UNIVERSITY OF ALMERIA

Department of Languages and Computation

PhD DISSERTATION

Algorithms for Processing of Spatial Queries using R-trees. The Closest Pairs Query and its Application on Spatial Databases

ANTONIO LEOPOLDO CORRAL LIRIA

Almeria, January 2002

UNIVERSITY OF ALMERIA

Department of Languages and Computation

Algorithms for the Processing of Spatial Queries using R-trees. The Closest Pairs Query and its Application on Spatial Databases

PhD Dissertation presented by **Antonio Leopoldo Corral Liria** in order to apply for the doctoral degree on Computer Science by the University of Almeria

Supervisors: Dr. Yannis Manolopoulos
Dr. José Samos

Almeria, January 2002

Abstract

This thesis addresses the problem of finding the K closest pairs between two spatial datasets (the so-called, K Closest Pairs Query, K-CPQ), where each set is stored in an index structure belonging in the R-tree family. There are two different techniques for solving this kind of distance-based query. The first is the incremental approach [HjS98, SML00], which computes the operation in the sense that the result elements are reported one-by-one in ascending order of distance. The second is the non-incremental alternative that we present in this thesis, which computes the operation reporting the K elements of the result all together at the end of the algorithm. Branch-and-bound has been the most successful technique for the design of algorithms that obtain the result of queries over tree-like structures. In this thesis, a general branch-and-bound algorithmic schema for obtaining the K optimal solutions of a given problem is proposed. Moreover, 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, in order to 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 spatial datasets. A wide range of values for the basic parameters affecting the performance of the algorithms is examined. The outcome of these studies is the designation of the algorithm that wins the performance award for each setting of parameter values. An experimental study of the behavior of the proposed K-CPQ branch-and-bound algorithms in terms of scalability of the dataset sizes and the value of K is also included. Finally, our non-incremental algorithms have been adapted to perform the simulation of the incremental closest pairs query.

Table of Contents

Chapter 1

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 data of modern life [MTT99]. In a computer system, these data are represented by points, line segments, regions, polygons, volumes and other kinds of 2D/3D geometric entities and are usually referred to as spatial objects. spatial databases include specialized systems like “Geographical Information Systems”, “CAD databases”, “Multimedia databases”, “Image databases”, etc. The role of spatial databases is continuously increasing in many modern applications during last years. 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 [Güt94]. The most basic form of such a manipulation is answering queries related to the spatial properties of data. Some typical spatial queries are the following.

- A “Point Location Query” seeks for the spatial objects that fall on a given point.
- A “Range Query” seeks for the spatial objects that are contained within a given region (usually expressed as a rectangle or a sphere).
- 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 spatial objects that satisfy a given spatial predicate [APR98, BKS93a, BKS94, LoR94, LoR96, PaD96, KoS97, HJR97a]. The spatial distance join [HjS98] was recently introduced in spatial databases to compute a subset of the Cartesian product of two spatial datasets, specifying an order on the result based on distance.
- Finally, a very common spatial query is the “Nearest Neighbor Query” that seeks for the spatial 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.

Branch-and-bound [HoS78, Iba87, BrB95] has been the most successful technique for the design of algorithms that obtains the result of queries based on tree-like 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 policy chosen, for instance Best-First and Depth-First which are used in most situations. Numerous branch-and-bound algorithms for spatial queries (exact query, range query, nearest neighbor query and spatial join) using spatial access methods have been studied in the literature [RSV01, YuM98]. Here, we show how these bounding functions and searching policies 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 thesis, we examine a spatial query, called "K Closest Pairs Query" (K-CPQ), that discovers the K pairs ($K \geq 1$) of spatial objects formed from two spatial 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 spatial 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 of America, while the second one may contain the most important populated places of North America (see Figure 5.1 in Chapter 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 value of K could be dependent on the budget of the authorities allocated for this purpose.

The fundamental assumption is that the two spatial datasets are stored in structures belonging in the family of R-trees [Gut84]. R-trees and their variants (R^+-trees [SRF87], R*-trees [BKS90], X-tree [BKK96], etc.) are considered an excellent choice for indexing various kinds of spatial data (points, line segments, rectangles, polygons, etc.) and have already been adopted in commercial systems (Informix [Bro01], Oracle [Ora01], etc). In this thesis, 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, in order to 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 policy; and the third 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 spatial 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.

Moreover, 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 one dataset, the closest object of the other 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 in terms of the size of the datasets and the number K of closest pairs is analyzed. In addition, we have executed experiments with the incremental algorithms [HjS98] and compared their results with a pseudo-incremental approach, where our non-incremental algorithms have been adopted to perform the incremental closest pairs query.

The organization of this thesis is the following. Chapter 2 discusses the incremental and non-incremental algorithmic approaches for the closest pairs query, as well as the motivation of this research. In Chapter 3, the K-CPQ, a review of the structure of R-trees and some useful metrics between MBRs are presented. In Chapter 4, the branch-and-bound technique is reviewed, a pruning heuristic, two updating strategies and three new non-incremental branch-and-bound algorithms for K-CPQ are introduced. Chapter 5 exhibits a detailed performance study of all algorithms for K-CPQs, including the effect of buffering, K-Self-CPQ, Semi-CPQ, K-FPQ, a scalability study and a performance comparison of the simulation of the incremental processing in order to obtain the K closest pairs incrementally. Finally, in Chapter 6 conclusions on the contribution of this thesis and related future research plans are presented.

Chapter 2

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 computes the operation in the sense that the number of desired elements in the result is reported one-by-one in ascending order of distance (pipelined fashion), i.e. the user can have part of the final result before the end of the algorithm execution. The incremental algorithms work in the sense that having obtained K elements of the result, to find the (K+1)-th, it is not necessary to restart the algorithm for K+1, but just to perform an additional step. The kernel of the incremental algorithms is a priority queue, build on a distance function associated to the kind of 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 of 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. In other words, the incremental algorithms are competitive when a small quantity of elements of the result is needed.

The second approach is the non-incremental [RKV95, CMT00], which computes the operation when K is known in advance and the K elements belonging to the result are reported 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 result elements just after the execution of the algorithm for K (M > K), he/she must restart the algorithm with M as input without reusing the previous result to obtain the rest M – K elements. However, the advantage of the non-incremental approach is that the pruning process during the execution of the algorithm is more effective even when K is large enough; this will be shown later, in the experimental chapter.

Numerous algorithms exist for answering distance-based queries. Most of these algorithms are focused on the nearest neighbors query (NNQ) over one multidimensional access method. 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, extensions of these algorithms can be applied to other recent multidimensional access methods (SR-tree [KaS97], A-tree [SYU00], etc.) for improving the I/O activity and the CPU cost.

To the authors' knowledge, for closest pairs query (CPQ) in spatial databases using R-trees, [HjS98, SML00, CMT00] are the more relevant references. 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] by using adaptive multi-stage processing, plane-sweep technique [PrS85] and other improvements based on sweeping axis and sweeping direction, for K-distance join and incremental distance join. In [CMT00], non-incremental recursive and iterative branch-and-bound algorithms are presented for solving the K-CPQ using points.

The incremental algorithms of [HjS98, SML00] for closest pairs queries adapted to R-trees store in the priority queue item pairs of the following types: node/node, node/obr, obr/node and object/object (where “node” is an entry of the R-tree internal node, and “obr” and “object” are the object bounding rectangle and the actual data item stored on the R-tree leaf nodes, respectively). While processing a pair of subtrees between the two R-trees, the algorithms of [HjS98] are likely to insert in the priority queue a significant portion of all the above four types of item pairs that can be formed from these subtrees. This fact advocates to the creation of a significantly large priority queue and it is a very important issue to take into account when a large number of elements is required in the result. The algorithms of [SML00] also reject part of the item pairs in the priority queue by making use of the plane-sweep technique and multi-stage processing. Due to its expected large size, the priority queue of [HjS98] is stored partially in main memory (with one part as a heap and another part as an unordered list) and partially in secondary memory (as a number of linked lists of pages in a file on disk). The designation of the size of each part is crucial for the performance and it depends on the arbitrary choice of a constant. In [SML00], the priority queue is also based on a hybrid storage scheme, but the designation of the in-memory heap and the disk-resident part is accomplished by an estimation formula. To reduce the size of the main priority queue and report only K elements on the result, in [HjS98, SML00] an extra data structure and an upper bounding function are needed, resulting to an algorithm incremental up to K, only.

The efforts described in the previous paragraph are mainly focused on the incremental approach, optimizing the required resources and the processing strategy. Here, our main objective is to improve the work presented in [CMT00] with respect to the design of branch-and-bound algorithms (recursive and iterative) in a non-incremental way for performing K-CPQ between two spatial datasets stored in indices belonging to the family of R-trees [Gut84]. We analyze the theoretical aspects of 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 are the kernel in the design of branch-and-bound algorithms for solving this kind of query. Moreover, we apply techniques for improving the performance with respect to the I/O activity (buffering) and response time (plane-sweep). Besides, 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 spatial datasets of different nature (line segments and points) in order to study the performance of the algorithms.

Chapter 3

3 The K Closest Pairs Query using R-trees

In this chapter, K-CPQ is defined and illustrated with an example. Moreover, a brief description of the R-trees is presented, pointing out the main characteristics of the R*-tree and its insertion procedure. Finally, some useful metrics between MBRs are proposed, which will be used in branch-and-bound algorithms for answering the K-CPQ.

3.1 Definition of the Query

We assume a point dataset *P* in the k-dimensional data space $D^{(k)}$ (≡ Â$^{(k)}$) such that for every p = $(p_1, p_2, \ldots , p_k) \in P$, $p_i \in$ Â and a metric distance function *dist* for a pair of points, i.e. *dist*:$P \times P \rightarrow$ Â$^{+}$. ∀p, q ,r ∈ *P*, the function *dist* satisfies the four following conditions:

$$\begin{array}{ll} dist(p,q) \geq 0 & < non-negativity > \\ dist(p,q) = 0 \Leftrightarrow p = q & < identity > \\ dist(p,q) = d(q,p) & < symmetry > \\ dist(p,q) \leq d(p,r) + d(r,q) & < \Delta - inequality > \end{array}$$

The more general expression for *dist* is called L_t-distance (L_t), L_t-metric, or Minkowski distance between two points, p = $(p_1, p_2, \ldots , p_k)$ and q = $(q_1, q_2, \ldots , q_k)$, in the k-dimensional data space, and it is defined as follows:

$$L_t(p,q) = \left(\sum_{i=1}^{k} |p_i - q_i|^t \right)^{1/t}, \quad if \ \ 1 \leq t \leq \infty$$

$$L_\infty(p,q) = \max_{1 \leq i \leq k} |p_i - q_i|, \qquad if \ \ t = \infty$$

For t = 2 we have the Euclidean distance and for t = 1 the Manhattan distance. They 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.

An important property of the L_t-distance function *(dimension distance property)* is that the value of this function (L_t-distance) for a given dimension (1≤i≤k) is always smaller than or equal to the total computation of the L_t-distance for all dimensions (k). Thereby, we can use this property for computing distances in some situations for a given dimension, since its computation is cheaper than the global one and we may obtain performance gain.

$$L_t(p,q,i) = |p_i - q_i| \leq L_t(p,q), \quad if \ \ 1 \leq i \leq k \ \ and \ \ 1 \leq t \leq \infty$$

The k-dimensional Euclidean space, $E^{(k)}$, is the pair ($D^{(k)}$, L_2). In other words, it is the k-dimensional data space $D^{(k)}$, equipped with the Euclidean distance (in the following we will use *dist* instead of L_2).

In the spatial database literature, two kinds of distance-based queries have gained attention: nearest or farthest neighbor and closest pairs queries, where a dataset is examined against a single query reference or another dataset, respectively.

Definition 1

Let two point sets, P = {p_1, p_2, ... , p_{NP}} and Q = {q_1, q_2, ... , q_{NQ}} in $E^{(k)}$, be stored in a spatial database. Then, the result of the K closest pairs query, K-CPQ(P, Q, K), is a set containing subsets of K (1£K£|P|·|Q|) different and ordered pairs of points from 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* ***Î*** *P, q_1, q_2, ..., q_K* ***Î*** *Q,*
(p_i, q_i) ¹ (p_j, q_j) "i ¹ j: "(p_i, q_j) Í P´Q – {(p_1, q_1), (p_2, q_2), ... , (p_K, q_K)},
dist(p_i, q_j) ³ dist(p_K, q_K) ³ dist($p_{(K-1)}$, $q_{(K-1)}$) ³ … ³ dist(p_2, q_2) ³ dist(p_1, q_1)}

Note that, due to ties of distances, the result of the K-CPQ may not be unique for an specific pair of point sets P and Q. The aim of the presented 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 by points to other *spatial data types* (point, line, region, etc.) is straightforward. We assume that the set of *k-dimensional polyhedra* forms a data type, which leads us to the common collection of *spatial data types*. A *k-dimensional polyhedron* is the union of a finite number of polytopes. A *k-dimensional polytope* is the convex hull of a finite set of k-dimensional points [PrS85]. A convex polytope is described by means of its boundary, which consists of *faces* (each face is a lower-dimensional convex polytope). If a polytope is k-dimensional, its (k-1)-faces are called *facets*, its (k-2)-faces are called *subfacets*, its 1-faces are *edges* and 0-faces are *vertices*. 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 modifications on the K-CPQ definition is the replacement of points p and q with spatial objects p and q with spatial extent p.G and q.G, respectively, in $E^{(k)}$ and the replacement of the distance between two points (dist) with the distance between two spatial objects.

3.2 Example of K-CPQ

In Figure 3.1 two sets, each consisting of three points in the plane are depicted. The first set is represented by crosses (Figure 3.1.a), while the second set by stars (Figure 3.1.b). The minimum bounding rectangle (MBR) covering each set is also depicted by a dashed rectangle. The solid line rectangle in each subfigure represents the whole data space (a part of the plane). To be able to distinguish visually the closest pairs of these sets, in Figure 3.1.c the two sets appear together. It is obvious that the 1-CP is (p3, q1) and the 2-CPs are (p3, q1) and (p2, q2). Using a ruler, it is easy to discover that the 3-CPs are (p3, q1), (p2, q2) and (p3, q2). The 4-CPs are (p3, q1), (p2, q2), (p3, q2) and (p3, q3). Note that the MBRs of the two datasets may partially or totally overlap. It is evident that the higher the portion of overlapping between the two MBRs the higher is the probability that more pairs with small distances will appear and an algorithm answering the closest pair query will have to examine more candidate pairs with distances that differ slightly.

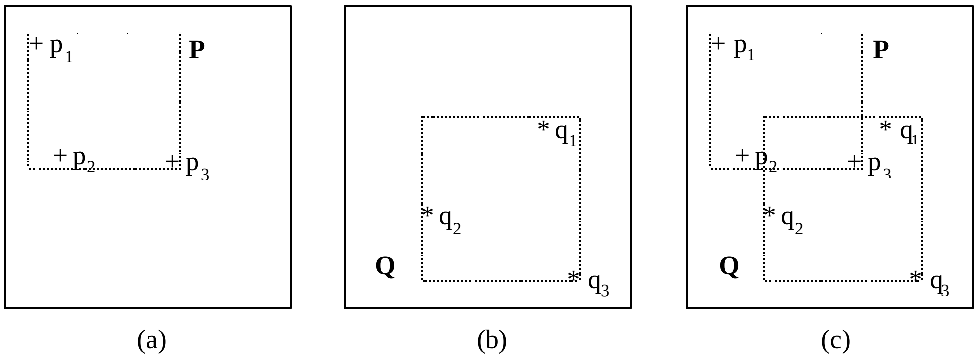


Figure 3.1: Two sets of points in the plane.

As stated earlier the two datasets are stored in structures of the family of R-trees. This means that the specific organization of data 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.

3.3 R*-tree

R-trees [Gut84] are hierarchical, height balanced multidimensional data structures based on B-trees [BaM72, Com79]. They are used for the dynamic organization of a set of k-dimensional geometric objects represented by their Minimum Bounding k-dimensional hyper-Rectangles (MBRs). These MBRs are characterize by "min" and "max" points of hyper-rectangles with faces parallel to the coordinate axis. Using the MBR instead of the exact geometrical representation of the object, its representational complexity is reduced to two points where the most important features of the object (position and extension) are maintained. Consequently, the MBR is an approximation widely employed [BKS93b].

Each R-tree node corresponds to the MBR that contains its children. The tree leaves contain pointers to the objects of the database, instead of pointers to children nodes. The nodes are implemented as disk pages. It must be noted that the rectangles that surround different nodes may be overlapping. 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 spatial search may demand visiting of 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 node contains entries of the form (MBR, Oid), such that MBR is the minimum bounding rectangle that encloses the spatial object determined by the identifier Oid. Every other node (internal) contains 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. An R-tree of class (m, M) has the characteristic that every node, except possibly for the root, contains 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 3.2 depicts some rectangles on the right and the corresponding R-tree on the left. Dotted lines denote the bounding rectangles of the subtrees that are rooted in inner nodes.

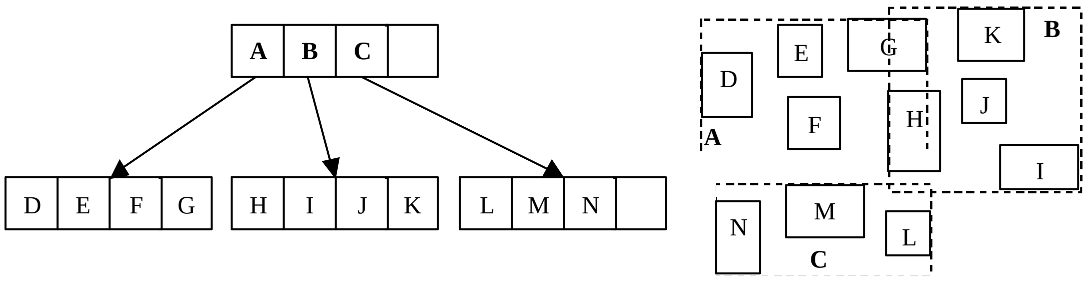


Figure 3.2: An example of an R-tree.

Like other tree-like index structures, an R-tree index partitions the multidimensional space by grouping objects in a hierarchical manner. A subspace occupied by a tree node in an R-tree 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. This characteristic of spatial containment between MBRs of R-tree nodes is commonly used by spatial join algorithms as well as distance-based query algorithms.

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

Many variations of R-trees have appeared in the literature (exhaustive surveys can be found in [Sam90a, GaG98, MTT99]). One of the most popular and efficient variations is the R*-tree [BKS90]. The R*-tree added two major enhancements to the R-tree, when a node overflow is caused. First, rather than just considering the area, the node-splitting algorithm in R*-tree also minimized the perimeter and overlap enlargement of the minimum bounding rectangles. Minimizing the overlap tends to reduce the number of subtrees to follow for search operations. Second, R*-tree introduced the notion of *forced reinsertion* to make the shape of the tree less dependent to the order of insertions. When a node overflows, it is not split immediately, but a portion of entries of the node is reinserted from the top of the tree. The reinsertion provides two important improvements: (1) it may reduce the number of splits needed and, (2) it is a technique for dynamically reorganizing the tree. 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. In this thesis, we choose R*-trees to perform our experimental study.

