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, building 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 $K$ 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 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 ANQ 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 KGNN query considering privacy preserving was proposed, and the existing KGNN algorithms (Papadias et al., 2005) for point locations were extended to regions in order to preserve user privacy. In (Zhu et al., 2010), the KGNN query in road networks based on network voronoi diagram was solved. In (Jiang et al., 2013), the KGNN and KGQNN 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 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 KGNN query was studied by (Lian and Chen, 2008; Li et al., 2014).

Therefore, the KGNN 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 \( K \) (\( \geq 1 \)) 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 \( \text{sumdist}(p, Q) = \sum_{i=0}^{M-1} \text{dist}(p, q_i) \), where \( \text{dist}(p, q_i) \) 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 (\( d(X) \)-distances) of a data point \( p \) to the set of query points \( Q \) by \( \text{sumdist}(p, Q) \).

Note that, while the sweep line approaches (moves away from) the median point(s), \( \text{sumdist} \) will be decreasing (increasing). This is proved in the Appendix. And, \( \text{sumdx}(p, Q) \leq \text{sumdist}(p, Q) \), for a data point \( p \in P \). Besides, we must emphasize that \( dx \)-distance (\( dx \text{dist}(p, q), \text{dist}(p, q) \)) is the distance function between two points \( p \) and \( q \) over the \( X \)-axis, an analogous expression is for \( dy \)-distance (\( dy \text{dist}(p, q), \text{dist}(p, q) \)) over the \( Y \)-axis. And the sum of \( dx \)-distances between one given point \( p \in P \) and all query points of \( Q \) is defined as \( \text{sumdx}(p, Q) = \sum_{i=0}^{M-1} dx \text{dist}(p, q_i) \).

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

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

Let \( p \) with \( \text{sumdx}(p, Q) \geq \delta \), then, for every \( p' \) with \( p'x \geq p_x \), \( \text{sumdx}(p', Q) \geq \text{sumdx}(p, Q) \). Moreover, \( \text{sumdx}(p', Q) \geq \text{sumdx}(p', Q) \). Thus, \( \text{sumdist}(p', Q) \geq \delta \) 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 \( \text{sumdx}(p, Q) \geq \delta \) 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 \( \text{sumdx}(p, Q) \geq \delta \) (termination condition 2). At this stage, MaxKHeap
Algorithm 1: GNPPS.

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

1: $i = \text{find}_\text{closest}_\text{point}(P,q[m])$ \hspace{1cm} // STEP 1: Preparation. $q[m]$ is the median point of query set $Q$.
2: $j = i - 1$
3: while $j > -1$ do
4: \hspace{1cm} if $\text{calc}_\text{sum}_\text{dist}(p[j],Q,\text{MaxKHeap}) == \text{err}_\text{code}_dx$ then \hspace{1cm} // Termination condition 1
5: \hspace{1cm} break
6: \hspace{1cm} while $i < N$ do
7: \hspace{1cm} \hspace{1cm} if $\text{calc}_\text{sum}_\text{dist}(p[i],Q,\text{MaxKHeap}) == \text{err}_\text{code}_dx$ then \hspace{1cm} // Termination condition 2
8: \hspace{1cm} \hspace{1cm} break

Algorithm 2: calc_sum_dist.

