Algorithms for Processing K-Closest-Pair Queries in Spatial Databases

A. Corral[1], Y. Manolopoulos[2], Y. Theodoridis[3], M. Vassilakopoulos[4]

[1]Department of Languages & Computation
University of Almeria, 04120 Almeria, Spain
E-mail: acorral@ual.es

[2]Department of Informatics
Aristotle University, 54006 Thessaloniki, Greece
E-mail: manolopo@csd.auth.gr

[3]Department of Informatics
University of Piraeus, 18534 Piraeus, Greece
E-mail: ytheod@unipi.gr

[4]Department of Informatics
Technological Educational Institute of Thessaloniki
P.O. Box 14561, 54101 Thessaloniki, Greece
E-mail: vasilako@it.teithe.gr

Abstract: This paper addresses the problem of finding the K closest pairs between two spatial datasets (the so called, K Closest Pairs Query, K-CPQ), where each dataset is stored in an R-tree. There are two different techniques for solving this kind of distance-based query. The first technique is the incremental approach, which returns the output elements one-by-one in ascending order of distance. The second one is the non-incremental alternative, which returns the K elements of the result all together at the end of the algorithm. In this paper, based on distance functions between two MBRs in the multidimensional Euclidean space, we propose a pruning heuristic and two updating strategies for minimizing the pruning distance, and use them in the design of three non-incremental branch-and-bound algorithms for K-CPQ between spatial objects stored in two R-trees. Two of those approaches are recursive following a Depth-First searching strategy and one is iterative obeying a Best-First traversal policy. The plane-sweep method and the search ordering are used as optimization techniques for improving the naive approaches. Besides, a number of interesting extensions of the K-CPQ (K-Self-CPQ, Semi-CPQ, K-FPQ (the K Farthest Pairs Query), etc.) are discussed. An extensive performance study is also presented. This study is based on experiments performed with real datasets. A wide range of values for the basic parameters affecting the performance of the algorithms is examined in order to designate the most efficient algorithm for each setting of parameter values. Finally, an experimental study of the behavior of the proposed K-CPQ branch-and-bound algorithms in terms of scalability of the dataset size and the K value is also included.

Keywords: Spatial databases, Branch-and-bound algorithms, Query processing, R-tree, Distance join, I/O and response time performance

1 Introduction

The term "Spatial Database" refers to a database that stores data for phenomena on, above or below the earth's surface [LaT92], or in general, various kinds of multidimensional entities of modern life (e.g. the layout of a VLSI design). In other words, a Spatial Database is a database system with the ability to handle geometric, geographic, or spatial data (i.e. data related to space). In a computer system, these data are represented by points, line segments, regions, polygons, volumes and other kinds of 2-d/3-d geometric entities and are usually referred to as spatial objects (from now on, simply objects). For example, a spatial database may contain polygons that represent building footprints from a satellite image, or points that represent the positions of cities, or line segments that represent roads. Spatial databases include specialized systems like Geographical databases, CAD databases, Multimedia databases, Image databases, etc. Recently, the role of spatial databases is continuously increasing in many modern applications; e.g. mapping, urban planning, transportation planning, resource management, geomarketing, archeology and environmental modeling are just some of these applications.

The key characteristic that makes a spatial database a powerful tool is its ability to manipulate spatial data, rather than simply to store and represent them. The basic form of such a database is answering queries related to the spatial properties of data. Some typical spatial queries are the following.

- A "Point Location Query" seeks for the objects that fall on a given point (e.g. the country where a specific city belongs).
- A "Range Query" seeks for the objects that are contained within a given region, usually expressed as a rectangle or a sphere (e.g. the pathways that cross a forest).
- A "Join Query" may take many forms. It involves two or more spatial datasets and discovers pairs (or tuples, in case of more than two datasets) of objects that satisfy a given spatial predicate [BKS93, HJR97, KoS97, LoR96, PaD96] (e.g. the pairs of boats and stormy areas, for boats sailing across a storm). The distance join [HjS98] was recently introduced to compute a subset of the Cartesian product of two datasets, specifying an order on the result based on distance (e.g. the pairs of hotels and archeological sites, ordered by driving distance up to 50 km between them).
- Finally, very common is the "Nearest Neighbor Query" that seeks for the objects residing more closely to a given object. In its simplest form, it discovers one such object (the Nearest Neighbor) [RKV95, HjS99]. Its generalization discovers K such objects (K Nearest Neighbors), for a given K (e.g. the K ambulances closer to a spot where an accident with K injured persons occurred).

Branch-and-bound [Iba87] has been the most successful technique for designing algorithms that answer queries on tree structures. Lower and upper bounding functions are the basis of the computational efficiency of branch-and-bound algorithms. Moreover, the computational behavior of this kind

of algorithms is highly dependent on the searching strategy chosen, for instance Best-First and Depth-First, which are used in most situations. Numerous branch-and-bound algorithms for queries (exact query, range query, nearest neighbor query and spatial join) using spatial access methods have been studied in the literature. Here, we show how these bounding functions and searching strategies perform when they are included in branch-and-bound algorithms for a special distance-based query, the K closest pairs query.

The distance between two objects is measured using some metric function over the underlying data space. The most common metric function is the Euclidean distance. We can use the Euclidean distance for expressing the concepts of "neighborhood" and "closeness". The concept of "neighborhood" is related to the discovery of all objects that are "near" to a given query object. The concept of "closeness" is related to the discovery of all pairs of objects that are "close" to each other. In this paper, we examine a query, called "K Closest Pairs Query" (K-CPQ), that discovers the K pairs ($K \geq 1$) of objects formed from two datasets that have the K smallest distances between them. The K-CPQ is a combination of join and nearest neighbor queries. Like a join query, all pairs of objects are candidates for the result. Like a nearest neighbor query, proximity metrics form the basis for pruning heuristics and the final ordering.

K-CPQs are very useful in many applications that use spatial data for decision making and other demanding data handling operations. For example, the first dataset may represent the cultural landmarks of the United States, while the second set may contain the most populated places of North America (see Figure 5.1 in Section 5). A K-CPQ will discover the K closest pairs of cities and cultural landmarks providing an order to the authorities for the efficient scheduling of tourist facilities creation, etc. The K value could be dependent on the budget of the authorities allocated for this purpose.

The fundamental assumption is that the two datasets are indexed by structures of the R-tree family [Gut84]. The R-tree and its variants (R^+-tree [SRF87], R*-tree [BKS90], etc.) are considered as excellent choices for indexing various kinds of spatial data (points, line segments, rectangles, polygons, etc.) and have already been adopted in commercial systems (e.g. Informix [Bro01], Oracle [Ora01]). In this paper, based on distance functions between MBRs (Minimum Bounding Rectangles) in the multidimensional Euclidean space, we present a pruning heuristic and two updating strategies for minimizing the pruning distance (i.e. the distance of the K-th closest pair found during the processing of the algorithm) and use them in the design of three different non-incremental branch-and-bound algorithms for solving the K-CPQ. Two of them are recursive algorithms following a Depth-First searching strategy; the third one is iterative following a Best-First traversal policy. The plane-sweep method and the search ordering are used as optimization techniques for improving the naive approaches. Moreover, an extensive performance study, based on experiments performed with real datasets, is presented. A wide range of values for the basic parameters affecting the performance of the

algorithms is examined. The outcome of the above studies is the determination of the algorithm outperforming all the others for each set of parameter values.

In addition, experimental results for three special cases of the query under consideration are examined: (1) the K Self Closest Pair Query (K-Self-CPQ), where both datasets refer to the same entity; (2) the Semi Closest Pair Query (Semi-CPQ), where for each object of the first dataset, the closest object of the second dataset is computed; and (3) the K Farthest Pairs Query (K-FPQ), finding the K farthest pairs of objects from two datasets. Besides, the scalability of the proposed algorithms is studied. That is, the increase of the I/O cost and response time of each algorithm is analyzed in terms of the dataset size and the number K of closest pairs.

The organization of this paper is as follows: Section 2 discusses the incremental and non-incremental algorithmic approaches for the CPQ, as well as the motivation of this research. In Section 3, the K-CPQ, a review of R-trees and some useful functions on pairs of MBRs are presented. In Section 4, a pruning heuristic, two updating strategies and three new non-incremental branch-and-bound algorithms for K-CPQ are introduced. Section 5 exhibits a detailed performance study of all algorithms for K-CPQs, including the effect of buffering, K-Self-CPQ, Semi-CPQ, K-FPQ and a scalability study. In Section 6, conclusions on the contribution of this paper and related future research plans are presented.

2 Related Work and Motivation

There are two approaches for solving distance-based queries. The first one is the incremental alternative [HjS95, HjS99, HjS98, SML00], which satisfies the query by reporting the desired elements of the result in ascending order of distance in a pipelined fashion (one-by-one), i.e. the user can have part of the final result before the end of the algorithm execution. In other words, when the incremental algorithms have obtained K elements of the result, then it is not necessary to restart the algorithm to find the (K+1)-th element but just to perform an additional step. The kernel of the incremental algorithms is a priority queue built on a distance function associated to the specific kind of the distance-based query. The strong point of this approach is that, when K is unknown in advance, the user stops when he/she is satisfied by the result. On the other hand, when the number of elements in the result grows, the amount of the required resources to perform the query increases too. Thus, incremental algorithms are competitive when a small quantity of elements of the result is needed.

The second approach is the non-incremental one [RKV95, CMT00], which assumes that K is known in advance and reports the K elements of the result all together at the end of the algorithm, i.e. the user can not have any result until the algorithm ends. The main issue of the non-incremental variant is to separate the treatment of the terminal candidates (the elements of the final result) from the rest of the candidates. Since the algorithm is not incremental, when one wants to obtain M results just after the execution of the algorithm for K (M > K), he/she must restart the algorithm with M as input with-

out reusing the previous result to obtain the remaining M-K elements. However, the advantage of the non-incremental approach is that the pruning process during the algorithm execution is more effective even when K is large enough, as it will be shown later in the experimental section.

Numerous algorithms exist for answering distance-based queries. Most of these algorithms focus in the nearest neighbors query (NNQ) on multidimensional access methods. The importance of NNQ is motivated by the great number of application fields such as GIS, CAD, pattern recognition, document retrieval, etc. For example, algorithms exist for k-d-trees [FBF77], quadtree-related structures [HjS95], R-trees [RKV95, HjS99], etc. In addition, similar algorithms can be applied to other recent multidimensional access methods for decreasing the I/O activity and the CPU cost.

To the authors' knowledge, [HjS98, SML00, CMT00] are the most relevant references for closest pairs queries (CPQ) in spatial databases using R-trees. In [HjS98], an incremental algorithm based on priority queues is presented for solving the distance join query and its extension for semi-distance join query. The techniques proposed in [HjS98] are enhanced in [SML00] for the K-distance join and incremental distance join by using adaptive multi-stage and plane-sweep techniques [PrS85], as well as other improvements based on sweeping axis and sweeping direction. In [CMT00], non-incremental recursive and iterative branch-and-bound algorithms are presented for solving the K-CPQ on points.

The first two efforts described in the previous paragraph follow the incremental approach, optimizing the required resources and the processing strategy. The motivation for this paper (our main objective), is to extend and enhance the work presented in [CMT00] with respect to the design of branch-and-bound algorithms (recursive and iterative) in a non-incremental way for answering K-CPQs between two datasets stored in an R-tree [Gut84]. To carry out this extension and enhancement, we study the distance functions and the branch-and-bound algorithms used to answer the K-CPQ. We propose a pruning heuristic and two updating strategies that comprise the kernel in the design of branch-and-bound algorithms for solving this kind of query. Besides, we apply techniques for improving the performance with respect to the I/O activity (buffering) and response time (plane-sweep). Moreover, we study extensions of our non-incremental algorithms for operations related to the K-CPQ, as K-Self-CPQ, Semi-CPQ, K-FPQ, etc. Finally, in our experiments we employ very large real datasets of different nature (line segments and points) to study the performance of the algorithms.

3 The K-Closest-Pair Query using R-trees

In this section, K-CPQ is defined and a brief description of R-trees is also presented, pointing out the main characteristics of the R*-tree. Moreover, some useful functions on pairs of MBRs, which will be used in branch-and-bound algorithms for answering the K-CPQ are introduced.

3.1 Definition of the Query

We assume a finite point dataset P in the d-dimensional data space $\Re^d$ and a metric distance function *dist* for a pair of points, i.e. *dist*: $P \times P \rightarrow \Re^+$. $\forall$p, q ,r $\in P$, the function *dist* satisfies the four following conditions: (1) dist(p, q) $\geq$ 0, "non-negativity". (2) dist(p, q) = 0 $\Leftrightarrow$ p = q, "identity". (3) dist(p, q) = dist(q, p), "symmetry". (4) dist(p, q) $\leq$ dist(p, r) + dist(r, q), "Δ-inequality".

The more general expression for *dist* between two points, p = (p_1, p_2, … , p_d) and q = (q_1, q_2, … , q_d) in the d-dimensional data space is called L_t-distance (L_t), L_t-metric or Minkowski distance. Its definition (included in this paper for clarity) is as follows:

$$L_t(p,q) = \left(\sum_{i=1}^{d} |p_i - q_i|^t \right)^{1/t}, \quad 1 \leq t < \infty \qquad \text{and} \qquad L_\infty(p,q) = \max_{1 \leq i \leq d} |p_i - q_i|, \quad t = \infty$$

For t = 2 we derive the Euclidean distance and for t = 1 the Manhattan distance. These are the most known L_t-metrics. Often, the Euclidean distance is used as the distance function but, depending on the application, other distance functions may be more appropriate.

A property of the L_t-distance function *(dimension distance property)* is that the value of this function for a given dimension (1 $\leq$ i $\leq$ d) is always smaller than or equal to the total computation of the L_t-distance for all d dimensions.

Dimension distance property:

$$L_t(p,q,i) = |p_i - q_i| \leq L_t(p,q), \qquad 1 \leq i \leq d \ \text{ and } \ 1 \leq t \leq \infty \tag{1}$$

The d-dimensional Euclidean space, $E^{(d)}$, is the pair ($\Re^d$, L_2). In other words, it is the d-dimensional data space $\Re^d$, equipped with the Euclidean distance (in the sequel, we will use *dist* instead of L_2). In the following, we formally define the K-CPQ.