We now present the insertion algorithm on the R*-tree more rigorously. Let T denote the pointer to the root of the R*-tree and let (S, P_S) be the pair to be inserted, where P_S is the address pointer of MBR S. Let L be the level at which (S, P_S) is to be inserted. Initially, L is the leaf level. Each level has an overflow flag indicating whether an overflow has occurred at this level. At then beginning, overflow flags at all levels are set to *off*.

Insert(S, P_S, L, T)

1. Select a node N at level L to insert S. This is carried out recursively from the root node to a node at level L. If N is a node at level L, return N. Otherwise, select a subtree to go down. This is done by choosing a proper entry in N. Two cases are considered.

1.1. If N is a parent of leaf nodes, choose the entry in N whose corresponding MBR needs the least overlap enlargement to contain S. If several MBRs in N need the same least overlap enlargement to contain S, choose the MBR whose area needs the least enlargement. (Note that the *area* of a MBR and the overlap of a MBR in N are different concepts). If again several MBRs in N have the same least area enlargement, choose the MBR with the smallest area.

1.2. If N is neither a leaf node nor a parent of leaf nodes, choose the MBR in N whose area needs the least enlargement. If several MBRs in N need the same least area enlargement, then choose the MBR with the smallest area.

Let (R, P) be the entry chosen. Replace T by P and continue the process of selecting a node at level L to insert S.

2. Let N be the node at level L returned from step 1. If N has available space to hold (S, P_S), insert it into N and exit. Otherwise, an overflow occurs. Execute step 3 to handle the overflow.

3. Check the overflow flag at level L to see whether this is the first overflow at this level. If the answer is yes (i.e., the overflow flags is still o*ff*) and N is not the root, set the overflow flag on and execute step 4; else go to step 5. Since the root level will always have one node, reinsertion at this level is useless. Therefore, if N is the root, go to step 5 directly to split the node.

4. Select *w* entries from the M + 1 entries in node N for reinsertion. This is a carried out as follows. First, compute the distances between the centers of the M + 1 MBRs and the center of the directory MBR for node N. Next, sort the M + 1 entries in descending order of the distances. The first *w* entries in this order will be used for reinsertion, and the directory MBR will be adjusted (i.e., reduced in size) accordingly. The selection tends to keep the remaining MBRs as close as possible. Experiments suggest that w = 0.3*M is a good choice.) Let (S_i, P_{Si}), $i = 1, ..., w$ be the *w* entries selected. Reinsert them in ascending order of the distances by calling *Insert*(S_i, P_{Si}, *L, T), i = w, w – 2, ...,1*. Experiments indicate that reinserting the *w* entries in ascending order of distances outperforms reinserting the *w* entries in descending order of the distances.

5. Split the M + 1 entries into two sets for distribution into two nodes. The constraint is that each node needs to have at least m entries. The split algorithm is described bellow.

5.1. For the *i-th* axis (dimension) in the k-dimensional space, sort the M + 1 entries first by the lower values and then by the higher values of their MBRs. For each sort, the following M – 2m + 2 different ways to partition the M + 1 entries into two sets are considered, The *j-th* way, 1 ≤ j ≤ M – 2m + 2, is to let the first set have the first m – 1 + j entries and the second set have the remaining entries. Note that the M – 2m + 2 different partitions enumerate all possible ways to split the M + 1 entries in such a way that each set has at least m entries and no entry in the first set appears before any entry in the second set according to the sort. For the *j-th* partition, let b_{j1} and b_{j2} be the MBRs of the MBRs in the first set and the second set, respectively. Let p_{j1} and p_{j2} be the perimeters of b_{j1} and b_{j2}, respectively. Let X_i denote the sum of the perimeters of all the MBRs for the *i-th* axis, $i = 1,\ldots,k$., as the following formula:

$$X_i = \sum_{j=1}^{M-2m+2} (p_{ij1} + p_{ij2}),\ \forall 1 \le i \le k$$

The axis with the smallest X_i is called the split axis. Only the partitions based on the sort the split axis will be considered for identifying the best split. By making an effort to minimize the perimeters, directory MBRs have the tendency to converge the squares. Since squares can be grouped more easily, smaller directory MBRs at the next level can be expected.

5.2. Among the M – 2m + 2 partitions along the split axis, find the partition with the smallest area overlap between b_{j1} and b_{j2}. If two or more partitions have the same smallest area overlap, choose the one with the least area sum.

5.3. Split the M + 1 entries based on the partition chosen in 5.2 and use the two new nodes to replace node N. Two entries corresponding to the two new nodes will be generated for the parent node N_P of N to replace the old entry. The MBR in each new entry is the MBR of the MBRs in the corresponding child node. If there is available room in N_P for the additional entry, replace the old entry by the two new entries in N_P and exit. Otherwise, with N replaced by N_P and the current level replaced by its parent level, go to step 3 to determine whether reinsertion or a split should be applied to handle the overflow. A split may be propagated all the way up to the root node. If the root node needs to be split, then a new root node will be created and the height of the R*-tree will be increased by 1.

Finally, we show a general classification of the multidimensional index structures based on MBRs (R-tree family). As we know, the indexes of the R-tree family belong to the category of data-driven access methods, since their structure adapts itself to the MBRs distribution in the space (i.e. the partitioning adapts to the object distribution in the embedding space). In contrast, the space-driven access methods (Grid File [NHS84], K-D-B-Tree [Rob81], linear quadtree [Sam90a, Sam90b], space filling curves [OrM84], etc.) are based on partitioning of the embedding space into regular cells according to some geometric criterion, independently of the distribution of the objects in the space [RSV01]. Table 3.1 shows the more representative multidimensional index structures based on MBRs. As the most important properties we identified the shape of the page regions (mainly MBRs), disjointedness and the most relevant decisions in the construction and maintenance of the index. As shape of the MBRs and intersections between an MBR and a sphere. Disjointedness means that the regions are not allowed to overlap. This is only guaranteed in R^+-tree [SRF87]. The criteria for choosing a subtree in performing an insert operation are based on volume, volume enlargement, etc. When more than one criterion is mentioned, the first has the highest weight, subsequent criteria are only applied if the first criterion yields a tie. The next column in the table summarizes the criteria for the choice of the split axis and the split plane. The last information in the table is if the index structure tries to perform a forced reinsert operation before splitting a page. Moreover, if the completeness means that the whole data space is covered with page regions, all the multidimensional access methods based on MBR are not complete.

Name	Region	Disjoint.	Criteria for Insert	Criteria for Splitting	Reinsert
R-tree	MBR	No	Volume enlargement Volume	Various algorithms	No
R*-tree	MBR	No	Overlap enlargement Volume enlargement Volume	Surface area Overlap Dead space coverage	Yes
R^+-tree	MBR	Yes	Volume enlargement The input MBR may be added in more than	Various algorithms based on packing criteria	No

			one leaf node		
X-tree	MBR	No	Overlap enlargement Volume enlargement Volume	Split history Surface/overlap Dead space coverage	No
SR-tree	Intersection MBR/Sphere	No	Proximity to centroid	Variance	Yes

Table 3.1: Multidimensional index structures based on MBRs and their properties.

3.4 Metrics Between Pairs of MBRs

Since the different algorithms for K-CPQ act on pairs of R-trees (R_P and R_Q), some metrics between MBRs (that can be used to increase performance of the algorithms) will be defined. Let N_P and N_Q be two internal nodes of R_P and R_Q, respectively. Each of these nodes has an MBR that contains all the points that reside in the respective subtree. In order for this rectangle to be the minimum bounding one, at least one point is located at each edge of the rectangle. Let M_P and M_Q represent the MBRs of N_P and N_Q, respectively. Let r_1, r_2, r_3 and r_4 be the four edges of M_P and s_1, s_2, s_3 and s_4 be 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 metrics between two MBRs. In case M_P and M_Q are disjoint we can define a metric that expresses the minimum possible distance of two points contained in different MBRs:

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

In case the MBRs of the two nodes intersect MINMINDIST(M_P, M_Q) equals 0 (the minimum possible distance between two objects in $E^{(k)}$). In any case (intersecting or disjoint MBRs) we can define the metrics

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

and

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

where, MAXMAXDIST expresses the maximum possible distance of any two points contained in two different MBRs. MINMAXDIST expresses an upper bound of distance for 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 3.3, two MBRs (M_P and M_Q) and their MINMINDIST, MINMAXDIST and MAXMAXDIST distances in $E^{(2)}$ are depicted. Recall that at least one point is located on each edge of each MBR.

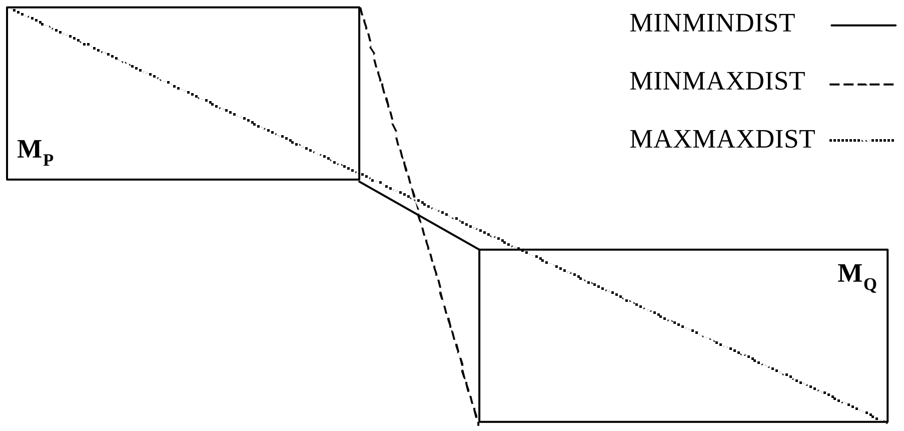


Figure 3.3: Two MBRs and their MINMINDIST, MINMAXDIST and MAXMAXDIST in $E^{(2)}$.

In the following we present algorithmic definitions of the above metrics (analogous definitions are presented in [RKV95, PaM97a], where metrics between a point and an MBR are provided) in $E^{(k)}$. Using these definitions, it is easy to devise efficient algorithms for calculating the metrics.

Let R(s, t) represent an MBR in $E^{(k)}$, where s = (s_1, s_2, ... , s_k) and t = (t_1, t_2, ... , t_k) ($s_i \le t_i$, for $1 \le i \le k$) are the endpoints of one of its major diagonals.

Definition 2
Given two MBRs $R_1(s, t)$ and $R_2(p, q)$ in $E^{(k)}$, MINMINDIST($R_1(s, t)$, $R_2(p, q)$) is defined as:

$$\sqrt{\sum_{i=1}^{k} y_i^2}$$

$$where, \qquad y_i = \begin{cases} p_i - t_i, & if\ p_i > t_i \\ s_i - q_i, & if\ s_i > q_i \\ 0, & otherwise \end{cases}$$

Proposition 1
The distance returned by MINMINDIST(R_1, R_2) is the minimum Euclidean distance from any point of the perimeter of R_1 to any point of the perimeter of R_2.

Proof:
The Euclidean distance from any point of the perimeter of R_1 to any point of the perimeter of R_2 is represented by the following formula:

$$dist(R_1, R_2) = \sqrt{\sum_{i=1}^{k} |a_i - b_i|^2}, \forall a_i, b_i : s_i \le a_i \le t_i \text{ and } p_i \le b_i \le q_i$$

and our target is to find the expression that minimizes dist(R_1, R_2), i.e. min{dist(R_1, R_2)}.
For proving this proposition we must study the behavior of dist(R_1, R_2) in each dimension *i* ($1 \le i \le k$), knowing whether R_1 and R_2 intersect or not. Two MBRs R_1(s, t) and R_2(p, q) in $E^{(k)}$ intersect if, and only if, exits at least on dimension *i* such that $s_i \le q_i$ and $p_i \le t_i$. On the other hand, two MBRs R_1(s, t) and R_2(p, q) in $E^{(k)}$ do not intersect if, and only if, $s_i > q_i$ or $p_i > t_i$ $\forall 1 \le i \le k$.
Case 1: R_1 and R_2 intersect in the dimension *i*.

The intersection condition of two MBRs is that $\exists 1 \leq i \leq k$: $s_i \leq q_i$ and $p_i \leq t_i$. Besides, we know that any point in the perimeter of R_1 at the dimension *i* satisfies $s_i \leq a_i \leq t_i$, and any point in the perimeter of R_2 at the dimension *i* satisfies $p_i \leq b_i \leq q_i$. Thereby, we obtain the following intersection conditions: ($s_i \leq a_i \leq q_i$ or $p_i \leq a_i \leq t_i$) and ($s_i \leq b_i \leq q_i$ or $p_i \leq b_i \leq t_i$). And we can deduce that exists at least one pair (a_i, b_i) on the intersection set such that $a_i = b_i$, then $|a_i - b_i| = 0$. Thus, when the MBRs R_1 and R_2 intersect in the dimension *i*, then this dimension does not intervene in the computation of min{dist(R_1, R_2)} and its value is 0.
Case 2: R_1 and R_2 do not intersect in the dimension *i*.
The condition of no intersection between two MBRS is that $\forall 1 \leq i \leq k$: $s_i > q_i$ or $p_i > t_i$. Besides, we know that any point in the perimeter of R_1 at the dimension *i* satisfies $s_i \leq a_i \leq t_i$ and any point in the perimeter of R_2 at the dimension *i* satisfies $p_i \leq b_i \leq q_i$. Thereby, we obtain the following no intersection conditions: ($q_i < s_i \leq a_i$ or $a_i \leq t_i < p_i$) and ($b_i \leq q_i < s_i$ or $t_i < p_i \leq b_i$). And we can deduce that for all dimensions ($1 \leq i \leq k$) $a_i > b_i$ or $b_i > a_i$, then $|a_i - b_i| > 0$. Thus, when the MBRs R_1 and R_2 do not intersect in the dimension *i*, then this dimension takes part in the computation of min{dist(R_1, R_2)} and its value is min{$|a_i - b_i|$}:

- If $s_i > q_i$, the min{dist(R_1, R_2)} = min{$|a_i - b_i|$} = $|s_i - q_i|$.
- If $p_i > t_i$, the min{dist(R_1, R_2)} = min{$|a_i - b_i|$} = $|t_i - p_i|$.

From the previous results we can conclude that min{dist(R_1, R_2)} = MINMINDIST(R_1, R_2). □

Three interesting metric properties related to MINMINDIST distance function can be enunciated as the following lemmas. The first one is called *MINMINDIST property.*

Lemma 1 MINMINDIST property
Given two MBRs $R_1(s, t)$ and $R_2(p, q)$ in $E^{(k)}$.
(1) If $t_i = s_i$ for $R_1(s, t)$ and $q_i = p_i$ for $R_2(p, q)$, then R_1 and R_2 degenerate into two points $s = (s_1, s_2, \ldots, s_k)$ and $p = (p_1, p_2, \ldots, p_k)$; then

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

(2) If $q_i = p_i$ for $R_2(p, q)$, then R_2 degenerates into a point $p = (p_1, p_2, \ldots, p_k)$; then.

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

In the same way, we can say: if $t_i = s_i$ for $R_1(s, t)$, then R_1 degenerating into a point $s = (s_1, s_2, \ldots, s_k)$; then MINMINDIST($R_1$, R_2) = MINDIST(s, R_2) [RKV95].

Proof:
Immediate from the definitions of MINMINDIST between two MBRs, the Euclidean distance function between two points and MINDIST(R, p) from the point p to an MBR R(s, t) [RKV95]. □

This property means the MINMINDIST of two MBRs is a generalization of the minimum distance between points and MBRs, since when those degenerate into two points is completely equivalent to the distance between two points, when one of the MBRs degenerates into a point is completely equivalent to MINDIST(R, p) from the point p to an MBR R(s, t) [RKV95]. This property allows us to apply MINMINDIST to pairs of any kind of element

(MBRs or points) stored in R-trees during the computation of branch-and-bound algorithms for K-CPQ.

The second metric property of MINMINDIST distance function is based on the *dimension distance property*, and it can be enunciated as the following lemma:

Lemma 2 Dimension MINMINDIST property
Given two MBRs $R_1(s, t)$ and $R_2(p, q)$ in $E^{(k)}$, then the value of MINMINDIST(R_1, R_2) for a given dimension 1£i£k is always smaller than or equal to the total computation of MINMINDIST(R_1, R_2).

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

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

Proof:
Immediate from the definition of MINMINDIST between two MBRs and the *dimension distance property.* □

The main usefulness of this property is due to the fact that its computation is cheaper than the MINMINDIST one and we may obtain performance gain in some situations (e.g. plane-sweep technique [PrS85]) for a given dimension. Figure 3.4 illustrates this property.

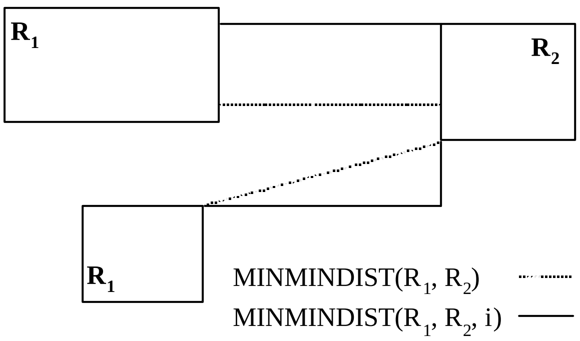


Figure 3.4: Dimension MINMINDIST property in $E^{(2)}$.

Another important metric property of the MBRs stored in two different R-trees related to the MINMINDIST distance function is called *MBRs MINMINDIST property.* This property can be enunciated as the following lemma:

Lemma 3 MBRs MINMINDIST property
Given two R-tree internal nodes N_P and N_Q (with MBRs M_{P0} and M_{Q0}) of two R-tree indices R_P and R_Q, respectively. These two internal nodes are enclosing two sets of MBRs {M_{P1}, M_{P2}, …, M_{PA}} and {M_{Q1}, M_{Q2}, …, M_{QB}}. Then

MINMINDIST(M_{Pi}, M_{Qj}) ³ MINMINDIST(M_{P0}, M_{Q0}): "1£i£A and "1£j£B
MINMINDIST(M_{P0}, M_{Qj}) ³ MINMINDIST(M_{P0}, M_{Q0}): "1£j£B
MINMINDIST(M_{Pi}, M_{Q0}) ³ MINMINDIST(M_{P0}, M_{Q0}): "1£i£A

Proof:
Immediate from the definition of MINMINDIST between two MBRs and the *MBR enclosure property* in R-trees. □

This property means 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}. This property allows us to limit the search space when we apply a branch-and-bound algorithm for K-CPQ. Figure 3.5 illustrates this property for A = B = 3.

