Plane-Sweep Algorithms for the K Group Nearest-Neighbor Query

George Roumelis[1], Michael Vassilakopoulos[2], Antonio Corral[3] and Yannis Manolopoulos[1]
[1]*Department of Informatics, Aristotle University, GR-54124 Thessaloniki, Greece*
[2]*Department of Electrical and Computer Engineering, University of Thessaly, GR-38221 Volos, Greece*
[3]*Department of Informatics, University of Almeria, 04120 Almeria, Spain*
groumeli@csd.auth.gr, mvasilako@uth.gr, acorral@ual.es, manolopo@csd.auth.gr

Keywords: Spatial Query Processing, Plane-Sweep, Group Nearest-Neighbor Query, Algorithms.

Abstract: One of the most representative and studied queries in Spatial Databases is the (K) Nearest-Neighbor (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 the recent years, considering data sets indexed by efficient spatial data structures. We study (K) GNNQs, considering non-indexed data sets, since this case is frequent in practical applications. And we present two (RAM-based) 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.

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 (Rigaux et al., 2002). It has 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 the 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 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 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. Note that, buiding indexes for the data sets would be needed only if several queries would be answered for the these data sets, which might evolve gradually in the course of time and not be completely replaced by new data sets.

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, (Preparata and Shamos, 1985). 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 (Hinrichs et al., 1988). For instance, the PS technique has been successfully applied in spatial query processing, mainly for intersection joins (Jacox and Samet, 2007).

In (Roumelis et al., 2014), 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 Section 2 that consider that 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 two (RAM-based) PS algorithms, 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. In the future, we plan to compare the best of our algorithms to existing index based solutions.

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

2 RELATED WORK AND MOTIVATIONS

GNN queries are introduced in (Papadias et al., 2004) 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 (Papadias et al., 2005). In (Papadias et al., 2004), the authors have developed three different methods were developed, 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 (Papadias et al., 2005) 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 KGNN. 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 KGNNs 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 policy. Experimental results showed that the performance of MBM is better than SPM and MQM for memory and disk resident queries, since it traverses the R-tree once and takes the query distribution area into account. Moreover, according to the comparison conducted in (Papadias et al., 2004), MBM is better than SPM in terms of node access and CPU cost while MQM is the worst.

In (Li et al., 2005), the authors propose two pruning strategies for KGNN 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 (Papadias et al., 2004).

A new method to evaluate a KGNN query for non-indexed data points using projection-based pruning strategies was presented in (Luo et al., 2007). 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 (Namnandorj et al., 2008), 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 (Hashem et al., 2010) an efficient algorithm for *K*GNN query considering privacy preserving was proposed, and the existing *K*GNN algorithms (Papadias et al., 2005) for point locations were extended to regions in order to preserve user privacy. In (Zhu et al., 2010), the *K*GNN query in road networks based on network voronoi diagram was solved. In (Jiang et al., 2013), the reverse top-*K* group nearest neighbor search is presented. In (Zhang et al., 2013), the *K*NN and *K*GNN queries are extended to get a new type of query, so-called *K* Nearest Group (*K*NG) query. It retrieves closest elements from multiple data sources, and it 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 (Lian and Chen, 2008; Li et al., 2014).

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 use the Plane-Sweep technique to solve the problem proposed in (Papadias et al., 2004), 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 Section 1).

3 PLANE-SWEEP ALGORITHMS FOR GNNQ

In this section we introduce two Plane-Sweep 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 $q_i \in Q$. 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). In the following we will denote the sum of distances (dx-distances) of a data point p to the set of query points Q by $sumdist(p,Q)$ ($sumdx(p,Q)$). 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. And, $sumdx(p,Q) \leq sumdist(p,Q)$, for a data point $p \in P$. Besides, we must emphasize that dx-distance ($dx\_dist(p,q)$, $\Delta x(p,q)$) is the distance function between two points p and q over the X-axis, an analogous expression is for dy-distance ($dy\_dist(p,q)$, $\Delta y(p,q)$) over the Y-axis. And the sum of dx-distances between one given point $p \in P$ and all query points of Q ($q_i \in Q$) is defined as $sumdx(p,Q) = \sum_{i=0}^{M-1} dx\_dist(p,q_i)$.