Definition. Let two point sets, P = {p_1, p_2, ... , p_{NP}} and Q = {q_1, q_2, ... , q_{NQ}} in $E^{(d)}$, be stored in a spatial database. Then, the result of the K closest pairs query K-CPQ(P, Q, K) is a set of ordered sequences of K (1 $\leq$ K $\leq$ |P|·|Q|) different pairs of points of P×Q, with the K smallest distances between all possible pairs of points that can be formed by choosing one point of P and one point of Q:

K-CPQ(P, Q, K) = {((p_1, q_1), (p_2, q_2), ... , (p_K, q_K)), p_1, p_2, ... , $p_K \in$ P, q_1, q_2, ..., $q_K \in$ Q:

(p_i, q_i) $\neq$ (p_j, q_j), i $\neq$ j, 1 $\leq$ i, j $\leq$ K and $\forall$(p_i, q_j) $\in$ P×Q – {(p_1, q_1), (p_2, q_2), ... , (p_K, q_K)},

dist(p_i, q_j) $\geq$ dist(p_K, q_K) $\geq$ dist(p_{K-1}, q_{K-1}) $\geq$ … $\geq$ dist(p_2, q_2) $\geq$ dist(p_1, q_1)}

Note that, due to ties of distances, the result of the K-CPQ may not be a unique ordered sequence for a specific pair of point sets P and Q. The aim of the proposed algorithms is to find one of the possible instances, although it would be straightforward to obtain all of them.

The extension of K-CPQ definition in terms of points to other *spatial data types* (line segment, region, rectangles, etc.) is straightforward. An object "obj" in a spatial database is usually defined by

several non-spatial attributes and one attribute of some *spatial data type*. This spatial attribute describes the object's spatial extent "obj.G", i.e. the location, shape, orientation and size of the object. In the spatial database literature, the terms: geometric description, shape description and spatial extension are often used instead of spatial extent. The single modification on the K-CPQ definition is the replacement of points p and q with objects p and q with spatial extent p.G and q.G, respectively, in $E^{(d)}$ and the replacement of the distance between two points (dist) with the distance between two objects, provided that a distance function can be defined between the type of the objects.

As stated earlier, the two datasets are stored in R-trees. This means that the specific data organization by R-trees should be taken into account in the design of efficient algorithms. In the next subsection, we briefly review the R-tree family.

3.2 R-trees

R-trees [Gut84] are hierarchical, height balanced multidimensional data structures, designed for using in secondary storage, and it is a generalization of B-trees [Com79] for multidimensional data spaces. They are used for the dynamic organization of a set of d-dimensional objects represented by their d-dimensional MBRs. These MBRs are characterized by "min" and "max" points of hyper-rectangles with faces parallel to the coordinate axes. Using the MBR instead of the exact geometrical representation of the object, its representational complexity is reduced to two points, where the most important object features (position and extension) are maintained. Consequently, the MBR is an approximation widely employed.

Each R-tree node corresponds to the MBR that contains its children. The tree leaves contain pointers to the database objects instead of pointers to child nodes. The nodes are implemented as disk pages. It must be noted that the rectangles that surround different nodes may overlap. Besides, a rectangle can be included (in the geometrical sense) in many nodes, but can be associated to only one of them. This means that a search may demand visiting many nodes, before confirming the existence or not of a given MBR.

The rules obeyed by the R-tree are as follows: leaves reside on the same level; each leaf contains entries of the form (MBR, Oid), such that MBR is the minimum bounding rectangle that encloses the object determined by the identifier Oid; internal nodes contain entries of the form (MBR, Addr), where Addr is the address of the child node and MBR is the minimum bounding rectangle that encloses MBRs of all entries in that child node; nodes (except possibly for the root) of an R-tree of class (m, M) contain between m and M entries, where $m \leq \lceil M/2 \rceil$ (M and m are also called maximum and minimum branching factor or fan-out); the root contains at least two entries, if it is not a leaf. Figure 3.1 depicts some rectangles on the left and the corresponding R-tree on the right. Dotted lines denote the bounding rectangles of the subtrees that are rooted in inner nodes.

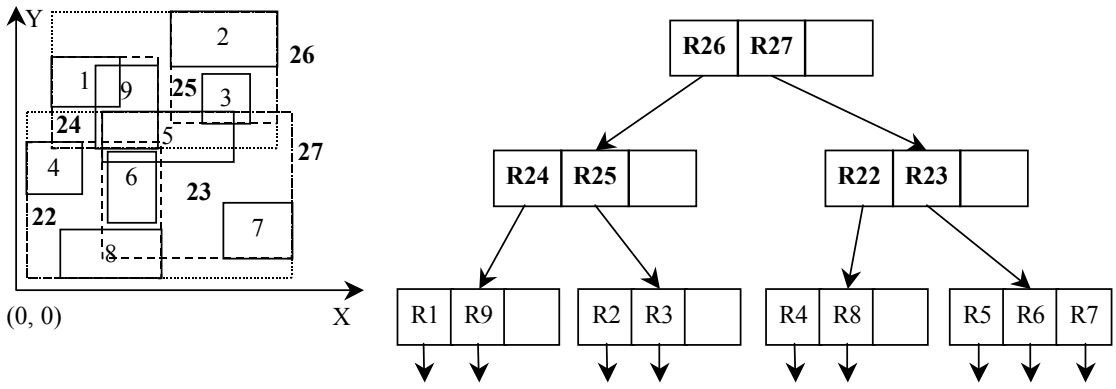


Figure 3.1. An example of an R-tree.

Like other tree access methods, an R-tree partitions the multidimensional space by grouping objects in a hierarchical manner. A subspace occupied by an R-tree node is always contained in the subspace of its parent node, i.e. the *MBR enclosure property*. According to this property, an MBR of an R-tree node (at any level, except at the leaf level) always encloses the MBRs of its descendent R-tree nodes. Spatial join algorithms as well as distance-based query algorithms commonly use this characteristic of spatial containment between MBRs of R-tree nodes.

Another important property of the R-trees is the *MBR face property* [RKV95]. This property means that every face of any MBR of an R-tree node (at any level) touches at least one point of some object in the spatial database. This characteristic of the MBR faces stored in R-tree nodes is used by distance-based query algorithms.

Many variations of R-trees have appeared in the literature (an exhaustive survey can be found in [GaG98]). One of the most popular and efficient variations is the R*-tree [BKS90]. The R*-tree added two major enhancements to the R-tree, in case that a node overflows. First, rather than just considering the area, the node-splitting algorithm in the R*-tree also minimized the perimeter and overlap enlargement of the minimum bounding rectangles. It tends to reduce the number of subtrees to follow for search operations. Second, the R*-tree introduced the notion of *forced reinsertion* to make the tree shape less dependent to the insertion order. When a node overflows, it is not split immediately, but a portion of entries of the node is reinserted from the tree root. The forced reinsertion provides two important improvements: (i) it may reduce the number of splits and, (ii) it is a dynamic technique for tree reorganization. With these two enhancements, the R*-tree generally outperforms R-tree. It is commonly accepted that the R*-tree is one of the most efficient R-tree variants. Thus, we choose R*-trees to perform our experimental study.

3.3 Functions on Pairs of MBRs

Since the different algorithms for K-CPQ act on pairs of R-trees (R_P and R_Q), some important functions on pairs of MBRs will be defined. Let N_P and N_Q be two internal nodes of R_P and R_Q, with M_P and M_Q the respective MBRs of N_P and N_Q. These MBRs contain all the points residing in the respec-

tive subtrees. In order for these MBRs to be the minimum ones, at least one point has to be located at each edge of their rectangles. The following functions of MBRs work for any number of dimensions, although in the examples are restricted to 2 dimensions. Let r_1, r_2, r_3 and r_4 be the four edges of M_P, whereas s_1, s_2, s_3 and s_4 are the four edges of M_Q. By *MinDist(r_i, s_i)* we denote the minimum distance between two points falling on r_i and s_i. Accordingly, by *MaxDist(r_i, s_i)* we denote the maximum distance between two points falling on r_i and s_i. In the sequel, we extend definitions of metrics between a point and an MBR that appear in [RKV95] and define a set of useful functions of two MBRs. In case M_P and M_Q are disjoint we can define a function that expresses the minimum possible distance of two points contained in different MBRs:

$$MINMINDIST(M_P, M_Q) = \min_{i,j}\{MinDist(r_i, s_j)\} \quad (2)$$

If the two nodes' MBRs intersect, *MINMINDIST(M_P, M_Q)* equals 0. In any case (either intersecting or disjoint MBRs) we can define the functions:

$$MINMAXDIST(M_P, M_Q) = \min_{i,j}\{MaxDist(r_i, s_j)\} \quad (3)$$

$$MAXMAXDIST(M_P, M_Q) = \max_{i,j}\{MaxDist(r_i, s_j)\} \quad (4)$$

MAXMAXDIST expresses the maximum possible distance of any two points contained in different MBRs. *MINMAXDIST* expresses an upper bound for the distance of at least one pair of points. More specifically, there exists at least one pair of points (contained in different MBRs) with distance smaller than or equal to *MINMAXDIST*. In Figure 3.2, two MBRs and their *MINMINDIST*, *MINMAXDIST* and *MAXMAXDIST* distances are depicted.

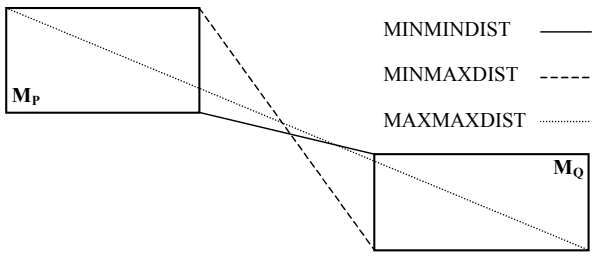


Figure 3.2: Two MBRs and their *MINMINDIST*, *MINMAXDIST* and *MAXMAXDIST* in $E^{(2)}$.

Let R(s, t) represent an MBR in $E^{(d)}$, where s = (s_1, s_2, ... , s_d) and t = (t_1, t_2, ... , t_d), such that $s_i \le t_i$, for $1 \le i \le d$, are the endpoints of one of its major diagonals. We present algorithmic definitions of the above functions in $E^{(d)}$. Using these definitions, it is easy to devise efficient algorithms for calculating the functions.

Definition. Given two MBRs R_1(s, t) and R_2(p, q) in $E^{(d)}$, *MINMINDIST(R_1(s, t), R_2(p, q))* is defined as:

$$MINMINDIST(R_1(s,t), R_2(p,q)) = \sqrt{\sum_{i=1}^{d} y_i^2}\ , \text{where } y_i = \begin{cases} p_i - t_i, & if\ p_i > t_i \\ s_i - q_i, & if\ s_i > q_i \\ 0, & otherwise \end{cases} \quad (5)$$

It is interesting that Equation 5 also holds between points or between an MBR and a point:

- If $t_i = s_i$ for $R_1(s, t)$ and $q_i = p_i$ for $R_2(p, q)$, R_1 and R_2 degenerate into two points $s = (s_1, s_2, \ldots, s_d)$ and $p = (p_1, p_2, \ldots, p_d)$; then:

$$MINMINDIST(R_1, R_2) = PointDistance(s, p) = \sqrt{\sum_{i=1}^{d} |s_i - p_i|^2}$$

- If $q_i = p_i$ for $R_2(p, q)$, R_2 degenerates into a point $p = (p_1, p_2, \ldots, p_d)$; then [RKV95]:

$$MINMINDIST(R_1, R_2) = MINDIST(R_1, p) = \sqrt{\sum_{i=1}^{d} y_i^2}\ , \text{where } y_i = \begin{cases} p_i - t_i, & if\ p_i > t_i \\ s_i - p_i, & if\ s_i > p_i \\ 0, & otherwise \end{cases}$$

Thus, *MINMINDIST* of two MBRs is a generalization of the distance between points and MBRs. This property allows us to apply *MINMINDIST* to pairs of any kind of elements (i.e. MBRs or points) stored in R-trees during the computation of branch-and-bound algorithms for K-CPQ.

Another property of the *MINMINDIST* function is based on the *dimension distance property*, and it can be stated as follows (easily proven by combining the definition of *MINMINDIST* between two MBRs (Equation 5) and the *dimension distance property* (Equation 1)).

Dimension MINMINDIST property:

Given two MBRs $R_1(s, t)$ and $R_2(p, q)$ in $E^{(d)}$, the value of $MINMINDIST(R_1, R_2)$ for a given dimension $1 \le i \le d$ is always smaller than or equal to $MINMINDIST(R_1, R_2)$.

$$MINMINDIST(R_1, R_2, i) \le MINMINDIST(R_1, R_2),\ \forall 1 \le i \le d$$

$$\text{such that } MINMINDIST(R_1, R_2, i) = y_i, \text{ where } y_i = \begin{cases} p_i - t_i, & if\ p_i > t_i \\ s_i - q_i, & if\ s_i > q_i \\ 0, & otherwise \end{cases}$$

The main usefulness of *MINMINDIST* for a given dimension is that it is computationally cheaper than *MINMINDIST* and thus we may obtain a performance gain in some situations (e.g. plane-sweep technique [PrS85]) for a given dimension.

A third important property of the MBRs stored in two different R-trees related to the *MINMINDIST* function is called *MBRs MINMINDIST property*. This property can be stated as follows (again, it is easily proven by combining the definition of *MINMINDIST* between two MBRs (Equation 5) and the *MBR enclosure property*).

MBRs MINMINDIST property:

Consider two R-tree internal nodes N_P and N_Q (with MBRs M_{P0} and M_{Q0}) of two R-trees R_P and R_Q, respectively. These two internal nodes are enclosing two sets of MBRs $\{M_{P1}, M_{P2}, \ldots, M_{PA}\}$ and $\{M_{Q1}, M_{Q2}, \ldots, M_{QB}\}$. Then (the proofs are simple and left as an exercise to the interested reader)

$$MINMINDIST(M_{Pi}, M_{Qj}) \geq MINMINDIST(M_{P0}, M_{Q0}): \forall 1 \leq i \leq A \text{ and } \forall 1 \leq j \leq B$$

$$MINMINDIST(M_{P0}, M_{Qj}) \geq MINMINDIST(M_{P0}, M_{Q0}): \forall 1 \leq j \leq B$$

$$MINMINDIST(M_{Pi}, M_{Q0}) \geq MINMINDIST(M_{P0}, M_{Q0}): \forall 1 \leq i \leq A$$

In other words, the minimum distance between two MBRs of two internal nodes N_P and N_Q (with MBRs M_{P0} and M_{Q0}) is always smaller than or equal to the minimum distance between one of the MBRs enclosed by M_{P0} and one of the MBRs enclosed by M_{Q0} (i.e. MINMINDIST is monotonically non-decreasing with the R-tree heights). This property allows us to limit the search space when we apply a branch-and-bound algorithm for K-CPQ. Figure 3.3 illustrates this property for A = B = 3.