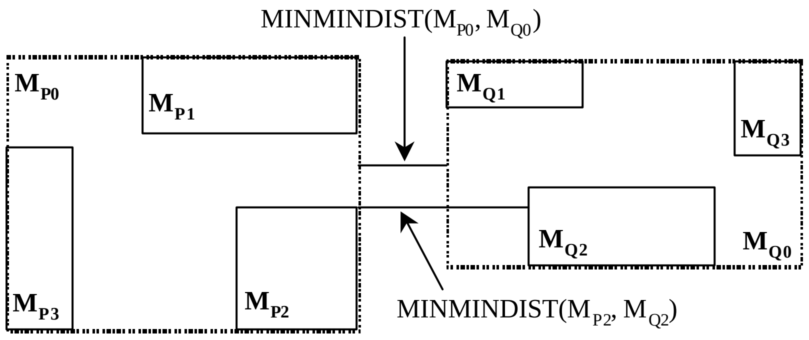


Figure 3.5: MBRs MINMINDIST property in $E^{(2)}$.

Definition 3

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

$$\sqrt{\sum_{i=1}^{k} y_i^2}$$

$$\textit{where,} \quad y_i = \max\{|s_i - q_i|, |t_i - p_i|\}$$

Proposition 2

The distance returned by MAXMAXDIST(R_1, R_2) is the maximum Euclidean distance from any point of the perimeter of R_1 to any point of the perimeter of R_2.

Proof:
Reasoning similar to proposition 1. □

Definition 4

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

$$\min_{1\le i\le k, 1\le j\le k} \left\{ \begin{array}{l} MAXDIST\left(F(R_1(s,t), i, s_i), F(R_2(p,q), j, p_j)\right), \\ MAXDIST\left(F(R_1(s,t), i, t_i), F(R_2(p,q), j, p_j)\right), \\ MAXDIST\left(F(R_1(s,t), i, s_i), F(R_2(p,q), j, q_j)\right), \\ MAXDIST\left(F(R_1(s,t), i, t_i), F(R_2(p,q), j, q_j)\right), \end{array} \right\}$$

*where a notation of the form **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 vertical to dimension **i** at value **x_i** (note that, for an MBR R(z, x), there are two faces vertical 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 the one face to each endpoint of the other face. Each face of dimension k has 2^{k-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** ¹ **i**.*

Note that definitions 2 and 3 lead to algorithms of O(k) time, while definition 4 to an algorithm of O(2^k) time. For small values of k (for example, k < 4) the cost of using definition 4 is not prohibitive. For larger values of k, an alternative definition could be used which gives an upper bound for the value produced by definition 4. This definition is presented in the following and leads to an algorithm of O(k) time. For each dimension j (1≤j≤k), it computes the minimum of the MAXDIST values of all the pairs of faces vertical 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-minimums). The computed value, in general, is larger than or equal to (an upper bound of) the minimum of the MAXDIST values of every possible pair of faces.

Definition 5

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

$$\sqrt{\min_{1\le j\le k}\left\{x_j^2 + \sum_{i=1, i\ne j}^{k} y_i^2\right\}}$$

$$\textit{where, } x_j = \min\left\{\left|s_j - p_j\right|, \left|s_j - q_j\right|, \left|t_j - p_j\right|, \left|t_j - q_j\right|\right\} \textit{ and}$$

$$y_i = \max\left\{\left| s_i - q_i\right|, \left| t_i - p_i\right|\right\}$$

Proposition 3

The distance returned by MINMAXDIST(R_1, R_2) is the minimum value of all maximum point distances between the closest parallel faces of the MBRs R_1 and R_2.

Proof: Reasoning similar to propositions 1 and 2. □

Definition 6

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

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

where $F(o_1)$ and $F(o_2)$ denote the set of faces of the object o_1 and o_2 in $E^{(k)}$, 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 $E^{(k)}$.

Theorem 1

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

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

Proof:

Immediate from the definition of MINMINDIST between two MBRs and the *MBR face property.* □

Theorem 2

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

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

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

Theorem 3
Given two MBRs $M_{P0}(s, t)$ and $M_{Q0}(p, q)$ in $E^{(k)}$, enclosing two set of objects O_1 = {o_{1i}: 1£i£N_1} and O_2 = {o_{2j}: 1£j£N_2}, respectively. The following is true:

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

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

From the previous metrics properties and theorems, we can deduce that MINMINDIST(R_1, R_2) and MAXMAXDIST(R_1, R_2) serve as lower and upper bounding function respectively, of the Euclidean distance from the K closest pairs of objects within the MBRs R_1 and R_2. And 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.

As in [HjS98, HjS99], as long as the distance functions are “consistent”, the algorithms based on them will work correctly. Informally, consistency means that no pair can have a smaller distance than a pair that we access during the processing of an algorithm over tree-like structures [HjS98, HjS99]. This constraint is clearly held by the Theorem 1, the *MBR MINMINDIST and MBR face properties*, 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.

Chapter 4

4 Algorithms for K-Closest Pairs Queries

In the following, a general branch-and-bound algorithmic approach to obtain the K optimal solutions of given problem Π is proposed. Moreover, based on distance functions between two MBRs, we present a pruning heuristic and two updating strategies for minimizing the pruning distance, in order to use them in the design 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 policy and one iterative following a Best-First traversal policy) for K-CPQ between spatial objects stored in two R-trees are presented. The plane-sweep method and the search ordering are used as optimization techniques for improving the naive approaches. Since the height of an R-tree depends on the number of spatial objects inserted (as well as in the order of insertions 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, a number of interesting extensions of the K-CPQ algorithms are shown.

4.1 Branch-and-bound algorithms

Branch-and-bound is a general technique for the design of algorithms [HoS78, BrB95]. This technique is used to traverse tree-like structures in order to find an optimal (minimal) solution to a given problem. An optimization (minimization) problem is generally defined as follows:

Π: minimize $f(x)$
subject to $x \in W$

where $W \subseteq X$ denotes a feasible region in the underlying space X. Namely, W is the set of feasible solutions satisfying the imposed constraints. The function $f: X \rightarrow \Re$, where $\Re$ is the set of real numbers, is called objective function. A feasible solution $x \in W$ is optimal (minimal) if no other feasible solution $y \in W$ satisfies $f(y) < f(x)$.

The underlying idea of branch-and-bound algorithm is to partition a given initial problem into a number of intermediate partial subproblems of smaller sizes. Every subproblem is characterized by properties related to the initial problem. The decomposition is repeatedly applied to the generated subproblems until each unexamined subproblem is decomposed, solved, or shown not to lead to an optimal solution to the original problem. Branch-and-bound is essentially a variant, or refinement, of backtracking technique that can take advantage of information about the optimality of partial solutions to avoid considering solutions that can not be optimal, hence reduce the search space significantly.

The notation in [Iba87] is commonly employed to formally define a branch-and-bound algorithm that will be needed in the sequel. Let Π denote an optimization problem and f denote the function to be minimized. The decomposition process applied to Π is represented by a tree B=(N, E), called branching tree, where N is a set of active nodes and E is a set of arcs. At the beginning, the root of B, denoted N_0, corresponds to the initial problem Π. During the processing of the algorithm, other active nodes N_i correspond to partial subproblems Π_i of

Π. The arc $(N_i, N_j) \in E$ if, and only if, N_j is generated from N_i by decomposition. The set of terminal active nodes of B, denoted T, are those partial subproblems that are solved without further decomposition. The level of $N_i \in B$, denoted by Level(N_i), is the length of the path from the root of B to N_i in B.

A branch-and-bound algorithm attempts to solve Π by examining only the minimum possible portion of N. It is accomplished by proceeding along the branches rooted at those active nodes N_i that are either solved or not yield an optimal solution of Π. A lower bounding function $g:N \rightarrow \Re$ is calculated for each active node when it is created in order to help eliminate unnecessary search. The lower bound represents the optimum (minimum) value of $f(N_i)$ for a partial subproblem Π_i, i.e. $g(N_i) \leq f(N_i)$ such that $N_i \in N$. The values of this lower bounding function satisfy the following properties, guaranteeing its consistency:

$g(N_i) \leq f(N_i)$	for $N_i \in N$,
$g(N_i) = f(N_i)$	for $N_i \in T$,
$g(N_i) \leq g(N_j)$	for $(N_i, N_j) \in E$ (N_j is a descendent of N_i by decomposition).

On the other hand, upper bounding functions $u:N \rightarrow \Re$ are calculated for each active node in order to try to update the value of the best feasible solution found so far (z). If good upper bounds are generated in the early stage of the computation of a branch-and-bound algorithm, and z is thereby set to relatively small values, the lower bound test would become powerful. The values of these upper bounding functions satisfy the following properties, guaranteeing its consistency:

$u(N_i) \geq f(N_i)$	for $N_i \in N$,
$u(N_i) = f(N_i)$	for $N_i \in T$.

It is not always assumed the $u(N_i)$ is computed for all the generated active nodes N_i. $u(N_i)$ is set to ∞ if the computation is not attempted for any N_i or a good bound is not found within the allotted computation time (i.e. $u(N_i)$ is computed for the initial active node ($u(N_0)$)) or a fixed number of initial active nodes at the beginning).

A typical branch-and-bound algorithm consists in four essential procedures: selection, branching, pruning and termination test.

Selection. At any step during the execution of the algorithm there exists a set of active nodes (N in the branching tree B) that have been generated but not yet examined. The selection procedure chooses a single active node from N, based on a selection function *s* applied to N. The following three searching policies are commonly used as selection functions:

Best-First search:	$s(N) = N_i$, such that $g(N_i)$ is the smallest lower bound,
Depth-First search:	$s(N) = N_i$, such that N_i is the active node with the largest depth,
Breadth-First search:	$s(N) = N_i$, such that N_i is the active node with the smallest depth.

In the selection procedure, the Best-First (Best-Bound) and Depth-First searching policies are used in most situations. Best-First search minimizes the number of partial subproblems decomposed before termination. However, it tends to consume an amount of requirements that is an exponential function of the problem size [Iba87], because the number of active nodes in the branching tree increases exponentially with the depth of the trees and the search tree grows in the shape an umbrella. It is attractive from the viewpoint of the computing time, because the total time is roughly proportional to the number of active nodes decomposed. Moreover, this searching policy has an important disadvantage, since in some cases the optimal solution is reached almost at the end of the entire computation of the branching tree. On the other hand, Depth-First search consumes an amount of space that is only a linear function of the height of the tree [Iba87], and its implementation is relatively easy, because we can use of the recursion in its implementation. In this traversal policy, the search tree consists in one active path from the root to a leaf with length equal to the height of the tree, and we have in memory one active path for each tree. A disadvantage of this searching policy is that it tends to consume time to exit, once it deviates of the branch where no optimal solution of the initial problem is located, therefore the number of active nodes decomposed from the branching tree is usually larger than those realized by the Best-First searching policy. However, as Depth-First search puts higher priority to those active nodes in larger depth, an “approximate” solution (although it may not be optimal) is usually available even if the computation is stopped before the normal termination.

Branching. A specific branching rule is used to generate new smaller subproblems from one selected by the selection procedure. Lower bounds for the new generated subproblems are calculated accordingly.

Pruning. A new created subproblem is deleted if its lower bound is greater than or equal to the value of the best feasible solution discovered so far (z).

Termination Test. The branch-and-bound algorithm terminates when the branching tree is empty or the value of the best feasible solution discovered so far is the optimal solution of the original problem.

Following the terminology proposed in [Iba87], we present the general description of the branch-and-bound algorithm to obtain the K optimal solutions of a given optimization (minimization) problem Π (K is smaller than or equal to all possible solutions of Π). The initial problem Π is characterized by the active node N_0 (the root of B). Moreover, $Opt_K = \{N_i \in T: f(N_i) \leq z\}$ stores the K optimal solutions ($Opt_K \subseteq T$) and all of them can not have the same value, z is the value of the K-th best feasible solution discovered so far, and s(N) selects the best active node N_i depending on the searching policy.

```
Algorithm Branch-and-Bound A = (B, T, f, g, u, s, Opt_K, z): K optimal solutions of Π.
A1 (initialization):
    $N = \{N_0\}$; $z = \infty$; $Opt_K = \emptyset$;
A2 (search and extract candidate):
    If $N = \emptyset$, Go to A8. Otherwise, $N_i = s(N)$; $N = N - \{N_i\}$;
A3 (update z based on upper bounding functions):
    If $u(N_i) < z$, $z = u(N_i)$;
A4 (T-test):
    If $N_i \in T$,
        If $f(N_i) \leq z$ and $|Opt_K| < K$,  $Opt_K = Opt_K \cup \{N_i\}$; $z = \max\{f(N_i): N_i \in Opt_K\}$;
        If $f(N_i) \leq z$ and $|Opt_K| = K$,  Remove $N_j \in Opt_K$, such that $f(N_j)$ is the maximum value;
                                               $Opt_K = Opt_K \cup \{N_i\}$; $z = \max\{f(N_i): N_i \in Opt_K\}$;
        If $f(N_i) > z$, $Opt_K = Opt_K$;
        Go to A2;
A5 (lower bound test):
    If $g(N_i) > z$, Go to A8;
A6 (branching):
    Decompose $N_i$ into a set of new active nodes $N_{i1}, N_{i2}, \ldots, N_{ij}$;
A7 (pruning):
    For every new active node $N_{ix}$ $1 \leq x \leq j$, If $g(N_{ix}) \leq z$, $N = N \cup \{N_{ix}\}$;
    Go to A2;
A8 (termination):
    Stop. If $z = \infty$, Π is unfeasible. If $z < \infty$, $Opt_K$ stores the K optimal solutions of Π.
```

Theorem 4 Finiteness and Correctness of branch-and-bound algorithm A to obtain the K optimal solutions
Under assumptions that branching tree B has a finite number of active nodes and each of the steps A1-A8 requires a finite computation time, the branch-and-bound algorithm A terminates in A8 in finite computation time. Upon termination, the value of z is equal to the maximum $f(N_i)$ such that N_i Î Opt_K and Opt_K stores the K optimal solutions, if P is feasible. If P is unfeasible, z = ¥ and Opt_K = Æ.

Proof:
The finite termination (finiteness) of A is an obvious consequence from the assumptions (B has a finite number of active nodes).
To prove the correctness, we first suppose that Π is feasible, i.e. Π has K optimal solutions stored in Opt_K and $z = \max\{f(N_i): N_i \in Opt_K\} < \infty$. Hence, N_i can be decomposed in $N_{i1}N_{i2}\ldots N_{ij}$ (A6), and this process can be repeatedly applied until $N_i \in T$. During the execution of the algorithm a decomposed $N_i \in N$ ($N \neq \varnothing$) is selected in A2 for the test.
If $N_i \in T$ and any ancestor of N_i is not discarded, then N_i is terminated and included in Opt_K if $f(N_i) \leq z$ in A4. If $N_i \notin T$, then the z value can be optionally updated by A3 using the upper bounding function u, and the processing of N_i continues. As $N_i \notin T$, if the lower bound test fails, the active node is decomposed in A6 and some descendent will be pruned in A7 if $g(N_i) > z$ and the selection process is started again (A2). On the other hand, $N_i \notin T$ can be terminated by the lower bound test in A5 ($g(N_i) > z$), and the algorithm concludes with $z = \max\{f(N_i): N_i \in Opt_K\}$ and storing in Opt_K the K optimal solutions.
The case of Π is unfeasible, i.e. $z = \infty$ and $Opt_K = \varnothing$, can be similarity treated and the details are omitted. □

The computational behavior of a branch-and-bound algorithm is highly dependent on its searching policy. First, we are going to introduce a lemma showing a basic property of Best-First searching policy. Then, based on it, we will present a theorem in order to show that Best-First is the best possible searching policy in the sense of minimizing *Total(A)* ≡ total number of active nodes decomposed before the termination of the branch-and-bound algorithm A. Also, we propose another theorem that proves the required memory space of the algorithm A, *M(A)*, for this traversal policy.

Lemma 4 Property of Best-First Searching Policy
Let A be the branch-and-bound algorithm using Best-First searching policy, which finds the K optimal solutions. Let U = $N_{j1}N_{j2}\ldots N_{jt}$ be the sequence of active nodes arranged in the order tested by A. Then $g(N_{j1})$ £ $g(N_{j2})$ £ … £ $g(N_{jt})$.

Proof:
We prove the lemma of an arbitrary *i* of $U_i = N_{j1}N_{j2}\ldots N_{ji}$ by induction.

1. For *i* = 1, $g(N_{j1}) \leq g(N_{j1})$ is trivially true.
2. We suppose that it is true for *i* and it must be satisfied for *i + 1.* Let Active(N_{ji}) the set of active nodes in A at the time of selection N_{ji} in the step A2. Let Descendent(N_{ji}) denote the set of active nodes in A that have ancestors in Active(N_{ji}) at the time of selection N_{ji} in the step A2. Namely, Descendent(N_{ji}) is the set of active nodes that will become active in the future (after a process of branching). Besides, the selection function of Best-First searching policy is g ($s \equiv g$) and it satisfies $g(N_j) \leq g(N_k)$, for $(N_j, N_k) \in E$, where N_k is a descendent of N_j by decomposition. Thereby, $g(N_{ji}) \leq g(N_k)$, for

any $N_k \in$ (Active(N_{ji}) $\cup$ Descendent(N_{ji})). Thus, it proves $g(N_{j1}) \leq g(N_{j2}) \leq \ldots \leq g(N_{ji}) \leq g(N_{j(i+1)})$ and completes the induction. □

Theorem 5 Optimality of Best-First Searching Policy with respect to *Total*
Let A be the branch-and-bound algorithm using Best-First searching policy, which finds the K optimal solutions. This A satisfies Total(A) £ |F – T|, where F is the set of active nodes (non-terminal and terminal) of B that represents the number of partial subproblems decomposed before the end of the branch-and-bound algorithm, F = {N_i Î N: g(N_i) £ z}.

