A New Plane-Sweep Algorithm for the K-Closest-Pairs Query

George Roumelis[1⋆], Michael Vassilakopoulos[2⋆], Antonio Corral[3⋆⋆], and Yannis Manolopoulos[1]

[1] Dept. of Informatics, Aristotle University, GR-54124 Thessaloniki, Greece
E-mail: {groumeli,manolopo}@csd.auth.gr
[2] Dept. of Computer Science and Biomedical Informatics,
University of Thessaly, 2-4 Papasiopoulou st., 35100 Lamia, Greece.
E-mail: mvasilako@dib.uth.gr
[3] Dept. of Informatics, University of Almeria, 04120 Almeria, Spain.
E-mail: acorral@ual.es

Abstract. One of the most representative and studied Distance-Based Queries in Spatial Databases is the K-Closest-Pairs Query (KCPQ). This query involves two spatial data sets and a distance function to measure the degree of closeness, along with a given number K of elements of the result. The output is a set of pairs of objects (with one object element from each set), with the K lowest distances. In this paper, we study the problem of processing KCPQs between RAM-based point sets, using *Plane-Sweep* (*PS*) algorithms. We utilize two improvements that can be applied to a *PS* algorithm and propose a new algorithm that minimizes the number of distance computations, in comparison to the *classic PS* algorithm. By extensive experimentation, using real and synthetic data sets, we highlight the most efficient improvement and show that the new *PS* algorithm outperforms the *classic* one, in most cases.

Keywords: Spatial Query Processing, Plane-Sweep, Closest-Pair 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 [11]. 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. While queries involving a single data set have been studied extensively in the literature, Distance Join Queries (DJQs) on spatial data like K-Closest-Pairs Query

⋆ 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.
⋆⋆ Supported by the Junta Andalucia research project [TIC-06114].

(KCPQ) has not been paid similar attention. For this reason, in this work we will improve this kind of DJQ for spatial data (points) in terms of execution time using the *plane-sweep (PS)* technique.

One of the most important techniques in the computational geometry field is the *PS* algorithm, which is a type of algorithm that uses a conceptual *sweep line* to solve various problems in the Euclidean plane, E^2, [10]. 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 [6]. The *PS* technique has been successfully applied in spatial query processing, mainly for intersection joins [8]. In the context of DJQs, the *PS* technique has been used to restrict all possible combinations of pairs of points from the two data sets.

In the context of computational geometry, in [6], the *PS* algorithm is applied to find the closest pair in a set of points, in an elegant way. Two improvements when a new pair can be formed are proposed. The first one examines only candidates which may form a new closest pair with the fixed point p on the sweep line that lie in a half-circle centered at this point, with radius δ (the current distance threshold). The second one, since the use of a half-circle in a *PS* algorithm is complex, examines only candidates within a boundary rectangle (a rectangle with width δ in X-axis and, height 2 * δ in Y-axis ($p + \delta$ and $p - \delta$ from p)). A critical observation made in [6] is that, as the sweep line passes through a fixed point, there is at most a constant number of points that need to be checked. But this property does not hold in the KCPQ, which is essentially a generalization of the Bichromatic Closest-Pair problem and the number of points with monochromatic color in this problem cannot be bounded (Section 5.1 of [13]). Moreover, the algorithm of [6] uses an array and a balanced binary tree (e.g. AVL-tree) to sort on both axes, while we will use one array for each data set, sorting on one axis (e.g. X). Finally, our proposed *PS* algorithm can be easily adapted to distance-based join query processing on disk resident data.

The contributions of this paper consist in the following:

1. We enhance the *classic PS* algorithm for KCPQ with two improvements (sliding window and sliding semi-circle), which were proposed in [6] for the closest-pair problem over one data set, and here have been adapted to KCPQ, where two data sets are involved.
2. We improve processing of the *classic PS algorithm* for KCPQ, with a new algorithm called Reverse Run Plane-Sweep, *RRPS* or *RR PS*, that minimizes Euclidean and sweeping axis distance computations.
3. We present results of an extensive experimentation, that compares the performance of the different algorithms and algorithmic improvements.

The paper is organized as follows. In Section 2, we review the related literature and motivate the research reported here. In Section 3, we describe the *classic PS* for KCPQ. In Section 4, a new *PS* algorithm for KCPQ is presented. In Section 5, a comparative performance study is reported. Finally, in Section 6, conclusions on the contribution of this paper and future work are summarized.

2 Related Work and Motivation

There are numerous papers that study processing of join queries, and recently, an exhaustive analysis of techniques for spatial join taking into account a filter-and-refinement approach appeared in [8]. We can classify the spatial join methods depending on whether the sets of objects involved are indexed or not, but in all cases the *PS* technique plays an important role for reducing the CPU cost [8].