A max binary heap (keyed by *sumdist* and called $MaxKHeap$) that keeps the K data points with the smallest sum of distances to the query points found so far is used. The *sumdist* of the root of the $MaxKHeap$ is denoted by δ. In case the heap is not full (it contains less than K points), p will be inserted in the heap, regardless of $sumdist(p,Q)$. Otherwise, for each data point p being compared with the query set Q, there are 2 cases:

1. *Case 1*: If $sumdx(p,Q)$ is larger than or equal to δ, then there is no need to calculate $sumdist(p,Q)$ (**rule 1**).
2. *Case 2*: If the $sumdist(p,Q)$ is smaller than δ, then p will be inserted in the heap (**rule 2**).

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'.

In the algorithms that we have developed, 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 the 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 with the smallest sum of distances to the query points.

In (Papadias et al., 2004) 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 second 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 second 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 first 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 set P decreasing the index i until the *termination condition 1* will be true or the points of set p will have finished (lines 3-5). Lastly, the algorithm sets the sweeping line at the point $p[i]$ and continues scanning the points of set P increasing the index i until the *termination condition 2* will be true or the points of the set P will have finished (lines 6-8).

The second 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 set P decreasing the index j until the *termination condition 1* will be true or the x-coordinate of the current point of set P 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 set P will have finished (lines 13-15). Lastly, the algorithm sets the sweeping line at the point $p[i]$ and continues scanning the points of set P 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). We must highlight that the function *calc_sum_dist_in* 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 : $distpc = calc\_dist(p,c)$
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*.

The following examples illustrate the execution of the algorithms. The point data set P 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 Figure 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.

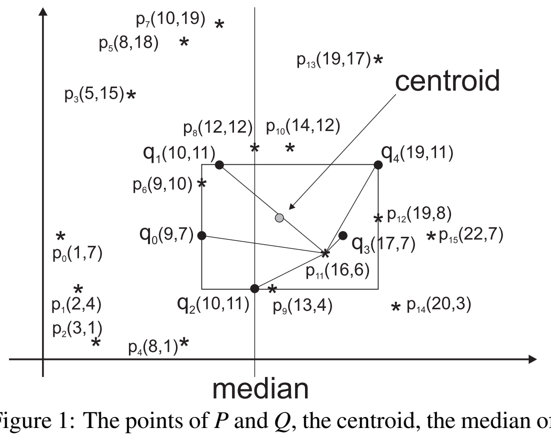

Figure 1: The points of P and Q, the centroid, the median of the query points and the initial position of the sweep line.

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 in-

Algorithm GNNPS

Input: Two X-sorted arrays of points $P = \{p[0], p[1], \cdots, p[N-1]\}$, $Q = \{q[0], q[1], \cdots, q[M-1]\}$, and $MaxKHeap$.
Output: $MaxKHeap$ storing the K Nearest Neighbors having smallest sums of distances to all query points.

```
i = find_closest_point(P, q[m])                      ▷ STEP 1 : Preperation. q[m] is the median point of query set Q.
j = i − 1
while j > −1 do                                      ▷ STEP 2 : Search in the range p[j].x ≤ q[m].x, descending j
    if calc_sum_dist(p[j−−], Q, MaxKHeap) == err_code_dx then      ▷ Termination condition 1
        break
while i < N do                                       ▷ STEP 3 : Search in the range p[i].x > q[m].x, ascending i
    if calc_sum_dist(p[i++], Q, MaxKHeap) == err_code_dx then      ▷ Termination condition 2
        break
```

Algorithm *calc_sum_dist*

Input: One point p, the sorted array of query points $Q = \{q[0], q[1], \cdots, q[M-1]\}$, 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() computes 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() computes the dx-distance between p and q[k] (Δx(p, 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)
```

serted 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 set P on the left of p_3 having sum of distances smaller than 52. The algorithm continues scanning the points of set P 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 cal-

Algorithm GNNPSC

Input: Two X-sorted arrays of points $P = \{p[0], p[1], \cdots, p[N-1]\}$, $Q = \{q[0], q[1], \cdots, q[M-1]\}$, and $MaxKHeap$.
Output: $MaxKHeap$ storing the K Nearest Neighbors having smallest sums of distances to all query points.