Proof:
We show that any active node N_i decomposed in A using Best-First searching policy satisfies $N_i \in F - T$.
As any $N_i \in T$ is terminated (accepted or discarded) by T-test (A4), it is enough to show that such $N_i \notin F$, since it is not decomposed. Moreover, recall that there is a $N_j \in T$ with $f(N_j) \leq z$. As any ancestor N_k of such N_j satisfies $g(N_k) \leq g(N_j) = f(N_j) \leq z$, then no such N_k is terminated. By the lemma 4, then such N_k is tested before any $N_j \notin F$ (N_k is tested in A5 (lower bound test) and terminated if $g(N_k) > z$).
Another situation is when $N_i \notin T$ and z is updated based on the upper bounding function in A3. In this case, again, some active node in N do not satisfy $g(N_j) > z'$, hence $N_i \in F$.
If $N_i \notin T$ and this active node is not terminated by A5 (lower bound test) since $g(N_i) \leq z$, then $N_i \in F$ and, it will be decomposed in A6 and all its descendent N_{ix} and they will be pruned in A7 according to $g(N_{ix}) \leq z$.
Finally, when $N_i \notin T$ and N_i is terminated by A5 since $g(N_i) > z$, then $N_i \in F$ and by the lemma 4 there is not any other active node $N_j \in N$ that satisfies $g(N_j) \leq z$ and the algorithm stops. For this reason, Best-First is the searching policy with the minimum decomposed actives nodes *(Total)*, since it takes into account the lower bounding function (g) as selection function and the minimization of z value. □

Theorem 6 Required memory space of Best-First Searching Policy
Let A be the branch-and-bound algorithm using Best-First searching policy, which finds the K optimal solutions. This branch-and-bound algorithm satisfies M(A) £ |F + T|.

Proof:
By the lemma 4, the algorithm A always chooses in the selection step the active node N_i with the minimum g value ($g(N_i)$) in the step A2 (because the selection function ($s \equiv g$) in Best-First traversal policy is g).
In the first testing step (T-test in A4), any terminal active node $N_i \in T$ is inserted in Opt_K ($f(N_i) \leq z$) or discarded ($f(N_i) > z$).
When the T-test and lower bound test are failed, even if all $N_i \in F$ are decomposed in the step A6, M(A) is bounded by the size of F (active nodes N_i that satisfies $g(N_i) \leq z$) plus the terminal active nodes N_i that satisfies $f(N_i) \leq z$. □

The last two theorems show us that Best-First searching policy is optimal with respect to the total number of active nodes decomposed before the termination of the branch-and-bound algorithm A, but the required memory space depends strongly on the z value. It means, if we apply an appropriate g lower bounding function in order to update z and an effective pruning heuristic, *M(A)* will be reduced considerably.

The branch-and-bound algorithms (recursive and iterative) of this thesis use the previous branch-and-bound technique. In our case, $\Pi \equiv$ obtain the K closest pairs of spatial objects from to spatial datasets stored in both R-trees (R_P and R_Q), f ≡ Euclidean distance between two spatial objects, g ≡ minimum Euclidean distance between two R-tree elements, u ≡ maximum Euclidean distance between two R-tree elements, and at the beginning, the level of N_0 is max{height(R_P), height(R_Q)}. For example, our iterative branch-and-bound algorithm consists of three steps.

In the first step, the value of the K-th best feasible solution discovered so far is set up ($z=\infty$). The branching tree is a minimum binary heap [CLR90] with a key that is equal to the lower bounding function (g), i.e. on the top of the binary heap will be the smallest lower bound.

In the second step, the four procedures described earlier should be established:
Selection: The selection procedure simply chooses the partial solution in the root of the heap, i.e. we pick the active node (candidate pair of R-tree elements) with the minimum key value.
Branching: The branching procedure creates a set of candidate pairs C from the combination of two R-tree nodes. The key of these candidate pairs is the value returned by the lower bounding function (minimum distance between two R-tree elements for internal nodes and the minimum distance between two spatial objects for leaf nodes). The cardinality of C verifies the following inequality: $m_P*m_Q \leq |C| \leq M_P*M_Q$, where ($m_P$, M_P) and (m_Q, M_Q) are the fan-out of R_P and R_Q, respectively.
Pruning: The pruning procedure compares the lower bounds of the new candidate pairs to the value of the K-th best feasible solution discovered so far (z) and deletes from the set C all candidate pairs if its key is greater that z, i.e. $g(C_i) > z$. The remainder candidate pairs of C ($g(C_i) \leq z$) are inserted in the minimum binary heap.
Termination Test: The algorithm stops if, and only if the minimum binary heap is empty or g(root(heap) > z).
In the third step, the value of z is equal to the maximum $f(N_i)$ such that $N_i \in Opt_K$, and Opt_K stores the K optimal solutions if Π is feasible. If Π is unfeasible, $z = \infty$ and $Opt_K = \emptyset$.

For our recursive branch-and-bound algorithm, we will have very similar steps if we do not use the recursion provided by the programming system (the cost of the recursive implementation is borne by the mechanism in the programming systems that support function calls, which use the equivalent of a built-in pushdown stack). The main difference with respect to the iterative branch-and-bound algorithm is how to organize the branching tree of active nodes. In this case, instead of using a LIFO stack we can use another minimum binary heap with key is equal to the smallest level of an active node (or the greatest depth).

In the description of our branch-and-bound algorithms (recursive and iterative) for K-CPQ we can apply optimization rules for reducing the number of candidate pairs in the Branching procedure that they will be inserted in the branching tree during the execution of the algorithm. For example, we can use the plane-sweep technique [PrS85] in order to reduce the number of candidate pairs for inserting in the branching tree. Moreover, the minimum binary heap (branching tree) can contain candidate pairs of entries corresponding to leaf nodes as well as internal nodes of the R-trees, thereby it can grow with the depth of the R-trees. For this reason, we can avoid this situation, separating the treatment of T (terminal active nodes) and N – T (non-terminal active nodes) elements in the Pruning step and insert in N only the non-terminal elements. In the previous general description of the branch-and-bound algorithm, the T-test (A4) should be included in the pruning step (A7) with the new decomposed active nodes, for avoiding the insertion of terminal active nodes in N and insert them in Opt_K, if it is possible. For example, with this variant for Best-First searching policy, the required memory space of the branch-and-bound algorithm A will be reduced, *M(A)* ≤ K + |F|.

In [HjS98] was proposed an interesting incremental distance join using backtracking technique, and it was improved using pruning heuristics (branch-and-bound). The technical details proposed in [HjS98] were enhanced in [SML00] by using adaptive multi-stage techniques, plane-sweep algorithm [PrS85] and other improvements based on sweeping axis and sweeping direction. The modifications on the general description of the previous branch-and-bound algorithmic schema for obtaining all possible solutions incrementally are the following:

1. Remove the Opt_K set.
2. Remove the steps A3 (update z based on upper bounding functions) and A5 (lower bound test).
3. Modify the step A4 (T-test). *If N_i Î T, return N_i.*
4. In the step A6 (branching) is necessary to consider three different branching methods (Basic, Evenly and Simultaneous).
5. In A7 (pruning) is necessary to insert in N all possible new active nodes. *For every new active node N_{ix} 1£x£j, N = N È {N_{ix}}.*
6. Best-first searching policy is used. The lower bounding function g ≡ minimum distance between two R-tree elements is employed.

If we want to obtain the K optimal solutions of Π in incremental manner (applying branch-and-bound technique), we must consider the step A3 for minimizing z based on an upper bounding function (u ≡ maximum distance between two R-trees elements) and stop when the number of reported pairs is equal to K.

In the next subsections we are going to design branch-and-bound algorithms for K-CPQ, following the branch-and-bound general schema in the algorithm A with the best searching policies (Depth-Fist and Best-First). Moreover, in both searches we are going to apply the plane-sweep technique [PrS85] in order to reduce the number of candidate pairs for processing in the query algorithm. Also, in the Depth-First searching policy we are going to make use of the calculated lower bounds, and sort them in ascending order for choosing the best candidate pair for processing in the same depth.

4.2 Pruning Heuristic and Updating Strategies

Based on the previous bounding functions, theorems and the general branch-and-bound algorithm A, we can 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) that are used in the pruning process.

First of all, we establish a data structure that supports the set Opt_K, which stores the K optimal solutions. This data structure will help us to update z, which is the value of the K-th best feasible solution discovered so far, when the algorithm processes terminal active nodes on the branching tree. This structure is organized as a maximum binary heap (called K-heap) and will hold pairs of spatial objects according to their distance. The pair of spatial objects with the largest distance resides on top of K-heap (on the root of the maximum binary heap). In the implementation of the branch-and-bound algorithms for K-CPQ we must consider the following conditions:

- Initially the K-heap is empty (z is initialized to ∞).
- The pairs of spatial objects discovered in the T-test are inserted in the K-heap until it gets full (z is maintained to ∞).
- Then, when a new pair of spatial objects is discovered in the T-test, and if its distance is smaller than the top of the K-heap, then the top is deleted and this new pair is inserted in the K-heap (z is equal to the distance of the pair of spatial objects located on the top of K-heap).

4.2.1 Pruning Heuristic

Given two MBRs M_{Pi} and M_{Qj} in $E^{(k)}$, stored in nodes of two R-tree indices R_P and R_Q, respectively. If MINMINDIST(M_{Pi}, M_{Qj}) > z, then the pair of MBRs (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, or it can be optionally updated using the upper bounding functions (MINMAXDIST and MAXMAXDIST).

In Figure 4.1 two R-tree nodes (dashed rectangles) containing two MBRs (thick-line rectangles) and the MINMINDIST distances (thin lines) 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.

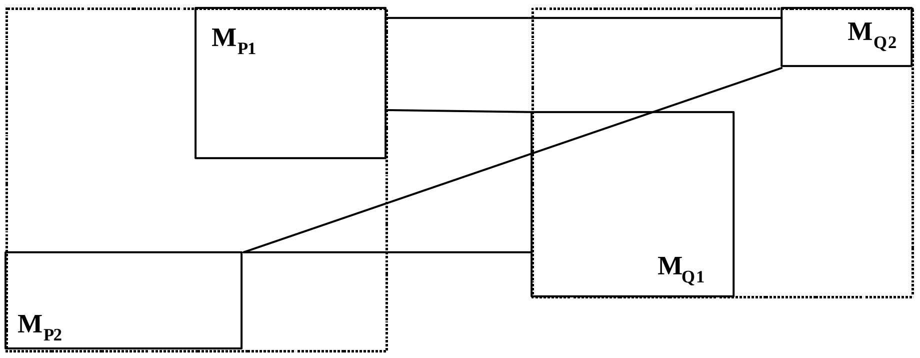


Figure 4.1: Two R-tree nodes and the pruning heuristic using MINMINDIST(M_{Pi}, M_{Qj}).

Moreover, we know the z value can be updated using the upper bounding functions MAXMAXDIST(R_1, R_2) and MINMAXDIST(R_1, R_2) for any K and K = 1, respectively. Although, if we apply these functions, the number of disk accesses will not be reduced and the computational cost can be increased, as in [ChF98] was proved for K-NNQ. Thereby, we can optionally use the following updating strategies based on upper bounding functions for updating z (trying to minimize its value, if it is possible).

4.2.2 Updating Strategy 1 (based on MINMAXDIST)

This first updating strategy can be done by making use of Theorem 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 R-tree indices 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 (i.e. if z' < z, then z = z'), where z' arises as follows:

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

In Figure 4.2 we have the same two R-tree nodes as in Figure 4.1, and in this case the MINMAXDIST distances between each pair of MBRs are depicted. 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.

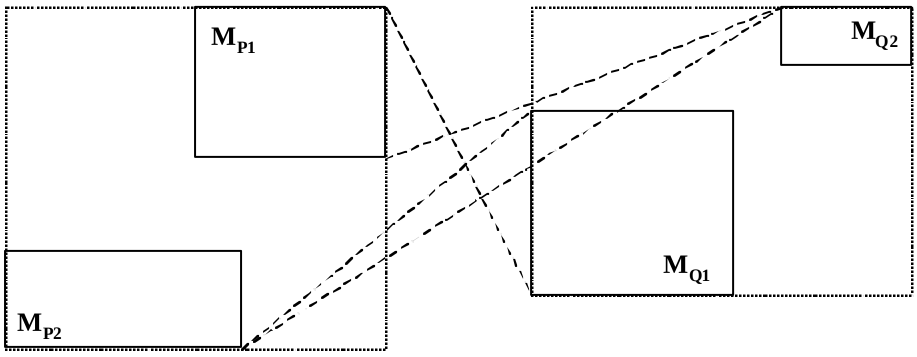


Figure 4.2: Two R-tree nodes and the updating strategy using MINMAXDIST(M_{Pi}, M_{Qj}).

4.2.3 Updating Strategy 2 (based on MAXMAXDIST)

This second updating strategy can be done by making use of Theorem 2 for any K. That is, given two R-tree nodes N_P and N_Q stored in internal nodes of R-tree indices R_P and R_Q, and enclosing two sets of MBRs {M_{Pi}: 1≤i≤|N_P|} and {M_{Qj}: 1≤j≤|N_Q|}, respectively. Then, z can be updated if, and only if z' has a smaller value (i.e. if z' < z, then z = z'), 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 (number of pairs is Total = $|N_P|\cdot|N_Q|$). 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_P, M_Q) stored in internal nodes is X(M_P, M_Q), where m_P and m_Q are the minimum fan-outs of R_P and R_Q, respectively.

$$X(M_P, M_Q) = m_P^{\text{level of } M_P} \times m_Q^{\text{level of } M_Q}$$

- Using X(M_P, M_Q), we can find the i-th element of the sorted list MxMxDList, until the following condition is satisfied:

$$\left(\sum_{i=0}^{Total-1} X\left(MxMxDList[i].M_P, MxMxDList[i].M_Q\right) \right) \geq K$$

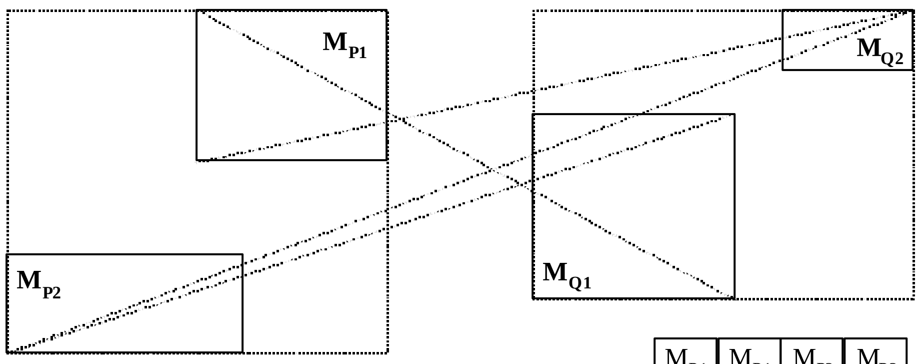
The following algorithm will return the i-th element of MxMxList that satisfies the previous condition:

```
i = 0; condition = false; coveringPairs = 0;
while (not(condition)) and (i < Total) do
    coveringPairs = coveringPairs + X(MxMxDList[i].$M_P$, MxMxDList[i].$M_Q$);
    if (coveringPairs $\geq$ K) then
        condition = true;
    else
        i = i + 1;
    endif
enddo
```

- Then, we can obtain z' = MxMxDList[i].MAXMAXDIST if (i < Total) is satisfied, otherwise (i = Total) z' = ∞. After that, we will update z, 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 policy.

In Figure 4.3 we have the same two R-tree nodes as in Figure 4.1, and in this case 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_P = level of M_Q = 1 (the level just before 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 update procedure works as follows: i = 0 [coveringPairs = 9 < 10; i = 1; continue;]; i = 1 [coveringPairs = 18 ≥ 10; stop;]; z' = MxMxDList[1].MAXMAXDIST = MAXMAXDIST(M_{P1}, M_{Q2}) = 9.70, and z = 9.70 because z < z' (9.70 < 15.35). But, if after this updating strategy we apply the pruning heuristic, then the algorithm will not prune any path, because MINMINDIST(M_{P2}, M_{Q2}) = 7.5 < z = 9.70.



	M_{P1}	M_{P1}	M_{P2}	M_{P2}
MxMxDList	M_{Q1}	M_{Q2}	M_{Q1}	M_{Q2}
	8.10	9.70	10.1	12.8
	0	1	2	3

Figure 4.3: Two R-tree nodes and the updating strategy using MAXMAXDIST(M_{Pi}, M_{Qj}).

However, for the iterative branch-and-bound algorithm following a Best-First searching policy, 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 $(S(X_i(M_P, M_Q)))$ 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 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 of the minimum number of spatial objects stored on the leaf nodes that can be generated by all pairs of MBRs ($\Sigma(X(M_P, M_Q)_i)$) stored in MxMxDHeap is larger than K, then we remove pairs of MBRs from MxMxDHeap until this sum is smaller than or equal to K, establishing 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, if z' < z holds.

After presenting these two updating strategies for minimizing the pruning distance (z), we must emphasize their use is optional (controlled by a parameter) at the beginning of the processing of the algorithm or a reduced number of candidate pairs of MBRs, because the cost of its computation is greater than the gain of updating z in order to apply at the pruning heuristic.

4.3 The Sorted Distances Recursive Algorithm

This first branch-and-bound algorithm follows a Depth-First searching policy making 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. 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 for two R-trees with the same height consists of the following steps.

KCPQ1 Start from the roots of the two R-trees and set z to ∞.

KCPQ2 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 those pairs of entries that have MINMINDIST $\leq$ z.

KCPQ3 If you access two leaf nodes, then calculate the distance of each possible pair of spatial 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 root of K-heap and insert the new pair in K-heap, updating z.

We must point out that on the leaf level of the R-tree an object (point or MBR) or MBR of another type of spatial 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 metric function returns the distance between two points if the two MBRs have degenerated in two points as we shown in the MINMINDIST property. In the second case, we must read the exact geometry of the pair of spatial objects (O_1, O_2) and calculate its distance **ObjectDistance**(O_1, O_2) as in [ChW83, ChW84, Ede85] for finding the minimum separation between two convex polygons or in [GJK88, LiC91] for convex objects.

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}). There may be ties between the values of MINMINDIST. That is, two or more pairs may have the same value of MINMINDIST. 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, but this special treatment needs time to carry out.