The KCPQ discovers the K pairs of data elements formed from two data sets that have the K smallest distances between them. The KCPQ is a combination of spatial join and nearest neighbor queries. Like a spatial join query, all pairs of objects are candidates for the result. Like a nearest neighbor query, proximity forms the basis for the final ordering. If both data sets are indexed by R-trees, the concept of synchronous tree traversal and Depth-First (DF) or Best-First (BF) traversal order can be combined for the query processing [3, 4, 7, 12]. In [7], incremental and non-recursive algorithms based on BF traversal using R-trees and additional priority queues for DJQs are presented. In [12], additional techniques as sorting and application of *PS* during the expansion of node pairs, and the use of the estimation of the distance of the K-th closest pair to suspend unnecessary computations of MBR distances are included to improve [7]. In [3, 4] non-incremental recursive (DF) and non-recursive (BF) algorithms are presented for solving the KCPQ, when the data sets are indexed by R*-trees. The main issue of the non-incremental variant is to separate the treatment of the terminal candidates (the elements of the leaf nodes) from the rest of the candidates (internal nodes) in the index data structures. In [4] the *PS* technique is also applied to limit the number of MBRs and points that must be paired up, thus reducing the number of distance computations. Finally, in [9] the *PS* technique has been used to answer the K Distance Join Query, another way to call to KCPQ, in order to find exactly K nearest pairs in non-incremental fashion, where R*-tree and Quadtree-like index structures are compared.

Recently, in [13] the *PS* technique is used to obtain the α-Distance for spatial query processing for fuzzy objects. Essentially, the computation of the α-Distance is to find the closest pair of qualified points of two fuzzy objects. The main property of this variant of the *PS* method is the use of two sweep lines to facility the search for the particular types of spatial queries with fuzzy objects, that has been presented in such research work.

Finally, the main motivation of this work is to improve the *classic PS algorithm* proposed in [4, 12] for KCPQs, in order to reduce the number of distance computations, making the query processing faster in terms of execution time.

3 Plane-Sweep in K-Closest-Pairs Query Processing

A common approach to performing spatial joins when both data sets are stored on disk is to partition the data until it is of a size that can be processed using internal memory *PS* algorithm [8]. The *classic PS* algorithm for KCPQ sweeps a scan line (*sweepline*) that is vertical to one of the axes through two sorted (in

relation to this axis) arrays of the two data sets. In general terms, this algorithm is an adaptation of the *PS* algorithm for intersection of MBRs presented in [2], considering two sets of points and a distance threshold δ (the distance of the K-th closest pair found so far) in the sweeping axis [4, 12]. This *classic PS* algorithm can be considered a *greedy* variant of the algorithm for the closest-pair problem described in [10, 6]. It is a greedy algorithm, because it always makes the choice that looks best at the moment. It also combines *PS* and *nested loop* techniques. Compared to the naive nested loop algorithm, except in unlikely situations, the *sweepline* limits the number of points that must be tested against one another [11].

In general, if we assume that the two point sets are P and Q, the *classic PS algorithm* consists of the following steps [4, 12]:

1. Sorting the entries of the two point sets, based on the coordinates of one of the axes in increasing or decreasing order. The axis for the *sweepline* can be established based on sweeping axis criteria (e.g. X-axis) and the order can be fixed by sweeping direction criteria (e.g. *forward sweep* (increasing order) or *backward sweep* (decreasing order)); both criteria are presented in [12].
2. After that, two pointers are maintained initially pointing to the first entry for processing of each sorted set of points. Assuming that X-axis is the sweeping axis and the order is increasing (from left to right, i.e *forward sweep*), let *pivot* be the point with the smallest X-value pointed by one of these two pointers, e.g. P, then the *pivot* is initialized to this point, $p_{pivot} \in P$.
3. Afterwards, the *pivot* must be paired up with the points stored in the other set of points ($q_j \in Q$) from left to right, satisfying $dx = q_j.x - p_{pivot}.x \leq \delta$, processing all points as candidate pairs where the *pivot* is fixed. After all possible pairs of entries that contain *pivot* have been paired up (i.e., the forward lookup stops when $q_j.x - p_{pivot}.x > \delta$ is verified), the pointer to the set of the *pivot* is increased to the next entry, *pivot* is updated with the point of the next smallest X-value pointed by one of the two pointers, and the process is repeated until one of the set of points is fully processed.