```
i = find_closest_point(P, q[m])                 ▷ STEP 1 : Preperation. q[m] is the median point of query set Q.
j = i − 1
c(x, y) = Calculate_Centroid_coord(Q)           ▷ calculate the coordinates of the Centroid
sumdistCQ = 0.0
for k = 0; k < M; k++ do                        ▷ for each query point q
    sumdistCQ += dist(c, q[k])
                                                ▷ STEP 2 : Search in the range p[j].x ≤ q[m].x, descending j
cont_search = true                              ▷ initialize the flag
while j > −1 and p[j].x > q[0].x do             ▷ for each point p[j] inside the query MBR in sweeping axis (X-axis)
    if calc_sum_dist_in(p[j−−], Q, c, sumdistCQ, MaxKHeap) == err_code_dx then    ▷ Termination condition 1
        cont_search = false
        break
if cont_search = true then
    while j > −1 do                             ▷ for each point p[j] on the left of the query MBR in sweeping axis
        if calc_sum_dist_out(p[j−−], Q, c, sumdistCQ, MaxKHeap) == err_code_dx then    ▷ Termination condition 1
            break
                                                ▷ STEP 3 : Search in the range p[i].x > q[m].x, ascending i
cont_search = true
while i < N and p[i].x < q[M−1].x do            ▷ for each point p[i] inside the query MBR in sweeping axis
    if calc_sum_dist_in(p[i++], Q, c, sumdistCQ, MaxKHeap) == err_code_dx then    ▷ Termination condition 2
        cont_search = false
        break
if cont_search = true then
    while i < N do                              ▷ for each point p[i] on the left of the query MBR in sweeping axis
        if calc_sum_dist_out(p[i++], Q, c, sumdistCQ, MaxKHeap) == err_code_dx then    ▷ Termination condition 2
            break
```

culations, 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 set P were examined.

GNNPSC starts (Step 1) by finding the first point of set P 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 set P 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 set P 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 set P on the right of the median query point q_2. The algorithm continues scanning the points of set P 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 set P were examined.

Algorithm *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.

```
function calc_sum_dist_in(p, Q, c, sumdistCQ, MaxKHeap)
    sumdist = 0.0, sumdx = 0.0
    if MaxKHeap is not full then
        for k = 0; k < M; k++ do                                  ▷ for each query point q
            sumdist+ = dist(p, q[k])          ▷ dist() computes the dx-distance between p and q[k]
        MaxKHeap.insert(p, sumdist)
        return sucessful_insertion
    else
        dpc = dist(p, c)                      ▷ dist() computes the distance between p and c
        if M · dpc − sumdistCQ ≥ MaxKHeap.root.dist then  ▷ prune p without computing distances
            return err_code_dist;                  ▷ not inserted because of sum of distances
        for k = 0; k < M; k++ do                                  ▷ for each query point q
            sumdx+ = dx_dist(p, q[k])   ▷ dx_dist() computes the dx-distance between p and q[k] (Δx(p, q[k]))
        if sumdx ≥ MaxKHeap.root.dist then                                          ▷ Rule 1
            return err_code_dx                ▷ exit k, all other points have longer distance
        for k = 0; k < M; k++ do                                  ▷ for each query point q
            sumdist+ = dist(p, q[k])           ▷ add the distance (dist) from the current point
        if sumdist < MaxKHeap.root.dist then                                        ▷ Rule 2
            MaxKHeap.insertFull(p, sumdist)
            return sucessful_insertion
        else
            return err_code_dist       ▷ not inserted because of sum of distances (sumdist)
```

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 (125*K*), 250000 (250*K*), 500000 (500*K*) and 1000000 (1000*K*) 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 PC with Intel Core 2 Duo, 2.2 GHz CPU with 4 GB of RAM and several GBs of secondary storage, with Ubuntu Linux v. 14.04, 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, (2) the number of points involved in calculations, and (3) the number of X-axis distance computations (*dx*-distance).

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 Figure 2, we depict the effect of the number of query points, N, on execution time of both 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). Analogous diagrams created for *dx*-distance and *dist* cal-

Algorithm *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.

