisan, L. (eds.) TAMC 2013. LNCS, vol. 7876, pp. 52–61. Springer, Heidelberg (2013)