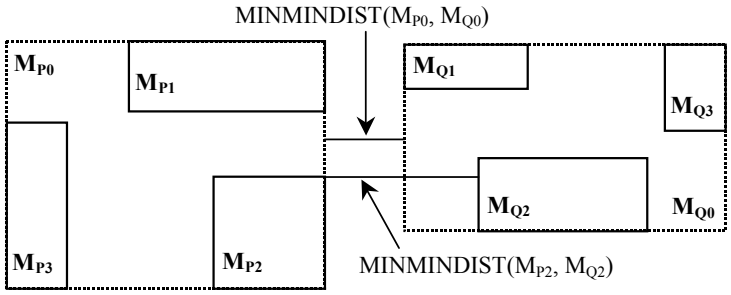


Figure 3.3: MBRs *MINMINDIST* property in $E^{(2)}$.

Definition. Given two MBRs $R_1(s, t)$ and $R_2(p, q)$ in $E^{(d)}$, *MAXMAXDIST($R_1(s, t)$, $R_2(p, q)$)* is defined as:

$$MAXMAXDIST(R_1(s,t), R_2(p,q)) = \sqrt{\sum_{i=1}^{d} y_i^2}, \text{ where } y_i = \max\{|s_i - q_i|, |t_i - p_i|\} \qquad (6)$$

Definition. Given two MBRs $R_1(s, t)$ and $R_2(p, q)$ in $E^{(d)}$, *MINMAXDIST($R_1(s, t)$, $R_2(p, q)$)* is defined as:

$$MINMAXDIST(R_1(s,t), R_2(p,q)) = \min_{1 \leq i \leq d, 1 \leq j \leq d} \begin{Bmatrix} MAXDIST(F(R_1(s,t), i, s_i), F(R_2(p,q), j, p_j)), \\ MAXDIST(F(R_1(s,t), i, t_i), F(R_2(p,q), j, p_j)), \\ MAXDIST(F(R_1(s,t), i, s_i), F(R_2(p,q), j, q_j)), \\ MAXDIST(F(R_1(s,t), i, t_i), F(R_2(p,q), j, q_j)), \end{Bmatrix} \qquad (7)$$

where *F(R(z, x), i, x_i)* denotes the face of the MBR R(z, x) containing all points with value x_i at coordinate i. In other words, it denotes the face that is orthogonal to dimension i at value x_i (note that, for an MBR R(z, x), there are two faces orthogonal to dimension i, one at value x_i and another at value z_i). The function *MAXDIST* calculates the maximum distance between two such faces from different

MBRs. For this calculation, it suffices to compare the distances between each endpoint of one face to each endpoint of the other face. Each face of dimension d has 2^{d-1} endpoints. For example, the set of endpoints of $F(R(z, x), i, x_i)$ consists of all the points with value x_i at coordinate i and with value either x_l, or z_l at each coordinate $l \neq i$.

Note that the definitions of *MINMINDIST* and *MAXMAXDIST* lead to algorithms of O(d) time, whereas the definition of *MINMAXDIST* results in an exponential algorithm, due to the computation of the distances (MAXDIST) between each endpoint of one face (2^{d-1}) and each endpoint of the other face (2^{d-1}) and the calculation of the minimum distance from 2d faces in one MBR against the other 2d faces in the other MBR. For small d values (e.g. d < 4) the cost of using definition of *MINMAXDIST* is not prohibitive. For larger d values, an alternative definition could be used which gives an upper bound for the value produced by definition of *MINMAXDIST*. This definition is presented in the following and leads to an O(d) algorithm. For each dimension j (where $1 \leq j \leq d$), it computes the minimum of the *MAXDIST* values of all the pairs of faces orthogonal to dimension j (the two faces of each pair belong in different MBRs). The final result is the minimum of all these j values (a minimum of sub-minima). In general, the computed value is larger than or equal to (an upper bound of) the minimum of the *MAXDIST* values of every possible pair of faces.

Definition. Given two MBRs $R_1(s, t)$ and $R_2(p, q)$ in $E^{(d)}$, an upper bound of $MINMAXDIST(R_1(s, t), R_2(p, q))$ is:

$$MINMAXDIST\left(R_1(s,t), R_2(p,q)\right) \leq \sqrt{\min_{1 \leq j \leq d}\left\{ x_j^2 + \sum_{i=1, i \neq j}^{d} y_i^2 \right\}} \tag{8}$$

where $x_j = \min\left\{ \left|s_j - p_j\right|, \left|s_j - q_j\right|, \left|t_j - p_j\right|, \left|t_j - q_j\right| \right\}$ and $y_i = \max\left\{ \left| s_i - q_i\right|, \left| t_i - p_i\right| \right\}$

Definition. Given two objects o_1 and o_2 in $E^{(d)}$, the minimum distance between them, denoted by $\|(o_1, o_2)\|$, is:

$$\left\|(o_1, o_2)\right\| = \min_{f_1 \in F(o_1), f_2 \in F(o_2)} \left\{ \min_{p_1 \in f_1, p_2 \in f_2} \left\{ dist(p_1, p_2) \right\} \right\} \tag{9}$$

where $F(o_1)$ and $F(o_2)$ denote the set of faces of the object o_1 and o_2 in $E^{(d)}$, respectively. Moreover, f_1 and f_2 are instances of the sets of faces $F(o_1)$ and $F(o_2)$. Here, *dist* is the Euclidean distance between two points p_1 and p_2 defined in $E^{(d)}$.

Lemma 1. Consider two MBRs $M_{P0}(s, t)$ and $M_{Q0}(p, q)$ in $E^{(d)}$, enclosing two set of objects $O_1 = \{o_{1i}: 1 \leq i \leq N_1\}$ and $O_2 = \{o_{2j}: 1 \leq j \leq N_2\}$, respectively. The following holds:

$$\forall (o_{1i}, o_{2j}) \in O_1 \times O_2, \; MINMINDIST(M_{P0}, M_{Q0}) \leq \left\|(o_{1i}, o_{2j})\right\| \tag{10}$$

Proof: From the definition of *MINMINDIST* between two MBRs and the *MBR face property*. □

Lemma 2. Consider two MBRs $M_{P0}(s, t)$ and $M_{Q0}(p, q)$ in $E^{(d)}$, enclosing two set of objects $O_1 = \{o_{1i}: 1 \le i \le N_1\}$ and $O_2 = \{o_{2j}: 1 \le j \le N_2\}$, respectively. The following is true:

$$\forall (o_{1i}, o_{2j}) \in O_1 \times O_2, \ \left\| (o_{1i}, o_{2j}) \right\| \le MAXMAXDIST(M_{P0}, M_{Q0}) \tag{11}$$

Proof: From the definition of *MAXMAXDIST* between two MBRs and the *MBR face property*. □

Lemma 3. Consider two MBRs $M_{P0}(s, t)$ and $M_{Q0}(p, q)$ in $E^{(d)}$, enclosing two set of objects $O_1 = \{o_{1i}: 1 \le i \le N_1\}$ and $O_2 = \{o_{2j}: 1 \le j \le N_2\}$, respectively. The following holds:

$$\exists (o_{1i}, o_{2j}) \in O_1 \times O_2, \ \left\| (o_{1i}, o_{2j}) \right\| \le MINMAXDIST(M_{P0}, M_{Q0}) \tag{12}$$

Proof: From the definition of *MINMAXDIST* between two MBRs and the *MBR face property*. □

From the previous properties and lemmas, we can deduce that *MINMINDIST(R_1, R_2)* and *MAXMAXDIST(R_1, R_2)* serve respectively as lower and upper bounding functions of the Euclidean distance from the K closest pairs of objects within the MBRs R_1 and R_2. In the same sense, *MINMAXDIST(R_1, R_2)* serves as an upper bounding function of the Euclidean distance from the closest pair of objects (K = 1) enclosed by the MBRs R_1 and R_2.

Usually, the distance functions are all based on a distance metric for points, dist(p_1, p_2), such as the Euclidean metric. As in [HjS98, HjS99], as long as the distance functions are "consistent", the algorithms based on them will work correctly. Informally, by *consistent*, it is meant that no pair can have a smaller distance than a pair that we access during the processing of an algorithm over tree access methods [HjS98]. In the case of R-trees, this means that if o_1 and o_2 are objects indexed by the R-trees R_P and R_Q, respectively, and R_1 and R_2 are the MBRs at leaf level that contain o_1 and o_2, respectively, then we must have MINMINDIST(R_1, R_2) ≤ ||(o_1, o_2)||. This constraint is clearly ensured by Lemma 1 *(lower-bounding property)*, the *MBR MINMINDIST property*, and the Euclidean distance properties: non-negativity and triangle inequality. Therefore, since our *MINMINDIST* function applied to R-tree elements is consistent, we can design algorithms based primarily on this distance function that will work correctly.

4 Algorithms for K-Closest Pairs Queries

In the following, based on functions between two MBRs, we present a pruning heuristic and two updating strategies for minimizing the pruning distance during the processing of branch-and-bound algorithms for K-CPQ. After that, three non-incremental branch-and-bound algorithmic approaches (two recursive following a Depth-First searching strategy and one iterative following a Best-First traversal policy) for K-CPQ between objects stored in two R-trees are presented. The unfamiliar reader is advised to study the algorithms presented in [CMT00], as a first reading. The plane-sweep method and the search ordering are used as optimization techniques for improving the naive approaches. Since the R-tree height depends on the number of inserted objects (as well as in the insertion order and the page

size), the two R-trees may have the same or different heights, and we study two alternatives to treat this case. Finally, some extensions of the K-CPQ algorithms are shown.

4.1 Pruning Heuristic and Updating Strategies

Based on the previous bounding functions and lemmas, we propose a pruning heuristic to discard pairs of MBRs, which will not contain the K closest pairs during the execution of the algorithm for reporting the result of K-CPQ. Besides, we present two updating strategies for minimizing the pruning distance z (distance of the K-th closest pair found so far), which are used in the pruning process.

First of all, we establish a data structure that stores the K closest pairs. This data structure will help updating z, which is the distance of the K-th closest pair discovered so far. This structure is organized as a maximum binary heap (called K-heap) and will hold pairs of objects according to their distance. The pair of objects with the largest distance resides in the K-heap root. In the implementation of the branch-and-bound algorithms for K-CPQ we must consider the following cases:

- Initially the K-heap is empty (z is initialized to ∞).
- The pairs of objects discovered at the leaf level are inserted in the K-heap until it gets full (z keeps the value of ∞).
- Then, if the distance of a new pair of objects discovered at the leaf level is smaller than the distance of the pair residing in the K-heap root, then the latter pair is extracted, this new pair is inserted in the K-heap and the root is updated with the pair with the largest distance (z is equal to the distance of the pair of objects residing in the K-heap root).

4.1.1 Pruning Heuristic

In Figure 4.1 two R-tree nodes (dotted rectangles) containing two MBRs (thick-line rectangles) and the *MINMINDIST* (thin lines) and *MINMAXDIST* (dashed lines) distances between each pair of MBRs are depicted. It is obvious that $MINMINDIST(M_{P2}, M_{Q2})$ is the largest one, $MINMINDIST(M_{P1}, M_{Q2})$ and $MINMINDIST(M_{P2}, M_{Q1})$ follow, and $MINMINDIST(M_{P1}, M_{Q1})$ is the smallest one. If, for example $MINMINDIST(M_{P1}, M_{Q2}) > z > MINMINDIST(M_{P2}, M_{Q1})$, the paths corresponding to (M_{P2}, M_{Q2}) and (M_{P1}, M_{Q2}) will be pruned.

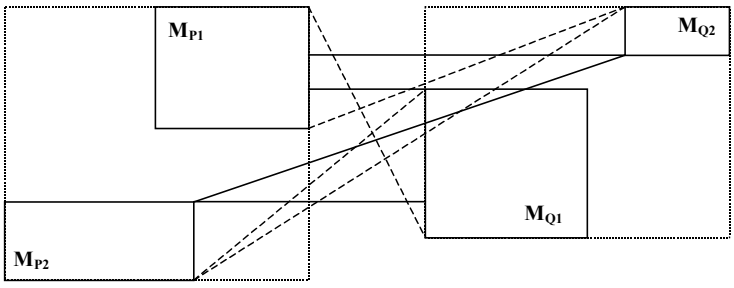


Figure 4.1: Two R-tree nodes and $MINMINDIST(M_{Pi}, M_{Qj})$, $MINMAXDIST(M_{Pi}, M_{Qj})$.

Given two MBRs M_{Pi} and M_{Qj} in $E^{(d)}$, stored in nodes of two R-trees R_P and R_Q, respectively. *If* $MINMINDIST(M_{Pi}, M_{Qj}) > z$, *then the pair* (M_{Pi}, M_{Qj}) *will be discarded*. z can be obtained from the distance of the K-th closest pair among all pairs that have been found so far. Moreover, the z value can be optionally updated using the upper bounding functions $MAXMAXDIST(R_1, R_2)$ and $MINMAXDIST(R_1, R_2)$ for any K and K = 1, respectively. However, if we apply these functions, the number of disk accesses will not be reduced and the computational cost can be increased, as was proved for K-NNQ in [ChF98]. Thereby, we can optionally use the following strategies based on upper bounding functions for updating z (trying to minimize its value, if possible).

4.1.2 Updating Strategy 1 (based on *MINMAXDIST*)

This first updating strategy uses Lemma 3 only for the case of K = 1. That is, given two R-tree nodes N_P and N_Q stored in internal nodes of the R-trees R_P and R_Q, and enclosing two sets of MBRs $\{M_{Pi}: 1 \le i \le |N_P|\}$ and $\{M_{Qj}: 1 \le j \le |N_Q|\}$, respectively. Then, z can be updated if, and only if z' has a smaller value, where z' is defined as follows:

$$z' = \min\left\{MINMAXDIST(M_{Pi}, M_{Qj}) : 1 \le i \le |N_P| \ \ and \ \ 1 \le j \le |N_Q|\right\} \qquad (13)$$

In Figure 4.1, the minimum *MINMAXDIST* (z') is the one of the pair (M_{P1}, M_{Q1}). Suppose that z' is smaller than z, thus z is updated with $MINMAXDIST(M_{P1}, M_{Q1})$. If after this updating strategy we apply the pruning heuristic, then the paths corresponding to (M_{P2}, M_{Q2}) and (M_{P1}, M_{Q2}) will be pruned, because $MINMINDIST(M_{P2}, M_{Q2}) > MINMINDIST(M_{P1}, M_{Q2}) > z$.