The pseudo-code of this algorithm, called **KCPQ_SortedDistancesRecursive**, is given as follows:

```
KCPQ_SortedDistancesRecursive(N_P, N_Q: R_node; K-heap: MaximumBinaryHeap_of_Pairs; z: Double)
01 if N_P and N_Q are INTERNAL nodes then
02     [Optionally try to update z using the Updating Strategies 1 (K = 1) or 2 for any K];
03     Total = N_P.numberOfEntries * N_Q.numberOfEntries;
04     Create_List(MINMINDISTLIST, Total);
05     i = 0; j = 0; t = 0;
06     for every entry e_i from N_P (i < N_P.numberOfEntries) do
07         for every entry e_j from N_Q (j < N_Q.numberOfEntries) do
08             MINMINDISTLIST[t].MINMINDIST = MINMINDIST(e_i.MBR, e_j.MBR);
09             MINMINDISTLIST[t].address_N_P = e_i.address;
10             MINMINDISTLIST[t].address_N_Q = e_j.address;
11             t = t + 1;
12         enddo
13     enddo
14     Sorting_List(MINMINDISTLIST, 0, (Total – 1)); t = 0;
15     while ((MINMINDISTLIST[t].MINMINDIST ≤ z) and (t < Total)) do
16         N_P' = ReadNode(MINMINDISTLIST[t].address_N_P);
17         N_Q' = ReadNode(MINMINDISTLIST[t].address_N_Q);
18         KCPQ_SortedDistancesRecursive(N_P', N_Q', K-heap, z);
19         t = t +1;
20     enddo
21     Destroy_List(MINMINDISTLIST);
22 else
23     i = 0; j = 0;
24     for every entry e_i from N_P (i < N_P.numberOfEntries) do
25         for every entry e_j from N_Q (j < N_Q.numberOfEntries) do
26             if e_i.address = NULL and e_j.address = NULL then
27                 dist = MINMINDIST(e_i.MBR, e_j.MBR);
28                 if K-heap.isFull() then
29                     if dist ≤ z then
30                         K-heap.deleteMaximum();
```

```
                        K-heap.insert(dist, $e_i$.MBR, $e_j$.MBR);
                        if K-heap.getMaximum() < z then
                            z = K-heap.getMaximum();
                        endif
                else
                    K-heap.insert(dist, $e_i$.MBR, $e_j$.MBR);
                endif
            else
                if MINMINDIST($e_i$.MBR, $e_j$.MBR) $\leq$ z then
                    $O_i$ = ReadObject($e_i$.address);
                    $O_j$ = ReadObject($e_j$.address);
                    dist = ObjectDistance($O_i$, $O_j$);
                    if K-heap.isFull() then
                        if dist $\leq$ z then
                            K-heap.deleteMaximum();
                            K-heap.insert(dist, $O_i$, $O_j$);
                            if K-heap.getMaximum() < z then
                                z = K-heap.getMaximum();
                            endif
                    else
                        K-heap.insert(dist, $O_i$, $O_j$);
                    endif
                endif
            endif
        enddo
    enddo
endif
```

In the previous algorithm, the two R-trees have the same height, and the spatial objects (points or MBRs) are stored directly in the leaf nodes of two R-trees (the pointers are NULL) or such objects are stored in two external files, in which case, the leaf nodes store the MBRs of the spatial objects and pointers to the exact geometric description of the object. The extensions to the other possibilities (the spatial objects are stored in the leaf nodes for one R-tree and in an external file for the another, and vice versa) is straightforward, and for R-trees with different heights, see subsection 4.6.

4.4 The Plane-Sweep Recursive Algorithm

Another improvement for a branch-and-bound algorithm making use of recursion (Depth-First traversal) is to exploit the spatial structure of the R-tree indices utilizing the plane-sweep technique [PrS85], in the same way as in the processing of spatial joins presented in [BKS93a].

Plane-sweep 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 for restricting all possible combinations of pairs of MBRs from two R-tree nodes $N_P = \{M_{Pi}: 1 \leq i \leq |N_P|\}$ and $N_Q = \{M_{Qj}: 1 \leq j \leq |N_Q|\}$ 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 in 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 (according to the choice of the sweeping direction [SML00]). Moreover, we must establish the dimension for the sweep-line (e.g. Sweeping_Dimension = 0 or X-axis) 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 we initialize **Pivot** 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 another R-tree node $\{M_{Qj}: 1 \leq j \leq |N_Q|\}$ from left to right that satisfies the MINMINDIST(**Pivot.MBR**, M_{Qj}, Sweeping_Dimension) $\leq$ 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 increasing 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. The pseudo code of the algorithm is given as follows:

```
PlaneSweep($N_P$, $N_Q$ : R_node; z: Double; ENTRIES: Set of Pairs of Entries)
01 Sorting_Node($N_P$, Sweeping_Dimension, Sweeping_Direction);
02 Sorting_Node($N_Q$, Sweeping_Dimension, Sweeping_Direction);
03 i = 0; j = 0;
04 while ((i < $N_P$.numberOfEntries) and (j < $N_Q$.numberOfEntries)) do
05     if ($N_P$.$e_i$.MBR.min[Sweeping_Dimension] < $N_Q$.$e_j$.MBR.min[Sweeping_Dimension]) then
06         k = j;
07         while (k < $N_Q$.numberOfEntries) do
08             if MINMINDIST($N_P$.$e_i$.MBR, $N_Q$.$e_k$.MBR, Sweeping_Dimension) ≤ z then
09                 ENTRIES.insert($N_P$.$e_i$, $N_Q$.$e_k$);
10             else
11                 break;
12             endif
13             k = k + 1;
14         enddo
15         i = i + 1;
16     else
17         k = i;
18         while (k < $N_P$.numberOfEntries) do
19             if MINMINDIST($N_P$.$e_k$.MBR, $N_Q$.$e_j$.MBR, Sweeping_Dimension) ≤ z then
20                 ENTRIES.insert($N_P$.$e_k$, $N_Q$.$e_j$);
21             else
22                 break;
23             endif
24             k = k + 1;
25         enddo
26         j = j + 1;
27     endif
28 enddo
```

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

For example, in Figure 4.4 we have a set of MBRs from two R-tree nodes ({M_{P1}, M_{P2}, M_{P3}, M_{P4}, M_{P5} and M_{P6}} and {M_{Q1}, M_{Q2}, M_{Q3}, M_{Q4}, M_{Q5}, M_{Q6} and M_{Q7}}). Without plane-sweep we must generate 6*7 = 42 pairs of MBRs. If we apply the plane-sweep technique over the X dimension (Sweeping_Dimension), this number of possible pairs will reduced considerably. First of all, we fix the MBR of the pivot entry **Pivot** = M_{P1} (shadowed MBR) and it must be paired up with {M_{Q1} and M_{Q2}}, because all pairs that can be formed from them have MINMINDIST($M_{\mathbf{P1}}$, M_{Qj}, Sweeping_Dimension) $\leq$ z and the other MBRs can be discarded ({M_{Q3}, M_{Q4}, M_{Q5}, M_{Q6}, and M_{Q7}}). In this case, we will obtain a set of 2 pairs of MBRs with the form {($M_{\mathbf{P1}}$, M_{Q1}) and ($M_{\mathbf{P1}}$, M_{Q2})}. When the processing has finished with the MBR of the **Pivot** = M_{P1}, the algorithm must establish the MBR of the pivot entry **Pivot** = M_{Q1} that is the MBR with next smallest value of the lower left corner and the process is repeated. At the end, the number of pairs of MBRs is 29, when the plane-sweep technique is applied.

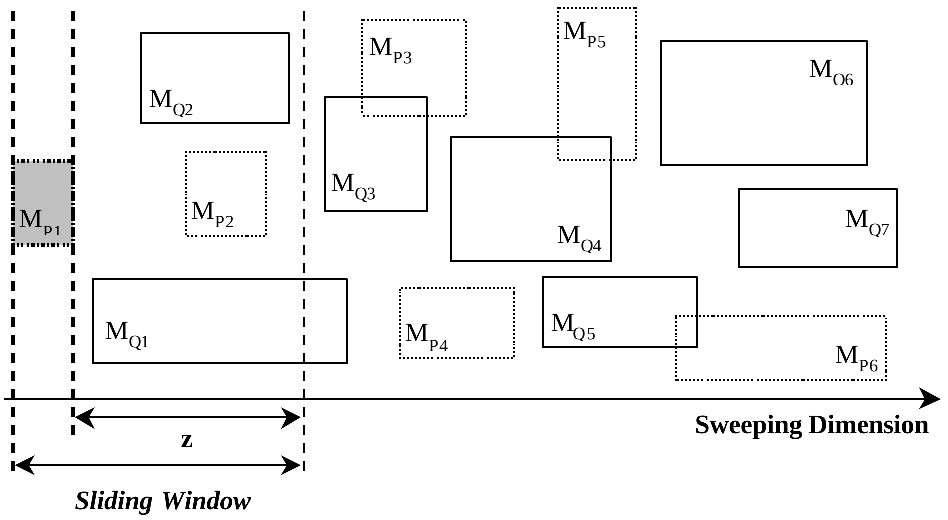


Figure 4.4: Using plane-sweep technique over MBRs from two R-tree nodes.

The 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 propagate downwards only for the pairs of entries with MINMINDIST smaller than or equal to z. The algorithm for two R-trees with the same height consists of the following steps.

KCPQ1 Start from the roots of the two R-trees and set z to ∞.

KCPQ2 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 in order to obtain the set of pairs of candidate entries, ENTRIES. Propagate downwards recursively only for those pairs of entries from ENTRIES that have MINMINDIST $\leq$ z.

KCPQ3 If you access two leaf nodes, apply the plane-sweep technique in order to obtain the set of candidate pairs of entries (ENTRIES). Then calculate the distance of each pair of spatial objects stored in ENTRIES. If this distance is smaller than or equal to z (the distance value of the K-th closest pair discovered so far), then remove the pair located in the root of K-heap and insert the new pair in K-heap, updating z.

In the example of Figure 4.5, suppose that we do not apply the updating strategies for reducing z. 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}).

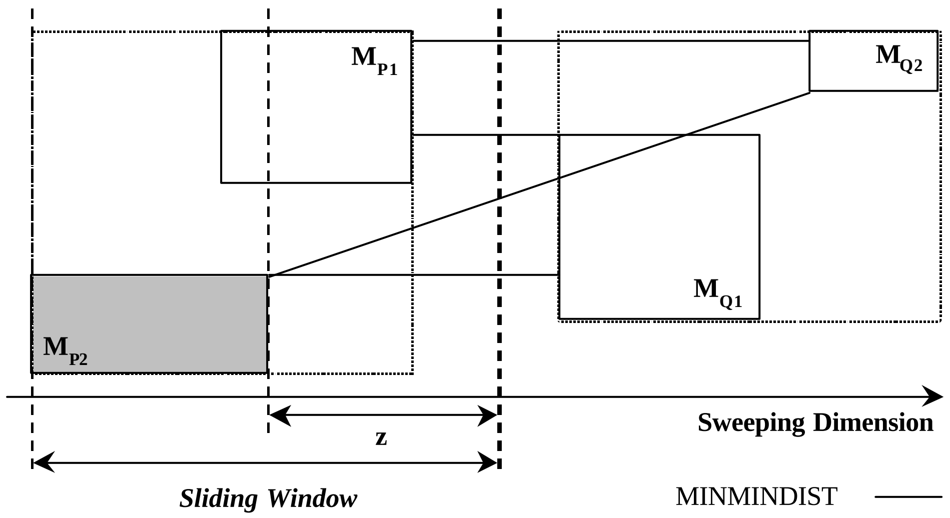


Figure 4.5: Applying the plane-sweep technique over the MBRs from two R-tree nodes.

The pseudo-code of this algorithm, called **KCPQ_PlaneSweepRecursive**, when the two R-trees have the same height, and the spatial objects (points or MBRs) are stored directly in the leaf nodes of two R-trees (the pointers are NULL) or such objects are stored in two external files (in which case, the leaf nodes store the MBRs of the spatial objects and pointers to the exact geometric description of the object), is given as follows:

```
KCPQ_PlaneSweepRecursive(N_P, N_Q: R_node; K-heap: MaximumBinaryHeap_of_Pairs; z: Double)
01 if N_P and N_Q are INTERNAL nodes then
02     [Optionally try to update z using the Updating Strategies 1 (K = 1) or 2 for any K];
03     Create_Set(ENTRIES);
04     PlaneSweep(N_P, N_Q, z, ENTRIES);
05     while not ENTRIES.isEmpty() do
06         Elem = ENTRIES.obtainPair();
07         if MINMINDIST(Elem.e_N_P.MBR, Elem.e_N_Q.MBR) ≤ z then
08             N_P' = ReadNode(Elem.e_N_P.address);
09             N_Q' = ReadNode(Elem.e_N_Q.address);
10             KCPQ_PlaneSweepRecursive(N_P', N_Q', K-heap, z);
11         endif
12     enddo
13     Destroy_Set(ENTRIES);
14 else
15     Create_Set(ENTRIES);
16     PlaneSweep(N_P, N_Q, z, ENTRIES);
17     while not ENTRIES.isEmpty() do
18         Elem = ENTRIES.obtainPair();
```

```
        if Elem.e_$N_P$.address = NULL and Elem.e_$N_Q$.address = NULL then
            dist = MINMINDIST(Elem.e_$N_P$.MBR, Elem.e_$N_Q$.MBR);
            if K-heap.isFull() then
                if dist $\leq$ z then
                    K-heap.deleteMaximum();
                    K-heap.insert(dist, Elem.e_$N_P$.MBR, Elem.e_$N_Q$.MBR);
                    if K-heap.getMaximum() < z then
                        z = K-heap.getMaximum();
                    endif
            else
                K-heap.insert(dist, Elem.e_$N_P$.MBR, Elem.e_$N_Q$.MBR);
            endif
        else
            if MINMINDIST(Elem.e_$N_P$.MBR, Elem.e_$N_Q$.MBR) $\leq$ z then
                $O_i$ = ReadObject(Elem.e_$N_P$.address);
                $O_j$ = ReadObject(Elem.e_$N_Q$.address);
                dist = ObjectDistance($O_i$, $O_j$);
                if K-heap.isFull() then
                    if dist $\leq$ z then
                        K-heap.deleteMaximum();
                        K-heap.insert(dist, $O_i$, $O_j$);
                        if K-heap.getMaximum() < z then
                            z = K-heap.getMaximum();
                        endif
                else
                    K-heap.insert(dist, $O_i$, $O_j$);
                endif
            endif
        endif
    enddo
    Destroy_Set(ENTRIES);
endif
```

4.5 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 as branching tree. Main-heap holds only pairs of addresses pointed to 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 large a K value or large spatial 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. The overall algorithm for two R-trees with the same height is as follows.

KCPQ1 Start from the roots of the two R-trees, set z to ∞ and initialize Main-heap.

KCPQ2 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 in order 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 its MBRs), which pair of MBRs has a MINMINDIST value smaller than or equal to z.

KCPQ3 If you access two leaf nodes, apply the plane-sweep technique in order to obtain the set of candidate pairs of entries (ENTRIES). Then calculate the distance of each pair of spatial objects stored in ENTRIES. If this distance is smaller than or equal to z (the distance value of the K-th closest pair discovered so far), then remove the pair located in the root of K-heap and insert the new pair in K-heap, updating z.

KCPQ4 If Main-heap is empty then stop.

KCPQ5 Get the pair on top of Main-heap. If this item has MINMINDIST > z then stop. Otherwise, repeat the algorithm from **KCPQ2** for the pair of R-tree nodes pointed by the addresses of this Main-heap item.

Note that ties between MINMINDIST values may also appear as in the sorted recursive algorithm. That is, two or more pairs ways have the same value of MINMINDIST. If this value is the minimum one, then more than one such pairs would appear near the top of the Main-heap. 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, but this special treatment requires time to carry out.

In the example of Figure 4.5, 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 item composed by <MINMINDIST(M_{P1} ,M_{Q1}), entryOfM_{P1}.address, entryOfM_{Q1}.address>.

The pseudo-code of this algorithm, called **KCPQ_PlaneSweepIterative**, is given as follows:

```
KCPQ_PlaneSweepIterative(Root_N_P, Root_N_Q: R_node; K-heap: MaximumBinaryHeap_of_Pairs)
01 Create_MinimumBinaryHeap(H);
02 z = ∞;
03 [Optionally try to update z using the Updating Strategies 1 (K = 1) or 2 for any K];
04 Create_Set(ENTRIES);
05 PlaneSweep(Root_N_P, Root_N_Q, z, ENTRIES);
06 while not ENTRIES.isEmpty() do
07     Elem = ENTRIES.obtainPair();
08     dist = MINMINDIST(Elem.e_N_P.MBR, Elem.e_N_Q.MBR);
09     if dist ≤ z then
10         H.insert(dist, Elem.e_N_P.address, Elem.e_N_Q.address);
11     endif
12 enddo
13 Destroy_Set(ENTRIES);
14 while not H.isEmpty() do
15     Pair = H.deleteMinimum();
16     if Pair.MINMINDIST ≤ z then
17         N_P = ReadNode(Pair.address_N_P);
18         N_Q = ReadNode(Pair.address_N_Q);
```

```
    if $N_P$ and $N_Q$ are INTERNAL nodes then
        [Optionally try to update z using the Updating Strategies 1 ($K = 1$) or 2 for any K];
        Create_Set(ENTRIES);
        PlaneSweep($N_P$, $N_Q$, z, ENTRIES);
        while not ENTRIES.isEmpty() do
            Elem = ENTRIES.obtainPair();
            dist = MINMINDIST(Elem.e_$N_P$.MBR, Elem.e_$N_Q$.MBR);
            if $dist \leq z$ then
                H.insert(dist, Elem.e_$N_P$.address, Elem.e_$N_Q$.address);
            endif
        enddo
        Destroy_Set(ENTRIES);
    else
        Create_Set(ENTRIES);
        PlaneSweep($N_P$, $N_Q$, z, ENTRIES);
        while not ENTRIES.isEmpty() do
            Elem = ENTRIES.obtainPair();
            if Elem.e_$N_P$.address = NULL and Elem.e_$N_Q$.address = NULL then
                dist = MINMINDIST(Elem.e_$N_P$.MBR, Elem.e_$N_Q$.MBR);
                if K-heap.isFull() then
                    if $dist \leq z$ then
                        K-heap.deleteMaximum();
                        K-heap.insert(dist, Elem.e_$N_P$.MBR, Elem.e_$N_Q$.MBR);
                        if K-heap.getMaximum() < z then
                            z = K-heap.getMaximum();
                        endif
                else
                    K-heap.insert(dist, Elem.e_$N_P$.MBR, Elem.e_$N_Q$.MBR);
                endif
            else
                if MINMINDIST(Elem.e_$N_P$.MBR, Elem.e_$N_Q$.MBR) $\leq z$ then
                    $O_i$ = ReadObject(Elem.e_$N_P$.address);
                    $O_j$ = ReadObject(Elem.e_$N_Q$.address);
                    dist = ObjectDistance($O_i$, $O_j$);
                    if K-heap.isFull() then
                        if $dist \leq z$ then
                            K-heap.deleteMaximum();
                            K-heap.insert(dist, $O_i$, $O_j$);
                            if K-heap.getMaximum() < z then
                                z = K-heap.getMaximum();
                            endif
                    else
                        K-heap.insert(dist, $O_i$, $O_j$);
                    endif
                endif
            endif
        enddo
        Destroy_Set(ENTRIES);
    endif
else
    H.makeEmpty();
endif
```

```
71 enddo
72 Destroy_MinimumBinaryHeap(H);
```

Notice again that in the previous algorithm, the two R-trees have the same height, and the spatial objects (points or MBRs) are stored directly in the leaf nodes of two R-tree (the pointers are NULL) or such objects are stored in two external files; in which case, the leaf nodes store the MBRs of the spatial objects and pointers to the exact geometric description of the object. The extensions to the other possibilities (the spatial objects are stored in the leaf nodes in one R-tree and in an external file for the another, and vice versa) is straightforward, and for R-trees with different heights, see the next subsection, when two interesting alternatives are proposed.