Input: One point $p$, the sorted array of query points $Q = \{q[0], q[1], \ldots, q[M-1]\}$, and MaxKHeap.
Output: Value successful_insertion or err_code_dist 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: \hspace{1cm} for $k = 0; k < M; k + +$ do \hspace{1cm} for each query point $q$ 5: \hspace{1cm} \hspace{1cm} distsum + = dist(p, q[k]) \hspace{1cm} // dist() computes the Euclidean distance between $p$ and $q[k]$ 6: \hspace{1cm} MaxKHeap.insert(p, sumdist) 7: \hspace{1cm} return successful_insertion 8: else 9: \hspace{1cm} for $k = 0; k < M; k + +$ do \hspace{1cm} for each query point $q$ 10: \hspace{1cm} \hspace{1cm} sumdx + = dx_dist(p, q[k]) \hspace{1cm} // dx_dist() computes the dx-distance between $p$ and $q[k]$ (dx(p, q[k])) 11: \hspace{1cm} if sumdx $\geq$ MaxKHeap.root.dist then \hspace{1cm} // Rule 1 12: \hspace{1cm} \hspace{1cm} return err_code_dx \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} exit $k$, all other points have longer distance 13: \hspace{1cm} \hspace{1cm} for $k = 0; k < M; k + +$ do \hspace{1cm} for each query point $q$ 14: \hspace{1cm} \hspace{1cm} \hspace{1cm} sumdist + = dist(p, q[k]) \hspace{1cm} // add the distance (dist) from the current point 15: \hspace{1cm} \hspace{1cm} if sumdist $\geq$ MaxKHeap.root.dist then 16: \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} MaxKHeap.insertFull(p, sumdist) 17: \hspace{1cm} \hspace{1cm} return successful_insertion 18: \hspace{1cm} else 19: \hspace{1cm} \hspace{1cm} return err_code_dist \hspace{1cm} // not inserted because of sum of distances (sumdist)

will contain the $K$ data points with the smallest sums of distances to the query points.

In (Papadias et al., 2004) it was proved that for every data point $p$ with $|Q| \cdot \text{dist}(p,c) \geq \delta + \text{sumlist}(c, Q)$, $p$ 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 $p$, $\text{sumdx}(p, Q) = |Q| \cdot |p.x - c.x|$.

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 $Q$ ($\text{sumdx}(p, Q)$):

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

The first algorithm (that is only based on median) is called GNPPS 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 GNNPS and it uses the helper
Algorithm 3: GNNPSC.

**Input:** Two X-sorted arrays of points $P = \{p[0], p[1], \ldots, p[N-1]\}$, $Q = \{q[0], q[1], \ldots, q[M-1]\}$, and MaxKHeap.

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

1: $i = \text{find closest point}(P, q[m])$ \hspace{1cm} \text{STEP 1: Preperation.} $q[m]$ is the median point of query set $Q$.
2: $j = i - 1$ \hspace{1cm} \text{STEP 2: Search in the range} $p[j].x \leq q[m].x$, descending $j$
3: $c(x,y) = \text{Calculate Centroid coord}(Q)$ \hspace{1cm} \text{STEP 3: Search in the range} $p[i].x > q[m].x$, ascending $i$
4: $\text{sumdistCQ} = 0$ \hspace{1cm} \text{initial the flag}
5: for $k = 0; k < M; k++$ do \hspace{1cm} \text{for each query point} $q$
6: $\text{sumdistCQ} += \text{dist}(c, q[k])$ \hspace{2cm} $\text{Termination condition 1}$
7: \hspace{1cm} cont_search = true \hspace{2cm} $\text{Termination condition 2}$
8: while $j > -1$ and $p[j].x > q[0].x$ do \hspace{1cm} \text{for each point} $p[j]$ inside the query MBR in sweeping axis (X-axis)
9: \hspace{2cm} if $\text{calc sum dist in}(p[j] --> , Q, c, \text{sumdistCQ}, \text{MaxKHeap}) == \text{err code} \_dx$ then \hspace{1cm} $\text{Termination condition 1}$
10: \hspace{2cm} break \hspace{2cm} $\text{Termination condition 2}$
11: \hspace{1cm} cont_search = false \hspace{2cm} $\text{Termination condition 2}$
12: if $\text{cont_search} = \text{true}$ then \hspace{1cm} $\text{calc sum dist out}(p[j --> ], Q, c, \text{sumdistCQ}, \text{MaxKHeap}) == \text{err code} \_dx$ \hspace{1cm} $\text{Termination condition 2}$
13: \hspace{2cm} if $\text{calc sum dist out}(p[j --> ], Q, c, \text{sumdistCQ}, \text{MaxKHeap}) == \text{err code} \_dx$ then \hspace{1cm} $\text{Termination condition 2}$
14: \hspace{2cm} break \hspace{2cm} $\text{Termination condition 2}$
15: \hspace{1cm} break \hspace{2cm} $\text{Termination condition 2}$
16: while $i < N$ and $p[i].x < q[M-1].x$ do \hspace{1cm} \text{for each point} $p[i]$ inside the query MBR in sweeping axis (X-axis)
17: \hspace{2cm} if $\text{calc sum dist in}(p[i --> ], Q, c, \text{sumdistCQ}, \text{MaxKHeap}) == \text{err code} \_dx$ then \hspace{1cm} $\text{Termination condition 2}$
18: \hspace{2cm} break \hspace{2cm} $\text{Termination condition 2}$
19: \hspace{2cm} cont_search = false \hspace{2cm} $\text{Termination condition 2}$
20: \hspace{1cm} break \hspace{2cm} $\text{Termination condition 2}$
21: while $i < N$ do \hspace{1cm} \text{for each point} $p[i]$ on the left of the query MBR in sweeping axis
22: \hspace{2cm} if $\text{calc sum dist out}(p[i --> ], Q, c, \text{sumdistCQ}, \text{MaxKHeap}) == \text{err code} \_dx$ then \hspace{1cm} $\text{Termination condition 2}$
23: \hspace{2cm} break \hspace{2cm} $\text{Termination condition 2}$
24: \hspace{1cm} break \hspace{2cm} $\text{Termination condition 2}$