```
function calc_sum_dist_out(p, Q, c, sumdistCQ, MaxKHeap)
    sumdist = 0.0, sumdx = 0.0
    if MaxKHeap is not full then
        for k = 0; k < M; k++ do                              ▷ for each query point q
            sumdist += dist(p, q[k])       ▷ dist() computes the dx-distance between p and q[k]
        MaxKHeap.insert(p, sumdist)
        return sucessful_insertion
    else
        dx = dx_dist(p, c)         ▷ dx_dist() computes the dx-distance between p and c (Δx(p,c))
        if M · dx ≥ MaxKHeap.root.dist then                   ▷ Rule 1
            return err_code_dx;            ▷ exit k, all other points have longer distance
        dy = dy_dist(p, c)         ▷ dy_dist() computes the dy-distance between p and c (Δy(p,c))
        distpc = √(dx² + dy²)
        if M · distpc ≥ MaxKHeap.root.dist + sumdistCQ then
            return err_code_dist_centroid;
        for k = 0; k < M; k++ do                              ▷ for each query point q
            sumdist += dist(p, q[k])
        if sumdist < MaxKHeap.root.dist then                  ▷ Rule 2
            MaxKHeap.insertFull(p, sumdist)
            return sucessful_insertion
        else
            return err_code_dist    ▷ not inserted because of sum of distances (sumdist)
```

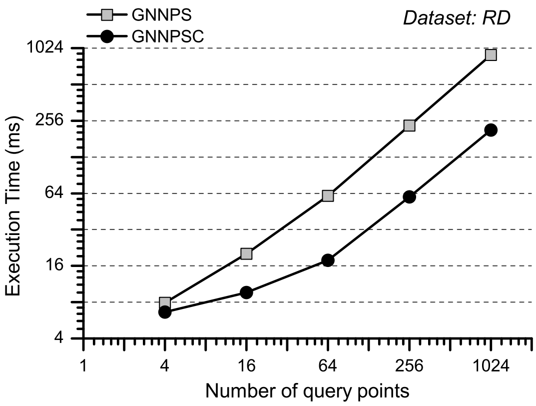

Figure 2: Execution time of the algorithms as a function of N (*RD* data set).

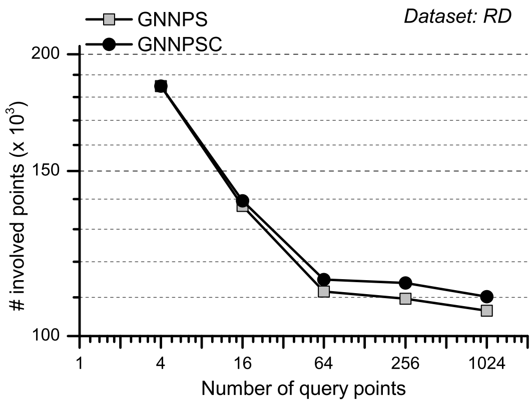

Figure 3: # Points involved in calculations of the algorithms as a function of N (*RD* data set).

culations had similar appearance. It is obvious that the increase of N 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 reference point of set P needs one calculation of distance while the computation of the sum of distances between the reference point and all query points needs N distance calculations).

For the same parameter settings and data set, in Figure 3, we depict the effect of N on the number of data set points involved in calculations. We observe that this number of points is reduced as N increases. The sums of distances of the points of data set P near the median are enlarged to a smaller extent, compared to the *sumdist* of the points outside the MBR. This enables the termination conditions and makes it possible to get nearest to the median query point. Moreover, we can observe in Figure 3 that GNNPSC needs more involved points and from Figure 2 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.

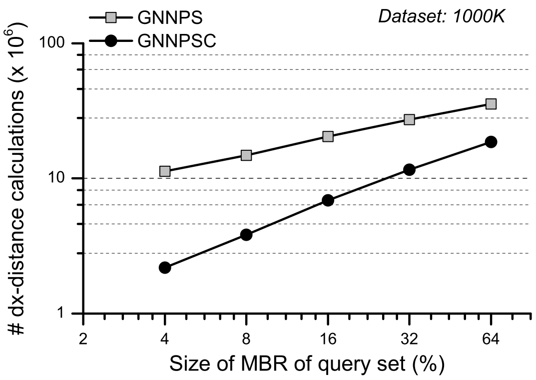

Figure 4: # *dx*-distance calculations of the algorithms as a function of the size of MBR (1000*K* data set).

In Figure 4, we depict the effect of the size of the query-set MBR, on *dx*-distance calculations of both algorithms for the 1000*K* data set (the number of group nearest neighbors, *K*, was equal to 8 and the number of query points was equal to 128).

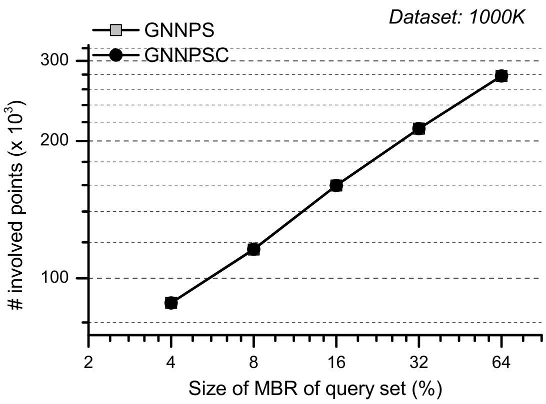

Figure 5: # Points involved in calculations of the algorithms as a function of the size of MBR (1000*K* data set).

Analogous diagrams created for executions time and distance calculations had similar appearance. 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 rate of increase. The size of MBR *M* 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 Figure 5, we depict the effect of the size of the query-set MBR on the number of points involved in calculations. We observe that this number of points is increased as *M* increases with a ratio smaller than 1.4. We observe in this figure that the number of points involved almost identical and the two lines are overlapped.

In Figure 6, we depict the effect of the number of group nearest-neighbors, *K*, on distance calculations of both algorithms for the *RDD* data set (the number of query points, *N*, was equal to 128 and the size of query-set MBR was 8% of the data set space).

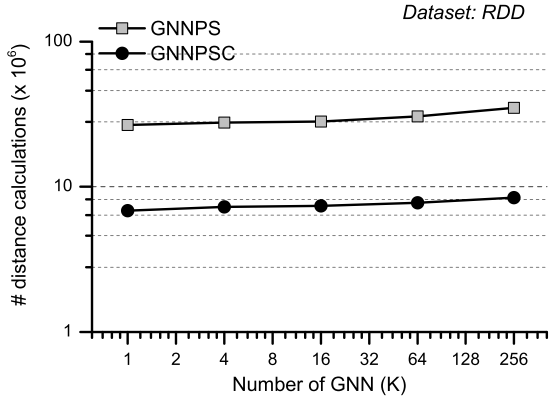

Figure 6: # distance calculations of the algorithms as a function of *K* (*RDD* data set).

Analogous diagrams created for execution times and *dx*-distance calculations had similar appearance. It is obvious that the increase of *K* does not significantly affect the execution time, *dx*-distance and complete distance (*dist*) calculations. For the same parameter settings and data set, in Figure 7, 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.

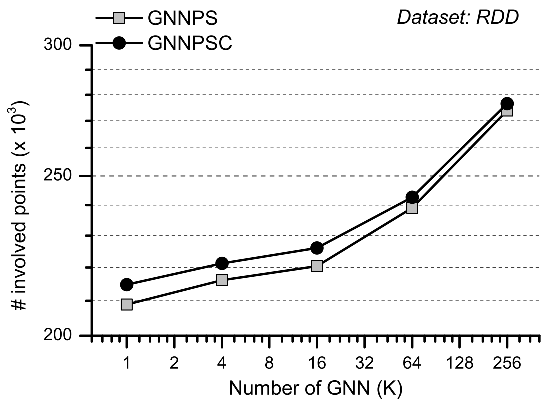

Figure 7: # Points involved in calculations of the algorithms as a function of *K* (*RDD* data set).

From the above experiments, we conclude that:

- The number of points of data sets (P) involved in the calculations of both 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.
- 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 decrease of the number of points involved in the calculations with respect to number of query points can be justified when the MBR size is 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 PS algorithms that can be efficiently applied on RAM-based data for processing the GNNQ. As the experimentation that we performed, using synthetic and real data sets, shows the use of median (in *GNNPS*) and, even more, the use of median and centroid (in *GNNPSC*), prunes the number of points involved in processing and the number of calculations.

Although, in this paper, we do not present a comparison of our algorithms with respect to the algorithms presented in (Papadias et al., 2004), comparing the results that we have presented to the results of (Papadias et al., 2004) for data sets of similar size (approximately 24.5K and 192/195K points) we observe that our algorithms achieve competitive performance.

This is an initial observation. A detailed comparison could be performed in the future, using the same data sets on the same machine. Moreover, the algorithms we present could be transformed / extended to work on high volume, disk resident data that are transferred in RAM in blocks. Moreover, 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),

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 (Ahn et al., 2013). To prove property **B**, for every point $p \in P$ and $q \in Q$, we use

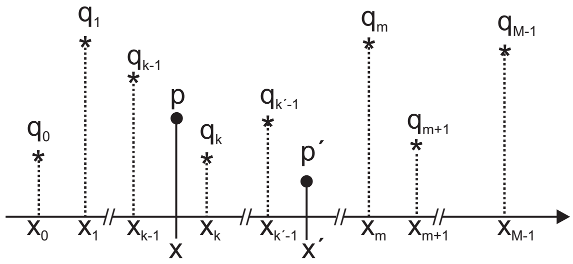

Figure 8: The point p has K query points on the left and the point p' ($p'.x > p.x$) has K' query points on the left.

$$\Delta x(p,q) = \begin{cases} p.x - q.x & \text{if } p.x \geq q.x \\ q.x - p.x & \text{if } p.x < q.x \end{cases}$$

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 (Figure 8), then:

$$\begin{aligned} sumdx(p,Q) &= \sum_{i=0}^{K-1}(p.x - q[i].x) + \sum_{i=K}^{M-1}(q[i].x - p.x) \\ &= Kp.x - \sum_{i=0}^{K-1} q[i].x + \sum_{i=K}^{M-1} q[i] - (M-K)p.x \\ &= (2K-M)p.x - \sum_{i=0}^{K-1} q[i].x + \sum_{i=K}^{M-1} q[i].x \end{aligned}$$

For another point $p' \in P$ with $p'.x > p.x$ which has K' query points on the left (Figure 8) 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:

$$\begin{aligned} \Delta sumdx &= sumdx(p',Q) - sumdx(p,Q) \\ &= (2K-M)(p'.x - p.x) \\ &\quad +2\left[(K'-K)p'.x - \sum_{i=K}^{K'-1} q[i].x\right] \end{aligned}$$

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:

$$\begin{aligned} \Delta sumdx &= (2K-M)(p'.x - p.x) + 2(K'-K)p'.x \\ &\quad -2\sum_{i=K}^{K'-1} q[i].x \end{aligned}$$

$$
\begin{aligned}
&\leq (2K-M)(p'.x-p.x)+2(K'-K)p'.x \\
&\quad -2(K'-K)p.x \\
&= 2(K-M)(p'.x-p.x) \\
&\quad +2(K'-K)(p'.x-p.x) \\
&= (2K-M+2K'-2K)(p'.x-p.x) \\
&= (2K'-M)(p'.x-p.x) < 0
\end{aligned}
$$

It is proven that for all points p on the left of the median query point the sum of dx-distances is strictly decreasing. □

ACKNOWLEDGEMENTS

Work supported by the GENCENG project (SYNERGASIA 2011 action, supported by the European Regional Development Fund and Greek National Funds); project number 11SYN 8 1213. Work also supported by the MINECO research project [TIN2013-41576-R] and the Junta de Andalucía research project [P10-TIC-6114].

REFERENCES

Ahn, H., Bae, S. W., and Son, W. (2013). Group nearest neighbor queries in the L_1 plane. In *TAMC Conference*, pages 52–61. Springer.

Hashem, T., Kulik, L., and Zhang, R. (2010). Privacy preserving group nearest neighbor queries. In *EDBT Conference*, pages 489–500. ACM.

Hinrichs, K., Nievergelt, J., and Schorn, P. (1988). Plane-sweep solves the closest pair problem elegantly. *Information Processing Letters*, 26(5):255–261.

Jacox, E. H. and Samet, H. (2007). Spatial join techniques. *ACM Trans. Database Syst.*, 32(1):7.

Jiang, T., Gao, Y., Zhang, B., Liu, Q., and Chen, L. (2013). Reverse top-k group nearest neighbor search. In *WAIM Conference*, pages 429–439. Springer.

Li, H., Lu, H., Huang, B., and Huang, Z. (2005). Two ellipse-based pruning methods for group nearest neighbor queries. In *ACM-GIS Conference*, pages 192–199. ACM.

Li, J., Wang, B., Wang, G., and Bi, X. (2014). Efficient processing of probabilistic group nearest neighbor query on uncertain data. In *DASFAA Conference*, pages 436–450. Springer.

Lian, X. and Chen, L. (2008). Probabilistic group nearest neighbor queries in uncertain databases. *IEEE Trans. Knowl. Data Eng.*, 20(6):809–824.

Luo, Y., Chen, H., Furuse, K., and Ohbo, N. (2007). Efficient methods in finding aggregate nearest neighbor by projection-based filtering. In *ICCSA Conference*, pages 821–833. Springer.

Namnandorj, S., Chen, H., Furuse, K., and Ohbo, N. (2008). Efficient bounds in finding aggregate nearest neighbors. In *DEXA Conference*, pages 693–700. Springer.

Papadias, D., Shen, Q., Tao, Y., and Mouratidis, K. (2004). Group nearest neighbor queries. In *ICDE Conference*, pages 301–312. IEEE.

Papadias, D., Tao, Y., Mouratidis, K., and Hui, C. K. (2005). Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.*, 30(2):529–576.

Preparata, F. P. and Shamos, M. I. (1985). *Computational Geometry - An Introduction*. Springer, New York, NY.

Rigaux, P., Scholl, M., and Voisard, A. (2002). *Spatial databases - with applications to GIS*. Elsevier, San Francisco, CA.

Roumelis, G., Vassilakopoulos, M., Corral, A., and Manolopoulos, Y. (2014). A new plane-sweep algorithm for the k-closest-pairs query. In *SOFSEM Conference*, pages 478–490. Springer.

Zhang, D., Chan, C., and Tan, K. (2013). Nearest group queries. In *SSDBM Conference*, page 7. ACM.

Zhu, L., Jing, Y., Sun, W., Mao, D., and Liu, P. (2010). Voronoi-based aggregate nearest neighbor query processing in road networks. In *ACM-GIS Conference*, pages 518–521. ACM.