4.1.3 Updating Strategy 2 (based on *MAXMAXDIST*)

A second updating strategy uses Lemma 2 for any K. That is, consider two internal R-tree nodes N_P and N_Q of the R-trees R_P and R_Q. N_P and N_Q enclose two sets of MBRs $\{M_{Pi}: 1 \le i \le |N_P|\}$ and $\{M_{Qj}: 1 \le j \le |N_Q|\}$, respectively. Then, z can be updated if, and only if z' has a smaller value, where z' can be obtained by the following procedure:

- MxMxDList is a set of all possible pairs of MBRs (M_{Pi}, M_{Qj}) that can be formed from the two internal nodes N_P and N_Q. $MAXMAXDIST(M_{Pi}, M_{Qj})$ is calculated for each pair of MBRs.
- MxMxDList is sorted in ascending order according to the *MAXMAXDIST* values (creating a sequence of pairs of MBRs with its respective *MAXMAXDIST* value).
- We know from the properties of the R-tree index structure that the minimum number of spatial objects stored on the leaf nodes that can be enclosed by two MBRs (M_{Pi}, M_{Qj}) stored in internal nodes is $X(M_{Pi}, M_{Qj})$, where m_P and m_Q are the minimum fan-outs of R_P and R_Q, respectively.

$$X(M_{Pi}, M_{Qj}) = m_P^{level\ of\ M_{Pi}} \times m_Q^{level\ of\ M_{Qj}}$$

- Using $X(M_{Pi}, M_{Qj})$, we can find the x-th element of the sorted list MxMxDList, until the following condition is satisfied, where Total = $|N_P| \cdot |N_Q|$.

$$\left(\sum_{x=0}^{Total-1} X\left(MxMxDList[x].M_{Pi}, MxMxDList[x].M_{Qj}\right) \right) \geq K$$

- Then, we can obtain z' = MxMxDList[x].*MAXMAXDIST* if (x < Total) is satisfied, otherwise (x = Total) z' = ∞. After that, we will update z with the z' value, if z' < z holds.

The previous procedure for updating z based on *MAXMAXDIST* must be applied locally to two internal nodes in recursive branch-and-bound algorithms following a Depth-First searching strategy.

In Figure 4.2 we have the same two R-tree nodes as in Figure 4.1, where the *MAXMAXDIST* distances between each pair of MBRs are depicted. Also, the sorted list MxMxDList with the value of *MAXMAXDIST* between all possible pairs is illustrated. For example, we suppose the level of M_{Pi} = level of M_{Qj} = 1 (the level just above the leaf level), $m_P = m_Q = 3$, K = 10, and z = 15.35 (at the current moment during the execution of the algorithm). The updating strategy works as follows: x = 0 [enclosedPairs = 9 < 10]; *x = 1* [enclosedPairs = 18 ≥ 10]; z' = MxMxDList[1].MAXMAXDIST = *MAXMAXDIST(M_{P1}, M_{Q2})* = 9.70, and z = 9.70 because z < z' (9.70 < 15.35). After this updating strategy we will apply the pruning heuristic with the new z value.

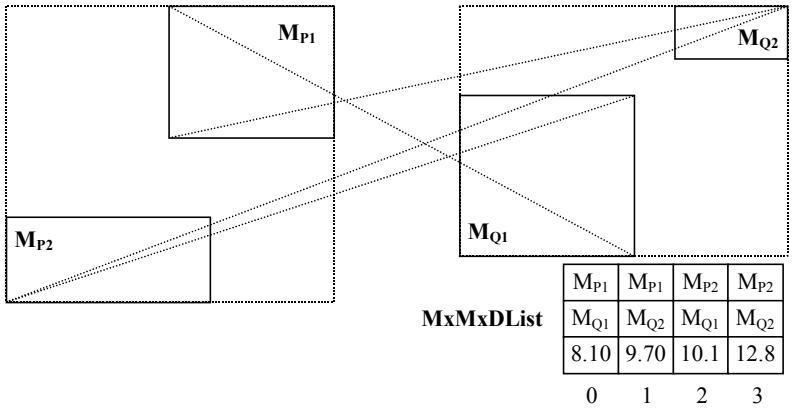


Figure 4.2: Two R-tree nodes and the updating strategy using *MAXMAXDIST(M_{Pi}, M_{Qj})*.

For the iterative branch-and-bound algorithm following a Best-First searching strategy, the global set of pairs of MBRs that take part in the above procedure for computing z' is the current set of pairs (set of all possible pairs of MBRs (M_{Pi}, M_{Qj}) that can be formed from the current two internal nodes N_P and N_Q) plus the pairs of MBRs already inserted in the main minimum binary heap. In this case, we will have a maximum binary heap, MxMxDHeap, with *MAXMAXDIST* as a key that stores globally all pairs of MBRs for which ($\Sigma(X(M_P, M_Q)_i)$) is smaller than or equal to K, and a hash table associated to this data structure to support locating a particular pair, as in [HjS98]. The procedure to update MxMxDHeap and z is very similar to the previous one for MxMxDList:

- When a candidate pair of MBRs (M_P, M_Q) is inserted in the main minimum binary heap, it is also inserted in MxMxDHeap. If this insertion causes the sum ($\Sigma(X(M_P, M_Q)_i)$) of the minimum number of objects stored in the leaves that can be generated by all pairs of MBRs stored in MxMxDHeap

be larger than K, then we remove pairs of MBRs from MxMxDHeap until this sum is smaller than or equal to K, setting z' to the *MAXMAXDIST* value of the last removed pair.

- When a candidate pair of MBRs (M_P, M_Q) is removed from the main minimum binary heap, it must also be removed from MxMxDHeap, if it is present.
- Then, we can update z with the z' value, if z' < z holds.

After presenting these two updating strategies for minimizing the pruning distance (z), we must emphasize that their use is optional (controlled by a parameter passed to the algorithms), since their computational cost is greater than the performance gain.

4.2 The Sorted Distances Recursive Algorithm

This first branch-and-bound algorithm follows a Depth-First searching strategy and makes use of recursion and the previous pruning heuristic and updating strategies. In addition, we employ the property that pairs of MBRs that have smaller *MINMINDIST* are more likely to contain the K closest pairs and to lead to a smaller z value. A heuristic that aims at improving this branch-and-bound algorithm when two internal nodes are accessed, is to sort the pairs of MBRs according to ascending order of *MINMINDIST* and to obey this order in propagating downwards recursively. This order of processing is expected to improve pruning of paths. Such an algorithm (SDR) for two R-trees with the same height appears in Figure 4.3.

We point out that at the R-tree leaf level an object (point or MBR) or MBR of another type of objects can be stored, together with a pointer to its exact geometry kept outside of the R-tree, e.g. in a sequential file. In the first case, we will calculate *MINMINDIST*, since this function returns the distance between two points if the two MBRs have degenerated to two points as shown in the *MINMINDIST* property. In the second case, we must read the exact geometry of the pair of objects (O_1, O_2) and calculate its distance ObjectDistance(O_1, O_2), using techniques presented in [ChW84, GJK88].

In the example of Figure 4.1, the order of paths that will be followed is: (M_{P1}, M_{Q1}), i.e. the one with the smallest *MINMINDIST* and then (M_{P2}, M_{Q1}). In such a case, there may be ties between the *MINMINDIST* values. This is likely to happen especially when the two datasets overlap. In that case, *MINMINDIST* will usually be 0. It is possible to get a further improvement by choosing the next pair in case of a tie using some heuristic (not following the order produced by the sorting method). In [CMT00] various such heuristics have been proposed and experimentally studied. Now, we will ignore this special treatment, since it does not significantly affect the behavior of the branch-and-bound algorithm with respect to the I/O activity, and it needs computational time to be carried out.

4.3 The Plane-Sweep Recursive Algorithm

Another improvement for a branch-and-bound algorithm making use of recursion (Depth-First traversal) is to exploit the R-tree structure utilizing the plane-sweep technique, which is a common technique for computing intersections [PrS85]. The basic idea is to move a line, the so-called sweep-line, perpendicular to one of the dimensions, e.g. X dimension, from left to right. We apply this technique

SDR1 Start from the two R-trees roots and set z to ∞.

SDR2 If you access two internal nodes, optionally try to minimize z using one of the two updating strategies (based on *MINMAXDIST* for K = 1 or *MAXMAXDIST* for any K). Calculate *MINMINDIST* for each possible pair of MBRs and sort these pairs in ascending order of *MINMINDIST*. Following this order, propagate downwards recursively only for the pairs of entries having *MINMINDIST* ≤ z.

SDR3 If you access two leaves, then calculate the distance of each possible pair of objects. If this distance is smaller than or equal to z (the distance of the K-th closest pair discovered so far), then remove the pair located in the K-heap root and insert the new pair in K-heap, updating z.

Figure 4.3: The SDR Algorithm

for restricting all possible combinations of pairs of MBRs from two R-tree nodes $N_P = \{M_{Pi}: 1 \le I \le |N_P|\}$ and $N_Q = \{M_{Qj}: 1 \le j \le |N_Q|\}$ from R_P and R_Q, respectively. If we do not use this technique, then we must create a set with all possible combinations of pairs of MBRs from two R-tree nodes ($|N_P| \cdot |N_Q|$) and process it as in the previous recursive algorithm.

In general, the technique consists of sorting the entries of the two current R-tree nodes, based on the coordinates of one of the corners of the MBRs (e.g. lower left corner) in increasing or decreasing order. First, the dimension for the sweep-line (e.g. Sweeping_Dimension = 0 or X-axis) is established based on sweeping axis criteria [SML00]. After that, two pointers are maintained initially pointing to the first entry of each sorted R-tree node. Let **Pivot** be the entry of the smallest value of the MBR with lower left corner pointed by one of these two pointers, e.g. M_{P1}, then **Pivot** is initialized to the entry associated to the MBR M_{P1}. The MBR of the pivot must be paired up with the MBRs of the entries stored in the other R-tree node $\{M_{Qj}: 1 \le j \le |N_Q|\}$ from left to right that satisfies the *MINMINDIST(**Pivot.MBR**, M_{Qj}, Sweeping_Dimension)* ≤ z, obtaining a set of entries for candidate pairs where the element **Pivot.MBR** is fixed. This partial set with respect to the MBR of the pivot entry will be added to a global set of candidate pairs of entries, called ENTRIES (empty at the beginning). After all possible pairs of entries that contain **Pivot.MBR** have been found, the pointer of the pivot node is increases to the next entry, **Pivot** is updated with the entry of the next smallest value of a lower left corner of MBRs pointed by one of the two pointers, and the process is repeated.

PSR1 Start from the two R-tree roots and set z to ∞.

PSR2 If you access two internal nodes, optionally try to minimize z using one of the two updating strategies (based on *MINMAXDIST* for K = 1 or *MAXMAXDIST* for any K). Apply the plane-sweep technique to obtain the set of pairs of candidate entries, ENTRIES. Propagate downwards recursively only for those pairs of entries from ENTRIES having *MINMINDIST* ≤ z.

PSR3 If you access two leaves, apply the plane-sweep technique to obtain the set of candidate pairs of entries (ENTRIES). Then calculate the distance of each pair of objects stored in ENTRIES. If this distance is smaller than or equal to z, then remove the pair located in the K-heap root and insert the new pair in K-heap, updating z.

Figure 4.4: The PSR Algorithm.

Notice that we apply *MINMINDIST*(M_{Pi}, M_{Qj}, *Sweeping_Dimension)* because in the plane-sweep technique, the sweep is only over one dimension (the best dimension according to the criteria suggested in [SML00]). Moreover, the search is only restricted to the closest MBRs with respect to the MBR of the pivot entry according to the current z value. No duplicated pairs are obtained, since the MBRs are always checked over sorted R-tree nodes. Also, the application of this technique can be viewed as a *sliding window* on the sweeping dimension with a width equal to the z value starting in the MBR of the pivot, where we only choose all possible pairs of MBRs that can be formed using the MBR of the pivot and the other MBRs from the remainder entries of the other R-tree node that fall into the current sliding window. We must point out that this sliding window has a length equal to z value plus the length of the MBR of the pivot on the sweeping dimension.

The PSR algorithm applies the plane-sweep technique for obtaining a reduced set of candidate pairs of entries from two R-tree nodes (ENTRIES) and it can be improved by sorting its pairs of MBRs according to ascending order of *MINMINDIST* or organizing ENTRIES as a minimum binary heap with *MINMINDIST* as a key. Then, it iterates in the set ENTRIES and propagates downwards only for the pairs of entries with *MINMINDIST* smaller than or equal to the z value. The PSR algorithm for two R-trees with the same height appears in Figure 4.4.

In the example of Figure 4.1, suppose that we do not apply the updating strategies for reducing z and z = 2. Then we apply the plane-sweep technique taking the sweeping dimension the X-axis. The set of pairs of MBRs produced is ENTRIES = {(M_{P1}, M_{Q1})}. We calculate *MINMINDIST(M_{P1}, M_{Q1})*, which is smaller than z. Thus, we propagate only for (M_{P1} ,M_{Q1}).

PSI1 Start from the two R-tree roots, set z to ∞ and initialize Main-heap.

PSI2 If you access two internal nodes, optionally try to minimize z using one of the two updating strategies (based on *MINMAXDIST* for K = 1 or *MAXMAXDIST* for any K). Apply the plane-sweep technique to obtain the set of candidate pairs of entries, ENTRIES. Insert into Main-heap only those pairs of addresses of entries stored in the current two internal R-tree nodes (and the *MINMINDIST* value of their MBRs), which have a *MINMINDIST* value smaller than or equal to z.

PSI3 If you access two leaves, apply the plane-sweep technique to obtain the set of candidate pairs of entries (ENTRIES). Then calculate the distance of each pair of objects stored in ENTRIES. If this distance is smaller than or equal to the z value, then remove the pair located in the K-heap root and insert the new pair in K-heap, updating z.

PSI4 If Main-heap is empty then stop.

PSI5 Get the pair from the Main-heap root. If this item has *MINMINDIST* > z then stop. Otherwise, repeat the algorithm from **PSI2** for the pair of R-tree nodes pointed by the addresses of this Main-heap item.

Figure 4.5: The PSI Algorithm.

4.4 The Plane-Sweep Iterative Algorithm

Unlike the previous ones, this branch-and-bound algorithm is iterative. In order to overcome recursion and to keep track of propagation downwards while accessing the two R-trees, a minimum binary heap, called Main-heap, is used. Main-heap holds only pairs of addresses pointing two R-tree nodes that will be processed during the execution of the algorithm and the *MINMINDIST* value of the pair of MBRs that encloses these two R-tree nodes. That is, the item structure for Main-heap is <*MINMINDIST*, $NodeAddressR_P$, $NodeAddressR_Q$>, and it allows us to store this data structure entirely in main memory even for a large K value or large datasets. The pair with the smallest *MINMINDIST* value resides on top of Main-heap (in the root of the minimum binary heap). This pair is the next candidate for processing. Also, we can apply the plane-sweep technique in this branch-and-bound iterative algorithm in the same way as in the recursive one. Such and algorithm (PSI) for two R-trees with the same height appears in Figure 4.5.

Note that ties between *MINMINDIST* values may also appear as in the sorted recursive algorithm. That is, two or more pairs may have the same *MINMINDIST* value. If this value is the minimum one, then more than one such pairs would appear close to the Main-heap root. As in the sorted recursive algorithm, we will ignore this special treatment, since it does not significantly affect the behavior of the branch-and-bound algorithm with respect to the I/O activity and it consumes computational time.

In the example of Figure 4.1, suppose that we consider the same situation (internal nodes). The set of pairs of MBRs produced by the application of plane-sweep technique is ENTRIES = $\{(M_{P1}, M_{Q1})\}$. We calculate $MINMINDIST(M_{P1}, M_{Q1})$, which is smaller than z. Thus, we insert in Main-heap only the entry $<MINMINDIST(M_{P1}, M_{Q1})$, entryOf$M_{P1}$.address, entryOf$M_{Q1}$.address >.

4.5 Treatment of Different Heights

When the two R-trees storing have different heights, the algorithms are slightly more complicated. In the recursive branch-and-bound algorithm, there are two approaches for treating different heights:

- The first approach is called “fix-at-root”. The idea is, when the algorithm is called with a pair of internal nodes at different levels, stop propagating downwards in the R-tree of the smaller level node, while propagation in the other R-tree continues until both nodes are located at the same level. Then, propagation continues in both subtrees as usual.
- The second approach is called “fix-at-leaves” and works in the opposite way. Recursion propagates downwards as usual. When the algorithm is called with a leaf on the one hand and an internal node on the other hand, downwards propagation stops in the R-tree of the leaf, while propagation in the other R-tree continues as usual.

The iterative algorithm can also be modified to deal with different heights by the “fix-at-leaves”, or the “fix-at-root” strategy. The only difference is that the recursive call is replaced by an insertion in the Main-heap.

The necessary modifications for applying these techniques of treating R-trees with different heights in recursive and iterative algorithms are presented in [CMT00], along with experimental results on the performance behavior of each approach.

4.6 Extending the K-CPQ Algorithms

Numerous operations can be extended from the branch-and-bound algorithms for K-CPQ. The K-Self-CPQ, Semi-CPQ, the K Farthest Pairs Query, and obtaining K or all closest pairs of objects with the distances within a range [Dist_Min, Dist_Max] ($0 \leq$ Dist_Min $\leq$ Dist_Max) are the more representative ones. Next, we will present these operations and the modifications in our branch-and-bound algorithms in order to carry them out.

4.6.1 K-Self-CPQ

A special case of K-CPQ is the called "K-Self-CPQ" where both datasets actually refer to the same entity. That is, the input dataset is joined with itself. Taking into account the K-CPQ definition, the result set of the K-Self-CPQ is given by the following expression:

$$\textbf{K-Self-CPQ(P, K)} = \{\text{K-CPQ(P, Q, K): } Q \equiv P\}$$

As an example from operational research, we may need to find the K pairs of facilities (hospitals, schools, etc.) that are closer than others in order to make a reallocation. In the terminology of Section 3, P and Q are identical datasets, and hence their entries are indexed in a single R-tree. The algorithms proposed in this paper are able to support this special case with only two slight modifications that correspond to necessary conditions on candidate results (p_i, p_j):

- (p_i, p_j) can be included in the result set if, and only if $i \neq j$ and
- (p_i, p_j) can be included in the result set if, and only if (p_j, p_i) is not already in the K-heap.

To improve the performance of the branch-and-bound algorithm for this query with respect to the candidate pairs in the result, we have included a hash table associated to the K-heap for testing whether the same or the symmetric of a given pair is already stored in K-heap or not.

4.6.2 Semi-CPQ

Another special case of closest pairs query is called "Semi-CPQ" ("distance semi-join" in [HjS98]). In Semi-CPQ for each object of the first dataset, the closest object of the second dataset is computed. The result set of Semi-CPQ is a sequence of pairs of objects given by the following definition:

Definition. Let two point sets, $P = \{p_1, p_2, \ldots, p_{NP}\}$ and $Q = \{q_1, q_2, \ldots, q_{NQ}\}$ in $E^{(d)}$. Then, the result of the semi closest pairs query, Semi-CPQ(P, Q), is a set of ordered sequences of |P| different pairs of points of P×Q, where each object in P forms a pair with its closest object (or one of its closest objects, if there is not only one such object) in Q:

$$\textbf{Semi-CPQ(P, Q)} = \{((p_1, q_1), (p_2, q_2), \ldots, (p_{|P|}, q_{|P|})), p_1, p_2, \ldots, p_{|P|} \in P, q_1, q_2, \ldots, q_{|P|} \in Q:$$
$$p_i \neq p_j, i \neq j, 1 \leq i, j \leq |P| \text{ and } \text{dist}(p_{|P|}, q_{|P|}) \geq \text{dist}(p_{|P|-1}, q_{|P|-1}) \geq \ldots \geq \text{dist}(p_1, q_1) \text{ and}$$
$$\forall (p_i, q_j) \in P \times Q - \{(p_1, q_1), (p_2, q_2), \ldots, (p_{|P|}, q_{|P|})\}, \text{dist}(p_i, q_j) \geq \text{dist}(p_i, q_i), 1 \leq i \leq |P|\}$$

Note that, due to ties of distances, the result of the Semi-CPQ may not be a unique ordered sequence for a specific pair of point sets P and Q. Our aim is to find one of the possible instances, although it would be straightforward to obtain all of them. The Semi-CPQ works by reporting a sequence of pairs of objects (p_i, q_i) in order of distance. Note that once we have determined the closest object q_i to a particular p_i, that p_i does not participate in other pairs. Unlike most join operations, the Semi-CPQ is not commutative, i.e. *Semi-CPQ(P, Q) ≠ Semi-CPQ(Q, P)*.

To implement this operation, we have transformed the recursive and iterative branch-and-bound algorithms for answering Semi-CPQ. These versions are similar to those proposed in [HjS98]. In the first version, called "GlobalObjects", we maintain a global list of objects belonging to leaves of the first R-tree. Each object is accompanied by the minimum distance to all the objects of the second R-tree visited so far. In the second version, called "GlobalAll", we maintain an analogous global list of objects. Moreover, we keep another global list of MBRs of the first R-tree, where each MBR is accompanied by the minimum *MINMAXDIST* value to all the MBRs of the second R-tree visited so far.

The Self-Semi-CPQ is an operation derived from Self-CPQ and Semi-CPQ, which, for one dataset, finds for each object its nearest neighbor. This operation is a Semi-CPQ where the input dataset is combined with itself: **Self-Semi-CPQ(P)** = {Semi-CPQ(P, Q): Q ≡ P }

The implementation of this operation is just a combination of the transformations of the Semi-CPQ with the constraint of Self-CPQ.

4.6.3 K-Farthest Pairs Query

In the same sense that we have defined the K-CPQ, it can be easily extended to find the K farthest pairs of objects from two datasets. The result set of the K-Farthest Pairs Query (K-FPQ) is given by the following definition.

Definition. Let two subsets of $E^{(d)}$, $P = \{p_1, p_2, \ldots, p_{NP}\}$ and $Q = \{q_1, q_2, \ldots, q_{NQ}\}$. The result of the K farthest pairs query K-FPQ(P, Q, K) is a set of ordered sequences of K ($1 \le K \le |P| \cdot |Q|$) different pairs of objects of P×Q, with the K largest distances between all possible pairs of objects that can be formed by choosing one object of P and one object of Q:

$$\textbf{K-FPQ(P, Q, K)} = \{((p_1, q_1), (p_2, q_2), \ldots, (p_K, q_K)), p_1, p_2, \ldots, p_K \in P, q_1, q_2, \ldots, q_K \in Q:$$
$$(p_i, q_i) \neq (p_j, q_j), i \neq j, 1 \le i, j \le K \text{ and } \forall (p_i, q_j) \in P \times Q - \{(p_1, q_1), (p_2, q_2), \ldots, (p_K, q_K)\},$$
$$dist(p_i, q_j) \le dist(p_K, q_K) \le dist(p_{K-1}, q_{K-1}) \le \ldots \le dist(p_2, q_2) \le dist(p_1, q_1)\}$$

In this case and in order to design a branch-and-bound recursive algorithm for solving K-FPQ by extending the K-CPQ algorithms, we take into consideration the following constraints:

(1) K-heap is organized as a minimum binary heap with *MAXMAXDIST* as a key. In this case, z is the distance value of the K-th farthest pair discovered so far and stored in K-heap (z = 0).

(2) If two internal nodes are accessed, *MAXMAXDIST* for each possible pair of MBRs is calculated and these pairs are sorted in decreasing order of *MAXMAXDIST*. Following this order, we propagate downwards recursively only for those pairs of entries that have *MAXMAXDIST* ≥ z.

(3) If two leaves are accessed, then the distance of each possible pair of objects is calculated. If this distance is larger than or equal to z, then the pair located in the K-heap root is removed and the new pair is inserted in K-heap, updating z.

Along the same lines, we extend the iterative branch-and-bound algorithm for K-CPQ to obtain one for K-FPQ. In this case, we only consider the following conditions:

(1) Main-heap is organized as a maximum binary heap and K-heap is organized as a minimum binary heap, with *MAXMAXDIST* as a key in both cases. Moreover, z is the distance of the K-th farthest pair discovered so far and stored in K-heap (z = 0).
(2) If two internal nodes are accessed, *MAXMAXDIST* for each possible pair of MBRs is calculated and the pairs of addresses of R-tree nodes (together with *MAXMAXDIST*), the MBRs of which have a *MAXMAXDIST* value larger than or equal to z, are inserted into Main heap.
(3) If two leaves are accessed, then the distance of each possible pair of objects is calculated. If this distance is larger than or equal to z, then the pair located in the K-heap root is removed and the new pair is inserted in K-heap, updating z.

4.6.4 Obtaining K- or All-Closest Pairs of objects with their distances within a range

The proposed algorithms can be also extended for obtaining the K closest pairs of objects with distances within a range, [Dist_Min, Dist_Max] ($0 \leq$ Dist_Min $\leq$ Dist_Max). This user-defined range determines the minimum and maximum desired distance for the query result. The necessary modifications of the branch-and-bound algorithms are the following:

(1) If two internal nodes are accessed, do not update z (by updating strategies based on *MINMAXDIST* or *MAXMAXDIST*). Calculate *MINMINDIST* for each possible pair of MBRs and recursively propagate downwards only for those pairs of MBRs with *MINMINDIST* $\leq$ Dist_Max.
(2) If two leaves are accessed, calculate the distance of each possible pair of objects. If this distance is in the range [Dist_Min, Dist_Max], insert the new pair in the K-heap and do not update z. If K-heap becomes full, remove the K-th closest pair (in the K-heap root) and insert the new one, updating the K-heap structure.

On the other hand, one may wish to obtain all possible pairs of objects with the distances within the interval [Dist_Min, Dist_Max]. In this case, neither K nor the K-heap size are known a priori and Dist_Max is the bound distance for the pruning heuristic. Apparently, when Dist_Max = ∞, our branch-and-bound algorithms degenerate in backtracking ones (obtaining all possible feasible solutions of a given problem), as when $K \geq |P| \cdot |Q|$, $|P|$ and $|Q|$ being the numbers of the objects stored in the R-trees R_P and R_Q, respectively. The modifications in the algorithms for this variant are the same to the previous ones, with only one difference: the management of the K-heap. In the worst case, the K-heap can grow as large as the product of all objects belonging to the two R-trees. That is, the size of K-heap can reach $|P| \cdot |Q|$ elements. Thus, it is not always feasible to store the K-heap in main memory, and we must use a hybrid memory / disk scheme and techniques based on range partitioning, as in [HjS98, SML00].

5 Experimental Results

This section provides the results of an extensive experimentation study aiming at measuring and evaluating the efficiency of the three K-CPQ algorithms proposed in Section 4, namely the Sorted Distances Recursive (SDR), the Plane-Sweep Recursive (PSR) and the Plane-Sweep Iterative (PSI) algorithms. In our experiments we used the R*-tree [BKS90] as the underlying disk-resident access method. In order to evaluate our branch-and-bound algorithms for K-CPQ we have taken into account several performance metrics. The effect of buffering and results over disjoint datasets are also studied, since these two parameters have an important influence on this kind of distance-based query. Moreover, we have adapted our K-CPQ algorithms to execute its more representative extensions: K-Self-CPQ, Semi-CPQ and K-FPQ. Finally, in this experimental section we have included studies on the scalability of the algorithms with varying the dataset sizes and K.

5.1 Experimental Settings

All experiments were run on an Intel/Linux workstation with 128 Mbytes RAM and several Gbytes of secondary storage. The programs were created using the GNU C++ compiler with maximum optimization (-O3). The page size was set to 4 Kbytes, resulting to an R*-tree node capacity M = 204; minimum capacity was set to $m = \lfloor M*0.4 \rfloor = 81$ since this m value yields the best performance according to [BKS90]. Moreover, the binary heaps (Main-heap and MxMxDHeap optionally) for the iterative algorithm were stored completely in main memory as well as the K-heap for the result.

In order to evaluate K-CPQ algorithms, we have used real datasets from [DCW97], performing new experiments and using different datasets with respect to [CMT00]. The particular datasets represented populated places (points), rail-roads (line segments), roads (line segments) and cultural landmarks (points) from the United States of America, Canada and Mexico with different cardinalities as shown in Table 5.1. Just as an indication, four of them are illustrated in Figure 5.1.

We have measured the performance of our K-CPQ algorithms based on the following five performance metrics to compare the algorithms in different aspects such as CPU cost and I/O activity.

(1) *Number of disk accesses*. It is the most representative parameter to measure the I/O activity, using or not additional buffers. The number of R*-tree nodes fetched from disk is reported as the number of disk accesses, and it may not exactly correspond to actual disk I/O, since R*-tree nodes can be found in the system buffers.

(2) *Response time.* Total query response times were measured for overall performance of the K-CPQ algorithms. The execution time is reported in seconds and represents the overall CPU time consumed, as well as the total I/O performed by the algorithms for this kind of distance join operation.

(3) *Number of distance computations*. The cost of computing distances between pairs of MBRs *(MINMINDIST)* and objects (line-line, line-point and point-point) constitutes a significant portion of the

	Cultural Landmarks	Populated Places	Rail-Roads	Roads
Canada	2,099	4,994	35,074	121,416
Mexico	1,087	4,293	10,060	92,392
USA	6,017	15,206	146,503	355,312
North America	9,203	24,493	191,637	569,120

Table 5.1. Cardinalities of the real datasets.

Figure 5.1: Four real-world datasets from [DCW97]: (a) roads of USA, (b) rail-roads of North America, (c) cultural landmarks of USA and (d) populated places of North America.

computational cost for this kind query. Thus, the total number of distance computations required by a K-CPQ algorithm provides a direct indication of its computational performance.

(4) *Number of subproblems created by decomposition (simply referred to as number of subproblems).* It is another important performance metric related to the query cost. It represents the number of pairs of MBRs created by decomposition before the algorithm termination and provides the number of partial subproblems considered during the algorithm execution. Thus, by minimizing this parameter we obtain the algorithm with the lowest computational cost.

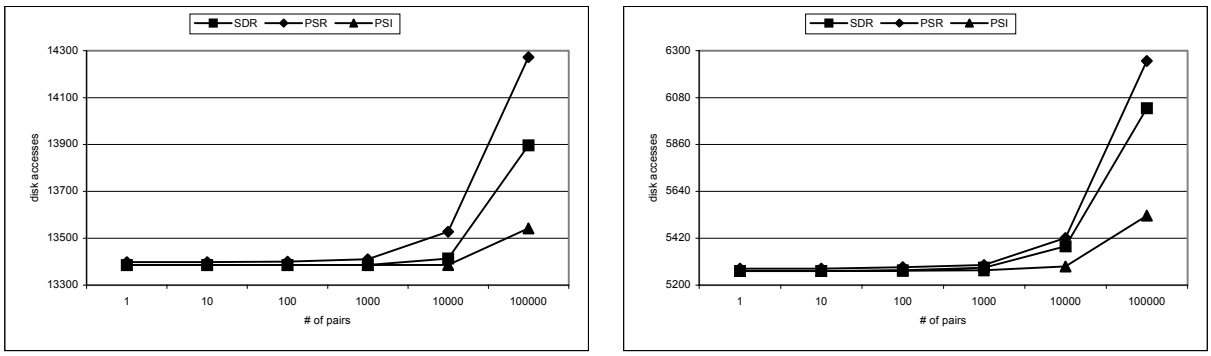
(5) *Number of insertions in the Main-heap for the iterative algorithm* (PSI). The task of managing the main binary heap (Main-heap) is largely CPU intensive as its size increases. Thus, the total number of insertions to the main binary required by the K-CPQ iterative algorithm provides a reasonable indication of its activity, since insertions are much more frequent than deletions.

5.2 Performance Comparison of K-Closest Pairs Query Algorithms

We proceed with the evaluation of the three algorithms for K-CPQ (SDR, PSR and PSI) as a function of K that varies from 1 to 100000, assuming zero buffer and obviously for the same workspaces. Fig-

ure 5.2.a illustrates the number of disk accesses for K-CPQ over the (USrr, USrd) configuration, where USrr and USrd are the rail-roads and roads of USA, respectively. On the other hand, Figure 5.2.b shows the same metric for the (NArr, NApp) configuration, where NArr and NApp are the rail-roads and populated places of North America, respectively. For this last configuration and in the sequel, when the R*-trees have different heights we will use the *fix-at-leaves* technique.

Figure 5.2 shows that the number of R*-tree nodes fetched from disk (I/O activity) of each algorithm gets higher as K increases, and PSI is better than the recursive alternatives in both configurations with similar I/O trends. Moreover, the deterioration is not smooth; after a threshold the cost increases slightly for large K values (this threshold was usually around K = 1000). This demonstrates that the iterative algorithm was more effective than the recursive ones in the pruning process in the absence of buffers, since it follows a Best-First searching strategy optimized with the plane-sweep technique.



(a) (b)

Figure 5.2: Comparison of the K-CPQ algorithms in terms of the number disk accesses without buffering and varying K for (a) (Usrr, USrd) and (b) (NArr, NApp) configurations.

For the (USrr, USrd) configuration, Table 5.2 compares the remaining performance parameters, i.e. total response time **(bold)**, number of distance computations *(italic)*, number of subproblems (regular) and the amount of Main-heap insertions (in parentheses) needed by each algorithm. For all K values, the plane-sweep technique needed a significantly reduced number of distance computations; this implies that the required response time was also considerably smaller than SDR (it does not use this optimization technique). This demonstrates that the plane-sweep method was very effective for this kind of distance-based query, since the number of possible pairs from the combination of two R*-tree nodes is also reduced considerably, as well as the number of insertion in the Main-heap. For instance, for small K values ($K \leq 1000$) PSR was slightly faster, and for large K values ($K \geq 10000$) the best was PSI. The explanation of this behavior is due to the fact that the recursive alternative traverses the R*-trees using a Depth-First searching strategy and it can deviate to the branches where no optimal solutions are located. Moreover, PSI is the algorithm with the minimum number of subproblems, since it follows a Best-First traversal. On the other hand, SDR was the worst alternative, because it combines all possible entries from two R*-tree nodes (depending on the fan-out (m, M) of the R*-trees, we have

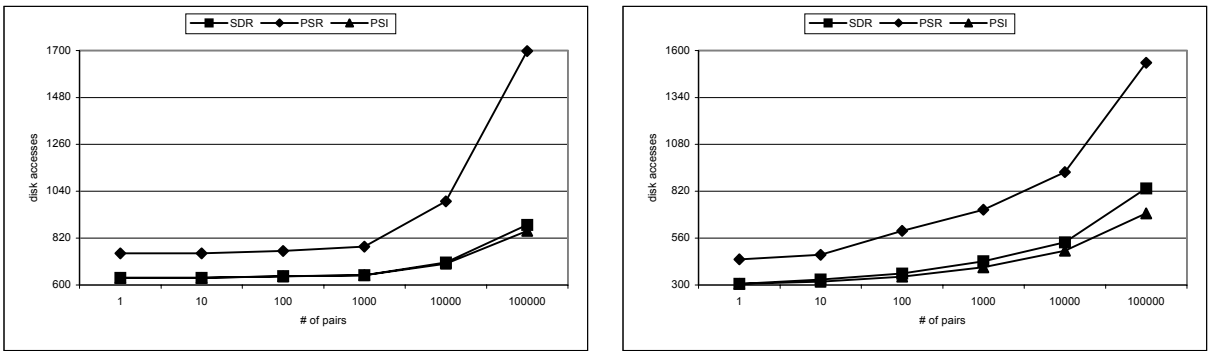
a list from 6561 to 41616 number of pairs), calculates its minimum distances, sorts them when they are internal; all these tasks consume significant CPU time.

	K=1	K=10	K=100	K=1000	K=10000	K=100000
SDR	**613.86** *140307590* 6692	**614.23** *140307590* 6692	**614.31** *140307590* 6692	**614.36** *140307590* 6692	**616.52** *140618716* 6706	**644.80** *145538868* 6947
PSR	**20.02** *3164690* 6698	**20.05** *3167352* 6698	**20.10** *3180129* 6699	**20.49** *3271461* 6704	**25.55** *4002726* 6763	**69.32** *9513814* 7135
PSI	**20.28** *3334834* 6692 (187022)	**20.30** *3336670* 6692 (187022)	**20.35** *3341956* 6692 (187022)	**20.59** *3391305* 6692 (187030)	**23.80** *3905617* 6692 (187133)	**48.65** *7454867* 6770 (187743)

Table 5.2: Comparison of the K-CPQ algorithms without buffering and varying K for the (USrr, USrd).

5.3 Results on Disjoint Datasets

In [CMT00] the effect of overlap between the datasets for K-CPQ was studied. In the absence of buffers, the conclusion was that: *the greater percentage of overlapping, the better performance of the iterative algorithm with respect to the recursive ones*. In order to verify this behavior, we performed experiments with datasets corresponding to disjoint workspaces. Figure 5.3.a illustrates the number of disk accesses for K-CPQ over the (MXrd, USrr) configuration, where MXrd are the roads of Mexico. On the other hand, Figure 5.3.b shows the same metric for the (CDrr, USpp) configuration, where CDrr are the rail-roads of Canada.



(a) (b)

Figure 5.3: Comparison of the K-CPQ algorithms in terms of the number of disk accesses without buffering and varying K for disjoint workspaces (a) (MXrd, USrr) and (b) (CDrr, USpp) configurations.

Figure 5.3 shows, like Figure 5.2, that the PSI performance is comparable to PSR (K must be really large to observe a slight outperformance of PSI with respect to PSR) without buffers for disjoint or overlapped workspaces, although the cost is notably smaller for disjoint datasets. Evidently, the algorithms are cheaper for disjoint workspaces than for overlapping ones, since the *MINMINDIST* values are large enough for disjoint datasets and the pruning is much more effective.

For the (MXrd, USrr) configuration, Table 5.3 compares the other performance metrics. For all K values and for all performance metrics, PSI outperforms SDR and PSR, proving that the iterative algorithms work better than the recursive ones in absence of buffers. For instance, if we consider the total response time consumed by the algorithms as the metric under consideration, PSI is on the average 90% and 80% faster than SDR and PSR, respectively.

	K=1	K=10	K=100	K=1000	K=10000	K=100000
SDR	**21.72** *4994760* 316	**21.74** *4994760* 316	**22.05** *5063198* 320	**22.32** *5121658* 322	**25.01** *5684082* 352	**34.46** *7312475* 440
PSR	**3.65** *703426* 373	**3.71** *712269* 373	**4.15** *782245* 379	**5.67** *968723* 389	**18.14** *2754905* 495	**77.76** *8890146* 848
PSI	**0.50** *115127* 316 (71116)	**0.56** *126827* 316 (71116)	**0.76** *160575* 319 (71116)	**1.37** *259818* 322 (71116)	**3.42** *571547* 349 (71123)	**12.70** *1658221* 426 (71139)

Table 5.3: Comparison of the K-CPQ algorithms without buffering and varying K for the (MXrd, USrr).

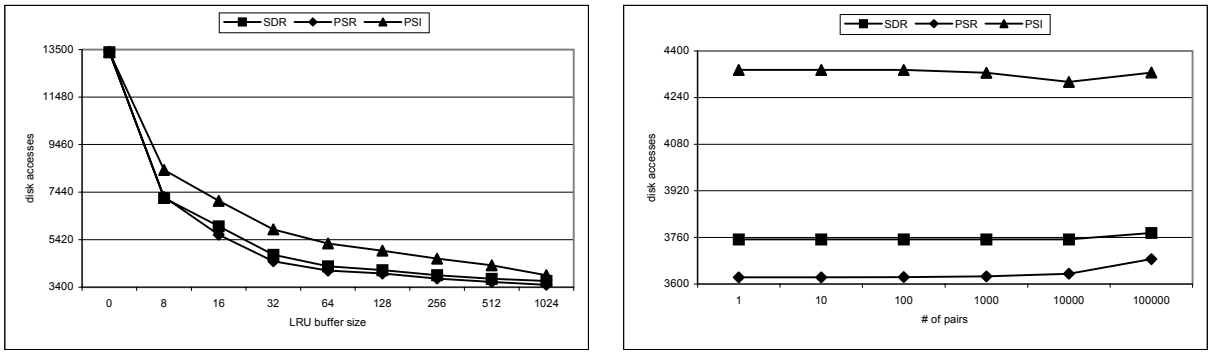
5.4 The Effect of Buffering

DBMS performance is sensitive to the size of buffers in main memory. There exist two basic research directions that aim at reducing the disk activity and enhancing the system throughput during query processing using buffers. The first one focuses on the availability of buffer pages at runtime by adapting memory management techniques for buffer managers used in operating systems to database systems [EfH84]. The second one focuses on query access patterns, where the query optimizer dictates the query execution plan to the buffer manager, so that the latter can allocate and manage its buffers accordingly [ChD85, COL92].

To speed up query processing, DBMSs use indices that may partially reside in main memory buffers. The buffering effect should be studied, since even a small number of buffer pages can drastically improve the overall performance. In DBMSs, the buffer manager is responsible for operations in the buffer pool, including buffer space assignment to queries, replacement decisions and buffer reads and writes in the event of page faults. When buffer space is available, the manager decides about the number of pages that are allocated to an activated query. This decision may depend on the availability of pages at runtime (page replacement algorithms), or the access pattern of queries (nature of the query). Following the former criterion, in [CVM01] several buffer pool structures, page replacement policies and buffering schemes for K-CPQ algorithms were analyzed, aiming at reducing the number of disk accesses. For the experiments of this section, we will adopt the best configuration for this kind of distance-based query that was proposed in [CVM01]: *LRU with a single buffer pool structure, using a global buffering scheme*.

For the experiments of this subsection, we are going to consider the workspace configuration (USrr, USrd) with different buffer sizes, B, varying from 0 to 1024 pages. This means that we have in memory a variable percentage of R*-tree nodes, depending on the number of buffer pages. Besides, the buffer does not use any global optimization criterion, i.e. the buffer pages are handled as the algorithms are required, depending on which R*-tree are located.

Figure 5.4.a shows that PSI presents an average excess of I/O activity around 14% and 18% for K = 1000 with respect to SDR and PSR, respectively, as can be noticed by the gap between the lines. Moreover, the influence of buffer is slightly greater for PSR than for SDR, due to the use of the plane-sweep technique. This behavior is due to the fact that recursion favors the most recently used pages (LRU) in the backtracking phase and this effect is preserved in case of large buffers. On the other hand, Figure 5.4.b illustrates that the gap for K-CPQ algorithms remains when the K value is incremented and B = 512 pages. For instance, the average I/O saving between PSR and SDR with increasing K (1 … 100000) is 3%, and PSR with respect to PSI is 16%. Again, this effect is due to the combination of recursion and LRU page replacement policy.

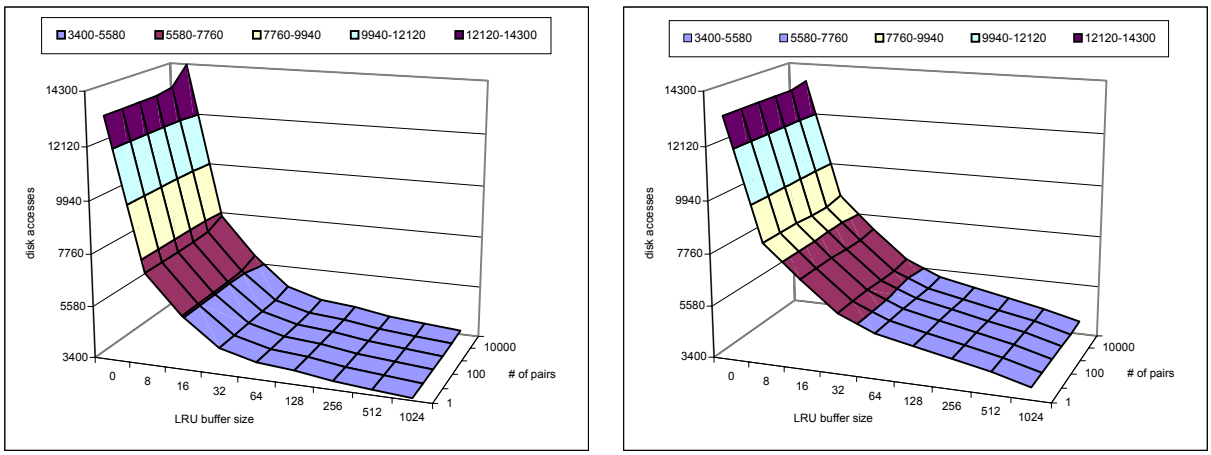


(a) (b)

Figure 5.4: Comparison of the K-CPQ algorithms in terms of the number of disk accesses using the (USrr, USrd) configuration: (a) varying the buffer size and K = 1000, (b) varying K and B = 512 pages.

Figure 5.5 illustrates the performance of the best K-CPQ recursive (PSR) and the iterative (PSI) algorithms as a function of buffer size (B ≥ 0). For PSR, when B ≥ 64, the savings in terms of the number of disk accesses are large and almost the same for all K values. However, the savings are considerably less when B ≤ 32, whereas for K = 100000 and B = 0 we can notice a characteristic peak. For PSI, the savings trend is similar to the PSR, but for high K values these savings become less than PSR. For instance, if we have available adequate buffer space, PSR is the best alternative for the number of disk accesses, since it provides an average I/O savings of 18% with respect to the PSI for K-CPQ using our buffering configuration.

From the results shown in Figure 5.5, we have obtained the percentage of I/O savings (induced by the use of buffer size B > 0 in contrast to using no buffer) of PSR and PSI. For PSR, the percentage of saving grows as the buffer size increases, for all K values. The trend of the behavior of PSI is almost the same to PSR, although the increase is 8% less in average with respect to the recursive algorithm.



(a) (b)

Figure 5.5: The number of disk accesses for (a) PSR and (b) PSI, as a function of the LRU buffer size (B) and the cardinality of the result (K).

From the results and conclusions of this subsection, we can notice that the influence of our buffer scheme according to [CVM01] is more important for the recursive K-CPQ algorithms (mainly for PSR) than for the iterative one (PSI), primarily due to the fact that the use of recursion in a Depth-First traversal and the plane-sweep technique is affected by our buffering scheme more than the case of a Best-First searching strategy implemented through a minimum binary heap.

5.5 K-Self-CPQ, Semi-CPQ and K-FPQ

The three more important extensions of our K-CPQ algorithms are the so-called K-Self-CPQ, Semi-CPQ and K-FPQ. First of all, we proceed with the evaluation of the three K-CPQ algorithms adapted to the K-Self-CPQ constrains. For the (NApp, NApp) configuration, Table 5.4 compares all our performance metrics (disk accesses are in brackets) for each adapted algorithm for K-Self-CPQ and B = 256 pages. For the number of disk accesses, these increasing trends are due to the fact that we must discard two kinds of candidate pairs (equal to and symmetric). The behavior of PSR, where for all K values we obtain the same number of disk accesses, is interesting. Respect to the total response time, for small K values ($K \leq 100$) PSR was the best alternative, but PSI is the fastest for large K values ($K \geq 1000$). Moreover, PSI was the algorithm with the minimum number of subproblems for all K values, and SDR was the worst algorithm for all metrics showed in this table. These results confirm our conjecture that the plane-sweep technique adapted to this kind of distance-based query reduces the number of distance computations, and this results in the reduction in response time. Besides, the Best-First traversal minimizes the number of subproblems and this effect can be shown for large K values.

Next, we report the results of our tests on the extension of the non-incremental algorithms for Semi-CPQ. We have implemented the recursive version of “GlobalObjects” (GOR), “GlobalAll” for recursive (GAR and GASR <sorting the pairs based on *MINMINDIST*>) and iterative (GAI) schema. We have not applied the plane-sweep technique, since in this case the z value is not global to the query

result and each object of the first dataset must maintain its own lower bound. This query can also be implemented using a nearest neighbor algorithm. For each object in the first R*-tree, we perform a nearest neighbor query in the second R*-tree, and sort the result once all neighbors have been calculated. We have called this procedure T+NNQ, since it consists of three steps: (1) traverse recursively the first R*-tree, accessing the object in order of appearance within each leaf; (2) for each object, perform a nearest neighbor query into the second R*-tree and (3) sort the results (array of object with its distances) in ascending order of distances.

	K=1	**K=10**	**K=100**	**K=1000**	**K=10000**	**K=100000**
SDR	{354} **8.68** *4216209* 200	{358} **9.68** *4372699* 208	{458} **21.95** *9367711* 438	{608} **36.13** *14628021* 686	{771} **52.91** *19855965* 968	{825} **81.61** *22284261* 1102
PSR	{350} **0.19** *35695* 216	{350} **0.26** *56015* 286	{350} **0.64** *154154* 518	{350} **1.86** *493696* 808	{350} **8.61** *1680911* 1052	{350} **107.72** *6233038* 1293
PSI	{354} **0.23** *57189* 200 (30276)	{358} **0.28** *77927* 208 (30276)	{464} **0.67** *175741* 438 (30276)	{614} **1.70** *437107* 686 (30276)	{778} **6.89** *1209604* 968 (30276)	{841} **37.14** *3505811* 1102 (30276)

Table 5.4: Comparison of the K-Self-CPQ algorithms for the (NApp, NApp) configuration, with varying K and B = 256 pages.

From these experiments, we have considered the (NApp, NArd) configuration without buffer. Obviously, we have reported 24,493 pairs in the result (cardinality of NApp). Table 5.5 compares the four performance metrics for this query. Our extensions obtain the best behavior with respect to the number of disk accesses, mainly "GlobalAll" iterative (GAI). However, for the other metrics, T+NNQ is better than our extensions, since it needs less distance computations. Also, we must highlight that T+NNQ needs a main memory array of objects with their distances for all objects indexed in the first R*-tree, whereas our "GlobalObjects" extension needs the same amount of main memory and "GlobalAll" needs memory for objects and MBRs from internal nodes. From these results, we can conclude that our extensions are adequate for Semi-CPQ with respect to the number of disk accesses without buffers, but they consume significant space and time resources to report the result.

	T+NNQ	**GOR**	**GAR**	**GASR**	**GAI**
Disk Accesses	94209	69894	45296	38962	38868
Response Time	29.52	1180.95	703.35	681.82	589.04
Distance Comp.	17007578	674087280	437822358	376471488	363864610
Sub. Decomp.		34771	22472	19305	19258

Table 5.5. Comparison of the Semi-CPQ algorithms for the (NApp, NArd) configuration without buffering.

Another extension of K-CPQ is to find the K farthest pairs of objects from two datasets (K-FPQ). For this purpose, we have implemented recursive and iterative extensions of our algorithms (without using the plane-sweep technique) for K-CPQ. The algorithms have been called: Non-Sorted Distances Recursive (NSDR), Sorted Distances Recursive (SDR) and Non-Sorted Distances Iterative (NSDI). Table 5.6 shows all our performance metrics (disk accesses are in brackets) for K-FPQ using the (USrr, USrd) configuration with a global LRU buffer of 256 pages. From these measurements, we can observe the reduced number of disk accesses needed for this query, even for large K values. The explanation is that *MAXMAXDIST* is the function for pruning in the extended branch-and-bound algorithms instead of *MINMINDIST*, and *MAXMAXDIST* is very effective in this case. In addition, SDR and NSDI have the best behavior, and they are notably better that NSDR. For the other performance measurements, SDR and NSDI are considerably better than NSDR. In particular, NSDI consumes slightly less time to report the result, although the number of distance computations is greater. This behavior is due to the sorting of *MAXMAXDIST* of all possible pairs of MBRs from two internal nodes that SDR needs to execute the query. Moreover, we have executed the algorithms for the (NArd, NApp) configuration (lines, points) over R*-trees with different heights and we have obtained similar results and conclusions with respect to the (USrr, USrd) configuration.

	K=1	**K=10**	**K=100**	**K=1000**	**K=10000**	**K=100000**
NSDR	{124} **13.32** *2913526* 140	{127} **13.66** *2977628* 144	{143} **16.13** *3454505* 169	{196} **24.68** *5008063* 252	{290} **48.18** *8393499* 426	{520} **173.64** *20184013* 1016
SDR	{7} **0.14** *43030* 3	{7} **0.15** *43030* 3	{7} **0.15** *43030* 3	{7} **0.17** *43030* 3	{12} **0.57** *124228* 8	{24} **1.85** *343390* 20
NSDI	{7} **0.11** *72488* 3 (29458)	{7} **0.11** *72488* 3 (29458)	{7} **0.11** *72488* 3 (29458)	{7} **0.13** *72488* 3 (29458)	{12} **0.59** *153686* 8 (29458)	{24} **1.92** *372848* 20 (29458)

Table 5.6: Comparison of the K-FPQ algorithms, with varying K and B = 256 pages for the (USrr, USrd) configuration.

5.6 Scalability of the Algorithms, Varying the Dataset Sizes and K

As already pointed out, we are going to study the scalability of the K-CPQ algorithms with respect to the dataset sizes and K. First of all, we will study the effect of varying the dataset sizes, fixing the K value, for the datasets with rail-roads (line segment) and roads (line segment) from California (CA), West USA (WU), United States of America (US), USA + Mexico (UX) and North America (NA) as shown in Table 5.7.

	Rail-Roads	Roads
California	11,381	21,831
West USA	81,043	244,385
USA	146,503	355,312
USA + Mexico	156,563	447,704
North America	191,637	569,120

Table 5.7: Cardinalities of the real datasets for studying the algorithm scalability.

Figure 5.6 shows that the performance (i.e. number of disk accesses) increases almost linearly with the increase of the cardinalities of the real datasets, even for large K values. The trends for two diagrams are very similar, since the savings in disk accesses using a global LRU buffer is very high. Moreover, in the presence of buffer, again, the PSR is the best alternative and PSI provides the largest number of disk accesses.

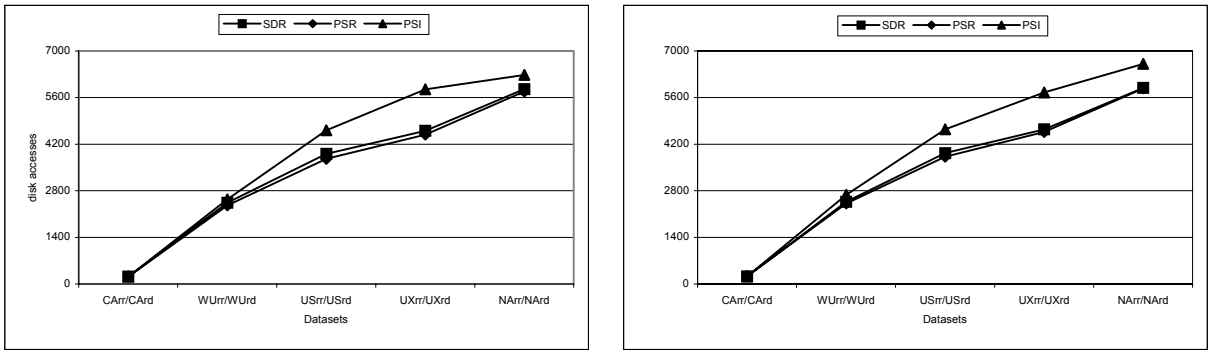


Figure 5.6: Comparison of the K-CPQ algorithms in terms of the number of disk accesses for B = 256 pages, using different configurations of the datasets in increasing size for (a) K = 1000 and (b) K = 100000.

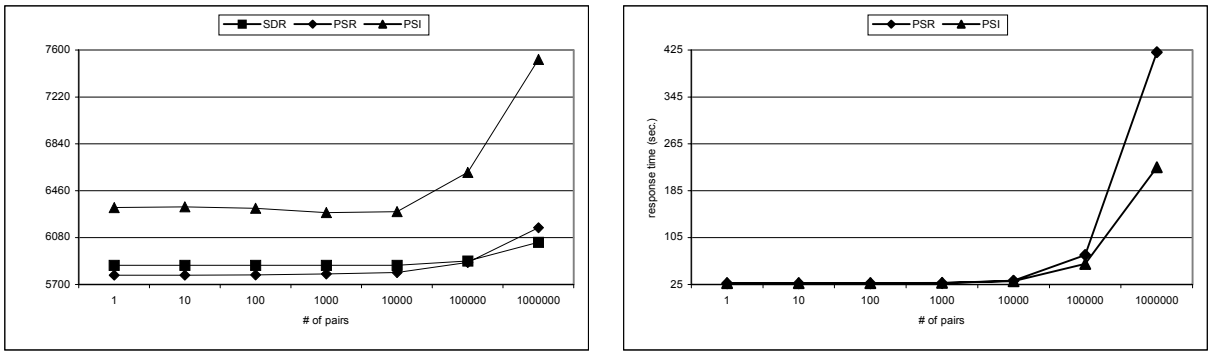
For the previous five configurations, Table 5.8 compares the other performance metrics for K = 100000 and B = 256. Clearly, PSI is the best algorithm for total response time, distance computations and subproblems. Also, we have executed experiments for the other K values, and the results were analogous to the ones of subsection 5.2 in all configurations: PSR won when $K \leq 1000$ and PSI when $K \geq 10000$. Besides, the increase of the performance was almost linear with the increase of the cardinalities of the real datasets for a given K, following the same trend to the disk accesses.

Another way to measure the scalability of our K-CPQ algorithms is to take into account their behavior with increasing K values using large real datasets. Figure 5.7.a shows that the number of disk accesses increases in a sub-linear way with the increase of the cardinalities of the result for the recursive alternative, using the (NArr, NArd) configuration and B = 256 pages. Namely, with increasing K values (1..1000000), the performance of PSR is not significantly affected; there is only a 6% extra cost, whereas for PSI this extra cost is about 16%. Moreover, SDR for K = 1000000 is slightly better than PSR, only 2%. Figure 5.7.b illustrates the response time for the fastest K-CPQ algorithms (PSR

and PSI) for the increase of cardinality of the result. For instance, they have very similar results for K ≤ 10000, but for K = 100000 and K = 10000000 PSI is 20% and 48% faster than PSR, respectively.

	CArr/Card	WUrr/Wurd	USrr/USrd	UXrr/UXrd	NArr/NArd
SDR	**37.25** *7933307* 388	**382.21** *86584811* 4132	**639.94** *145538868* 6947	**726.38** *165453915* 7963	**891.82** *202997777* 9924
PSR	**19.66** *2583242* 503	**55.58** *7410232* 4289	**68.56** *9513814* 7135	**71.65** *10021597* 8083	**75.39** *10539267* 10149
PSI	**13.67** *1838280* 388 (12720)	**37.47** *5455709* 3964 (139850)	**48.58** *7454867* 6770 (187743)	**50.56** *7787764* 7697 (122471)	**59.97** *9045976* 9699 (191207)

Table 5.8: Comparison of the K-CPQ algorithms for K = 100000 and B = 256 pages, using different configurations of the real datasets in increasing size.



(a) (b)

Figure 5.7: Comparison of the K-CPQ algorithms in terms of the (a) number of disk accesses and (b) total response time, with varying K (1..1000000), B = 256 pages and (NArr, NArd) configuration.

Table 5.9 presents the other performance metrics for the (NArr, NArd) configuration and B = 256, varying K from 1 to 1000000. From these results we can conclude that PSR was the best when K ≤ 1000 and PSI when K ≥ 10000, with respect to the time response and the number of distance computations. Besides, PSI is the algorithm with the smallest number of subproblems for all K values, whereas it needs only a 48% extra of insertions in the Main-heap to carry out the query from K = 1 to K = 1000000. On the other hand, SDR is the worst, since it does not use the plane-sweep technique for reducing the number of distance computations and avoiding intermediate sorting processes.

	K=1	K=10	K=100	K=1000	K=10000	K=100000	K=1000000
SDR	**855.24** *196679155* 9601	**855.58** *196679155* 9601	**855.75** *196679155* 9601	**856.41** *196679155* 9601	**858.26** *197186117* 9626	**892.39** *202997777* 9924	**1116.02** *236040339* 11612
PSR	**26.88** *4305737* 9607	**26.89** *4307645* 9608	**26.97** *4318106* 9610	**27.41** *4386909* 9618	**31.46** *4979372* 9668	**75.31** *10539267* 10149	**421.45** *44331311* 12700
PSI	**27.32** *4439791* 9601 (156659)	**27.34** *4441921* 9601 (156671)	27.39 *4452615* 9601 (156739)	**27.68** *4501059* 9601 (156970)	**30.77** *4956503* 9601 (157822)	**60.05** *9045976* 9699 (191207)	**225.13** *28188435* 10416 (298557)

Table 5.9. Comparison of the K-CPQ algorithms with varying K (1..1000000), B = 256 pages and (NArr, NArd) configuration.

6 Conclusions and Open Problems

Efficient processing of K-CPQs is of great importance in spatial databases due to the wide area of applications that may address such queries. Although popular in computational geometry literature [PrS85], the closest pair problem has not gained special attention is spatial database research. Certain other problems of computational geometry, including the "all nearest neighbor" problem (that is related to the closest pair problem), have been solved for external memory systems [GTV93]. To the best of the authors' knowledge, [HjS98, CMT00, SML00] are the only references to this type of queries. In this paper, based on the properties of distance functions between two MBRs in the multidimensional Euclidean space, we propose a pruning heuristic and two updating strategies for minimizing the pruning distance to apply them in the design of three non-incremental branch-and-bound algorithms for K-CPQ between objects indexed in two R-trees, extending and enhancing the work presented in [CMT00]. Two of the algorithms are recursive, following a Depth-First searching strategy and one is iterative, obeying a Best-First traversal policy. The plane-sweep method and the search ordering (this heuristic is based on the ordering of *MINMINDIST*) are used as optimization techniques for improving the naive approaches. Furthermore, some interesting extensions of the K-CPQ are presented: K-Self-CPQ, Semi-CPQ, Self-Semi-CPQ, K-FPQ and a method to obtain the K or all closest pairs of objects with the distances within a range [Dist_Min, Dist_Max].

In the experimental section, we have used an R-tree variant (R*-tree) in which the objects are stored directly in the tree leaves. Moreover, an extensive experimentation was also included, which resulted to several conclusions about the efficiency of each algorithm (disk accesses, response time, distance computations and subproblems) with respect to K, the size of the underlying buffer, the disjointedness of the workspaces and the algorithmic scalability. The more important conclusions for the K-CPQ algorithms over overlapped or disjoint workspaces are listed as follows:

- The Sorted Distances Recursive (SDR) algorithm has a good performance with respect to the number of disk accesses when we include a global LRU buffer for all configurations. But, it consumes much time for reporting the results, since it must combine all possible entries from two internal R-tree nodes in a temporary list of pairs of MBRs, compute its *MINMINDIST* for each pair, and sort this list of pairs in ascending order of *MINMINDIST*.
- The Plane-Sweep Recursive (PSR) algorithm is the best alternative with regards to the I/O activity when buffer space is available, since the combination of recursion in a Depth-First traversal and LRU page replacement policy favors this performance metric. Moreover, this algorithm is the fastest for small and medium K values, since it reduces the distance computations using the plane-sweep technique.
- The Plane-Sweep Iterative (PSI) algorithm is the best alternative for the number of disk accesses without buffer, but when we have a global LRU buffer this behavior is inverted, since the Best-First traversal implemented through a minimum binary heap is less affected in contrast to the combination of recursion in a Depth-First searching strategy with an LRU replacement policy. Moreover, this algorithm is the fastest for large K values, since it obtains the minimum number of distance computations and subproblems (Best-First traversal and plane-sweep technique) in this case. Also, it is interesting to observe the small number of insertions in the Main-heap even for very large K values, because our pruning heuristic based on *MINMINDIST* is very effective in non-incremental branch-and-bound algorithms for K-CPQ.
- K does not radically affect the relative performance with respect to the number of disk accesses, since the increase of this function grows sublinearly with the increase of K.
- The number of disk accesses grows almost linearly with the increase of the dataset cardinalities; this trend is noticed for the other performance metrics, too.
- In general, the PSI and PSR response times are significantly lower than SDR' one (one order of magnitude for large datasets), while disk accesses keep comparable. Therefore, PSI is preferable when enough main memory resources are available to store the Main-heap (PSI outperforms the two other recursive algorithms, except for disk accesses), otherwise PSR is the best alternative.

We have also implemented and presented experimental results for three special cases of closest pairs queries: K-Self-CPQ, where both datasets actually refer to the same entity, Semi-CPQ, where for each element of the first dataset, the closest object of the second dataset is computed, and K-FPQ, where the K farthest pairs of objects from to datasets are found. Again, the iterative variants have the best overall performance, although the recursive ones are I/O competitive in the presence of buffers.

Future work on CPQs may include: (1) The study of multi-way K-CPQs where tuples of objects are expected to be the answers, extending related work on multi-way spatial joins [PMT99]. (2) The analytical study of K-CPQs, extending related work on spatial joins [TSS98] and nearest neighbor queries [PaM97]. (3) The extension of our K-CPQ algorithms using multidimensional data for exact result or

approximate K-closest pairs query (the degree of inexactness can be specified by an upper bound ε and indicates the reported answer and the exact closest pair distance) in a sense similar to the approximate nearest neighbor searching [AMN98].

References

[AMN98] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman and A.Y. Wu: "An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions", *Journal of the ACM*, Vol. 45, No.6, pp.891-923, 1998.

[BKS90] N. Beckmann, H.P. Kriegel, R. Schneider and B. Seeger: "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *Proceedings ACM SIGMOD Conference*, pp.322-331, 1990.

[BKS93] T. Brinkhoff, H.P. Kriegel and B. Seeger: "Efficient Processing of Spatial Joins Using R-trees", *Proceedings ACM SIGMOD Conference*, pp.237-246, 1993.

[Bro01] P. Brown: *Object-Relational Database Development: A Plumber's Guide,* Prentice Hall, 2001.

[ChD85] H.T. Chou and D.J. DeWitt: "An Evaluation of Buffer Management Strategies for Relational Database Systems", *Proceedings 11th VLDB Conference*, pp.127-141, 1985.

[ChF98] K.L. Cheung and A.W. Fu: "Enhanced Nearest Neighbour Search on the R-tree", *ACM SIGMOD Record*, Vol.27, No.3, pp.16-21, 1998.

[ChW84] F.Y. Chin and C.A. Wang: "Minimum Vertex Distance Between Separable Convex Polygons", *Information Processing Letters*, Vol. 18, No.1, pp.41-45, 1984.

[CMT00] A. Corral, Y. Manolopoulos, Y. Theodoridis and M. Vassilakopoulos: "Closest Pair Queries in Spatial Databases", *Proceedings ACM SIGMOD Conference*, pp.189-200, 2000.

[COL92] C.Y. Chan, B.C. Ooi and H. Lu: "Extensible Buffer Management of Indexes", *Proceedings 18th VLDB Conference*, pp.444-454, 1992.

[Com79] D. Comer: "The Ubiquitous B-tree", *ACM Computing Surveys*, Vol.11, No.2, pp.121-137, 1979.

[CVM01] A. Corral, M. Vassilakopoulos and Y. Manolopoulos: "The Impact of Buffering for the Closest Pairs Queries using R-trees", *Proceedings 5th ADBIS Conference,* pp.41-54, 2001.

[DCW97] Digital Chart of the World: Real spatial datasets of the world at 1:1,000,000 scale. 1997. Downloadable from: http://www.maproom.psu.edu/dcw.

[EfH84] W. Effelsberg and T. Harder: "Principles of Database Buffer Management", *ACM Transactions on Database Systems*, Vol.9, No.4, pp.560-595, 1984.

[FBF77] J.H. Friedman, J.L. Bentley and R.A. Finkel: "An Algorithm for Finding Best Matches in Logarithmic Expected Time", *ACM Transactions on Mathematical Software*, Vol.3, No.3. pp.209-226, 1977.

[GaG98] V. Gaede and O. Günther: "Multidimensional Access Methods", *ACM Computing Surveys*, Vol.30, No.2, pp.170-231, 1998.

[GJK88] E.G. Gilbert, D.W. Johnson and S.S. Keerthi: "A Fast Procedure for Computing the Distance Between Complex Objects in Three-dimensional Space", *IEEE Journal of Robotics and Automation*, Vol.4, No. 2, pp.193-203, 1988.

[GTV93] M.T. Goodrich, J.J. Tsay, D.E. Vengroff and J.S. Vitter: "External-Memory Computational Geometry", *Proceedings 34th FOCS Conference*, pp.714-723, 1993.

[Gut84] A. Guttman: “R-trees: A Dynamic Index Structure for Spatial Searching”, *Proceedings ACM SIGMOD Conference*, pp.47-57, 1984.

[HJR97] Y.W. Huang, N. Jing and E.A. Rundensteiner: “Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations”, *Proceedings 23rd VLDB Conference*, pp.396-405, 1997.

[HjS95] G.R. Hjaltason and H. Samet: “Ranking in Spatial Databases”, *Proceedings 4th SSD Conference,* pp.83-95, 1995.

[HjS98] G.R. Hjaltason and H. Samet: “Incremental Distance Join Algorithms for Spatial Databases”, *Proceedings ACM SIGMOD*, pp.237-248, 1998.

[HjS99] G.R. Hjaltason and H. Samet: “Distance Browsing in Spatial Databases”, *ACM Transactions on Database Systems*, Vol.24 No.2, pp.265-318, 1999.

[Iba87] T. Ibaraki: *Annals of Operations Research,* Scientific Publishing Company, 1987.

[KoS97] N. Koudas and K.C. Sevcik: “Size Separation Spatial Join”, *Proceedings ACM SIGMOD Conference*, pp.324-335, 1997.

[LaT92] R. Laurini and D. Thomson: *Fundamentals of Spatial Information Systems,* Academic Press, London, 1992.

[LoR96] M.L. Lo and C.V. Ravishankar: “Spatial Hash-Joins”, *Proceedings ACM SIGMOD Conference*, pp.247-258, 1996.

[Ora01] Oracle Technology Network: “Oracle Spatial User’s Guide and Reference”, 2001. Downloadable from: http://technet.oracle.com/doc/Oracle8i_816/inter.816/a77132.pdf

[PaD96] J.M. Patel and D.J. DeWitt: “Partition Based Spatial-Merge Join”, *Proceedings ACM SIGMOD Conference*, pp.259-270, 1996.

[PaM97] A.N. Papadopoulos and Y. Manolopoulos: “Performance of Nearest Neighbor Queries in R-Trees”, *Proceedings 6th ICDT Conference*, pp.394-408, 1997.

[PMT99] D. Papadias, N. Mamoulis and Y. Theodoridis: “Processing and Optimization of Multi-way Spatial Joins Using R-trees”, *Proceedings 18th ACM PODS Conference*, pp.44-55, 1999.

[PrS85] F.P. Preparata and M.I. Shamos: *Computational Geometry: an Introduction,* Springer Verlag, 1985.

[RKV95] N. Roussopoulos, S. Kelley and F. Vincent: “Nearest Neighbor Queries”, *Proceedings ACM SIGMOD Conference*, pp.71-79, 1995.

[SML00] H. Shin, B. Moon and S. Lee: “Adaptive Multi-Stage Distance Join Processing”, *Proceedings ACM SIGMOD Conference*, pp.343-354, 2000.

[SRF87] T. Sellis, N. Roussopoulos and C. Faloutsos: “The R^+-tree: a Dynamic Index for Multi-Dimensional Objects”, *Proceedings 13th VLDB Conference*, pp.507-518, 1987.

[TSS98] Y. Theodoridis, E. Stefanakis and T. Sellis: “Cost Models for Join Queries in Spatial Databases”, *Proceedings 14th ICDE Conference*, pp.476-483, 1998.