The remaining statements of $\text{calc sum dist in}$ from line 12 (12-22) are the same as $\text{calc sum dist}$.

The following examples illustrate the execution of the algorithms. The point data set $P$ is defined as $P = \{p_0(7,1); p_1(2,4); p_2(3,1); p_3(3,13); p_4(8,2); p_5(8,18); p_6(9,10); p_7(10,19); p_8(12,12); p_9(13,4); p_{10}(14,12); p_{11}(16,6); p_{12}(19,8); p_{13}(19,17); p_{14}(20,3); p_{15}(22,7)\}$, and the point query set $Q$ is defined as $Q = \{q_{10}(7,9); q_{11}(10,11); q_{12}(12,4); q_{13}(17,7); q_{14}(19,11)\}$. 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.

In GNNPSC, firstly (in Step 1) the algorithm searches for the point of the $P$ set which is on the right of the median of $q_{14}(12,4)$ query point (line 1). That is $p_6(13,4)$. 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 $\text{sumdist}(p_8, Q) = 30.209$ and the point $p_8$ is inserted in the MaxKHeap (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 $\text{calc sum dist}$ function (lines 9-19) is executed. The sum of distances is $\text{sumdist}(p_7, Q) = 61.108$ larger than the MaxKHeap.root.dist = 30.209 (condition in
Algorithm 4: calc_sum_dist_in.