4.6 Treatment of Different Heights

When the two R-trees storing the two spatial datasets 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, not to propagate 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 node on the one hand and an internal node on the other hand, downwards propagation stops in the R-tree of the leaf node, 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 effect of each approach.

4.7 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 spatial objects with the distances within a range [Dist_Min, Dist_Max] ($0 \leq$ Dist_Min $\leq$ Dist_Max) are the more representative ones. Now, we are going to present these operations and the modifications in our branch-and-bound algorithms in order to carry out them.

4.7.1 K-Self-CPQ

A special case of K-CPQ is the called "K-Self-CPQ" where both spatial datasets actually refer to the same entity. That is, the input spatial 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:

K-Self-CPQ(P, Q, K) = {K-CPQ(P, Q, K): P = Q}

As an example from the 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 Chapter 3, P and Q are identical datasets, and hence their entries are indexed in a single R-tree. The algorithms proposed in this thesis 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 result set.

To improve the performance of the branch-and-bound algorithm for this derived 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.

4.7.2 Semi-CPQ

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

Definition 7
Let two spatial datasets, P = {p_1.G, p_2.G, ... , p_{NP}.G} and Q = {q_1.G, q_2.G, ... , q_{NQ}.G} with spatial extent p_i.G and q_j.G in $E^{(k)}$, be stored in a spatial database. Then, the result of the semi-closest pairs query is a sorted sequence of |P| pairs of spatial objects, Semi-CPQ(P,Q) Î P ´ Q, where each spatial object in P forms a pair with the closest spatial object in Q:

Semi-CPQ(P, Q) *= {(p_i.G, q_i.G): p_i.G Î P, q_i.G Î Q, p_k.G ¹ p_j.G if k ¹ j,*
dist(p_s.G, q_s.G) £ dist(p_t.G, q_t.G) if s £ t}

The Semi-CPQ works by reporting a sequence of pairs of spatial objects (p_i.G, q_i.G) in order of distance. Note that once we have determined the closest spatial object q_i.G to a particular p_i.G, that p_i.G does not participate in other pairs with q_i.G. Unlike most join operations, the Semi-CPQ is not commutative, that is, 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 spatial objects belonging to leaf nodes of the first R-tree. Each spatial 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 spatial 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 spatial dataset, finds for each spatial object its nearest neighbor. This operation is a Semi-CPQ where the input spatial dataset is combined with itself.

Self-Semi-CPQ(P, Q) = {Semi(P, Q): P = Q}

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

4.7.3 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 spatial objects from two spatial datasets. The result set of the K-Farthest Pairs Query (K-FPQ) is given by the following definition.

Definition 8

Let two spatial datasets, P = {p_1.G, p_2.G, ... , p_{NP}.G} and Q = {q_1.G, q_2.G, ... , q_{NQ}.G} with spatial extent p_i.G and q_j.G in $E^{(k)}$, be stored in a spatial database. Then, the result of the K farthest pairs query (K-FPQ(P,Q)) is a set containing subsets of K (1£K£|P|·|Q|) different and sorted pairs of spatial objects from P´Q, with the K largest distances between all possible pairs of spatial objects that can be formed by choosing one spatial object of P and one spatial object of Q:

K-FPQ(P, Q, K) *= {{(p_1.G, q_1.G), (p_2.G, q_2.G), ..., (p_K.G, q_K.G)},*
p_1.G, p_2.G, ... , p_K.G Î P ,q_1.G, q_2.G, ..., q_K.G Î Q, (p_i.G, q_i.G) ¹ (p_j.G, q_j.G) "i ¹ j:
"(p_i.G, q_j.G) Í P´Q – {(p_1.G, q_1.G), (p_2.G, q_2.G), ..., (p_K.G, q_K.G)},
dist(p_1.G, q_1.G) ³ dist(p_2.G, q_2.G) ³ … ³ dist(p_K.G, q_K.G) ³ dist(p_i.G, q_j.G)}

In this case, if we want to design a branch-and-bound recursive algorithm for solving K-FPQ by extending the K-CPQ algorithms, when the two spatial datasets are stored in two R-trees, we must take into consideration the following constraints:

1. Now, K-heap must be 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 you access two internal nodes, calculate MAXMAXDIST for each possible pair of MBRs and sort these pairs in decreasing order of MAXMAXDIST. Following this order, propagate downwards recursively only for those pairs of entries that have MAXMAXDIST $\geq$ z.
3. If you access two leaf nodes, then calculate the distance of each possible pair of spatial objects. If this distance is larger than or equal to z, then remove the pair located in the root of K-heap and insert the new pair in K-heap, updating z.

In the same terms, we can extend the iterative branch-and-bound algorithm for K-CPQ in order to obtain one for K-FPQ. In this case, we must only consider the following conditions:

1. Now, Main-heap must be organized as a maximum binary heap and K-heap as a minimum binary heap with MAXMAXDIST as a key in both cases. Moreover, z is the distance value of the K-th farthest pair discovered so far and stored in K-heap (z = 0).
2. If you access two internal nodes, calculate MAXMAXDIST for each possible pair of MBRs. Insert into Main-heap only those pairs of addresses of R-tree nodes (and MAXMAXDIST), which pair of MBRs have a MAXMAXDIST value larger than or equal to z.

3. If you access two leaf nodes, then calculate the distance of each possible pair of spatial objects. If this distance is larger than or equal to z, then remove the pair located in the root of K-heap and insert the new pair in K-heap, updating z.

4.7.4 Obtaining the K or All Closest Pairs of Spatial Objects with their Distances within a Range

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

1. If two internal nodes are accessed, do not update z (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 leaf nodes are accessed, then calculate the distance of each possible pair of spatial 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, we must remove the K-th closest pair (in the root of K-heap) and insert the new one, updating the K-heap structure.

On the other hand, one may wish to obtain all the possible pairs of spatial objects with the distances within the interval [Dist_Min, Dist_Max]. In this case, neither K nor the size of K-heap are known a priori and Dist_Max is the bound distance for the pruning heuristic. Of course, when Dist_Max = ∞, our branch-and-bound algorithms degenerate in backtracking ones (obtaining all possible feasible solutions of a given problem), as when $K \geq |P|\cdot|Q|$, where $|P|$ and $|Q|$ are the numbers of the spatial 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 spatial 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].

Chapter 5

5 Experimental Results

This chapter provides the results of an extensive experimentation study aiming at measuring and evaluating the efficiency of the three K-CPQ algorithms proposed in Chapter 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 index structure, instead of packed R-tree [RoL85, KaF93, LEL97], since we are interested in dynamic environments. In order to evaluate our non-incremental branch-and-bound algorithms for K-CPQ we have taken into account several performance measurements (I/O activity, response time, number of distance computations and subproblem decomposed). We have also measured the number of insertions in the Main-heap for the iterative algorithm (PSI). 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 in order to execute its more representative extensions as K-Self-CPQ, Semi-CPQ and K-FPQ. Finally, in this experimental chapter we have treated the study of the scalability of the algorithms (varying the dataset sizes and K), and a performance comparison of the simulation of the incremental processing in order to obtain the K closest pairs incrementally.

5.1 Experimental Settings

All experiments have been run on a 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 heap for the result (K-heap), because if we store some part of them in secondary memory (file on disk) we must take into account the I/O activity over this data structure.

Spatial datasets We have used spatial data of different nature in our experiments: points and line segments. The spatial objects were represented directly in the leaf nodes of the R*-trees. This is not always practical, especially for complex and variable size objects such as polygons. The other alternative is to store the spatial objects in an external file, in which case the leaf nodes store the MBRs of the spatial objects and pointers to the objects.

In order to evaluate K-CPQ algorithms, we have used real spatial datasets from [DCW97]. 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.

	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 spatial datasets.

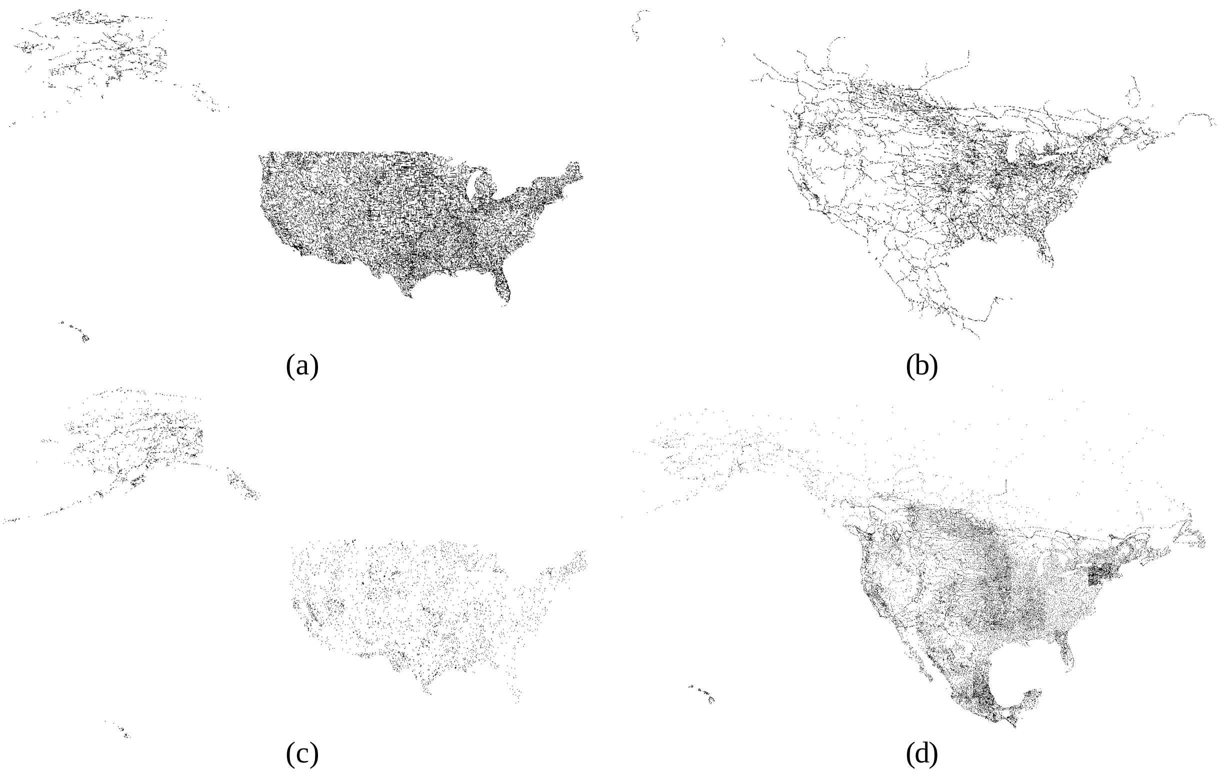


Figure 5.1: Four real-world spatial datasets from [DCW97]: (a) roads of USA, (b) rail-roads of North America, (c) cultural landmarks of USA and (d) populated places of North America.

Performance Measurements 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 computational cost and I/O activity.

1. **Number of disk accesses**. It is the more 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 it represents the overall CPU time consumed by the algorithms as well as all data structures included on them.

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 computational cost of this kind of distance join operation.

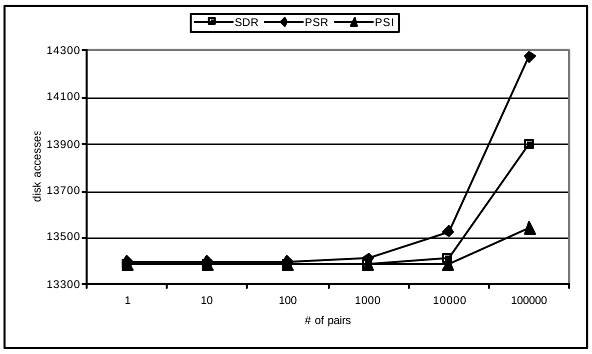
Thus, the total number of distance computations required by a K-CPQ algorithm provides a direct indication of its computational performance.

4. **Number of subproblems decomposed**: It is another important performance measurement related to computational cost of this query. It represents the number of pairs of MBRs decomposed before the termination of the algorithms and provides the number of partial subproblems considered during the execution of the algorithms. 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 (especially for PSI)**: The task of managing the main binary heap (Main-heap) is largely CPU intensive with the increase of its size. 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 the insertions are much more frequent than the deletions.

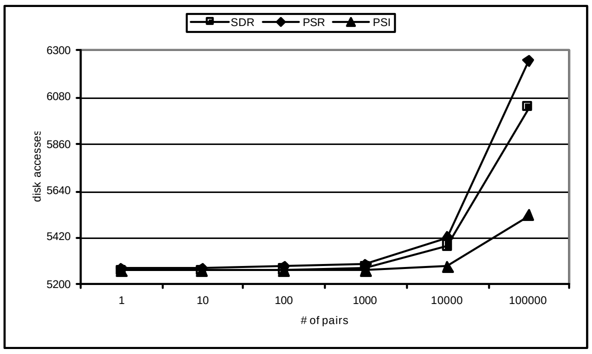
5.2 Performance Comparison of K-Closest Pairs Query Algorithms

We proceed with the evaluation of the three algorithms for K-CPQ (SDR, PSR and PSI) with K (cardinality of the result) varying from 1 to 100000, assuming zero buffer and obviously for the same workspaces. Figure 5.2.a illustrates the number of disk accesses for K-CPQ over (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 (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 following, 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 configuration with similar I/O trends. Moreover, the deterioration is not smooth; after a threshold the cost increases considerably (this threshold was usually around K = 1000). This demonstrates that the iterative algorithm was very effective in the pruning process in absence of buffers, since it follows a Best-First searching policy optimized with the plane-sweep technique.



(a)



(b)

Figure 5.2: Comparison with respect to the disk accesses of the K-CPQ algorithms without buffer and varying K for (a) (USrr, USrd) and (b) (NArr, NApp) configurations.

For the (USrr, USrd) configuration, Table 5.2 compares the other performance parameters: total response time (bold), number of distance computations (italic), number of subproblems decomposed (regular) and the amount of Main-heap insertions (in parentheses) needed by each algorithm. For all K values, the plane-sweep technique reduced the number of distance computations significantly; 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 ≤ 1000) PSR was the fastest, and for large K values (K ≥ 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 policy and it can deviate of the branches where no optimal solutions are located. Moreover, PSI is the algorithm with the minimum number of subproblems decomposed according to Theorem 5, 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 internals, and all these tasks consume time.

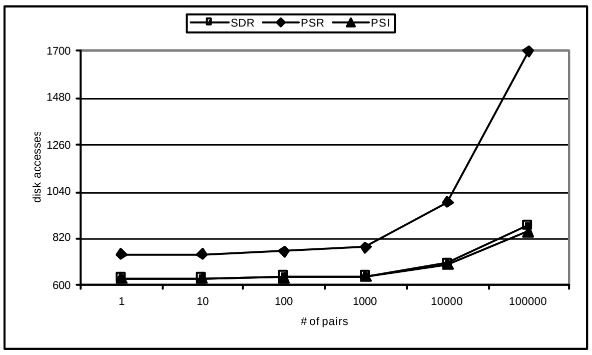
	K = 1	K = 10	K = 100	K = 1000	K = 10000	K = 100000
SDR	**613.86** *140307590* 6692	**614.23** *140307590* 6692	**614.31** *140307590* 6692	**614.36** *140307590* 6692	**616.52** *140618716* 6706	**644.80** *145538868* 6947
PSR	**20.02** *3164690* 6698	**20.05** *3167352* 6698	**20.10** *3180129* 6699	**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 performance of the K-CPQ algorithm without buffer and varying K for (USrr, USrd) configuration. In particular, total response time (bold), number of distance computation (italic), number of subproblems decomposed (regular) and the amount of Main-heap insertions (in parenthesis).

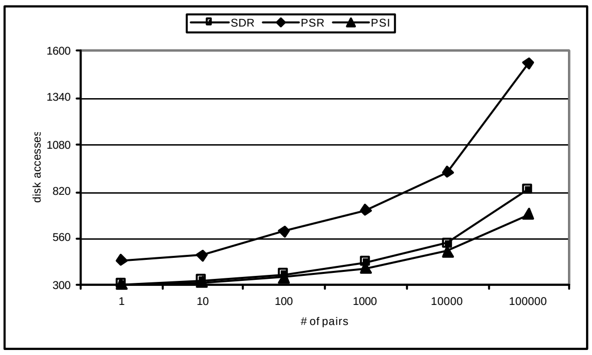
5.3 Results on Disjoint Datasets

In [CMT00], the effect of overlap between the datasets for K-CPQ was studied, and the conclusion without buffers was: *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 (MXrd, USrr) configuration, where MXrd are the roads of Mexico. On the other hand, Figure 5.3.b shows the same metric for (CDrr, USpp) configuration, where CDrr are the rail-roads of Canada.

Figure 5.3 shows, as Figure 5.2, that PSI clearly improves performance with respect to the recursive alternatives (SDR and 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, and the explanation of this behavior is due primarily to the fact that the values of MINMINDIST are large enough for disjoint datasets and the pruning is much more effective.



(a)



(b)

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

For the (MXrd, USrr) configuration, Table 5.3 compares the other performance measurements. 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 measurement, PSI is in 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 performance of K-CPQ algorithms without buffer and varying K for (MXrd, USrr) configuration

5.4 The Effect of Buffering

The use of buffers is very important in DBMSs, since it can improve the performance substantially (reading data from the disk is significantly more expensive than reading from a main memory buffer). There exist two basic research directions that aim at reducing the disk I/O 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, JoS94, BJK97]. 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 buffer accordingly [SaS82, ChD85, COL92].

DBMSs use indices to speed up query processing (e.g. various spatial databases use R-trees). Indices may partially reside in main memory buffers. The buffering effect should be studied, since even a small number of buffer pages can substantially improve the global database 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 first decision criterion, in [CVM01] several buffer pool structures, page replacement policies and buffering schemes for K-CPQ algorithms were analyzed, in order to reduce the number of disk accesses. For the experiments of this subsection, we will adopt the best configuration for this kind of distance-based query that was proposed in [CVM01] and in Figure 5.4 is depicted: *LRU with a single buffer pool structure, using a global buffering scheme.*

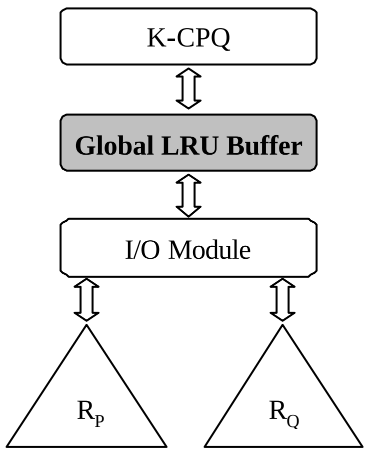
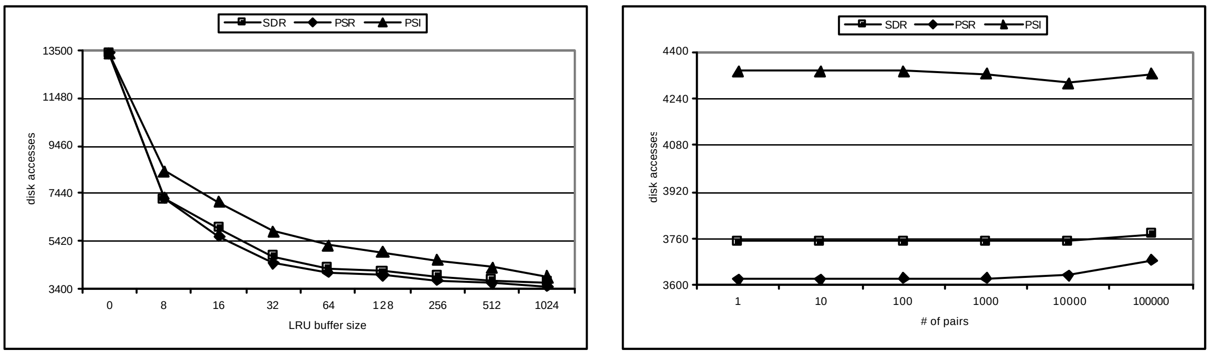


Figure 5.4: Buffer configuration.

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. It means that we have in memory a variable percentage of the R*-tree nodes, depending on the number of pages the buffer. Besides, the buffer does not use any global optimization criterion, i.e. the pages on the buffer are handled as the algorithms are required, depending on which R*-tree are located.

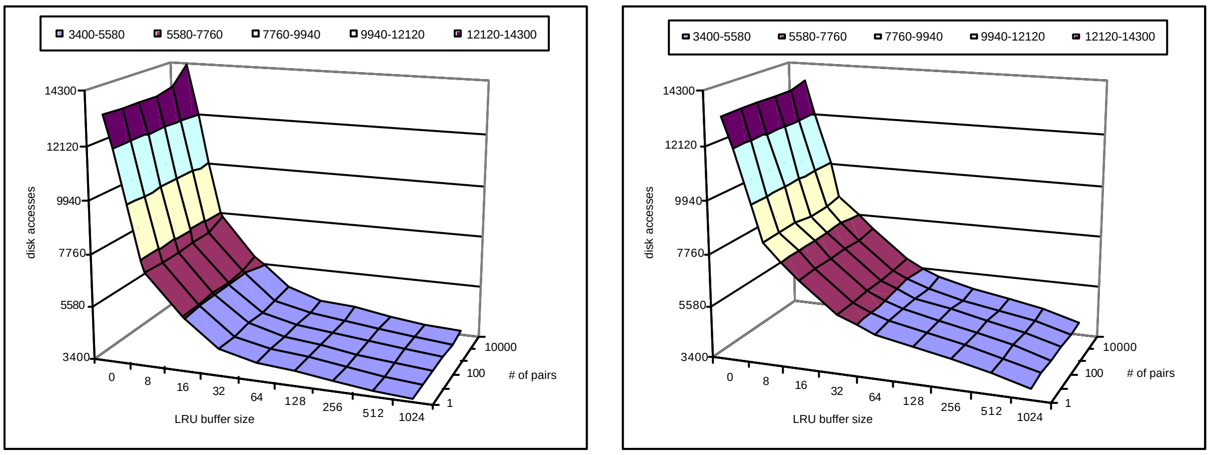
Figure 5.5.a shows that PSI presents an average excess of I/O activity around 14% and 18% for K = 1000 with respect to SDR and PSR, respectively, as can be noticed by the gap between the lines. Moreover, the influence of buffer is greater for PSR than for SDR, due to the use of the plane-sweep technique. This behavior is due to the fact that the recursion favors the most recently used pages (LRU) in the backtracking phase and this effect is conserved in case of large buffers. On the other hand, Figure 5.5.b illustrates that the gap for K-CPQ algorithms is maintained when the K value is incremented and B = 512 pages. For instance, the average I/O saving between PSR and SDR with the increase of 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 5.5: Comparison of the disk accesses for the K-CPQ algorithms using the (USrr, USrd) configuration: (a) varying the buffer size and K = 1000, (b) varying K and B = 512 pages.

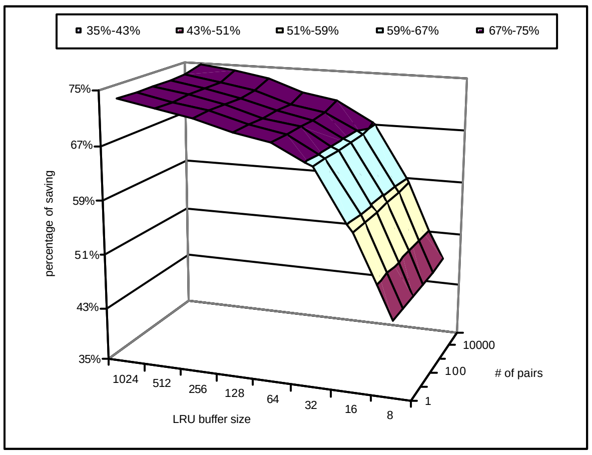
Figure 5.6 illustrates the performance of the best K-CPQ recursive (PSR) and the iterative (PSI) algorithms as a function of buffer size (B ≥ 0). For PSR, when B ≥ 64, the savings in terms of disk accesses are large and almost the same for all K values. However, the savings are considerably less when 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 enough 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.

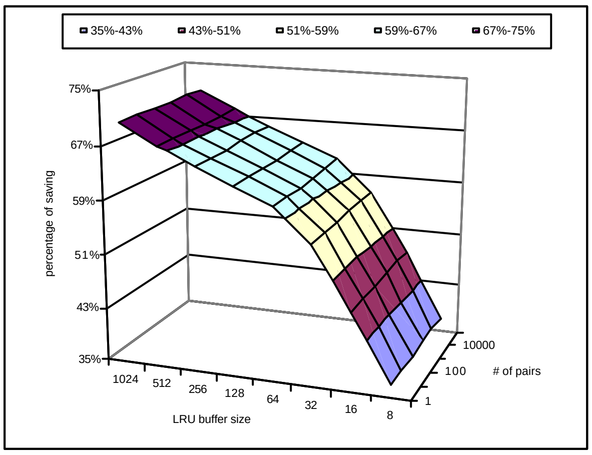


(a) (b)

Figure 5.6: 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).

In Figure 5.7, the percentage of I/O saving (induced by the use of buffer size B > 0 in contrast to not using any buffer) of PSR (Figure 5.7.a) and PSI (Figure 5.7.b) is depicted. For PSR, the percentage of saving grows as the buffer size increases, for all K values, although it is slightly bigger for K = 100000. The trend of the behavior of PSI is almost the same that PSR, although the increase is 8% less in average with respect to the recursive algorithm.





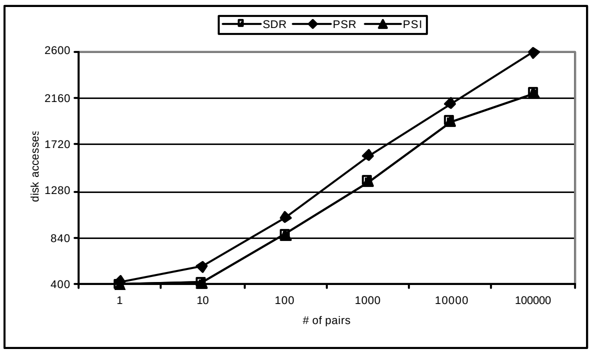
(a) (b)

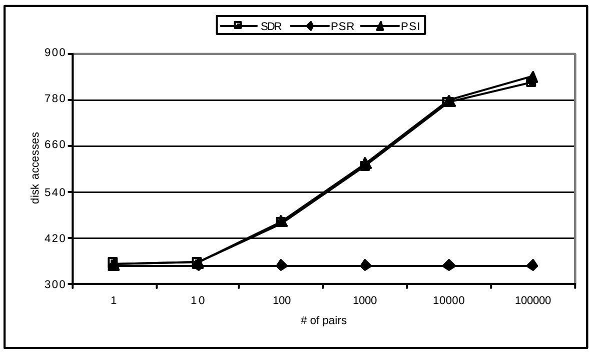
Figure 5.7: The percentage of saving for (a) PSR and (b) PSI, as a function of the LRU buffer size (B) and the cardinality of the result (K).

From the results and conclusions of this subsection, we can notice that the influence of our buffer configuration 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 configuration more than the case of a Best-First searching policy 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. Figure 5.8 shows the number of disk accesses for K-Self-CPQ algorithms for the configuration (NApp, NApp), without buffer (Figure 5.8.a) and with B = 256 pages (Figure 5.8.b). From these figures we can observe the same influence of buffering for this extension with respect to K-CPQ. These increase trends are due to the fact that we must discard two kinds of candidate pairs (equal to and symmetric). The behavior of PSR with large LRU buffer size, where for all K values we obtain the same number of disk accesses is interesting.





(a) (b)

Figure 5.8: Comparison with respect to the disk accesses of the K-Self-CPQ algorithms, varying K for (NApp, NApp) configuration (a) without buffer and (b) with B = 256 pages.

For the same workspace configuration (NApp, NApp), Table 5.4 compares the other performance measurements needed by each adapted algorithm for K-Self-CPQ, following similar behavior with respect to the K-CPQ. In particular, for small K values ($K \leq 100$) PSR was the best alternative, but when K is incremented ($K \geq 1000$), PSI is the fastest. Moreover, PSI was the algorithm with the minimum number of subproblems decomposed for all K values, and SDR was the worst algorithm for all measurements 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 implies the reduction in response time. Besides, the Best-First traversal minimizes the number of subproblems decomposed and this effect can be clearly shown for large K values.

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

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

Next, we report the results of experiments testing the extension of our 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 value of z is not global to the result of the query 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:

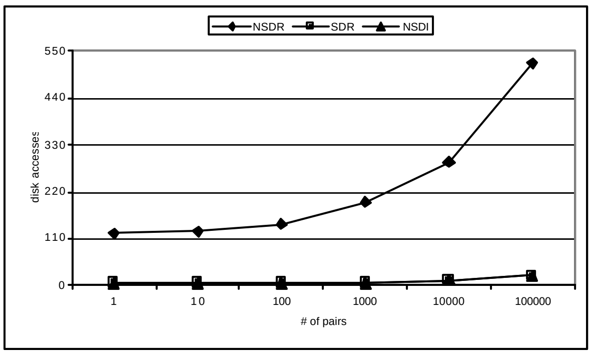
1. Traverse recursively the first R*-tree, accessing the object in order of appearance within each leaf node;
2. For each object, perform a nearest neighbor query over the second R*-tree and,
3. Sort the results (array of object with its distances) in ascending order of distances.

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 measurements for this query. Our extensions obtain the best behavior with respect to the number of disk accesses, mainly "GlobalAll" iterative (GAI). But, for the other measurements, T+NNQ is better than our extensions, since it needs less distance computations. Also, we must highlight that T+NNQ needs to store in main memory an array of objects with its distances for all objects indexed in the first R*-tree, and 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 disk accesses without buffers, but they consume a lot of memory resources and time for reporting the result.

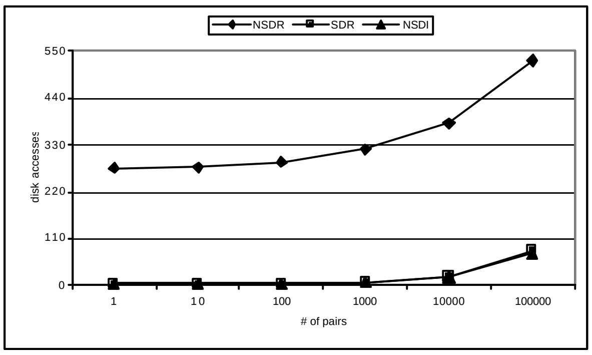
	T+NNQ	GOR	GAR	GASR	GAI
Disk Accesses	94209	69894	45296	38962	38868
Response Time	29.52	1180.95	703.35	681.82	589.04
Distance Comp.	17007578	674087280	437822358	376471488	363864610
Sub. Decomp.		34771	22472	19305	19258

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

Another extension of K-CPQ is to find the K farthest pairs of objects from two spatial datasets. 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). Figure 5.9.a shows the number of disk accesses with global LRU buffer of 256 pages for K-FPQ using the (USrr, USrd) configuration, whereas Figure 5.9.b illustrates the same metric for (NArd, NApp) configuration. From these figures, we can observe the reduced number of disk accesses needed for this query, even for large K values. The explanation of this behavior is due to the fact that MAXMAXDIST is the distance metric for pruning in the extended branch-and-bound algorithms instead of MINMINDIST, and MAXMAXDIST is very effective in this case. Moreover, the execution of the algorithms for (NArr, NApp) configuration (lines, points) over R*-trees with different heights is more expensive that (USrr, USrd) configuration (lines, lines) over R*-trees with the same heights and more objects to combine. This behavior is primarily due to the different height treatment (fix-at-leaves) that we must consider in this case at the leaf level. Another conclusion from the figures is that SDR and NSDI have the best behavior, and they are notably better that NSDR. For large K values NSDI is slightly better than SDR in the number of disk accesses over (NArr, NApp) configuration.



(a)



(b)

Figure 5.9: Comparison with respect to the disk accesses of the K-FPQ algorithms, varying K and B = 256 pages for (a) (USrr, USrd) and (b) (NArr, NApp) configurations.

Table 5.6 presents the (USrr, USrd) configuration, comparing the other performance measurements for K-FPQ. Again, 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 based MAXMAXDIST of all possible pairs from two internal nodes that SDR needs to execute the query.

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

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

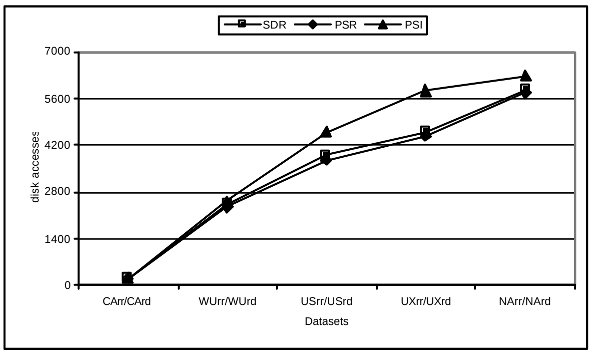
5.6 Scalability of the Algorithms, Varying the Dataset Sizes and K

As already pointed out, we are going to see 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 (its cardinality), fixing the value of K, for 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.

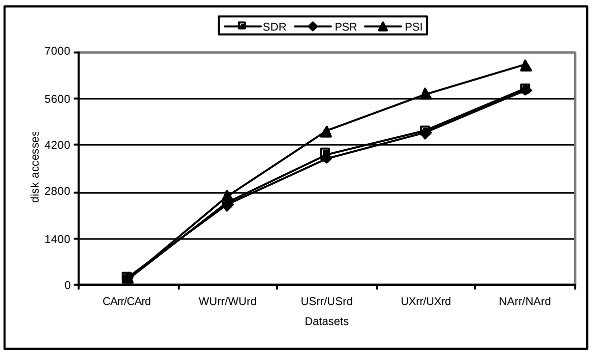
	Rail-Roads	Roads
California	11,381	21,831
West USA	81,043	244,385
USA	146,503	355,312
USA + Mexico	156,563	447,704
North America	191,637	569,120

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

Figure 5.10 shows that the performance (disk accesses) increases almost linearly with the increase of the cardinalities of the real spatial datasets, even for large K values. The trends for two diagrams are very similar, since the saving in disk accesses using a global LRU buffer is very high. Moreover, in presence of buffer, again, the PSR is the best alternative and PSI provides the greatest number of disk accesses.



(a)



(b)

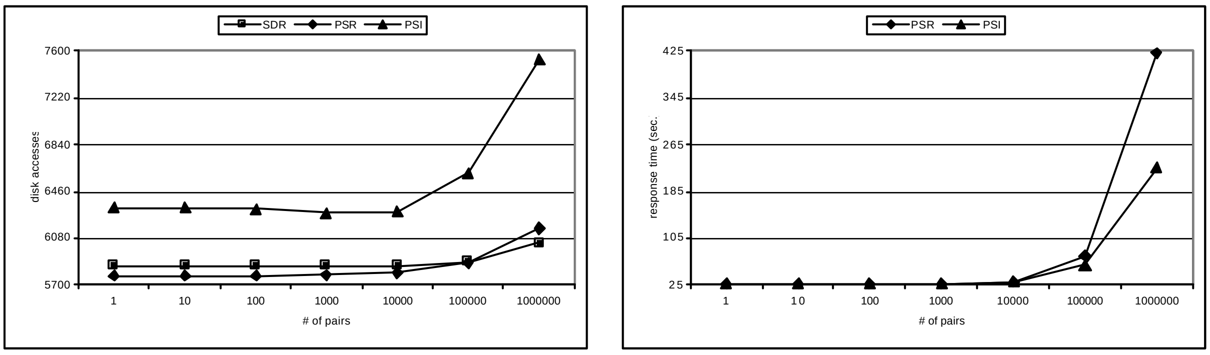
Figure 5.10: Comparison with respect to the disk accesses of K-CPQ algorithms for B = 256 pages, using different configurations of the spatial datasets in increasing size for (a) K = 1000 and (b) K = 100000.

For the previous five configurations, Table 5.8 compares the other performance measurements for K = 100000 and B = 256. Clearly, PSI is the best algorithm for total response time, distance computations and subproblems decomposed. 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 ≤ 1000 and PSI when K ≥ 10000. Besides, the increase of the performance was almost linear with the increase of the cardinalities of the real spatial datasets for a given K, following the same trend to the disk accesses.

	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 performance of K-CPQ algorithms for K = 100000 and B = 256 pages, using different configurations of the real spatial datasets in increasing size.

Another way to measure the scalability of our K-CPQ algorithms is to take into account its behavior for the increase of K using large real spatial datasets. Figure 5.11.a shows that the disk accesses increases in a sub-linear way with the increase of the cardinalities of the result (K) for the recursive alternative, using the (NArr, NArd) configuration and B = 256 pages. Namely, by increasing K values (1..1000000), the performance of PSR is not significantly affected, there is only an extra cost of 6%, whereas for PSI this extra cost is about 16%. Moreover, SDR for K = 1000000 is slightly better than PSR, only 2%. Figure 5.11.b illustrates the response time for the fastest K-CPQ algorithms (PSR and PSI) for the increase of cardinality of the result (K). For instance, they have very similar result for K ≤ 10000, but for K = 100000 and K = 10000000 PSI is 20% and 48% faster than PSR, respectively.



(a) (b)

Figure 5.11: Comparison with respect to the (a) disk accesses and (b) total response time for K-CPQ algorithms varying K (1..1000000), B = 256 pages and (NArr, NArd) configuration.

Table 5.9 represents the other performance measurements for (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 $\leq$ 1000 and PSI when K $\geq$ 10000, with respect to the time response and distance computations. Besides, PSI is the algorithm with the smallest number of subproblems decomposed for all K values, according to Theorem 5; and it only needs a 48% extra of insertions in the Main-heap in order 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 computation 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 performance for K-CPQ algorithms varying K (1..1000000), B = 256 pages and (NArr, NArd) configuration.

5.7 Simulating the Incremental Processing

In this subsection we focus on the closest pairs query when we do not know in advance how many pairs will be needed by the user before the end of the query. In this case, we need to restart the K-CPQ algorithm as the value of K changes. In contrast, in the case of the incremental closest pairs query algorithm, we need to recall the algorithm to obtain just the next pairs. Basically, there are two ways for using a K-CPQ algorithm to perform an incremental closest pairs query: (1) execute a non-incremental K-CPQ algorithm each time we need a new pair, and (2) invoke a K-CPQ algorithm for every X pairs. In the second method, the K-CPQ algorithms can be adapted to perform the incremental simulation more suitable in order to use it in a incremental closest pairs query. The idea is to create variants of our non-incremental K-CPQ algorithms that discover the closest pairs after K_1-th and before K_2-th closest pair. In particular, after finding the K_1-th closest pair and knowing we must search $K_2 > K_1$ closest pairs, we can use the distance of the K_1-th closest pair as lower bound of pruning distance only for pairs of objects at the leaf level, when the K-CPQ algorithm is restart with $K = K_2$ (really, K is set to $K_2 – K_1$, since the K_1 closest pairs would be excluded from the search and all the new closest pairs will have a distance larger than or equal to the distance of the K_1-th closest pair). The procedure that obtains the K closest pairs incrementally for every X pairs can be expressed as the following steps (pseudo-incremental approach):

I1 $K_1 = X$, $d_{K1} = \infty$, $z = \infty$.
I2 If $K_1 = K$ then stop.
I3 If $K_1 > K$ then $K_1 = K_1 – K$.
I4 We execute one of our non-incremental K-CPQ algorithms (SDR, PSR or PSI) for $K = K_1$ and it discovers the K_1 closest pairs.
I5 We keep the distance of the K_1-th closest pair (d_{K1}) as the lower bound of pruning distance.
I6 We set $K_1 = K_1 + X$, $z = \infty$ and using d_{K1} as the lower bound of pruning distance only for pairs of objects at the leaf level (i.e. we must choose the pair of object (O_i, O_j) that satisfies $d_{K1} \leq$ **ObjectDistance**(O_i, O_j) $\leq z$), go to step **I2**.

We can notice that some complications arise if there are pairs with the same distance of K_1-th closest pair and the procedure described above may rediscover the pairs with the same distance (d_{K1}) in the step **I4**. However, it is easy to keep track of such pairs and discard them before the execution of such a step.

In this set of experiments, we have simulated a situation where the users repeatedly require a set of X closest pairs at a time, until a total of K closest pairs are generated. This X was set to K divided by 5 ($X = K / 5$). In general, $X = K / \alpha$ where $1 \leq \alpha \leq K$. For example, if we want to obtain the $K = 10000$ closest pairs incrementally, $X = 2000$ and our non-incremental branch-and-bound K-CPQ algorithms were restarted for $K_1 = 2000$, 4000, 6000, 8000 and 10000. The choice of the X value is crucial for the global performance of our pseudo-incremental approach, the smaller X value (it implies large α value) the greater the cumulative cost is (the number of restarts of out non-incremental K-CPQ algorithms will be smaller). In order to compare our pseudo-incremental approach, we have used the incremental distance join algorithms (Evenly (EVN) and Simultaneous (SML)) proposed in [HjS98] as guidelines for obtaining the incremental results of the query (i.e. obtain the K closest pairs incrementally).

Table 5.10 shows the performance measurements of the cumulative cost for the K (from 10 to 100000) closest pairs obtaining incrementally (the number of disk accesses is represented between brackets), using the (NArr, NArd) configuration, B = 256 pages and for incremental algorithms D_T was set to 0.05. From the Table 5.10, the incremental algorithms are faster that our pseudo-incremental approach for small K values ($K \leq 1000$), but when the number of pairs required is large enough ($K \geq 10000$) the cumulative cost of our pseudo-incremental alternative (depending on X) is less than the cost of the incremental alternatives. This behavior is primarily due to the fact that the incremental algorithms need to calculate many distances and manage a high volume of pairs (MBR/MBR, MBR/Object, Object/MBR and Object/Object) in the main priority queue when the K value is large. Moreover, the nature of spatial data involved in the query plays an important role, since if there are many pairs of objects with the distance equal to 0, they will appear at the beginning of the main priority queue and they will be returned very fast as result of the query. For instance, in this configuration (NArr, NArd) there are many pairs with distance equal to 0 (i.e. many pairs of lines intersect) and they are reported very fast with the incremental algorithms. Also, it is very interesting the few number of disk accesses of the incremental algorithms, the reason is due to the effect of buffering.

	K = 10	K = 100	K = 1000	K = 10000	K = 100000
PSR	{28879} **135.53** *21534874* 48039	{28885} **133.92** *21561917* 48045	{28899} **135.10** *21760964* 48065	{28955} **142.77** *22995530* 48193	{29099} **196.52** *31236986* 48992
PSI	{31585} **121.98** *22204422* 48005 (783340)	{31640} **122.25** *22231960* 48005 (783408)	{31602} **123.04** *22413559* 48016 (783943)	{31487} **128.88** *23500204* 48047 (785684)	{31607} **168.42** *30596429* 48299 (794014)
EVN	{46} **0.17** *37366* 205 (2164)	{153} **0.87** *174964* 1073 (5611)	{267} **6.12** *1111370* 8458 (23419)	{2379} **173.70** *11671499* 88423 (176286)	{12618} **4878.18** *34533114* 335771 (912182)
SML	{37} **1.82** *486694* 36 (3517)	{123} **7.96** *1845509* 209 (4919)	{591} **63.77** *14873784* 1740 (17142)	{2731} **346.78** *76027757* 13717 (97355)	{6705} **1612.45** *198694099* 109703 (772570)

Table 5.10: Comparison of the cumulative cost for the stepwise incremental execution, varying K depending on X (X = K / 5) and B = 256 pages for (NArr, NArd) configuration.

Here, the conclusions are very similar to the results reported on [HjS98], the incremental algorithms outperform our pseudo-incremental approach if small or medium part of the result is needed, while if we require a large or the entire distance join result, the performance of the incremental algorithms is penalized. On the other hand, our pseudo-incremental approach obtain interesting performance measurements only if X is good enough, since these cumulative performance measurements are proportional to the α value, where $X = K / \alpha$.

Chapter 6

6 Conclusions and Future Work

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 author's knowledge, [HjS98, CMT00, SML00] are the only references to this type of queries. Branch-and-bound has been the most successful technique for the design of algorithms that obtain the result of queries over tree-like structures. In this thesis, we have proposed a general branch-and-bound algorithmic schema for obtaining the K optimal solutions of a given problem. Moreover, 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, in order to apply them in the design of three non-incremental branch-and-bound algorithms for K-CPQ between spatial objects indexed in two R-trees. Two of those approaches are recursive, following a Depth-First searching policy and one is iterative, obeying a Best-First traversal policy. The plane-sweep method and the search ordering (this heuristic is based on orderings of the MINMINDIST metric) 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 spatial objects with the distances within a range [Dist_Min, Dist_Max].

In the experimental chapter, we have used an R-tree variant (R*-tree) in which the spatial objects are stored directly in the leaf nodes of the tree. 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 decomposed) with respect to K, the size of the underlying buffer schema, the disjointedness of the workspaces and the scalability of the algorithms. 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 disk accesses when we include a global LRU buffer for all configurations. But, it consumes so 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 for 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 to this performance measurement. 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 affected in a minor degree in contrast to the combination of the recursion in a Depth-First searching policy 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 decomposed (Best-First traversal 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 disk accesses, since the increase of this metric grows in sub-linear way with the increase of K. But, the other performance measurements are affected in a major degree.
- The number of disk accesses grows almost linearly with the increase of the cardinalities of the spatial datasets involved in the query, and this trend is maintained for the other performance measurements.

We have also implemented and presented experimental results for three special cases of closest pairs queries: (1) K-Self-CPQ, where both datasets actually refer to the same entity, (2) Semi-CPQ, where for each element of the first dataset, the closest object of the second dataset is computed, and (3) 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 when we have buffer. Finally, the incremental algorithms outperform our pseudo-incremental approach if small or medium part of the result is needed, while if we require a large or the entire distance join result, the performance of the incremental algorithms is penalized and our pseudo-incremental approach obtained interesting performance measurements only if X is good enough.

Future work may include:

- 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 [MaP99, PMT99].
- The analytical study (cost model) of K-CPQs, extending related work on spatial joins [HJR97b, TSS98], nearest neighbor queries [PaM97b, BBK97] and taking into account the effect of buffering [LeL98].
- The extension of our K-CPQ algorithms using different spatial access methods that coexist in the same spatial database system, as the case of R-trees and Linear Region Quadtrees [CVM99].
- 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 and indicates the maximum tolerance between the reported answer and the exact closest pair distance) in a sense similar to the approximate nearest neighbor searching proposed in [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 (JACM), Vol.45, No.6, pp.891-923, 1998.

[APR98] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel and J.S. Vitter: "Scalable Sweeping-Based Spatial Join", *Proceedings 24th VLDB Conference*, pp.570-581, New York, NY, 1998.

[BaM72] R. Bayer and E.M. McCreight: "Organization and Maintenance of Large Ordered Indices", *Acta Informatica*, Vol.1, No.3, pp.173-189. 1972.

[BBK97] S. Berchtold, C. Böhm, D.A. Keim and H.P. Kriegel: "A Cost Model for Nearest Neighbor Search in High-Dimensional Data Space", *Proceedings 18th ACM PODS Symposium (PODS'97)*, pp.78-86, Tucson, AZ, 1997.

[BJK97] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza and N. MacNaughton: "The Oracle Universal Server Buffer", *Proceedings 23rd VLDB Conference*, pp.590-594, Athens, Greece, 1997.

[BKK96] S. Berchtold, D.A. Keim and H.P. Kriegel: "The X-tree: An Index Structure for High-Dimensional Data", *Proceedings 22nd VLDB Conference*, pp.28-39, Bombay, India, 1996.

[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 1990 ACM SIGMOD Conference*, pp.322-331, Atlantic City, NJ, 1990.

[BKS93a] T. Brinkhoff, H.P. Kriegel and B. Seeger: "Efficient Processing of Spatial Joins Using R-trees", *Proceedings 1993 ACM SIGMOD Conference*, pp.237-246, Washington, DC, 1993.

[BKS93b] T. Brinkhoff, H.P. Kriegel and R. Schneider: "Comparison of Approximations of Complex Objects Used for Approximation-based Query Processing in Spatial Database Systems", *Proceedings 9th IEEE International Conference on Data Engineering (ICDE'93)*, pp.40-49, Vienna, Austria, 1993.

[BKS94] T. Brinkhoff, H.P. Kriegel, R Schneider and B Seeger: "Multi-Step Processing of Spatial Joins", *Proceedings 1994 ACM SIGMOD Conference*, pp.197-208, Minneapolis, Minnesota, 1994.

[BrB95] G. Brassard and P. Bratley: "Fundamentals of Algorithmics", *Prentice Hall*. 1995.

[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, Stockholm, Sweden, 1985.

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

[ChW83] F.Y. Chin and C.A. Wang: “Optimal Algorithms for the Intersection and the Minimum Distance Problems Between Planar Polygons”, *IEEE Transactions on Computers*, Vol.32, No.12, pp.1203-1207, 1983.

[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.

[CLR90] T.H. Cormen, C.E. Leiserson and R.L. Rivest: “Introduction to Algorithms”, *Mc-Graw-Hill Companies Inc*. 1990.

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

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

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

[CVM99] A. Corral, M. Vassilakopoulos and Y. Manolopoulos: “Algorithms for Joining R-Trees and Linear Region Quadtrees”, *Proceedings 6th of Advances in Spatial Databases (SSD’99)*, pp.251-269, Hong Kong, China, 1999.

[CVM01] A. Corral, M. Vassilakopoulos and Y. Manolopoulos: “The Impact of Buffering for the Closest Pairs Queries using R-trees”, *Proceedings 5th of Advances in Databases and Information Systems (ADBIS’01)*, pp.41-54, Vilnius, Lithuania, 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.

[Ede85] H. Edelsbrunner: “Computing the Extreme Distances Between Two Convex Polygons”, *Journal of Algorithms*, Vol.6, No.2, pp.213-224, 1985.

[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 Annual IEEE Symposium on Foundations of Computer Science (FOCS'93)*, pp.714-723, Palo Alto, CA, 1993.

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

[Güt94] R.H. Güting: "An Introduction to Spatial Database Systems", *VLDB Journal*, Vol.3, No.4, pp.357-399, 1994.

[HJR97a] 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, Athens, Greece, 1997.

[HJR97b] Y.W. Huang, N. Jing and E.A. Rundensteiner: "A Cost Model for Estimating the Performance of Spatial Joins Using R-trees". *Proceedings 9th International Conference on Scientific and Statistical Database Management (SSDBM'97)*, pp.30-38. Olympia, WA, 1997.

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

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

[HoS78] E. Horowitz and S. Sahni: "Fundamentals of Computer Algorithms", *Computer Science Press*, 1978.

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

[JoS94] T. Johnson and D. Shasha: "2Q: a Low Overhead High Performance Buffer Management Replacement Algorithm", *Proceedings 20th VLDB Conference*, pp.439-450, Santiago, Chile, 1994.

[KaF93] I. Kamel and C. Faloutsos: "On Packing R-trees", *Proceedings of the 2nd International Conference on Information and Knowledge Management*, pp.490-499, Washington DC, USA, 1993.

[KaS97] N. Katayama and S. Satoh: "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries", *Proceedings 1997 ACM SIGMOD Conference*, pp.369-380, Tucson, AZ, 1997.

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

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

[LeL98] S.T. Leutenegger and M.A. Lopez: "The Effect of Buffering on the Performance of R-Trees", *Proceedings 14th IEEE International Conference on Data Engineering (ICDE'98)*, pp.164-171, Orlando, FL, 1998.

[LEL97] S.T. Leutenegger, J.M. Edgington and M.A. Lopez: "STR: A Simple and Efficient Algorithm for R-Tree Packing", *Proceedings of the 13th International Conference on Data Engineering*, pp.497-506, Birmingham, United Kingdom, 1997.

[LiC91] M.C. Lin and J.F. Canny: "A Fast Algorithm for Incremental Distance Calculation", *Proceedings of IEEE International Conference on Robotics and Automation*, pp.1008-1014, Sacramento, CA, 1991.

[LoR94] M.L. Lo and C.V. Ravishankar: “Spatial Joins Using Seeded Trees”, *Proceedings 1994 ACM SIGMOD Conference*, pp.209-220, Minneapolis, Minnesota, 1994.

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

[MaP99] N. Mamoulis and D. Papadias: “Integration of Spatial Join Algorithms for Processing Multiple Inputs”, *Proceedings 1999 ACM SIGMOD Conference*, pp.1-12, Philadelphia, PA, 1999.

[MTT99] Y. Manolopoulos, Y. Theodoridis and V. Tsotras: “Advanced Database Indexing”, *Kluwer Academic Publishers*, 1999.

[NHS84] J. Nievergelt, H. Hinterberger and K.C. Sevcik: “The Grid File: An Adaptable, Symmetric Multikey File Structure”, *ACM Transactions on Database Systems (TODS)*, Vol.9, No.1, pp.38-71, 1984.

[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.

[OrM84] J.A. Orenstein and T.H. Merrett: “A Class of Data Structures for Associative Searching”, *Proceedings 3rd ACM PODS Symposium (PODS’84)*, pp.181-190, Waterloo, Canada, 1984.

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

[PaM97a] A.N. Papadopoulos and Y. Manolopoulos: “Nearest Neighbor Queries in Shared-Nothing Environments”, *Geoinformatica*, Vol.1, No.4, pp.369-392, 1997.

[PaM97b] A.N. Papadopoulos and Y. Manolopoulos: “Performance of Nearest Neighbor Queries in R-Trees”, *Proceedings 6th International Conference on Database Theory (ICDT'97)*, pp.394-408, Delphi, Greece, 1997.

[PMT99] D. Papadias, N. Mamoulis and Y. Theodoridis: “Processing and Optimization of Multi-way Spatial Joins Using R-trees”, *Proceedings 18th ACM PODS Symposium (PODS'99)*, pp.44-55, Philadelphia, PA, 1999.

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

[Rob81] J.T. Robinson: “The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes”, *Proceedings 1981 ACM SIGMOD Conference*, pp.10-18, Ann Arbor, MI, 1981.

[RoL85] N. Roussopoulos and D. Leifker: “Direct Spatial Search on Pictorial Databases Using Packed R-Trees”, *Proceedings 1985 ACM SIGMOD Conference*, pp.17-31, Austin, TX, 1985.

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

[RSV01] P. Rigaux, M. Scholl and A. Voisard: “Spatial Databases with Application to GIS”, *Morgan Kaufmann Publishers Inc.*, 2001.

[Sam90a] H. Samet: “The Design and Analysis of Spatial Data Structures”, *Addison-Wesley*, 1990.

[Sam90b] H. Samet: “Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS”, *Addison-Wesley*, 1990.

[SaS82] G.M. Sacco and M. Schkolnick: “A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model”, *Proceedings 8th VLDB Conference*, pp.257-262, Mexico City, Mexico, 1982.

[SML00] H. Shin, B. Moon and S. Lee: “Adaptive Multi-Stage Distance Join Processing”, *Proceedings 2000 ACM SIGMOD Conference*, pp.343-354, Dallas, TX, 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, Brighton, UK, 1987.

[SYU00] Y. Sakurai, M. Yoshikawa, S. Uemura and H. Kojima: “The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation”, *Proceedings 26th VLDB Conference*, pp.516-526, Cairo, Egypt, 2000.

[TSS98] Y. Theodoridis, E. Stefanakis and T. Sellis: “Cost Models for Join Queries in Spatial Databases”, *Proceedings 14th IEEE International Conference on Data Engineering (ICDE'98)*, pp.476-483, Orlando, FL, 1998.

[YuM98] C.T. Yu and W. Meng: “Principles of Database Query Processing for Advanced Applications”, *Morgan Kaufmann Publishers Inc.*, 1998.