Highlight that the *PS technique* applies the distance function over the sweeping axis (in this case, the X-axis) because in the *PS technique*, the sweep is only over one axis (the best axis according to the criteria suggested in [12]). Moreover, the search is only restricted to the closest points with respect to the *pivot* according to the current *distance threshold* (δ). No duplicated pairs are obtained, since the points are always checked over sorted sets. Note that this kind of processing is called *forward sweep*, since it scans from left to right (or from right to left, *backward sweep*) the sorted sets in order to obtain pairs of points that will have a distance smaller than or equal to δ.

Clearly, the application of this technique can be viewed as a *sliding strip* on the sweeping axis with a width equal to the δ value starting from the *pivot* (i.e., $[0, \delta]$ in the X-axis), where we only choose all possible pairs of points that can be formed using the *pivot* and the other points from the remainder entries of

the other set of points that fall into the current *strip*. See in Figure 1[4] left, the strip in light grey color.

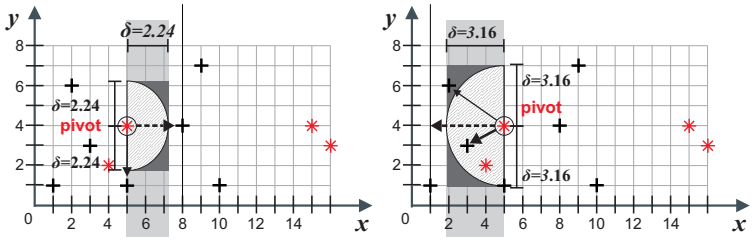

Fig. 1. *Classic* (left) and *RR* (right) *PS* algorithms, using sliding window or semi-circle.

According to the ideas proposed in [6] to improve the *PS* applied to the closest-pair problem over one set of points, here we will propose two improvements of the previous *classic PS* algorithm over two data sets to reduce the number of point-point distance computations on KCPQ algorithms.

1. An intuitive way to save the number of point-point distance computations is to bound the other axis (not only the sweeping axis) by δ as is illustrated in Figure 1 left. In this case, the search space is now restricted to the closest points inside the *window* with width δ and a height $2 * \delta$ (i.e., $[0, \delta]$ in the X-axis and $[-\delta, \delta]$ in the Y-axis, from the *pivot*). Clearly, the application of this technique can be viewed as a *sliding window* on the sweeping axis with a width equal to δ and height equal to $2 * \delta$ starting from the *pivot*. And we only choose all possible pairs of points that can be formed using the *pivot* point and the remainder points of the other data set that fall into the current *window*. See in Figure 1 left, the *window* in dark grey color.
2. If we try to reduce even more the search space, we can only select those points inside the *semi-circle* (or half-circle) centered in the *pivot* with radius δ (remember that the equation of all points $t = (t.x, t.y) \in E^2$ that fall inside the circle, centered in the point $p_{pivot} = (p_{pivot}.x, p_{pivot}.y) \in P$ and radius δ is $(t.x - p_{pivot}.x)^2 + (t.y - p_{pivot}.y)^2 \leq \delta^2$). And the application of this new improvement can be viewed as a *sliding semi-circle* with radius δ along the sweeping axis and centered on the *pivot* point, choosing only the points that fall inside the current *semi-circle*. See in Figure 1 left, the *semi-circle* centered at *pivot*.

4 Reverse Run Plane-Sweep algorithm for KCPQs

The *Reverse Run Plane-Sweep (RRPS, or RR PS)* algorithm is based on two concepts. First, every point that is used as a *reference point* (pivot) forms a *run*

[4] In both parts of Figure 1, a dotted arrow points to a point not included in the KCPQ result (paired with *pivot*), due to large dx; a thin arrow points to a point not included in the result, due to large distance to *pivot*; a thick arrow points to a point included in the KCPQ result (paired with *pivot*); the value of K is 3.

with other subsequent points of the same set. A *run* is a continuous sequence of points of the same set that doesn't contain any point from the other set. For each set, we keep a *left limit*, which is updated (moved to the right) every time that the algorithm concludes that it is only necessary to compare with points of this set that reside on the right of this limit. Each point of the *active run* (*reference point*) is compared with each point of the other set (*current point*) that is on the left of the first point of the *active run*, until the *left limit* of the other set is reached. Second, the *reference points* (and their *runs*) are processed in ascending X-order (the sets are X-sorted before the application of the *RRPS* algorithm). Each point of the *active run* is compared with the points of the other set (*current points*) in the opposite order (descending X-order).

A max binary heap (keyed by pair distances and called *MaxKHeap*) that keeps the K closest point pairs found so far is utilized. For each point of the *active run* being compared with a *current point*, there are 2 cases. *Case 1:* If the distance between this pair of points d is smaller than the distance of δ, then the pair will be inserted in the heap (rule 1). In case the heap is not full (it contains less than K pairs), the pair will be inserted in the heap, regardless of the pair distance. *Case 2:* If the distance between this pair of points in the sweeping axis dx is larger than or equal to δ, then there is no need to calculate the distance of the pair (rule 2). The *left limit* of the other set must be updated at the index value of the point being compared (a comparison with a point of the other set having an index value smaller than, or equal to the updated *left limit* will have X-distance larger than dx and is unnecessary).

Moreover, if the rightmost *current point* has an index value equal to the left limit of its set, then all the points of the *active run* will have larger dx from all the *current points* of the other set and the relevant pairs need not participate in computations (the algorithm advances to the start of the next run - rule 3).

The *RRPS* algorithm is depicted in Algorithm 1. The following example illustrates its operation. Let's consider the points of the right part of Figure 1 (depicting a snapshot of the *RRPS* operation), presented, in commonly sorted X-order, in Table 1. To simplify the algorithm operation (the stopping conditions),

i	0	1	2			3	4	5	6			7
$P[i]$	(1,1)	(2,6)	(3,3)			(5,1)	(8,4)	(9,7)	(10,1)			$(\infty, -)$
j				0	1					2	3	4
$Q[j]$				(4,2)	(5,4)					(15,4)	(16,3)	$(\infty, -)$

Table 1. The points of Figure 1 in X-sorted order.

a sentinel point with X-coordinate equal to ∞ is added to each set (line 2) . In case the two point sets overlap in X-dimension, initialization sets i (j) equal to 0, since the first run of P (Q) set starts at $P[0]$ ($Q[0]$). Moreover, initialization sets the left limit *leftp* (*leftq*) equal to -1 (line 5), since the first P (Q) point to be used in comparisons is $P[leftp{+}1]$ ($Q[leftq{+}1]$).

Since $P[0].x < Q[0].x$ (line 7), the *active run* is from the P set and consists of $P[0], P[1]$ and $P[2]$ ($P[2]$ is the last point of P before $Q[j]$). Each of the points of the *active run* should be compared with each of the *current points* of Q in reverse X-order which form the sequence $Q[j-1], \ldots, Q[leftq{+}1]$. However, since

Algorithm 1 Reverse Run Plane-Sweep

Input: $P[0..N-1], Q[0..M-1]$: X-sorted arrays of points. K: positive int
Output: *MaxKHeap*: binary Max Heap storing the K closest pairs between P and Q

```
    // Initialization
 1: i ← 0   j ← 0   continue ← TRUE
 2: P[N].x ← ∞   Q[M].x ← ∞      // sentinel points for simpler stoping conditions
 3: if P[N−1].x ≤ Q[0].x then     i ← N      // the sets do not overlap
 4: if Q[M−1].x ≤ P[0].x then     j ← M      // the sets do not overlap
 5: leftp ← −1   leftq ← −1      // comparisons start at P[leftp + 1], Q[leftq + 1]
    // Main Algorithm. P[i] (Q[j]): start of next P-run (Q-run)
 6: while continue do
 7:     if P[i].x < Q[j].x then      // the active run is from the P set
 8:         while P[i].x < Q[j].x do      // while active run unfinished. P[i]: ref_point
 9:             if j − 1 = leftq then      // Q[j − 1]: last cur_point - rule 3
10:                 advance i to next P-run and break while l.8
11:             for k = j − 1 downto leftq + 1 do      // Q[k]: cur_point
12:                 if MaxKHeap is not full then
13:                     calculate distance d b/t ref_point (P[i]) and cur_point (Q[k])
14:                     insert (ref_point, cur_point) with key d into MaxKHeap
15:                 else
16:                     calculate x-distance dx b/t ref_point (P[i]) and cur_point (Q[k])
17:                     if dx ≥ key of MaxKHeap root then      // dx ≥ δ - rule 2
18:                         leftq ← k and break for l.11
19:                     calculate distance d b/t ref_point (P[i]) and cur_point (Q[k])
20:                     if d < key of MaxKHeap root then      // d < δ - rule 1
21:                         insert (ref_point, cur_point) with key d into MaxKHeap
22:             increment i      // update ref_point P[i]
23:     else if j < M then      // the active run is from the (unfinished) Q set
24:         P[N].x ← Q[M − 1].x + 1      // P[N] should be < Q[M], since ...
    // ... else l.23 handles equal X-values b/t P and Q points
25:         while Q[j].x ≤ P[i].x do      // while active run unfinished. Q[j]: ref_point
26:             if i − 1 = leftp then      // P[i − 1]: last cur_point - rule 3
27:                 advance j to the next Q-run start and break while l.25
28:             for k = i − 1 downto leftp + 1 do      // P[k]: cur_point
29:                 if MaxKHeap is not full then
30:                     calculate distance d b/t ref_point (Q[j]) and cur_point (P[k])
31:                     insert (ref_point, cur_point) with key d into MaxKHeap
32:                 else
33:                     calculate x-distance dx b/t ref_point (Q[j]) and cur_point (P[k])
34:                     if dx ≥ key of MaxKHeap root then      // dx ≥ δ - rule 2
35:                         leftp ← k and break for l.28
36:                     calculate distance d b/t ref_point (Q[j]) and cur_point (P[k])
37:                     if d < key of MaxKHeap root then      // d < δ - rule 1
38:                         insert (ref_point, cur_point) with key d into MaxKHeap
39:             increment j      // update ref_point Q[j]
40:         P[N].x ← Q[N].x      // revert the P sentinel at the maximum real X-value
41:     else continue ← FALSE      // the points of both sets have been processed
```

$j-1 =$*leftq* (line 9), i is advanced to 3 ($P[3]$ is the start of the next P run) and processing of the *active run* is broken (rule 3).

During the next iteration of the "while" loop at line 6, $P[3].x > Q[0].x$. Thus the *active run* is from the Q set (line 23[5]) and consists of $Q[0]$ and $Q[1]$ ($Q[1]$ is the last point of Q before $P[i]$) and each of these points will be compared with each of the *current points* of P in reverse X-order which form the sequence $P[i-1], \ldots, P[$*leftp*$+1]$ ($P[2], \ldots, P[0]$). The pairs $(P[2], Q[0])$, $(P[1], Q[0])$ and $(P[0], Q[0])$ are inserted in the non-full heap ($K = 3$). The pair $(P[2], Q[1])$ is inserted in the heap, replacing $(P[1], Q[0])$, since its distance is smaller than δ (rule 1). The pair $(P[1], Q[1])$ is not inserted in the heap, due to its distance. The pair $(P[0], Q[1])$ is not inserted in the heap, due to $dx \geq \delta$ (rule 2). *leftp* is advanced to 0 (only comparisons with P points after $P[0]$ are necessary) and "for" loop at line 28 is broken. The *active run* ends with $i = 3$ and $j = 2$.

During the next iteration of the while loop at line 6, the *active run* consists of $P[3], P[4], P[5]$ and $P[6]$. Each of these points will be compared with $Q[1]$ and $Q[0]$. The pair $(P[3], Q[1])$ is inserted in the heap, replacing $(P[0], Q[0])$ since its distance is smaller than δ (rule 1). Next, the pair $(P[3], Q[0])$ is inserted in the heap, replacing $(P[3], Q[1])$, since its distance is smaller than δ (rule 1). The pair $(P[4], Q[1])$ is not inserted in the heap, due to $dx \geq \delta$ (rule 2). *leftq* is advanced to 1 (only comparisons with Q points after $Q[1]$ are necessary) and "for" loop at line 28 is broken (so the comparison of $P[4]$ with $Q[0]$ is avoided). During the next iteration of the "while" loop at line 8, $j - 1 =$ *leftq* (line 9), meaning that the rest of the P run will be skipped, saving comparisons (rule 3). The *active run* ends with $i = 7$ and $j = 2$.

During the next iteration of the while loop at line 6, the *active run* consists of $Q[2]$ and $Q[3]$. Each of these points will be compared with each point of the sequence $P[6], \ldots, P[1]$. The pair $(P[6], Q[2])$ is not inserted in the heap, due to $dx \geq \delta$ (rule 2). *leftp* is advanced to 6 (only comparisons with P points after $P[0]$ are necessary) and "for" loop at line 28 is broken (so the comparisons of $Q[2]$ with $P[5], \ldots, P[1]$ are avoided).

During the next iteration of the "while" loop at line 25, $i-1$ is equal to *leftp* (line 26), meaning that the rest of the Q run (in fact, only point $Q[3]$) will be skipped, saving comparisons (rule 3). The *active run* ends with $i = 7$ and $j = 3$.

During the next iteration of the while loop at line 6, since $P[i = 7] = Q[j = 3] = \infty$ and $j = M$, processing of the two sets is completed.

Note that the *classic PS* algorithm always processes pairs from left to right, even when the distance of the *pivot* point to its closest point of the other set is large (this is likely, since, *runs* of the two sets are in general interleaved). On the contrary, *RRPS* processes pairs of points in opposite X-orders, starting

[5] Note that the extra check "$j < M$" at line 23 guarantees that we have not reached the sentinel of the Q set and is necessary, since the "else" part of the main loop (Lines 23-39) handles the case of equal X-values between points and, at the time of this check, both sentinels equal ∞. At line 24, only for the duration of this "else" part, we set the P sentinel to a value between the last Q point and the Q sentinel (to terminate the loop at line 24 when the last run belongs to the Q set).

from pairs consisting of points that are the closest possible, avoiding further processing of pairs that is guaranteed not to be part of the result and substituting distance computations by simpler dx computations, when possible. This way, δ is expected to be updated more fastly and the processing cost of *RRPS* to be lower. In the specific example described previously, the *classic PS* algorithm would perform 9 distance computations, 15 dx computations, 8 heap insertions and would examine 18 pairs. *RRPS* performed 7 distance computations, 7 dx computations, 6 heap insertions and examined 10 pairs.

5 Experimentation

In order to evaluate the behavior of the proposed algorithms, we have used 6 real spatial data sets of North America, representing cultural landmarks (NAcl with 9203 points) and populated places (NApp with 24493 points), roads (NArd with 569120 line-segments) and railroads (NArr with 191637 line-segments). To create sets of points, we have transformed the MBRs of line-segments from NArd and NArr into points by taking the center of each MBR (i.e., |NArd| = 569120 points, |NArr| = 191637 points). Moreover, in order to get the double amount of points from NArr and NArd, we chose the two points with min and max coordinates of the MBR of each line-segment (i.e., |NArdD| = 1138240 points, |NArrD| = 383274 points). The data of these 6 files were normalized in the range $[0, 1]$ and the files were combined in pairs, excluding the combinations of NArr with NArrD and NArd with NArdD (since these data sets are correlated, due to the way D versions were created), we made 13 combinations of input sets. We have also created synthetic clustered data sets of 125000, 250000, 500000 and 1000000 points, with 125 clusters in each data set (uniformly distributed in the range $[0 - 1]$), where for a set having N points, $N/125$ points were gathered around the center of each cluster, according to Gaussian distribution. We made 4 combinations of synthetic data sets by combining two separate instances of data sets, for each of the above 4 cardinalities. For each of these 17 (=13+4) combinations of data sets, we executed the *classic PS* algorithm and the *RRPS* algorithm, using a sliding strip, a sliding window and a sliding semi-circle, for K equal to 1, 10, 100, 1000 and 10000. This sums up to 510 experiments (17 combinations $\times$ 2 algorithms $\times$ 3 variations $\times$ 5 K values). 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 KCPQ, not counting reading from disk files to main memory and sorting.
2. The number of Euclidean distance computations (*dist*).
3. The number of X-axis distance computations (dx).

5.1 Performance comparison of *PS* Algorithms for *K*CPQs

In the following, out of the large amount of results obtained from experimentation, some representative results are presented. In the upper (lower) part of

Table 2, the execution time in milliseconds of the *classic* (*RRPS*) algorithm, for two real and two synthetic data set combinations, for the sliding strip, window and semi-circle variations and for all K values are depicted. First, it is observed that among the algorithmic variations, the sliding semi-circle is constantly the most efficient one in both algorithms. Second, it is observed that the *RRPS* algorithm outperforms the *classic* one in all cases for the NApp-NArdD combination, for $K > 1$ for the 250KC-250KC combination, for $K = 1$ / sliding strip and for $K > 1$ / sliding window or semi-circle for the 1000KC-1000KC combination and for $K > 100$ for the NArr-NArd combination. In Figure 2,

PS	NApp-NArdD			NArr-NArd			250KC-250KC			1000KC-1000KC		
K	Strip	Window	sCircle	Strip	Window	sCircle	Strip	Window	sCircle	Strip	Window	sCircle
1	7.7	7.4	7.5	9.0	8.3	7.6	21.5	15.5	12.1	133.2	87.6	67.9
10	9.4	8.7	8.2	19.9	16.4	12.5	45.1	31.2	21.9	255.8	165.8	119.0
100	15.2	13.1	10.5	39.8	32.3	20.9	121.4	81.2	53.4	740.5	464.5	322.5
1000	34.3	28.1	19.6	98.1	76.9	46.1	329.2	221.0	140.9	2193.3	1378.3	936.7
10000	131.6	109.7	77.9	300.0	241.4	145.8	806.2	570.0	349.6	5353.3	3561.7	2281.1

RRPS	NApp-NArdD			NArr-NArd			250KC-250KC			1000KC-1000KC		
K	Strip	Window	sCircle	Strip	Window	sCircle	Strip	Window	sCircle	Strip	Window	sCircle
1	5.8	5.7	5.4	9.0	8.5	7.7	21.0	15.8	12.3	127.3	85.8	67.4
10	7.5	7.1	6.2	19.6	16.7	12.6	40.1	28.6	20.3	219.3	147.1	105.6
100	13.0	11.2	8.2	39.2	32.2	21.1	96.3	66.7	43.5	537.0	355.6	238.1
1000	30.4	24.8	15.5	94.7	74.2	44.2	247.6	172.0	106.4	1457.5	970.0	625.4
10000	108.9	88.5	56.2	276.1	214.6	124.8	667.5	475.5	288.5	3848.3	2615.0	1632.5

Table 2. Execution times of the *classic* (above) and *RR* (below) *PS* algorithms.

the relative performance of the two algorithms is depicted for the NApp-NArdD (upper-left diagram), NArr-NArd (upper-right diagram), 250KC-250KC (lower-left diagram) and 1000KC-1000KC (lower-right diagram) combinations, for all K values and all algorithmic variations. The percentages depicted express the fraction of the difference of the execution time of the *classic* minus the execution time of the *RRPS* algorithm, over the execution time of the *classic* algorithm (called gain). In other words, they express how much time is saved (positive values) or wasted (negative values) when the *RRPS* replaces the *classic* algorithm. These two figures were created by the same data that are depicted in Table 2 and they visualize the above conclusions about the relative performance of the two algorithms. Note that the variation of gain values depends on the distributions of the data sets and the value of K, both of which affect the number of computations each algorithm performs and how fast it approaches a good δ.

In Table 3 we summarize the results of relative performance for all the 255 (=510/2) cases of experimental comparisons performed. A "$-$" expresses gain $\leq -1.5\%$ (the *classic* algorithm is more efficient), a "$\times$" expresses $-1.5\% <$ gain $< 1.5\%$ (the two algorithms are almost equal) and a "$+$" expresses gain $\geq 1.5\%$ (the *RRPS* algorithm is more efficient). Moreover, in each row the minimum and maximum gain is shown. The *RRPS* algorithm is more efficient in 217 (or in 85% of the) cases (in fact, gain $\geq 5\%$ in 76% of the cases), the two algorithms are equal in 22 cases, while the *classic* algorithm is more efficient in 16 cases. Moreover, in all experiments the sliding semi-circle was the most efficient variation for both algorithms.

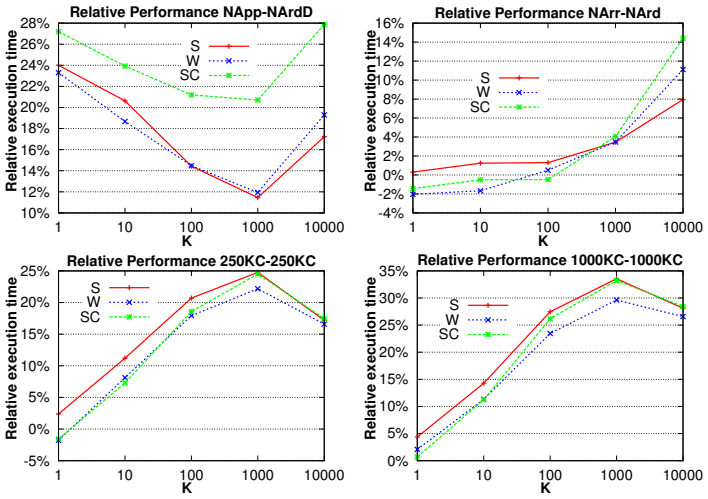

Fig. 2. Relative performance of the *classic* and *RR PS* algorithms.

Algorithmic variant:	Sliding Strip					Sliding Window					Sliding Semi-Circle					gain %	
Set combinations K:	10^0	10^1	10^2	10^3	10^4	10^0	10^1	10^2	10^3	10^4	10^0	10^1	10^2	10^3	10^4	min	max
NAcl−NApp	+	×	+	+	+	−	×	+	+	+	−	×	+	+	+	-4.4	39.6
NArr−NArd	×	×	×	+	+	−	−	×	+	+	×	×	×	+	+	-2.0	14.4
NArrD−NArd	−	−	×	+	+	−	−	×	+	+	−	−	−	+	+	-9.9	11.3
NArrD−NArdD	+	+	×	+	+	−	×	×	+	+	−	×	×	+	+	-2.6	10.6
All other (9/13) real data combinations	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	3.5	36.4
125KC−125KC	×	+	+	+	+	−	+	+	+	+	×	+	+	+	+	-2.5	19.8
250KC−250KC	+	+	+	+	+	−	+	+	+	+	−	+	+	+	+	-1.8	24.8
500KC−500KC	+	+	+	+	+	×	+	+	+	+	×	+	+	+	+	-0.5	29.7
1000KC−1000KC	+	+	+	+	+	+	+	+	+	+	×	+	+	+	+	0.7	33.5

Table 3. Summary of the relative performance of the *classic* and *RR PS* algorithms.

In Table 4, the relative gains in *dist* and *dx* computations of using the *RRPS* algorithm instead of the *classic* algorithm, utilizing only (due to space limitations) the semi-circle variant in both algorithms, for the same data set combinations and K values of Table 2, are depicted. It is obvious that the use of the *RRPS* algorithm saves both *dist* and *dx* computations. The percentages of gain varies significantly. Studying the rest of the results of the 510 experiments performed, we reach the same conclusion: the use of the *RRPS* algorithm always saves *dist* and *dx* computations, but the percentage of gain varies significantly and depends on the data sets combination, the algorithmic variation and K. Moreover, there is no linear or other obvious relation of these percentages to the execution time gain of using the *RRPS* algorithm instead of the *classic* algorithm. We plan to investigate this relation futher in future work.

	NApp-NArdD		NArr-NArd		250KC-250KC		1000KC-1000KC	
K	*dist*	*dx*	*dist*	*dx*	*dist*	*dx*	*dist*	*dx*
1	80.5%	86.1%	91.6%	47.0%	36.4%	16.7%	38.2%	12.9%
10	56.3%	67.1%	83.2%	18.9%	33.7%	19.0%	38.4%	19.9%
100	37.9%	36.9%	75.7%	7.6%	37.8%	23.8%	40.3%	29.6%
1000	41.2%	16.9%	59.7%	3.0%	38.2%	25.4%	33.3%	34.0%
10000	47.2%	7.0%	52.5%	1.9%	25.6%	16.4%	26.6%	28.0%

Table 4. Relative gain in *dist* and *dx* computations of *RRPS* (semi-circle variant).

6 Conclusions and future work

In this paper, we studied the problem of answering the KCPQ between two point sets that reside on RAM, using *PS* algorithms. We utilized two improvements (sliding window and sliding semi-circle) and proposed a new algorithm (*RRPS*) that minimizes the number of *dist* and *dx* computations, in comparison to the *classic PS* algorithm. By extensive experimentation using real and synthetic data sets, we concluded that the semi-circle improvement is the most efficient one, while the *RRPS* algorithm outperformes the *classic* one in 76% (85%) of the cases with a performance gain $\geq 5\%$ (1.5%) and may approach 40%.

In future work, we plan to extend the *RRPS* algorithm for finding closest pairs between non point data sets, like MBR sets that are stored in nodes of two trees and are combined during processing of distance join queries.

The development of the *RRPS* algorithm is the first step in developing a *PS* algorithm for the K-Closest-Pairs query for data sets that cannot be completely transferred to RAM, due to their large cardinalities. For such cases, the *PS* algorithm should utilize the available RAM and process the data sets in parts, minimizing not only distance computations but disk accesses too.

References

1. N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference, pp. 322-331, 1990.
2. T. Brinkhoff, H.P. Kriegel, B. Seeger. Efficient Processing of Spatial Joins Using R-trees. SIGMOD Conference, pp. 237-246, 1993.
3. A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos. Closest Pair Queries in Spatial Databases. SIGMOD Conference, pp. 189-200, 2000.
4. A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos. Algorithms for Processing K-Closest-Pair Queries in Spatial Databases. Data & Knowledge Engineering, 49(1), pp. 67-104, 2004.
5. A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Conference, pp. 47-57, 1984.
6. K. Hinrichs, J. Nievergelt & P. Schorn. Plane-Sweep Solves the Closest Pair Problem Elegantly. Information Processing Letters, 26(5), pp. 255-261, 1988.
7. G.R. Hjaltason, H. Samet. Incremental Distance Join Algorithms for Spatial Databases. SIGMOD Conference, pp. 237-248, 1998.
8. E.H. Jacox, H. Samet. Spatial Join Techniques. TODS 32(1), article 7, pp. 1-44, 2007.
9. Y.J. Kim, J. Patel. Performance Comparison of the R*-tree and the Quadtree for kNN and Distance Join Queries. IEEE TKDE, 22(7), pp. 1014-1027, 2010.
10. F.P. Preparata, M.I. Shamos, Computational Geometry: An Introduction, Springer-Verlag, 1985.
11. P. Rigaux, M. Scholl, A. Voisard. Introduction to Spatial Databases: Applications to GIS. Morgan Kaufmann, 2000.
12. H. Shin, B. Moon, S. Lee. Adaptive and Incremental Processing for Distance Join Queries. IEEE TKDE, 15(6), pp. 1561-1578, 2003.
13. K. Zheng, X. Zhou, P.C. Fung, K. Xie. Spatial query processing for fuzzy objects. VLDB Journal, 21, pp. 729-751, 2012.