Input: One point \( p \), set of query points \( Q \), centroid \( c \), its sum of distances to all query points \( \text{sumdist}_CQ \) 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, dist_CQ, MaxKHeap)
2: \( \sum \text{dist} = 0.0, \sum \text{dx} = 0.0 \)
3: if MaxKHeap is not full then
4: for \( k = 0; k < M; k++ \) do
5: \( \sum \text{dist} += \text{dist}(p,q[k]) \) \( \triangleright \) for each query point \( q \)
6: MaxKHeap insert(\( p, \sum \text{dist} \)) \( \triangleright \) for each query point \( q \)
7: return successful_insertion
8: else
9: \( \text{dpc} = \text{dist}(p,c) \) \( \triangleright \) dist() computes the distance between \( p \) and \( c \)
10: if \( \text{M - dpc - sumdist}_CQ \geq \text{MaxKHeap.root} \cdot \text{dist} \) then \( \triangleright \) prune \( p \) without computing distances
11: return err_code_dist; \( \triangleright \) not inserted because of sum of distances
12: for \( k = 0; k < M; k++ \) do
13: \( \sum \text{dx} += \text{dist}_d(p,q[k]) \) \( \triangleright \) dx_dist() computes the dx-distance between \( p \) and \( q[k] \) \( \triangleright \) Rule 1
14: if \( \sum \text{dx} \geq \text{MaxKHeap.root} \cdot \text{dist} \) then \( \triangleright \) exit \( k \), all other points have longer distance
15: return err_code_dx; \( \triangleright \) add the distance (dist) from the current point
16: for \( k = 0; k < M; k++ \) do
17: \( \sum \text{dist} += \text{dist}(p,q[k]) \) \( \triangleright \) for each query point \( q \)
18: if \( \sum \text{dist} < \text{MaxKHeap.root} \cdot \text{dist} \) then \( \triangleright \) MaxKHeap insertFull(\( p, \sum \text{dist} \)) \( \triangleright \) Rule 2
19: return successful_insertion
20: else
21: return err_code_dist \( \triangleright \) not inserted because of sum of distances (sumdist)

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

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 \( \text{sumdist}(p_6,Q) = 29.716 \) 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 \( p_3 \) and \( p_4 \) the sum of distances are \( \text{sumdist}(p_3,Q) = 60.317 \) and \( \text{sumdist}(p_4,Q) = 43.299 \), respectively; both larger than the MaxKHeap.root.dist and the points are rejected. In the sixth iteration, the point \( p_3 \) has \( \sum \text{dx}(p_3,Q) = 52 \) (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 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 \( \text{sumdist}(p_9,Q) = 27.835 \) 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 \( \text{sumdist}(p_{10},Q) = 30.570 \) and it is rejected. The next iteration will try the point \( p_{11} \) which has \( \text{sumdist}(p_{11},Q) = 26.599 \) 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 \( \text{sumdist}(p_{12},Q) = 28 \) 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 set \( P \) were examined.

GNPSC 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 \( \text{sumdist}(c,Q) = 23.374 \).
Algorithm 5: calc_sum_dist_out.

Input: One point $p$, set of query points $Q$, centroid $c$, its sum of distances to all query points $\sum_{dist} CQ$ 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($p, Q, c, \sum_{dist} CQ, \text{MaxKHeap}$)
2: \hspace{1cm} $\sum_{dist} = 0.0$, $\sum_{dx} = 0.0$
3: \hspace{1cm} if MaxKHeap is not full then
4: \hspace{2cm} for $k = 0; k < M; k + +$ do \hspace{1cm} $\triangleright$ for each query point $q$
5: \hspace{3cm} $\sum_{dist} + = \text{dist}(p,q[k])$ \hspace{1cm} $\triangleright$ $\text{dist}()$ computes the $dx$-distance between $p$ and $q[k]$
6: \hspace{2cm} MaxKHeap.insert($p, \sum_{dist}$)
7: \hspace{2cm} return successful_insertion
8: \hspace{1cm} else
9: \hspace{2cm} $dx = \text{dx}_\triangleright$ (p,c) \hspace{1cm} $\triangleright$ $\text{dx}_\triangleright$ (p,c) computes the $dx$-distance between $p$ and $c$ ($\Delta x(p,c)$)
10: \hspace{2cm} if $M \cdot dx \geq \text{MaxKHeap.root.dist}$ then \hspace{1cm} $\triangleright$ Rule 1
11: \hspace{3cm} return err_code_dx;
12: \hspace{2cm} $dy = \text{dy}_\triangleright$ (p,c) \hspace{1cm} $\triangleright$ $\text{dy}_\triangleright$ (p,c) computes the $dy$-distance between $p$ and $c$ ($\Delta y(p,c)$)
13: \hspace{2cm} $\text{dist}_{pc} = \sqrt{dx^2 + dy^2}$
14: \hspace{2cm} if $M \cdot \text{dist}_{pc} \geq \text{MaxKHeap.root.dist} + \sum_{dist} CQ$ then \hspace{1cm} $\triangleright$ for each query point $q$
15: \hspace{3cm} return err_code_dist_centroid;
16: \hspace{2cm} for $k = 0; k < M; k + +$ do \hspace{1cm} $\triangleright$ Rule 2
17: \hspace{3cm} $\sum_{dist} + = \text{dist}(p,q[k])$
18: \hspace{2cm} if $\sum_{dist} < \text{MaxKHeap.root.dist}$ then \hspace{1cm} MaxKHeap.insert_Full($p, \sum_{dist}$)
19: \hspace{2cm} return successful_insertion
20: \hspace{2cm} else
21: \hspace{3cm} return err_code_dist \hspace{1cm} $\triangleright$ not inserted because of sum of distances ($\sum_{dist}$)

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 $\sum_{dist}(p_8,Q) = 30.209$ 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 $\sum_{dist}(p_7,Q)$ 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 $p_9$ with $\sum_{dist}(p_9,Q) = 29.716$ is inserted (calc_sum_dist_in-lines 9-20), while points $p_5$ and $p_4$ are rejected with $\sum_{dist}(p_5,Q) = 60.137$ and $\sum_{dist}(p_4,Q) = 43.299$ respectively, both larger than the MaxKHeap.root.dist = 29.716 with the point $p_9$. The next point $p_3$ is the last point to be examined because it has $\sum_{dx}(p_3,Q) = 52$ 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 $\sum_{dist}(p_9,Q) = 27.835$ 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 $\sum_{dist}(p_{10},Q) = 30.370$ and it is rejected. The next iteration will try the point $p_{11}$ which has $\sum_{dist}(p_{11},Q) = 26.599$ 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 $\sum_{dx}(p_{12},Q) = 28$ 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.

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. $|RRD| = 1138240$ points and $|RRD| = 385274$ 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 distribution is common for natural properties of systems within Science). The first real data set \((\text{CL})\) 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 \((dx\text{-distance})\).

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 \(dx\text{-distance} \) and \(dist\) calculations 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. \(\text{GNNPSC}\) needs less time than \(\text{GNNPS}\), 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 \(\text{sumdist}\) 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 \(\text{GNNPSC}\) needs more involved points and from Figure 2 it is the fastest. This behaviour could be due to that in function \(\text{calc\_sum\_dist}\) 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 \(\text{GNNPSC}\) rather than being the cause of termination of the scanning.

In Figure 4, we depict the effect of the size of the query-set MBR, on \(dx\text{-distance}\) calculations of both 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 execution 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). 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.

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 $GNPSC$ remains always lower than the execution time of $GNPSC$, 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.

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


APPENDIX

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

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

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

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:

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

For another point \( p' \in P \) with \( p'.x > p.x \) which has \( K' \) query points on the left (Figure 8) and \( M - K' \) query points on the right, it is:

\[
\text{sumdx}(p', Q) = (2K' - M)p'.x - \sum_{i=0}^{K-1} q[i].x + \sum_{i=K}^{M-1} q[i].x
\]

The difference between \( dx \)-distances of the points \( p' \) and \( p \) is:

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

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

\[
(\Delta \text{sumdx}) = (2K - M)(p'.x - p.x) + 2(K' - K)p'.x \\
\leq (2K - M)(p'.x - p.x) + 2(K' - K)p'.x \\
= 2(K - M)(p'.x - p.x) \\
+ 2(K' - K)(p'.x - p.x) \\
= (2K - M + 2K' - 2K)(p'.x - p.x) \\
= (2K' - M)(p'.x - p.x) < 0
\]

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

For both types of cardinality of the query set \( Q \) and for the case \( p.x < p'.x < q[m].x \) it is: