Available online at www.sciencedirect.com

ScienceDirect

Information Sciences 177 (2007) 2207–2237

INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL

www.elsevier.com/locate/ins

A performance comparison of distance-based query algorithms using R-trees in spatial databases

Antonio Corral [*], Jesús M. Almendros-Jiménez

Department of Languages and Computing, University of Almeria, 04120 Almeria, Spain

Received 6 November 2004; received in revised form 20 July 2006; accepted 22 December 2006

Abstract

Efficient processing of distance-based queries (DBQs) is of great importance in spatial databases due to the wide area of applications that may address such queries. The most representative and known DBQs are the K Nearest Neighbors Query (KNNQ), ρ Distance Range Query (ρDRQ), K Closest Pairs Query (KCPQ) and ρ Distance Join Query (ρDJQ). In this paper, we propose new pruning mechanism to apply them in the design of new Recursive Best-First Search (RBFS) algorithms for DBQs between spatial objects indexed in R-trees. RBFS is a general search algorithm that runs in linear space and expands nodes in best-first order, but it can suffer from node re-expansion overhead (i.e. to expand nodes in best-first order, some nodes can be considered more than once). The R-tree and its variations are commonly cited spatial access methods that can be used for answering such spatial queries. Moreover, an exhaustive experimental study was also included using R-trees, which resulted to several conclusions about the efficiency of proposed RBFS algorithm and its comparison with respect to other search algorithms (Best-First Search (BFS) and Depth-First Branch-and-Bound (DFBnB)), in terms of disk accesses, response time and main memory requirements, taking into account several important parameters as maximum branching factor (*Cmax*), cardinality of the final query result (K), distance threshold (ρ) and size of a global LRU buffer (B). In general RBFS is competitive for KNNQ and KCPQ where the maximum branching factor (*Cmax*) is large enough (even better than DFBnB and very close to BFS), and it is a good alternative when we have main memory limitations in our computer due to high process overload in our system, since it is linear space consuming with respect to the height of the R-trees. Nevertheless, RBFS is the worst alternative for ρDRQ and ρDJQ. DFBnB is also a linear space algorithm and it obtains the same behavior as BFS for ρDRQ and ρDJQ; and it is the best when an LRU buffer was included. Finally, we have been able to check experimentally that BFS is the best for all DBQs, but it can consume many main memory resources to perform spatial queries.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Spatial databases; Query processing; Search algorithms; Distance-based queries; R-trees; Performance study

[*] Corresponding author. Tel.: +34 950015844; fax: +34 950015129.
E-mail addresses: acorral@ual.es (A. Corral), jalmen@ual.es (J.M. Almendros-Jiménez).

0020-0255/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2006.12.012

1. Introduction

A *spatial database system* (SDBS) is a database system that offers spatial data types in its data model and query language and supports spatial data types in its implementation, providing at least spatial indexing and efficient spatial query processing [32]. In a computer system, these spatial data are represented by points, line-segments, regions, polygons, volumes and other kinds of 2-d/3-d geometric entities and are usually referred to as spatial objects. The spatial access methods (SAMs) manage these spatial data to accelerate the spatial query processing, and one of the most cited SAM is the R-tree [18]. The main reason that has caused the active study of spatial database management systems (SDBMS) comes from the needs of the existing applications such as geographical information systems (GIS), computer-aided design (CAD), very large scale integration design (VLSI), multimedia information systems (MIS), data warehousing, etc. Recently, the role of spatial databases is continuously increasing in many modern applications; e.g. mapping, urban planning, transportation planning, resource management, geo-marketing, archeology and environmental modeling are just some of these applications.

The key characteristic that makes a spatial database a powerful tool is its ability to manipulate spatial data, rather than simply to store and represent them. The most basic form of such a system is answering spatial queries related to the spatial properties of data. Some typical spatial queries are the following: (1) *Point query* [18] that finds all spatial objects that contain a given query point. (2) *Range query* [18] that seeks for the spatial objects that are contained within a given region (usually expressed as a rectangle or a sphere). (3) *Spatial join* [6,21,23,19,15,8] that reports all pairs of spatial objects from two spatial datasets that satisfy a spatial predicate. (4) Finally, very common is the *nearest neighbor query* [30,20] that seeks for the spatial object residing more closely to a given reference spatial object. *Spatial aggregate queries* [32] are usually variants of the nearest neighbor problem, and they involve a specific region of space and ask for the value of some aggregate functions (e.g. count, sum, min, max, average, etc.) for which we have measurements for this given region.

The distance functions are typically based on a distance metric (satisfying the non-negative, identity, symmetry and Δ-inequality properties) defined on points in the data space. A general distance metric is called L_t-distance (L_t) or Minkowski distance between two points, $p = (p_1, p_2, \ldots, p_d)$ and $q = (q_1, q_2, \ldots, q_d)$, in the d-dimensional data space, D^d, and it is defined as follows:

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

For $t = 2$ we have the Euclidean distance and for $t = 1$ the Manhattan distance. They are the most known L_t-distances. Often, the Euclidean distance is used as the distance function but, depending on the application, other distance functions may be more appropriate. The *d-dimensional Euclidean space*, E^d, is the pair (D^d, L_2). That is, the d-dimensional Euclidean space is the d-dimensional data space, D^d, with the Euclidean distance. In the following we will use d_p instead of L_2 as the *Euclidean distance* between two points and it will be the basis for DBQs studied on this paper.

In spatial database applications, the nearness or farness of spatial objects is examined by performing DBQs. The most known DBQs in the spatial database framework that report the top K answers (K is a positive number given by the users) are the K nearest neighbors (KNNQ) and the K closest pairs queries (KCPQ), where the nearest neighbor and the closest pair are particular cases for $K = 1$. They are considered DBQs, because they use distance functions to measure the degree of nearness between spatial objects, reporting only the top K. Other kind of DBQs that involve distance thresholds and report all spatial objects that fall on a distance range are called ρ *Distance Range Query* (ρDRQ) and ρ *Distance Join Query* (ρDJQ). The former reports all spatial objects from a spatial objects dataset that fall on the distance range defined by $[\rho_1, \rho_2]$ with respect to a query object, and the latter finds all the possible pairs of spatial objects from two different spatial objects datasets, having a distance between ρ_1 and ρ_2 of each other (a special case of ρDJQ is *Buffer Query* [8], where two spatial datasets are involved and $\rho_1 = 0$ and $\rho_2 > 0$). Their definitions for point datasets (the extension of these definitions to other complex spatial objects is straightforward) are the following:

Definition (*K Nearest Neighbors Query* (*KNNQ*)). Let P be set of points ($P \neq \emptyset$) in E^d and a positive real number, K. Then, the result of the K nearest neighbors query with respect to a query point q is a set $\mathrm{KNNQ}(P, q, K) \subseteq P$ of ordered sequences of K ($1 \leqslant K \leqslant |P|$, where $|P|$ is the number of points in the dataset P, i.e. its cardinality) different points of P, with the K smallest distances from the query object q:

$$\mathrm{KNNQ}(P,q,K) = \left\{ \begin{array}{l} (p_1, p_2, \ldots, p_K) \in P : p_i \neq p_j, i \neq j, 1 \leqslant i,j \leqslant K \quad \text{and} \\ \quad \forall p_i \in P - \{p_1, p_2, \ldots, p_K\}, \\ d_p(p_1, q) \leqslant d_p(p_2, q) \leqslant \cdots \leqslant d_p(p_K, q) \leqslant d_p(p_i, q) \end{array} \right\}$$

Definition (ρ *Distance Range Query* (ρ*DRQ*)). Let P be a set of points ($P \neq \emptyset$) in E^d, and a distance range defined by $[\rho_1, \rho_2]$ such that $\rho_1, \rho_2 \in \Re^+$ and $\rho_1 \leqslant \rho_2$. Then, the result of the ρ distance range query with respect to a query point q is a set $\rho\mathrm{DRQ}(P, q, \rho_1, \rho_2) \subseteq P$, which contains all points ($p_i \in P$) that fall on the distance range $[\rho_1, \rho_2]$ from the query point q:

$$\rho\mathrm{DRQ}(P, q, \rho_1, \rho_2) = \{p_i \in P : \rho_1 \leqslant d_p(p_i, q) \leqslant \rho_2\}$$

Definition (*K Closest Pairs Query* (*KCPQ*)). Let P and Q be two sets of points ($P \neq \emptyset$ and $Q \neq \emptyset$) in E^d, and a positive real number, K. Then, the result of the K closest pairs query is a set $\mathrm{KCPQ}(P, Q, K) \subseteq (P \times Q)$ of ordered sequences of K ($1 \leqslant K \leqslant |P| \cdot |Q|$) different pairs of points of $P \times Q$, 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:

$$\mathrm{KCPQ}(P,Q,K) = \left\{ \begin{array}{l} ((p_1, q_1), (p_2, q_2), \ldots, (p_K, q_K)) \in (P \times Q) : \\ p_1, p_2, \ldots, p_K \in P, q_1, q_2, \ldots, q_K \in Q \quad \text{and} \\ (p_i, q_i) \neq (p_j, q_j), i \neq j, 1 \leqslant i,j \leqslant K \quad \text{and} \\ \forall (p_i, q_j) \subseteq P \times Q - \{(p_1, q_1), (p_2, q_2), \ldots, (p_K, q_K)\}, \\ d_p(p_1, q_1) \leqslant d_p(p_2, q_2) \leqslant \cdots \leqslant d_p(p_K, q_K) \leqslant d_p(p_i, q_j) \end{array} \right\}$$

Definition (ρ *Distance Join Query* (ρ*DJQ*)). Let P and Q be two sets of points ($P \neq \emptyset$ and $Q \neq \emptyset$) in E^d, and a distance range defined by $[\rho_1, \rho_2]$ such that $\rho_1, \rho_2 \in \Re^+$ and $\rho_1 \leqslant \rho_2$. Then, the result of the ρ distance join query is a set $\rho DJQ(P, Q, \rho_1, \rho_2) \subseteq (P \times Q)$, which contains all the possible different pairs of points that can be formed by choosing one point of P and one point of Q, having a distance between ρ_1 and ρ_2 of each other:

$$\rho\mathrm{DJQ}(P, Q, \rho_1, \rho_2) = \{(p_i, q_j) \subseteq P \times Q : \rho_1 \leqslant d_p(p_i, q_j) \leqslant \rho_2\}$$

DBQs are very useful in many applications that use spatial data for decision making and other demanding data handling operations. For example, we can use two spatial datasets that represent the cultural landmarks and the most populated places of the United States of America. A KNNQ can report the K nearest populated places to Chicago in increasing order of its distances, and a KCPQ can discover the K closest pairs of cities and cultural landmarks providing a increase order based on its distances. On the other hand, a ρDRQ will find all cultural landmarks that are within 50 kilometers from Chicago, and a ρDJQ will return populated place-cultural landmark pairs that are within 10 kilometers of each other (it is an example of Buffer Query [8]).

Numerous algorithms exist for answering DBQs using R-trees or other tree-like structures. Most these algorithms are focused in the KNNQ over R-trees as spatial access method, from the incremental [20,31] or non-incremental [30,10] point of view. For the KCPQ in spatial databases using R-trees, [19,33,15,16,29] are the most representative references in the literature. In [19,33], incremental algorithms based on Best-First Search and additional priority queues were presented; and in [15,16], non-incremental algorithms following Depth-First and Best-First Searches were proposed for solving the KCPQ in spatial databases (in [19] is studied KCPQ with spatial constraints). *Non-incremental processing* is a technique that reports the elements of a query result all together, at the end of the algorithm's execution, assuming that the cardinality of the result K is known in advance, i.e. the user cannot have any result until the algorithm ends. *Incremental processing* is a technique that reports the desired elements of a query result in ascending order of distance in a pipelined fashion (one-by-one), i.e. the user can have part of the final result before the end of the algorithm's execution. In other words, when an incremental algorithm has obtained K elements of the result, it is not necessary to restart the algorithm to find the $(K+1)$th element, but just to perform an additional step. ρDRQ [32] and ρDJQ [8] have not been studied in-depth in the literature of spatial databases, because they are nature extensions of KNNQ and KCPQ, respectively.

In this paper we mainly propose new Recursive Best-First Search algorithms in non-incremental way for the most representative DBQs (KNNQ, ρDRQ, KCPQ and ρDJQ) over R-trees and compare them with respect to the Best-First Search (BFS) and Depth-First Branch-and-Bound (DFBnB) algorithms. Moreover, we also present an extensive performance study based on experimental results of the DBQ algorithms applied on R*-trees [1] with synthetic (uniform) and real datasets, primarily in terms of the I/O activity and response time, drawing conclusions based on the performance comparison taking into account several execution parameters to provide useful insights on the choice of search algorithms to researchers and developers.

The rest of the paper is organized as follows. In Section 2, we review previous work on search algorithms and on DBQs using R-trees; and we describe the main contributions of this paper. Section 3 presents a survey of the R-tree as the most cited spatial access method; the distance functions and pruning mechanisms used in the design of branch-and-bound algorithms for DBQs using R-trees; and finally, the Recursive Best-First Search algorithms for KNNQ, ρDRQ, KCPQ and ρDJQ over R-trees indexing spatial objects are presented. In Section 4, experimental results of the proposed algorithms (RBFS) are presented and compared with respect to other R-tree implementations of search algorithms (BFS and DFBnB) for DBQs using different data distributions (uniform and real) and taking into account the buffering effect for KCPQ and ρDJQ. Finally, Section 5 presents the conclusions of this paper and gives directions for future work.

2. Related works and motivation

Search is a fundamental problem-solving technique and it consists on a systematic exploration of the space in order to find one or more goal solutions that have specified properties. In this paper, we focus on the search space on trees, where leaf nodes are goal nodes and the number of children of a node is refereed by *branching factor* of such node. An important application framework of search algorithms is the query processing over tree-like structures in spatial databases, e.g. algorithms for DBQs over R-trees. In this paper, we deal with a new search algorithm, following a Recursive Best-First traversal policy, for DBQs in non-incremental way using R-trees.

2.1. Search algorithms

In general, a search algorithm is an algorithm that takes a problem as input and returns a solution to such a problem, usually after evaluating a number of possible solutions. In the context of tree structures, a search algorithm is a strategy to decide which node is the next to explore [38]. Here, we review the most representative search algorithms that are widely used for problem solving, they are Best-First Search (BFS), Depth-First Branch-and Bound (DFBnB) and Recursive Best-First Search (RBFS). An excellent survey can be found in [38].

Best-First Search maintains a partial expanded tree, and each iteration expands the node with minimum cost, among all nodes that have been generated but not expanded yet, until the optimal result is obtained. To maintain a partial expanded tree is necessary to manage a global list, so-called *global_list*, and its size can be exponential in the search depth. Special cases of BFS are the A^* algorithm [26] and Breadth-First search. For example, the latter never generates a node until all the nodes at shallower levels have been previously generated, i.e. the nodes are visited level by level and the cost function is the depth of the node in the tree.

Depth-First Branch-and-Bound starts from the root, and with a global upper bound u of the lowest-cost solution found so far, it always selects the most recently generated node or the deepest node to be expanded next. Whenever a new leaf node is reached whose cost is less than u, u is updated with this lower cost. Whenever a new internal node is selected for expansion whose cost is larger than u, it is pruned, since all descendents of such internal node must have costs at least as large as their ancestors. In order to find good candidate nodes quickly, the new generated child nodes should be searched in an increasing order of their costs, and it is called *node-ordering*. Moreover, DFBnB uses space that is only linear in the search depth and it is very easy to implement using the recursion, which is available in most of the programming languages. Iterative-Deepening [38] can be considered a variant of DFBnB, which performs a series of DFBnB iterations using global variables for pruning.

Recursive Best-First Search [22] is a Best-First Search that runs in space that is linear (using the recursion) with respect to the maximum search depth (i.e. RBFS is a linear-space best-first search algorithm). It works by maintaining on the recursion stack the complete path to the current node being expanded, as well as all immediate siblings of nodes on that path, along with the cost of the best node in the subtree expanded below each sibling. Whenever the cost of the current node exceeds some other node in the previously expanded portion of the tree, the algorithm backs-up to their deepest common ancestor, and continues the search down the new path. The algorithm maintains a separate local pruning threshold for each subtree diverging from the current search path.

Fig. 1 contains the algorithmic descriptions in pseudo-code of BFS (1.a), DFBnB (1.b) and RBFS (1.c); where *root* represents the root node and *cost*(n) is the cost of a node n. For BFS algorithm, the *global_list* can be implemented as a priority queue sorted by node cost values and it is used to maintain the nodes that have been generated but not expanded yet and it starts with BFS (root). DFBnB algorithm uses *node-ordering* and, it starts with DFBnB (root) and initially $u \leftarrow \infty$ (u is a global upper bound of the lowest cost values found so far, and it is used to prune branches whose cost is greater than or equal to this bound). Finally, RBFS algorithm starts with RBFS(root, cost(*root*), ∞) and *Fb*[n] is the *stored value* of node n and u is a local upper bound. *Fb*[i] denotes the current stored value of the ith child in the sorted order of children, and the *stored value* of a node n(*Fb*[n]) is a lower bound of the static values (the *static value* of a node n corresponds

a

```
BFS(root)
01 global_list ← ∅; n ← root;
02 while (n is not a goal node)
03   expand n, generating and evaluating all its children;
04   insert all its children into global_list;
05   delete n from global_list;
06   n ← minimum-cost node in global_list;
```

Best-First Search algorithm.

b

```
DFBnB(n)
01 generate all x children of n: n_1, n_2, ..., n_x;
02 evaluate and sort them in increasing order of cost;
03 for (i ← 1 to x)
04   if (cost(n_i) < u)
05     if (n_i is a goal node) u ← cost(n_i);
06     else DFBnB(n_i);
07   else return;
08 return;
```

Depth-First Branch-and-Bound algorithm.

c

```
RBFS(n, Fb[n], u)
01 if (cost(n) > u) return cost(n);
02 if (n is a goal node) exit with optimal goal node n;
03 if (n has no children) return ∞;
04 for (each child n_i of n)
05   if (cost(n_i) < Fb[n]) Fb[i] ← max{Fb[n], cost(n_i)};
06   else Fb[i] ← cost(n_i);
07 sort n_i and Fb[i] in increasing order of Fb[i];
08 if (only one child) Fb[2] ← ∞;
09 while (Fb[1] ≤ u and Fb[1] < ∞)
10   Fb[1] ← RBFS(n_1, Fb[1], min{u, Fb[2]});
11   insert n_1 and Fb[1] in increasing-sorted order of Fb[i];
12 return Fb[1];
```

Recursive Best-First Search algorithm.

Fig. 1. Search algorithms.

to its cost, *cost*(*n*)) of its children, and it should be set to the maximum of their parent's stored values and its own static value. The stored value is passed down to its children and it is used to control the node re-expansion, following a best-first order. The *u* parameter for RBFS is a local cost threshold of the subtree below a given node, and it is used to stop the exploration of the subtree below such a node.

If we compare these three search algorithms, we can observe that there are two major differences between BFS and DFBnB. One is that DFBnB runs in space linear in the search depth, whereas BFS can require space exponential in the search depth (i.e. the main drawback of BFS is its memory requirements, and in some applications it is severely space-limited). The other difference is that DFBnB may expand nodes whose costs are larger than the optimal cost, whereas BFS does not. On the other hand, as DFBnB, the space complexity of RBFS is linear in the search depth; and as BFS, RBFS expands nodes in best-first order, but it suffers from node re-expansion overhead. Fig. 2 illustrates how the search algorithms (BFS, DFBnB and RBFS) work on a binary tree (the numbers in the nodes are the costs) in order to find the optimal goal (node 4), with the numbers next to the nodes represent the order in which the nodes are visited [38].

DFBnB and RBFS maintain in memory the current search path using the recursion, and thus their space complexity is linear in the search depth. For example, DFBnB traverses as far as possible along each branch (if the pruning condition is not satisfied) before backtracking. We can also observe the node re-expansion overhead for RBFS, when the nodes 1 and 2 are visited twice (node 1 $(2,4)$ and node 2 $(3,8)$). For the interested reader, in [38], the complete trace of the RBFS algorithm for this binary tree is thoroughly described. BFS maintains the expanded tree partially in main memory (global_list) and each cycle expands the node of minimum cost, and thus, in the worst case, BFS can require space (amount of main memory) that can be exponential in the search depth. For example, if depth = 4, the maximum size of *global_list* before reaching the optimal goal is 4, $2^{\text{depth}-2}$ $([0,1,2,3,5,4,6]$ since the nodes 0, 1 and 2 have been deleted before its expansion during the execution of the algorithm).

2.2. Distance-based queries using R-trees

Numerous algorithms exist for answering DBQs in spatial databases, but most these algorithms are focused in the KNNQ and KCPQ over spatial access methods as R-trees, from the *incremental* [20,19,33] or *non-incremental* [30,15,16] point of view. The first KNN search algorithm for R-trees was proposed in [30], and it traverses the R-tree in a DFBnB manner improved with node-ordering, accessing pages in an order induced by the hierarchy of the index structure. Assume that we want to search the nearest neighbor ($K = 1$) of point query q in the R-tree R. Starting from the root, all entries are sorted according to their *MinDist* [30] values from q, and the entry with the smallest *MinDist* values is visited first. The process is repeated recursively until the leaf level, where a potential nearest neighbor is found. During the backtracking phase to the upper levels, the algorithm visits entries whose *MinDist* values are smaller than or equal to the distance of the nearest neighbor found so far. This algorithm was enhanced in [10], proving that any page can be pruned by using *MinMaxDist* [30] distance function (between a point and an MBR), since its computation can cause an additional overhead. The performance of this algorithm was shown to be suboptimal in [28] and the worst-case space complexity of this algorithm is O($h * Cmax$), where *Cmax* is the maximum branching factor of the R-tree and h is its height.

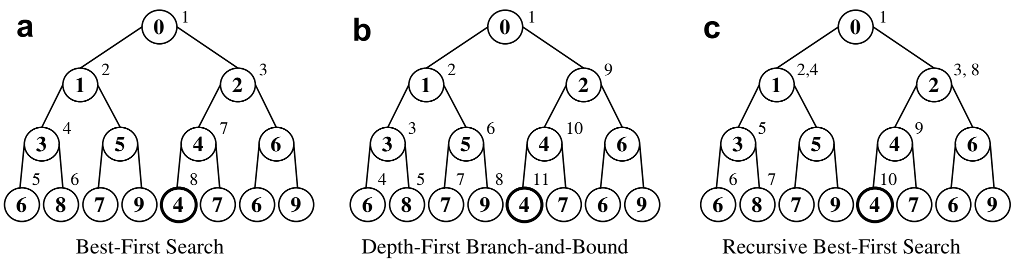

Fig. 2. The order of visited nodes in the search algorithms: (a) BFS, (b) DFBnB and (c) RBFS.

In some applications, the number of desired neighbors (K) is fixed in advance, but in other applications this number is unknown. In the last case is when the incremental nearest neighbor algorithms (ranking queries) [20] are useful. The way to implement this kind of algorithm is using a BFS algorithm, keeping a minimum binary heap [13], *global_list*, with the entries of the nodes visited so far (internal and leaf nodes). Therefore, the nodes of the R-tree are accessed in the order of increasing distance to the query point. This algorithm was proved to be I/O optimal in [2], since it only visits the necessary nodes for obtaining the desired result. Moreover, the worst-case space complexity (maximum size of *global_list*) of this algorithm is $O(n)$, where n is the total number of elements in the R-tree [3]. A version of KNNQ following Breadth-First search, which can be considered a special case of BFS, for multi-user environments can be found in [9].

ρDRQ is a special case of *Region Query* [17], which permits search regions to have arbitrary orientations and shape. In our case, the region query is defined by a point query and an interval of distances, generating different circular shapes (e.g. if $\rho_1 = 0$ and $\rho_2 > 0$, then the region is a circle; if $\rho_1 = \rho_2 \neq 0$, then the region is a circumference; if $\rho_1 = \rho_2 = 0$, then the spatial objects intersect or they can be identical; etc.). ρDRQ can be considered a *Buffer Query* [32] over one spatial dataset, since a *buffer*, in the spatial database terminology, refers to a region constructed outward from an isolated spatial object (e.g. point query) to a specific distance [11]. This kind of DBQ has not been studied in-depth using R-trees for spatial databases, since the researchers have paid more attention to the *Window Query* (the region query is a rectangle, where the faces are parallel to the coordinate axes) [18,1]. For high-dimensional data spaces using R-tree-like index structures, *ρ-Similarity Query* is reviewed in [3], adapting the KNNQ algorithms of [30,20]. The *ρ-Similarity Query* [3] finds all high-dimensional points (high-dimensional feature vectors), which distance from a high-dimensional query point is below (or equal to) a given distance threshold ρ.

For the KCPQ in spatial databases using R-trees, [19,33,15,16,29] are the most representative references in the literature. In [15,16] non-incremental recursive (DFBnB) and non-recursive (BFS) algorithms were presented for solving the KCPQ in spatial databases. Recently, in [29], the KCPQ with spatial constraints is addressed from the non-incremental processing point of view. The main issue of the non-incremental variant is to separate the treatment of the terminal candidates (the elements of the leaf nodes) from the rest of the candidates (internal nodes). The worst-case space complexity of the recursive algorithm (DFBnB) is $O((h_1 * Cmax_1) + (h_2 * Cmax_2))$, where $Cmax_1$ and $Cmax_2$ are the maximum branching factor of each R-tree (R_1 and R_2) involving in the query and, h_1 and h_2 are their respective heights.

In [19,33], incremental and non-recursive algorithms based on Best-First Search (BFS) and additional priority queues using R-trees for distance join queries were presented. The application of BFS for KCPQ is similar to the case of KNNQ. The techniques proposed in [19] were enhanced in [33], using the plane-sweep during the combination of the entries of the nodes and an estimation of the distance value of the Kth closest pair in order to avoid unnecessary computation of MBRs distances and insertion of the main minimum binary heap (*global_list*). The strong point of this incremental approach is that, when K is unknown in advance, the user stops when he/she is satisfied by the result. On the other hand, when the number of elements in the result grows, the amount of the required resources, to perform the query, increases as well. Thus, incremental algorithms are competitive when a small quantity of elements in the result (K) is needed, while it is penalized if large part or the entire join result is desired [19]. The worst-case space complexity of this algorithm, which follows a Best-Fist search, is $O(n * m)$, where n and m are the total number of elements in each of the two R-trees.

ρDJQ is a generalization of the *Buffer Query* [8] (it is characterized by two spatial datasets and a distance threshold), which permits search pairs of spatial objects from the two input datasets that are within distance ρ from each other. In our case, the distance threshold is a range of distances defined by an interval of distances $[\rho_1, \rho_2]$ (e.g. if $\rho_1 = 0$ and $\rho_2 > 0$, then we have the definition of *Buffer Query* and if $\rho_1 = \rho_2 = 0$, then we have the *spatial intersection join*, which retrieves all different intersecting spatial object pairs from two distinct spatial datasets [6]). This query is also related to the *Similarity Join* [24,37], where the problem of deciding if two objects are similar is reduced to the problem of determining if two high-dimensional points are within a certain distance of each other. In [8], the Buffer Query is solved for non-point (lines and regions) spatial datasets using R-trees, where efficient algorithms for computing the minimum distance for lines and regions, pruning techniques for filtering in a Depth-First Branch-and-Bound algorithm (performance comparisons with other search algorithms are not included), and extensive experimental results are presented.

Table 1
Search algorithms and DBQs using R-trees in spatial databases

	KNNQ	ρDRQ	KCPQ	ρDRJ
DFBnB	[30,10]	$\times$	[15,16]	[8]
RBFS	$\times$	$\times$	$\times$	$\times$
BFS	[20,31]	$\times$	[19,33]	$\times$

Other complex DBQs using R-trees have been studied in the literature of the spatial databases, as *Iceberg Distance Join* and *K Nearest Neighbors Join* queries. In [34], the Iceberg Distance Join Query is studied for hash-based algorithms and index-based methods (R-trees). This DBQ involves two spatial datasets, a distance threshold ρ and a cardinality threshold K $(K \geqslant 1)$. The answer is a set of pairs of spatial objects from the two input datasets that are within distance ρ from each other, provided that the first spatial object appears at least K times in the join result. On the other hand, in [4], the K Nearest Neighbors Join Query is studied for R-tree-based data structures. This DBQ involves two spatial datasets and a cardinality threshold K $(K \geqslant 1)$. The answer is a set of pairs of spatial objects from the two input datasets that includes, for each of the spatial objects of the first dataset, the pairs formed with each of its K nearest neighbors in the second dataset.

2.3. Motivation and contributions

BFS and DFBnB searches have been widely studied on DBQs using R-trees in spatial databases. BFS typically requires space that can be exponential in the heights of the R-trees (in the worst-case), and it expands the minimum number of nodes in order to find the optimal solution. DFBnB expands some nodes which costs can be larger than the optimal cost, however it runs in linear space. In Table 1, we compare the most employed search algorithms (DFBnB, RBFS and BFS) adapted to the most representative DBQs (KNNQ, ρDRQ, KCPQ and ρDJQ) using R-trees in spatial databases, and the most important conclusion from this comparison is that RBFS has not been investigated yet as search algorithm in order to report solutions to DBQs using R-trees.

RBFS expands nodes in best-first order and runs in linear space, although it can suffer from node re-expansion overhead, since the recursion is used to avoid additional data structures (e.g. priority queues). Therefore, the main contribution of this paper is to design new RBFS algorithms for DBQs in non-incremental way over R-trees, and compare them experimentally using real and synthetic (uniform) datasets with respect to the other ones (BFS and DFBnB), taking into account several execution parameters. Adapting these RBFS algorithms to others DBQs (as Iceberg Distance Join, K Nearest Neighbors Join, etc.) using R-trees is a subject for future study.

3. Recursive Best-First search algorithms for DBQs using R-trees

In this section, a brief description of R-trees is presented, pointing out the main characteristics of the R*-tree. Moreover, distance functions between MBRs and pruning mechanisms, which will be used in algorithms for answering the DBQ are reviewed. Based on distance functions between MBRs, pruning mechanisms and the RBFS traversal policy, we design search algorithms for processing DBQ using R-trees in non-incremental way. The non-incremental processing on R-trees can be achieved by separating the treatment of the leaf and the internal nodes according to the particular constraint of a DBQ.

3.1. R-trees

An R-tree [18] is a hierarchical, height balanced multidimensional data structure, designed to be used in secondary storage and it is a generalization of B-trees [12] for multidimensional data spaces. The R-trees are considered as excellent choices for indexing various kinds of spatial data (points, rectangles, line-segments, polygons, etc.) and have been adopted in known commercial systems (e.g. Informix [7], Oracle [27], etc.). They are used for the dynamic organization of a set of d-dimensional geometric objects represented by their Minimum Bounding *d*-dimensional hyper-Rectangles (MBRs). These MBRs are characterized 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 spatial object (position and extension) are maintained. Consequently, the MBR is an approximation widely employed, and the R-trees 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).

The rules obeyed by the R-tree are as follows. (1) Leaves reside on the same level. (2) 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 ($\neq$ *null*) and stored in a separate file on disk (if Oid = *null*, the geometry of the spatial object is simple (e.g. points or MBRs) and it is stored directly in the R-tree leaf nodes). (3) Every other node (internal) contains entries of the form (MBR, Addr), where Addr is the address of the child node (internal or leaf node) and MBR is the minimum bounding rectangle that encloses MBRs of all entries in that child node. (4) An R-tree of class $(Cmin, Cmax)$ has the characteristic that every node, except possibly for the root, contains between *Cmin* and *Cmax* entries, where $Cmin \leqslant \lceil Cmax/2 \rceil$ (*Cmax* and *Cmin* are also called maximum and minimum branching factors or fan-out). (5) The root contains at least two entries, if it is not a leaf. Fig. 3 depicts some points (p_i) and MBRs (M_i) on the left and the corresponding R-tree $(2, 3)$ on the right, where the R-tree nodes can be implemented as disk pages. Dotted lines denote the bounding rectangles of the subtrees that are rooted in inner nodes.

Like other spatial tree-like index structures, an R-tree 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 property of spatial containment between MBRs stored in R-tree nodes is commonly used by spatial join as well as DBQ. Another important property of the R-trees that store spatial objects in a spatial database is the *MBR face property* [30]. 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. DBQ algorithms mainly use this last property.

Many variations of R-trees have appeared in the literature, a recent monograph about R-trees (theory and applications) has been published in [25]. One of the most popular and efficient R-tree variation is the R*-tree [1]. 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 becomes overflowed, 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 and it is commonly accepted that the R*-tree is one of the most efficient R-tree variants. In this paper, we choose R*-trees to perform our experimental study.

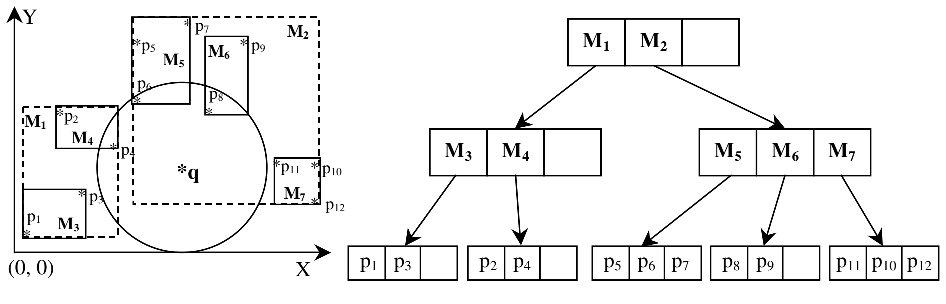

Fig. 3. An example of an R-tree from a set of 2-dimensional points.

3.2. Distance function and pruning mechanism

Usually, in the R-tree structure, we have one type of element: MBRs, although if the geometric description of the spatial objects is simple (e.g. points or MBRs) they can store directly on the leaf nodes, instead of their MBRs. In the general case, when we consider complex spatial objects, the exact geometry of the spatial objects is stored in a separate file on disk, whereas the MBRs of such spatial objects are store in the leaf nodes of the R-tree. Thus, computing the distance between MBRs is much less expensive than computing the distance between spatial objects. Moreover, when a R-tree leaf node is visited by a DBQ algorithm, we must distinguish if Oid = *null* (computing distance between points or MBRs) or Oid $\neq$ *null* (computing distances between spatial objects).

If we assume that the spatial datasets are indexed on any spatial tree-like structure belonging to the R-tree family, then the main objective while answering these types of queries is to reduce the search space. In [15,16], a generalization of the function that calculates the minimum distance between points and MBRs (*MinDist*) was presented. We can apply this distance function to pairs of any kind of elements (MBRs or points) stored in R-trees during the processing of algorithms. $MinDist(M_1, M_2)$ calculates the minimum distance between two MBRs M_1 and M_2. If any of the two (both) MBRs degenerates (degenerate) to a point (two points), then we obtain the minimum distance between a point and an MBR [30] (between two points, d_p).

Definition ($MinDist(M_1, M_2)$). Given two MBRs $M_1 = (a, b)$ and $M_2 = (c, d)$, in E^d,

$$\begin{aligned} &M_1 = (a,b), \text{ where } a = (a_1, a_2, \ldots, a_d) \quad \text{and} \quad b = (b_1, b_2, \ldots, b_d) \text{ such that } a_i \leqslant b_i\ \forall 1 \leqslant i \leqslant d, \\ &M_2 = (c,d), \text{ where } c = (c_1, c_2, \ldots, c_d) \quad \text{and} \quad d = (d_1, d_2, \ldots, d_d) \text{ such that } c_i \leqslant d_i\ \forall 1 \leqslant i \leqslant d, \end{aligned}$$

we define $MinDist(M_1, M_2)$ as follows:

$$MinDist(M_1, M_2) = \sqrt{\sum_{i=1}^{d} l_i^2}, \quad \text{such that } l_i = \begin{cases} c_i - b_i, & \text{if } c_i > b_i \\ a_i - d_i, & \text{if } a_i > d_i \\ 0, & \text{otherwise} \end{cases}$$

The most representative properties of *MinDist* distance function with respect to KCPQ and ρDJQ (for KNNQ and ρDRQ are similar, taking into account that one MBR corresponds to the query point [30]) are the following [16]:

- $MinDist(M_1, M_2)$ serves as *lower bound function* of the Euclidean distance from the K closest pairs of spatial objects enclosed by the MBRs M_1 and M_2 (lower-bounding property). It is an important property between $MinDist(M_1, M_2)$ and the distance of two spatial objects O_1 and O_2, which is the basis for the pruning mechanism. It says, given two MBRs M_1 and M_2 in E^d, enclosing two set of spatial objects $SO_1 = \{O_{1i} : 1 \leqslant i \leqslant |SO_1|\}$ and $SO_2 = \{O_{2j} : 1 \leqslant j \leqslant |SO_2|\}$, respectively. For all pair of spatial objects (O_{1i}, O_{2j}) belonging to $SO_1 \times SO_2 : MinDist(M_1, M_2) \leqslant \partial(O_{1i}, O_{2j})$. $\partial(O_1, O_2)$ is the minimum distance between the two spatial objects O_1 and O_2, defined as

$$\partial(O_1, O_2) = \min_{f_1 \in F(O_1), f_2 \in F(O_2)} \left(\min_{p_1 \in f_1, p_2 \in f_2} (d_p(p_1, p_2)) \right)$$

  where $F(O_1)$ and $F(O_2)$ denote the set of *faces* of the spatial objects O_1 and O_2 in E^d, respectively. Moreover, f_1 and f_2 are instances of the sets of faces $F(O_1)$ and $F(O_2)$.
- *MinDist* is *monotonically non-decreasing* with the R-tree heights. That is, given two MBRs M_3 and M_4, descendents of M_1 and M_2, respectively; then $MinDist(M_3, M_4)$ is always larger than or equal to $MinDist(M_1, M_2)$, i.e. $MinDist(M_3, M_4) \geqslant MinDist(M_1, M_2)$. This property comes from the *MBR enclosure property* in R-trees, where a subspace occupied by an R-tree node is always contained in the subspace of its parent node; equivalently, 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.

From the previous properties of *MinDist* distance function, the general pruning mechanism for DBQs over internal R-tree nodes (or leaf nodes when Oid = *null*, storing points or MBRs) using DFBnB and BFS traversal policies is the following: "*if MinDist*$(M_1, M_2) > z$, *then the pair of MBRs* (M_1, M_2) *will be discarded* (M_2 *can be the query point for KNNQ*)", where z is the distance value of the Kth nearest neighbor (KNNQ), or the Kth closest pair (KCPQ) that has been found so far, during the processing of the algorithm. And z is equal to the maximum distance threshold (ρ_2) for ρDRQ or ρDJQ.

In a similar way, when the leaf R-tree nodes are processed and Oid $\neq$ *null* (exact geometry of the spatial objects is stored in a separate file on disk), apart of the previous pruning mechanism between two MBRs using *MinDist*, we have to use the following condition for DBQs: "*if* $\partial(O_1, O_2) > z$, then the pair of spatial objects (O_1, O_2) *will be discarded for the final result*".

3.3. The Recursive Best-First Search algorithm for KNNQ (KNNQ_RBFS)

In this subsection we present a Recursive Best-First Search algorithm in non-incremental way (different treatment of internal and leaf R-tree nodes) to find the K nearest neighbors spatial objects. It is a linear space Best-First Search algorithm, since it runs in space that is linear with respect to the height of the R-tree using the recursion instead of an additional priority queue (*global_list*) to process internal R-tree nodes and visits the nodes in best-first order.

In order to design algorithms for processing KNNQ in a non-incremental way (K must be fixed in advance), an extra data structure that holds the K nearest neighbors is needed. This data structure is organized as a maximum binary heap [13], called *Kheap*, and holds spatial objects according to their distances (the K spatial objects from the query point with the smallest distance processed so far). The spatial object with the largest distance resides on top of the *Kheap* (the root), and we will prune the unnecessary subtrees of the R-tree using this distance value. Initially, the *Kheap* is empty and the distance of all elements inside this data structure is infinity. The spatial objects discovered at the leaf level are inserted in the *Kheap* until it gets full. Then, when a new spatial object is discovered at the leaf level, if its distance is smaller than the top of the *Kheap*, then the root is deleted and this new spatial object is inserted in the *Kheap* (updating this data structure).

In general, our Recursive Best-First algorithm for KNNQ using R-trees extends the RBFS algorithm proposed in [22] and illustrated in Fig. 1c, using *MinDist* as cost function and a pruning mechanism similar to the general pruning strategy (DFBnB and BFS) for DBQs over R-tree nodes. This algorithm uses a *local upper bound* (μ, local distance threshold of the subtree below a given MBR, which is mainly used to stop the exploration of such a subtree) for each recursive call over internal R-tree nodes. Therefore, the main idea of this algorithm for KNNQ is to traverse the subtree below a given MBR as long as it contains MBRs whose distance values (*MinDist*) are smaller than or equal to the local upper bound (μ), and at this point the algorithm returns the minimum distance value among all MBRs located in the traversed subtree. This distance value is stored for each MBR and serves also as an indicator if the subtree below the MBR has been already traversed. In this way, the essential information of the *quality* of the subtree is preserved for possible subsequent revisits. And thus, the algorithm permits to search already traversed subtrees in depth-first manner, in order to create a sequence of MBRs following a best-first order based on *MinDist* values.

The pseudo-code of the Recursive Best-First Search algorithm for KNNQ is shown in Fig. 4, so-called KNNQ_RBFS, where the spatial objects can be stored in the leaves of the R-tree with height h or external to it (using the MBR stored in the leaf nodes). To achieve the general behavior of RBFS for KNNQ using R-trees, besides the *static* distance value, *MinDist*(MBR, q), for each MBR on internal R-tree nodes, KNNQ_RBFS also maintains a *stored* distance value for each MBR, *Fb*(MBR, q). The stored value of an MBR is just a lower bound on the *MinDist* values of its children (i.e. it is a backed-up distance value), and it also serves as an indicator if the subtree below the MBR has been already traversed. *Fb*(MBR, q) is initialized with *MinDist*(MBR, q) when the MBR is considered at the first time, but when the algorithm traverses completely the subtree below the MBR up to the local upper bound and returns, this *Fb* value associated to the MBR is updated with the minimum distance value between the items in the subtree. In order to traverse the subtree below a given MBR, the local upper bound must be at least as large as its stored distance value (in this case, the algorithm maintains a local minimum binary heap associated to the current internal R-tree nodes (*FbHeap*, which item structure is $\langle Fb, MinDist, NodeAddress\rangle$) in the recursion stack, organized by their MBR *Fb* values). The recursive call associated to a given MBR will not return until the stored distances values of all items in the subtree below it exceed its local upper bound, and its new stored distance value will be updated to

```
double KNNQ_RBFS(queryPoint, KNearestObjects, nodeAddr, fValue, FbValue, $\mu$, level)
node = readNode(nodeAddr);
if (not(node.leafNode))
  FbHeap = new MinBinaryHeap(node.numOfEntries);
  for (i = 1; (i $\leq$node.numOfEntries); i++)
    MinDistValue = MinDist(node.entries[i].MBR, queryPoint);
    if (fValue < FbValue)
      Fbi = max{FbValue, MinDistValue};
    else
      Fbi = MinDistValue;
    if (((MinDistValue $\geq$Fbi) or (level $\neq$1)) and (Fbi $\leq$KNearestObjects.get_z()))
      heapElem.Fb = Fbi;                          heapElem.MinDist = MinDistValue;
      heapElem.NodeAddress = node.entries[i].Addr;  FbHeap.insert(heapElem);
  if (FbHeap.isEmpty())
    returnedValue = $\infty$;
  else
    while (not(FbHeap.isEmpty()))
      heapElem = FbHeap.deleteMinimum();
      if ((heapElem.Fb $\leq\mu$)and (heapElem.Fb < KNearestObjects.get_z()))
        if (FbHeap.isEmpty())
          minimumFb = $\infty$ ;
        else
          minimumFb = FbHeap.findMinimumDistance();
        $\mu'$ = min{$\mu$ minimumFb};
        heapElem.Fb = KNNQ_RBFS(queryPoint, KNearestObjects,
            heapElem.NodeAddress, heapElem.MinDist, heapElem.Fb, $\mu'$, level – 1);
        FbHeap.insert(heapElem);
      else
        returnedValue = heapElem.Fb;              FbHeap.makeEmpty();
  delete FbHeap;
else
  for (i = 1; (i $\leq$node.numOfEntries); i++)
    if (node.entries[i].Oid = null)
      dist = MinDist(node.entries[i].MBR, queryPoint);
      if (KNearestObjects.isFull())
        if (dist < KNearestObjects.get_z())
          KNearestObjects.deleteMaximumAndInsert(dist, node.entries[i].MBR);
      else
        KNearestObjects.insert(dist, node.entries[i].MBR);
    else
      $O_1$ = readObject(node.entries[i].Oid);          dist =$\partial$($O_1$, queryPoint);
      if (KNearestObjects.isFull())
        if (dist < KNearestObjects.get_z())
          KNearestObjects.deleteMaximumAndInsert(dist, $O_1$);
      else
        KNearestObjects.insert(dist, $O_1$);
  returnedValue = KNearestObjects.get_z();
return returnedValue;
```

Fig. 4. Pseudo-code of the Recursive Best-First Search algorithm to find the *K* nearest neighbors from a *queryPoint* using an R-tree, KNNQ_RBFS.

the minimum of these distance values. Thus, for a given MBR that have been previously expanded, its stored distance value (*Fb*) will be larger than the static one (*MinDist*). In this way, the essential information of the subtree of a given MBR is preserved for possible subsequent revisits, i.e. the MBR (according to the *Fb* value) can be considered in the future. The updated *Fb* value of a given MBR serves also as an indicator if the MBR have been already traversed. If $Fb(MBR, q) > MinDist(MBR, q)$, then the MBR have been traversed previously and the *Fb* value for its descendents should be set to the maximum of the stored distance value of the associated MBR in its node ancestor (MBR_ancestor) and its own static value (i.e. the stored distance value of a given MBR is a lower bound of the stored distance values of its descendents) as in line 07 in Fig. 4. $Fb(M_i, q) = \max\{Fb(MBR\_ancestor_i, q), MinDist(M_i, q)\}$, such that M_i belongs to the current internal R-tree node and $1 \leqslant i \leqslant$ numOfEntries. In this way, the algorithm allows to search already traversed subtrees recursively following best-first order, without unnecessary backtrackings. On the other hand, the new value of the local upper bound of a given MBR (before the recursive call) is obtained from the minimum of the local upper bound of the associated MBR in its node ancestor (MBR_ancestor) and the minimum stored distance value of the remainder MBRs in the current internal R-tree node (infinity, if there is no more MBRs in such internal node) as in line 23 in Fig. 4. $\mu(M_i) = \min\{\mu(MBR\_ancestor_i), Fb(M_j, q)\}$, such that M_i and M_j belong to the current internal R-tree node, $1 \leqslant i, j \leqslant$ numOfEntries and $i \neq j$. The initial call is $KNNQ\_RBFS(q, KNearestObjects, rootNodeAddress, \infty, \infty, \infty, h-1)$. The level of the R-tree root is $h-1$ and 0 for the leaves. An important condition of the algorithm is to avoid the revisit (re-read) of leaf R-tree nodes (line 10 in Fig. 4), since it must not consider leaf nodes (terminal nodes) that have been traversed previously $(Fb(M_i, q) > MinDist(M_i, q))$ at level immediately over the leaves (level = 1). The KNNQ_RBFS algorithm searches the K nearest neighbors (of spatial objects indexed by an R-tree with height h and maximum fan-out $Cmax$) from a query point q (queryPoint) in best-first order and the space complexity is $O(h * Cmax)$ in the worst case. The proofs for its correctness and complexity (space and time) are very similar to the original RBFS, which can be found in [22].

If we compare the search algorithm in Fig. 1c (RBFS) with respect to KNNQ_RBFS in Fig. 4, we can observe that the former is equivalent to the internal R-tree node treatment in the latter. For instance, lines 4–8 in RBFS correspond to lines 4–14 in KNNQ_RBFS, where the sorting of *Fb*[*i*] is implicit in the processing of the minimum binary heap (FbHeap). Moreover, the *while* loop for the recursive calls in lines 9–11 of RBFS (to visit new nodes o previously visited nodes in order to follow best-first order using the recursion) are equivalent to lines 16–27 in KNNQ_RBFS. Finally, the *return* line 12 in RBFS corresponds to the line 46 in KNNQ_RBFS, taking into account the internal and leaf R-tree nodes treatment.

Note that, due to ties of distances, the result of the KNNQ may not be a unique ordered sequence for a specific point dataset (lines 34 and 41 in Fig. 4). The aim of the proposed algorithm is to find one of the possible instances, although it would be straightforward to obtain all of them.

We can also deduce from the previous recursive algorithm that the pruning mechanism for RBFS is different to the one of DFBnB and BFS for internal R-tree nodes. It is based on *Fb* values instead of *MinDist* values, and it is a composed condition (line 18 in Fig. 4): "*if* $\min\{Fb(M_1, M_2)\} > \mu$ or $\min\{Fb(M_1, M_2)\} \geqslant z$, *then the pair of MBRs* (M_1, M_2) *will be discarded* (M_2 *can be the query point for KNNQ*)", where μ is the local upper bound and z is the distance value of the *K*th nearest neighbor that has been found so far (pruning distance). On the other hand, for leaf R-tree nodes the pruning mechanism is based on *MinDist* (Oid = *null*) or Δ distance functions (Oid $\neq$ *null*) as for BFS and DFBnB (lines from 30 to 45 in Fig. 4).

As an example, suppose that we want to find the three nearest neighbors ($K = 3$) to query point q in the R-tree given in Fig. 3, where the spatial objects are points which are stored in the leaf nodes with the following coordinates: $p_1 = (2,8), p_2 = (6,27), p_3 = (10,14), p_4 = (14,21), p_5 = (17,37), p_6 = (17,28), p_7 = (26,41)$, $p_8 = (30,26), p_9 = (36,38), p_{10} = (46,17), p_{11} = (37,18), p_{12} = (46,12)$, and the query point $q = (25,20)$. Moreover, the MBRs $(M_i = [(\min_x, \min_y), (\max_x, \max_y)])$ of the internal R-tree nodes correspond to $M_1 = [(2,8),(14,27)], M_2 = [(17,12),(46,41)], M_3 = [(2,8),(10,14)], M_4 = [(6,21),(14,27)], M_5 = [(17,28),$ $(26,41)], M_6 = [(30,26),(36,38)]$ and $M_7 = [(37,12),(46,18)]$. We show the steps of the algorithm before each recursive call KNNQ_RBFS with the most representative values associated to the current MBR in internal R-tree nodes (MBR, Fb, MinDist, μ', μ), and the content of the *Kheap* (KNearestObjects) $\langle(p_i, \text{dist}(p_i, q)), (p_j, \text{dist}(p_j, q)), (p_k, \text{dist}(p_k, q))\rangle$, which is empty at the beginning $\langle(*, \infty), (*, \infty), (*, \infty)\rangle$.

$(M_2, \mathbf{0.00000}, 0.00000, 11.00000, \infty)$.
$(M_6, \mathbf{7.81025}, 7.81025, 8.00000, 11.00000)$.
$\langle(*, \infty), (p_9, 21.09502), (p_8, 7.81024)\rangle, z = \infty$.
$(M_5, \mathbf{8.00000}, 8.00000, 11.00000, 11.00000)$.
$\langle(p_5, 18.78829), (p_6, 11.31370), (p_8, 7.81024)\rangle, z = 18.78829$.
$(M_1, \mathbf{11.00000}, 11.00000, 12.16553, \infty)$.
$(M_4, \mathbf{11.04536}, 11.04536, 12.16553, 12.16553)$.

$\langle(p_6, 11.31370), (p_4, 11.04536), (p_8, 7.81024)\rangle, z = 11.31370$.
The algorithm terminates because the updated **Fb** value after the recursive call is equal to z in line 18 in Fig. 4 (**if** condition is false), $Fb = 11.31370 = z$. And the number of nodes accessed is 6.

If we use the extension of the BFS algorithm to solve the KNNQ using R-trees (KNNQ_BFS in Fig. 5) to find the three nearest neighbors ($K = 3$) in non-incremental way (the *global_list* is implemented as a minimum binary heap (*GlobalMinDistHeap*) and represented its content by square brackets [...]), it executes the following steps:

Insert M_1 and M_2: $[(M_2, 0.00000), (M_1, 11.00000)], z = \infty$.
Extract $(M_2, \mathbf{0.00000}) : [(M_1, 11.00000)], z = \infty$.
Insert M_5, M_6 and M_7: $[(M_6, 7.81025), (M_5, 8.00000), (M_1, 11.00000), (M_7, 12.16553)], z = \infty$.
Extract $(M_6, \mathbf{7.81025}) : [(M_5, 8.00000), (M_1, 11.00000), (M_7, 12.16553)], z = \infty$.
$\langle(*, \infty), (p_9, 21.09502), (p_8, 7.81024)\rangle, z = \infty$.
Extract $(M_5, \mathbf{8.00000}) : [(M_1, 11.00000), (M_7, 12.16553)], z = \infty$.
$\langle(p_5, 18.78829), (p_6, 11.31370), (p_8, 7.81024)\rangle, z = 18.78829$.
Extract $(M_1, \mathbf{11.00000}) : [(M_7, 12.16553)], z = 18.78829$.
Insert M_3 and M_4: $[(M_4, 11.04536), (M_7, 12.16553), (M_3, 16.15549)], z = 18.78829$.
Extract $(M_4, \mathbf{11.04536}) : [(M_7, 12.16553), (M_3, 16.15549)], z = 18.78829$.
$\langle(p_6, 11.31370), (p_4, 11.04536), (p_8, 7.81024)\rangle, z = 11.31370$.
The algorithm terminates because, $MinDist(M_7, q) = 12.16553 > z = 11.31370$. And the number of nodes accessed is 6.

Finally, the extension of the DFBnB algorithm to solve the KNNQ using R-trees (KNNQ_DFBnB) to find the three nearest neighbors ($K = 3$) in non-incremental way is illustrated in Fig. 6. We show the steps of the algorithm before each recursive call KNNQ_DFBnB with the most representative values associated to the current MBR in internal R-tree nodes and z (MBR, MinDist, z), and the content of the *Kheap* (KNearestObjects) $\langle(p_i, \text{dist}(p_i, q)), (p_j, \text{dist}(p_j, q)), (p_k, \text{dist}(p_k, q))\rangle$, which is empty at the beginning $\langle(*, \infty), (*, \infty), (*, \infty)\rangle$.

```
KNNQ_BFS(queryPoint, KnearestObjects, rootAddr)
01 node = readNode(rootAddr);
02 GlobalMinDistHeap = new MinBinaryHeap();
03 for (i = 1; (i ≤ node.numOfEntries); i++)
04   heapElem.MinDist = MinDist(node.entries[i].MBR, queryPoint);
05   heapElem.address = node.entries[i].Addr;
06   GlobalMinDistHeap.insert(heapElem);
07 while (not(GlobalMinDistHeap.isEmpty()))
08   heapElem = GlobalMinDistHeap.deleteMinimum();
09   if (heapElem.MinDist ≤ KNearestObjects.get_z())
10     if (heapElem.address ≠ NULL_ADDR)
11       node = readNode(heapElem.address);
12     if (not(node.leafNode))
13       for (i = 1; (i ≤ node.numOfEntries); i++)
14         MinDistValue = MinDist(node.entries[i].MBR, queryPoint);
15         if (MinDistValue ≤ KNearestObjects.get_z())
16           heapElem.MinDist = MinDistValue;
17           heapElem.address = node.entries[i].Addr;
18           GlobalMinDistHeap.insert(heapElem);
19     else
20-34  // The same treatment as KNNQ_RBFS for leaf nodes, from lines 30 to 44
35   else
36     GlobalMinDistHeap.makeEmpty();
37 delete GlobalMinDistHeap;
```

Fig. 5. Pseudo-code of the Best-First Search algorithm to find the K nearest neighbors from a *queryPoint* using an R-tree, KNNQ_BFS.

```
KNNQ_DFBnB(queryPoint, KnearestObjects, nodeAddr)
01 node = readNode(nodeAddr);
02 if (not(node.leafNode))
03   LocalMinDistHeap = new MinBinaryHeap(node.numOfEntries);
04   for (i = 1; (i ≤ node.numOfEntries); i++)
05     heapElem.MinDist = MinDist(node.entries[i].MBR, queryPoint);
06     heapElem.address = node.entries[i].Addr;
07     LocalMinDistHeap.insert(heapElem);
08   while (not(LocalMinDistHeap.isEmpty()))
09     heapElem = LocalMinDistHeap.deleteMinimum();
10     if (heapElem.MinDist ≤ KNearestObjects.get_z())
11       KNNQ_DFBnB(queryPoint, KNearestObjects, heapElem.address);
12     else
13       LocalMinDistHeap.makeEmpty();
14   delete LocalMinDistHeap;
15 else
16-30 // The same treatment as KNNQ_RBFS for leaf nodes, from lines 30 to 44
```

Fig. 6. Pseudo-code of the Depth-First Branch-and-Bound Search algorithm to find the K nearest neighbors from a *queryPoint* using an R-tree, KNNQ_DFBnB.

$(M_2, \mathbf{0.00000}, \infty)$.
$(M_6, \mathbf{7.81025}, \infty)$.
$\langle(*, \infty), (p_9, 21.09502), (p_8, 7.81024)\rangle, z = \infty$.
$(M_5, \mathbf{8.00000}, \infty)$.
$\langle(p_5, 18.78829), (p_6, 11.31370), (p_8, 7.81024)\rangle, z = 18.78829$.
$(M_7$, **12.16553**, 18.78829).
$\langle(p_{11}, 12.16553), (p_6, 11.31370), (p_8, 7.81024)\rangle, z = 12.16553$.
$(M_1$, **11.00000**, 12.16553).
$(M_4$, **11.04536**, 12.16553).
$\langle(p_6, 11.31370), (p_4, 11.04536), (p_8, 7.81024)\rangle, z = 11.31370$.
The algorithm terminates because, $MinDist(M_3, q) = 16.15549 > z = 11.31370$. And the number of nodes accessed is 7. And in this example, we can observe the main drawback of the DFBnB search algorithm, when subtrees are traversed where no optimal solutions are located (e.g. subtree below M_7), and hence it needs additional node accesses before finding the desired result.

From the previous steps of KNNQ_RBFS executions (algorithm in Fig. 4), we can notice that the *Fb* values (in bold style) before each recursive call (after each item extraction from *FbHeap* when $Fb = MinDist$ (the MBR is considered at the first time), satisfying the pruning condition) are the same as MinDist values (also in bold style) after each item extraction from *global_list* in the BFS algorithm. It means that both follow a best-first order, although KNNQ_RBFS can suffer from internal R-tree node re-expansion overhead when the value of K increases, i.e. some internal R-tree nodes need to be read (visited) more than once. For instance, if we consider the previous example and we want to find the four nearest neighbors ($K = 4$) in non-incremental way using KNNQ_RBFS algorithm, the execution follows the next steps:

$(M_2, \mathbf{0.00000}, 0.00000, 11.00000, \infty)$.
$(M_6$, **7.81025**, 7.81025, 8.00000, 11.00000).
$\langle(*, \infty), (*, \infty), (p_9, 21.09502), (p_8, 7.81024)\rangle, z = \infty$.
$(M_5$, **8.00000**, 8.00000, 11.00000, 11.00000).
$\langle(p_7, 21.02380), (p_5, 18.78829), (p_6, 11.31370), (p_8, 7.81024)\rangle, z = 21.02380$.
$(M_1, \mathbf{11.00000}, 11.00000, 12.16553, \infty)$.
$(M_4$, **11.04536**, 11.04536, 12.16553, 12.16553).
$\langle(p_5, 18.78829), (p_6, 11.31370), (p_4, 11.04536), (p_8, 7.81024)\rangle, z = 18.78829$.
$(M_2, \mathbf{12.16552}, 0.00000, 16.15549, \infty)$. **Internal R-tree Node Re-expansion Overhead**.

(M_7, **12.16552**, 12.16552, 16.15549, 16.15549).
$\langle(p_{11}, 12.16552), (p_6, 11.31370), (p_4, 11.04536), (p_8, 7.81024)\rangle, z = 12.16552$.
The algorithm terminates because the updated *Fb* value after the recursive call is equal to z in line 18 in Fig. 4 (**if** condition is false), $Fb = 12.16552 = z$. And the number of nodes accessed is 8 (the internal R-tree node pointed by M_2 is read twice). For the interested reader is not so difficult to detect that for BFS the number of nodes accessed is only 7, because we have only to extract M_7 from the *global_list* and read the pointed leaf R-tree node.

Finally, if we want to find the K nearest neighbors from a query spatial object (*queryObject*) instead of a query point (*queryPoint*) as in the algorithm of Fig. 4 (similarly for KNNQ_BFS and KNNQ_DFBnB), we have to introduce in the algorithms the following modifications:

1. Two additional parameters in the KNNQ_RBFS function (similarly for KNNQ_BFS and KNNQ_DFBnB), instead of *queryPoint*:
   queryObject and *MBROfQueryObject*.
2. Treatment on internal R-tree nodes (line 05 in Fig. 4, lines 04 and 14 in Fig. 5 and line 05 in Fig. 6):MinDistValue = MinDist(node.entries[i].MBR, *MBROfQueryObject*);
3. Treatment on leaf R-tree nodes (lines 30–44 in Fig. 4, lines 20–34 in Fig. 5and lines 16–30 in Fig. 6):**if** (node.entries[i].Oid = *null*) dist = ∂(node.entries[i].MBR, *queryObject*);**else** dist = ∂(O_1, *queryObject*);

3.4. The Recursive Best-First Search algorithm for ρDRQ (ρDRQ_RBFS)

The adaptation of the algorithms for KNNQ to the ρDRQ is not so difficult. For the non-incremental processing, we have to consider the following modifications:

1. For the treatment of internal R-tree nodes (KNNQ_BFS and KNNQ_DFBnB), the pruning mechanism is: *if* $MinDist(M, q) > \rho_2$, *then the MBR M will be discarded.* For KNNQ_RBFS, the pruning mechanism is (line 18 in Fig. 4): *if* $\min\{Fb(M, q)\} > \mu$ *or* $\min\{Fb(M, q)\} \geqslant \rho_2$, *then the MBR M will be discarded.* It means that for all cases, ρ_2 will be used as pruning distance instead of the distance of Kth nearest neighbor that has been found so far. Moreover, in the line 10 in Fig. 4 (no revisit condition for leaf R-tree nodes) the function call *KNearestObjects.get_z*( ) must be also replaced by ρ_2.
2. For the treatment of leaf R-tree nodes (for all traversal policies), the spatial object in the distance range $[\rho_1, \rho_2]$ is selected for the final result (lines 34 and 41 in Fig. 4), using *MinDist* if Oid = *null* (points or MBRs) or Δ distance functions if Oid $\neq$ *null* (complex spatial objects).
3. The result of the query must not be ordered (for the three algorithms, KNNQ_BFS, KNNQ_DFBnB and KNNQ_RBFS). That is, the *Kheap* is unnecessary (the returned value in line 45 in Fig. 4 is always ρ_2). Therefore, the data structure that holds the result set is (instead of *Kheap*) a file of records (*resultFile*) of two fields, where the first field will be the distance and the second one will be a pointer to spatial objects in the spatial dataset (or to R-tree leaf nodes if simple spatial objects (points or MBRs) are stored directly in the R-tree).
4. If we want to find a set of spatial objects that fall on a distance range $[\rho_1, \rho_2]$ from a query spatial object (*queryObject*) instead of a query point (*queryPoint*), we have to consider the same three modifications as for KNNQ_RBFS algorithm. This treatment is the same for the three traversal policies (KNNQ_BFS, KNNQ_DFBnB and KNNQ_RBFS).

A special case of ρDRQ is to consider $\rho_1 = 0$ and $\rho_2 > 0$ (i.e. the region (range) is a circle centered in the query object, see the left chart of Fig. 3), which will be used in the experimental section.

3.5. The Recursive Best-First Search algorithm for KCPQ (KCPQ_RBFS)

The main advantage of the recursive algorithms is that they transform a global problem into smaller local ones at each tree level (stored in the recursion stack) and that we can apply pruning mechanisms on every subproblem for reducing the search space. Moreover, for improving the I/O and CPU cost of the Recursive Best-First Search algorithm for KCPQ, two techniques can be used. The first technique aims at reducing the number of I/O operations: it consists in using a *global LRU buffer* [15,16]. The second mechanism for improving the performance aims at reducing the CPU cost by using the *distance-based plane-sweep technique* [16] to avoid processing all the possible combinations of pairs of R-tree items from two leaf nodes. For a RBFS algorithm, we will also apply this technique into internal R-tree nodes, apart of organizing these items as local minimum binary heap (*FbHeap*), which item structure is now $\langle Fb, MinDist, Fb\_SD, NodeAddress_P, NodeAddress_Q\rangle$. Moreover, to design the RBFS algorithm for KCPQ in a non-incremental way (K must be fixed in advance) for R-trees indexing spatial objects, the *Kheap* holds pairs of spatial objects according to their distances (the K pairs of spatial objects with the smallest distance processed so far).

In general, the *distance-based plane-sweep technique* for KCPQ using R-trees [16] consists of sorting the entries of the two current R-tree nodes, based on the coordinates of one of the corners of the MBRs (e.g. lower left corner) in increasing order. Afterwards, the dimension for the sweep-line (sweeping dimension) is established (e.g. SD = 1 or X-axis), and a perpendicular line to the sweeping dimension is moved from left to right, discarding pairs of MBRs (M_i, M_j) if $MinDist(M_i, M_j, SD) > z$ (note that $MinDist(M_i, M_j, SD) = l_{SD}$ in the definition of *MinDist* distance function). In our case, for RBFS and internal R-tree nodes we have to consider an additional treatment for the sweeping dimension (similar to *MinDist* and *Fb* in KNNQ_RBFS). Therefore, we have also to store $Fb(M_i, M_j, SD)$ of a pair of MBRs (M_i, M_j) for the sweeping dimension (so-called, *Fb_SD*), which represents the minimum distance value on the sweeping dimension among the pairs of MBRs in the subtrees bellow them, $Fb(M_i, M_j, SD) = \max\{Fb(MBR\_ancestor_i, MBR\_ancestor_j, SD), MinDist(M_i, M_j, SD)\}$. And, pairs of MBRs (M_i, M_j) can be discarded if $Fb(M_i, M_j, SD) > z$.

Fig. 7 describes by steps the non-incremental RBFS algorithm for processing the KCPQ (KCPQ_RBFS) between two sets of points (P and Q) indexed in two R-trees (R_P and R_Q) with the same height (z is the distance value of the Kth closest pair of points found so far; at the beginning $z = \infty$).

As KNNQ_RBFS algorithm, if we want to find the K closest pairs between two sets of spatial objects (or a set of points and a set of spatial objects), we have to consider in all cases the ∂ distance function for the computation of distances at leaf level, except for the cases (point, point), (point, MBR), (MBR, point) and (MBR, MBR) that using *MinDist* distance function is enough. Moreover, when the two R-trees storing the two spatial datasets have different heights, the algorithms are slightly more complicated. For the recursive algorithms, KCPQ_RBFS is one of them, there are two approaches for treating different heights: *fix-at-leaves* and *fix-at-roots* [15,16], and one of them can be adopted. As KNNQ_RBFS algorithm and due to ties of distances, KCPQ_RBFS obtains one of the possible instances of ordered sequences of K different pairs of spatial objects, although it would be straightforward to report all of them.

For the interested readers, Figs. 8 and 9 describe by steps the non-incremental BFS and DFBnB algorithms, respectively, for processing the KCPQ (KCPQ_BFS and KCPQ_DFBnB [16]) between two sets of points (P

KCPQ_RBFS1 Start from the roots of the two R-trees.

KCPQ_RBFS2 If two internal nodes are accessed, then calculate **MinDist(M_i, M_j, SD)**, **Fb(M_i, M_j, SD)**, **MinDist(M_i, M_j)** and **Fb(M_i, M_j)** for each possible pair of MBRs stored in the nodes in the same manner as for KNNQ_RBFS. Select pairs of MBRs if $Fb(M_i, M_j, SD) \leq z$ (*distance-based plane-sweep technique* [16]) and insert (Fb, MinDist, Fb_SD, $NodeAddress_P$, $NodeAddress_Q$) in a local minimum binary heap (*FbHeap*) organized with *Fb* as key, satisfying the no revisit condition of leaf R-tree nodes ((($Fb(M_i, M_j) > MinDist(M_i, M_j)$ and ($level_{RP} = 1$) and ($level_{RQ} = 1$)) or ($Fb(M_i, M_j) > z$)). Extracting the pair of MBRs, (M_x, M_y), with the minimum *Fb* value, propagate downwards *recursively* those pairs while $Fb(M_x, M_y) \leq \mu$ *and* $Fb(M_x, M_y) < z$, updating and the *Fb* values of such pair of MBRs when the algorithm returns from the recursive call in the same way as for KNNQ_RBFS.

KCPQ_RBFS3 If two leaf nodes are accessed, then calculate $MinDist(p_i, q_j)$ of each possible pair of points, and discard possible pairs of points according to *distance-based plane-sweep technique* [16]. If this distance is smaller than z, then remove the root of the *Kheap* and insert the new pair of points (p_i, q_j), updating this structure and z.

Fig. 7. Recursive Best-First Search algorithm for KCPQ using R-trees, KCPQ_RBFS.

KCPQ_BFS1 Start from the roots of the two R-trees and initialize the global minimum binary heap, *GlobalMinDistHeap*.

KCPQ_BFS2 If you access two internal nodes (including the roots), then compute $MinDist(M_i, M_j, SD)$ and $MinDist(M_i, M_j)$ for each possible pair of MBRs stored in the two nodes. Select pairs of MBRs if $MinDist(M_i, M_j, SD) \leq z$ (following distance-based plane-sweep technique [16]) and insert (MinDist(M_i, M_j), NodeAddressP, NodeAddressQ) into *GlobalMinDistHeap*, which is organized with *MinDist* as key.

KCPQ_BFS3 *The same treatment as* ***KCPQ_RBFS3*** *for pairs of leaf nodes.*

KCPQ_BFS4 If the global minimum binary heap, *GlobalMinDistHeap*, is empty, then stop.

KCPQ_BFS5 Extract the item (MinDist(M_x, M_y), NodeAddressP, NodeAddressQ) on top of the global minimum binary heap, *GlobalMinDistHeap*. If this item has MinDist(M_x, M_y) > z, then stop. Else, repeat the algorithm from **KCPQ_BFS2** for the two nodes pointed by *NodeAddressP* and *NodeAddressQ*.

Fig. 8. Best-First Search algorithm for KCPQ using R-trees, KCPQ_BFS.

KCPR_DFBnB1 Start from the roots of the two R-trees.

KCPR_DFBnB2 If you access two internal nodes, then compute $MinDist(M_i, M_j, SD)$ and $MinDist(M_i, M_j)$ for each possible pair of MBRs stored in the two nodes. Select pairs of MBRs if $MinDist(M_i, M_j, SD) \leq z$ (following the distance-based plane-sweep technique [16]) and insert (MinDist(M_i, M_j), NodeAddressP, NodeAddressQ) into a local minimum binary heap *(LocalMinDistHeap)* organized with MinDist(M_i, M_j) as key. Extracting from *LocalMinDistHeap* the pair of MBRs, (M_x, M_y), with the minimum *MinDist* value, propagate downwards *recursively* those pairs of MBRs that have $MinDist(M_x, M_y) \leq z$.

KCPQ_DFBnB3 *The same treatment as* ***KCPQ_RBFS3*** *for pairs of leaf nodes.*

Fig. 9. Depth-First Branch-and-Bound Search algorithm for KCPQ, KCPQ_DFBnB.

and Q) indexed in two R-trees (R_P and R_Q) with the same height (z is the distance value of the Kth closest pair of points found so far; at the beginning $z = \infty$).

3.6. The Recursive Best-First Search algorithm for ρDJQ (ρDJQ_RBFS)

The adaptation of the algorithms from KCPQ to the ρDJQ is very similar to the one from KNNQ to ρDRQ. For the non-incremental processing, we have to consider the following modifications:

1. For the treatment of internal R-tree nodes (KCPQ_BFS and KCPQ_DFBnB), the pruning mechanism is: *if* $MinDist(M_1, M_2) > \rho_2$, *then the pair of MBRs* (M_1, M_2) *will be discarded*. For KCPQ_RBFS, the pruning mechanism is: *if* $min\{Fb(M_1, M_2)\} > \mu$ *or* $\min\{Fb(M_1, M_2)\} \geqslant \rho_2$, *then the pair of MBRs* (M_1, M_2) *will be discarded*. It means that for all cases, ρ_2 will be used as pruning distance instead of the distance of Kth closest pair that has been found so far.
2. For the treatment of leaf R-tree nodes (for all traversal policies), the pair of spatial objects in the distance range $[\rho_1, \rho_2]$ is selected for the final result, using *MinDist* if Oid $=$ *null* (points or MBRs) or Δ if Oid $\neq$ *null* (complex spatial objects) distance functions, as KCPQ_RBFS, and
3. The result of the query must not be ordered (for the three algorithms, KCPQ_BFS, KCPQ_DFBnB and KCPQ_RBFS). That is, the *Kheap* is unnecessary. Therefore, the data structure that holds the result set is (instead of *Kheap*) a file of records (*resultFile*) of three fields, where the first field will be the distance and the second and the third ones will be two pointers to spatial objects in the spatial datasets involved on the query (or to R-tree leaf nodes if simple spatial objects (points or MBRs) are stored directly in the R-trees).

A special case of ρDJQ is the *Buffer Query* [8] where two spatial datasets and a distance threshold are involved. For this kind of ρDJQ, we will establish $\rho_1 = 0$ and $\rho_2 > 0$ in the experimental section.

4. Experimental results

This section provides the results of an extensive experimentation study aiming at measuring and evaluating the efficiency of the new Recursive Best-First Search algorithms for KNNQ, ρDRQ, KCPQ and ρDJQ proposed in Section 3, namely RBFS. And we have compared them with respect to the BFS and DFBnB implementations from the non-incremental point of view in order to draw conclusions about its real usefulness. In particular, Section 4.1 describes the experimental settings, Section 4.2 compares query algorithms, given a spatial objects, finding other spatial objects based on spatial predicates NN (KNN) and *within* (ρDRQ), whereas Section 4.3 compares spatial distance join algorithms, taking into account NN (KCP) and *within* (ρDJQ) spatial predicates. Finally, in Section 4.4 a summary from the experimental results is reported.

4.1. Experimental settings

In our experiments we used the R*-tree [1] as the underlying disk-resident spatial access method and minimum capacity was set to $Cmin = \lfloor Cmax * 0.4 \rfloor$, since this *Cmin* value yields the best performance according to [1]. In order to evaluate the algorithms for DBQs we have taken into account several performance metrics (disk accesses, response time and main memory requirements). Finally, the effect of buffering for KCPQ and ρDJQ is also studied, since this parameter has an important influence on this kind of DBQ.

All experiments were run on an Intel/Linux workstation with a Pentium 4 2.4 GHz processor (the operating system running in this computer was RedHat Linux 9), 512 Mbytes RAM and several GBytes of secondary storage, using the *gcc* compiler. Moreover, the binary heaps for the algorithms (BFS (global) and RBFS (locals)) were stored completely in main memory as well as the *Kheap* for the final result of KNNQ and KCPQ. We have used synthetic datasets (uniform distributions, UN1 and UN2) that contain 100,000 2-dimensional points (the size of files is 2.8 MBytes) and real-life datasets (the 2-dimensional data space is normalized to have unit length). Real 2-dimensional datasets are data of California: (1) from [36] that contains 98,451 points (MBRs of streams (line-segments), which have been transformed to points by taking the middle point of each segment, CAS, and the file size is 2.7 MBytes); and (2) from Sequoia benchmark [35] that consists of 62556 points (populated places, CAP, and the file size is 1.8 MBytes). The used point datasets are depicted in Fig. 10: (a) uniform, (b) CAS and (c) CAP.

We have measured the performance of our DBQ algorithms based on the following three performance metrics to compare the algorithms in different aspects such as I/O activity, CPU cost and main memory requirements:

1. *Number of Disk Accesses* (DA). It is the most representative parameter to measure the I/O activity, using or not additional buffers. The number of R*-tree nodes fetched from disk is reported as the number of disk accesses, and it may not exactly correspond to actual disk I/O, since R*-tree nodes can be found in the system buffers.
2. *Response Time* (RT). The response time (total query runtime) was measured for overall execution time of the algorithm. This measure is reported in seconds (time) and represents the overall CPU time (number of distance computations) consumed, as well as the I/O time (number of page accesses) performed by the algorithms (i.e. response time = CPU time + I/O time). The indexes construction was not taken into account for the total response time.

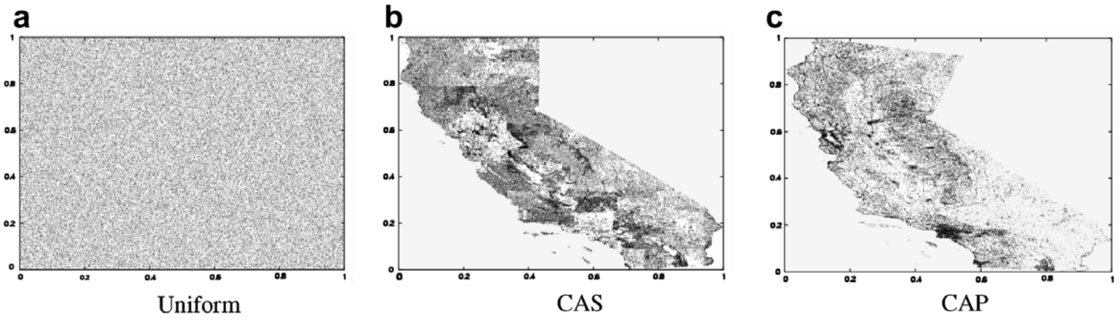

Fig. 10. Datasets used in the experiments: (a) uniform, (b) CAS and (c) CAP.

3. *Maximum Heap Size* for the BFS algorithms (MHS). The task of managing the global binary heap (BFS) can be CPU intensive as its size increases. Thus, the maximum size (in KBytes) of the minimum binary heap required by the BFS algorithms provides a reasonable indication of its main memory requirements, since this search algorithm requires space that can be exponential (in the worst-case) in the height of the R*-trees.

4.2. Performance comparison of KNNQ and ρDRQ algorithms

In our first experiment, we have measured the I/O activity and the main memory requirements (only for BFS) for KNNQ (query point at (0.5, 0.5)), using uniform (UN1) and real (CAP) datasets, and varying the maximum branching factor (*Cmax*) for $K = 10{,}000$ (for lower K values the results are almost the same for all search algorithms). The different values for *Cmax* are 25 (1/2 Kbyte), 50 (1 Kbyte), 102 (2 Kbytes), 204 (4 Kbytes) and 409 (8 Kbytes). The most representative structural characteristics of the R*-trees from the synthetic (uniform) and real datasets, depending of *Cmax*, are shown in Table 2, where *Hei* represents the height of the R*-trees and *NON* is the number of nodes (internals, leaves) of the R*-trees. Moreover, the file size of the R*-trees (in MBytes) varies slightly with *Cmax*. For instance, in the case of *Cmax* = 25: sizeofRtree(U1) = sizeofRtree(U2) = 3.9, sizeofRtree(CAP) = 2.4, sizeofRtree(CAS) = 3.9; and for *Cmax* = 409: sizeofRtree(U1) = sizeofRtree(U2) = 3.8, sizeofRtree(CAP) = 2.3, sizeofRtree(CAS) = 3.8.

Table 3 shows the number of R*-tree nodes fetched from disk of each search algorithm (BFS, RBFS and DFBnB) for KNNQ. The main conclusion of this first experiment is that BFS obtains the best performance, since it is I/O optimal for NNQ [2], although it consumes some additional Kbytes (*in italic*) of main memory for the global minimum binary heap (MHS), e.g. 4.45 Kbytes for CAP and *Cmax* = 204. Obviously, the number of disk accesses decreases as *Cmax* increases and it is almost proportional with this increment (e.g. for CAP, from *Cmax* = 25 to *Cmax* = 204 (8.16), the DA is 614 and 79 (7.77), respectively). We can also observe an interesting behavior for RBFS, it becomes better as *Cmax* increases (closer to BFS), even it is better than DFBnB for *Cmax* = 204 and equal to BFS for *Cmax* = 409 (for low values of *Cmax*, RBFS is inefficient). It is owing to the fact that, the larger R*-tree node is, the cheaper the search can become following a best-first order (i.e. the number of internal R*-tree nodes involved in the query processing is smaller). Moreover, for large *Cmax* values (the number of R*-tree nodes and the height of the R*-trees are smaller, see Table 2), the RBFS algorithm is less affected by the internal node re-expansion overhead (apart of the no revisit condition for leaf R*-tree nodes for non-incremental processing, line 10 in Fig. 4). For example, for CAP and *Cmax* = 25, the

Table 2
Structural characteristics of the R*-trees (heights and number of nodes), based on *Cmax*, for uniform and real datasets

	Cmax = 25		*Cmax* = 50		*Cmax* = 102		*Cmax* = 204		*Cmax* = 409	
	Hei	*NON*	*Hei*	*NON*	*Hei*	*NON*	*Hei*	*NON*	*Hei*	*NON*
UN1	4	(307, 5471)	4	(73, 2723)	3	(20, 1344)	3	(5, 687)	2	(1, 345)
UN2	4	(308, 5441)	4	(75, 2685)	3	(19, 1352)	3	(5, 682)	2	(1, 336)
CAP	4	(191, 3394)	3	(45, 1698)	3	(13, 835)	3	(4, 418)	2	(1, 213)
CAS	4	(327, 5512)	4	(80, 2769)	3	(20, 1369)	3	(5, 686)	2	(1, 344)

Table 3
KNNQ results in terms of disk accesses and MHS (in Kbytes), using the three search algorithms, varying *Cmax* and fixing $K = 10{,}000$ for uniform and real datasets

	Cmax = 25		*Cmax* = 50		*Cmax* = 102		*Cmax* = 204		*Cmax* = 409	
	UN1	CAP	UN1	CAP	UN1	CAP	UN1	CAP	UN1	CAP
DFBnB	916	964	543	354	234	166	156	113	51	47
RBFS	4354	3978	1293	550	282	228	149	101	51	47
BFS	648	614	327	309	168	161	92	79	51	47
MHS	15.14	12.94	10.38	7.86	6.23	5.53	10.8	4.45	5.39	3.33

height of the R*-tree is 4 and the number of internal nodes is 191, RBFS is penalized for internal node re-expansion (DA = 3978, which is 6.5 times more expensive than BFS). On the other hand, when *Cmax* = 204 (height = 3), RBFS needs only a 27% of disk accesses more than BFS and for *Cmax* = 409 (height = 2, i.e. one root and 213 leaves) the number of node accesses is the same. Finally, note that the amount of the required main memory resources (MHS) to perform the KNNQ, following a BFS algorithm (KNNQ_BFS), decreases with the increase of *Cmax*, because R*-trees with large height values, for this performance measure, affect negatively to the execution of this kind of DBQ.

In the second experiment for KNNQ, we have also measured the I/O activity and the main memory requirements (only for BFS) for the same KNNQ configuration, varying the cardinality of the final result (*K*) from 1 to 10,000 (i.e. the query result size) and fixing the maximum branching factor (*Cmax*) to 204 (4 Kbytes is a good choice, because it is the data page size for Linux operating system (version 2.4 of the kernel) [5]). As Tables 3 and 4 shows the same performance metrics of each search algorithm and again BFS obtains the best performance for all *K* values, although for low *K* the three algorithms gets the same results. The number of accesses to R*-tree nodes of each algorithm gets higher as *K* increases (the same behavior for MHS in the BFS algorithm), and RBFS is better than DFBnB if we compare the recursive variants for this *Cmax* value. Moreover, the deterioration is not smooth; after a threshold the cost increases slightly for large *K* values (this threshold is around $K = 1000$). This increment in DA does not depend directly on the increase of *K*, since this parameter is not a structural parameter of the R*-tree as *Cmax* (*K* is a numerical limit, which represents the cardinality of the final query result). Note that for KNNQ we have not shown the response time (in seconds), because for all cases (search algorithms, *K* and *Cmax* values, data distributions, etc.) these times were less than 0.05 s, although BFS in some cases was marginally faster than the other search algorithms (DFBnB and RBFS).

The same experiments have been run for ρDRQ ($\rho_1 = 0$ and $\rho_2 > 0$, i.e. the region (range) is a circle centered in the query object and radius ρ_2). The results for real datasets (the uniform data follow similar trend), when the *Cmax* value is varied and $\rho_2 = 0.3$, are very interesting in terms of disk accesses. DFBnB and BFS: 1452 (25), 719 (50), 368 (102), 190 (204) and 102 (409). RBFS: 12830 (25), 1310 (50), 608 (102), 286 (204) and 102 (409). DFBnB and BFS have the same number of disk accesses in all cases; and RBFS is much more expensive except for large *Cmax* values (i.e. it also becomes better when *Cmax* increases). For example, when *Cmax* = 204, RBFS needs 50% more disk accesses than DFBnB and BFS. The explanation of this behavior for DFBnB and BFS is owing to that the pruning distance for ρDRQ is always ρ_2, and it does not change with the execution of the algorithm (it is not refined as in KNNQ), and for DFBnB there is no possibility to follow subtrees where no desired solutions are located. This fact affects negatively to RBFS, because the internal R*-tree node revisit is produced in all levels of the R*-tree for the same ρ_2 value (only for large *Cmax* values (409) and height = 2, RBFS gets the same results as DFBnB and BFS). That is, the more levels the R*-tree has, the more internal node revisits the RBFS produces. Remember that BFS consumes main memory during its execution (minimum binary heap, called *GlobalMinDistHeap*) and it is (in Kbytes): 22.7 (25), 11.2 (50), 5.7 (102), 2.9 (204) and 1.6 (409). The required memory (for ρDRQ, it is very small quantity, because of the spatial query is over one R*-tree) decreases with the increase of *Cmax*, due mainly to the height of the index and the node size.

As for KNNQ, we have also measured the I/O activity and the main memory requirements (only for BFS) for ρDRQ, varying the distance threshold ρ_2 (0.1, 0.2, 0.3, 0.4 and 0.5) and fixing the maximum branching

Table 4
KNNQ results in terms of disk accesses and MHS (in Kbytes), using the three search algorithms, varying *K* and fixing *Cmax* = 204 for uniform and real datasets

	$K = 1$		$K = 10$		$K = 100$		$K = 1000$		$K = 10{,}000$	
	UN1	CAP	UN1	CAP	UN1	CAP	UN1	CAP	UN1	CAP
DFBnB	3	3	3	4	4	4	17	17	156	113
RBFS	3	3	3	4	4	4	20	17	149	101
BFS	3	3	3	4	4	4	16	16	92	79
MHS	3.25	2.78	3.25	2.78	3.25	2.78	8.27	4.45	10.8	4.45

Table 5
ρDRQ results in terms of disk accesses and MHS (in Kbytes), using the three search algorithms, varying ρ_2 and fixing *Cmax* = 204 for uniform and real datasets

	$\rho_2 = 0.1$		$\rho_2 = 0.2$		$\rho_2 = 0.3$		$\rho_2 = 0.4$		$\rho_2 = 0.5$	
	UN1	CAP	UN1	CAP	UN1	CAP	UN1	CAP	UN1	CAP
DFBnB	41	43	112	99	230	190	391	312	585	385
RBFS	59	54	184	140	382	286	663	484	1004	603
BFS	41	43	112	99	230	190	391	312	585	385
MHS	0.63	0.66	1.73	1.53	3.58	2.95	6.09	4.86	9.13	6.00

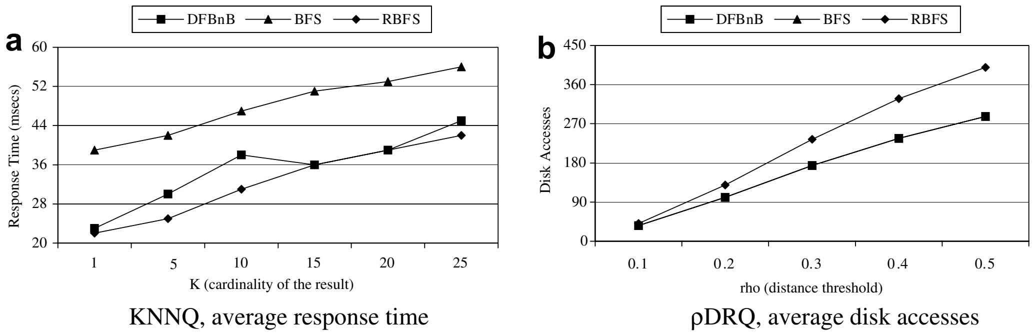

Fig. 11. Average response time and disk accesses for KNNQ and ρDRQ, using the three search algorithms, varying K and ρ_2, for real data and *Cmax* = 204.

factor (*Cmax*) to 204. Table 5 shows the number of disk accesses and again, DFBnB and BFS obtain the best (and the same) performance for all ρ_2 values. This performance measure (DA) for each search algorithm gets higher as ρ_2 increases (the same behavior for MHS in BFS); and RBFS is always worse than DFBnB if we compare the recursive variants. This increment in DA does not depend directly on the increase of ρ_2, since this parameter (as K for KNNQ) is not a structural parameter of the R*-tree as *Cmax* (ρ_2 is only a distance threshold). Note that for ρDRQ, as KNNQ, we have not shown the response time (in seconds), because for all cases (search algorithms, ρ_2 and *Cmax* values, data distributions, etc.) these times were less than 0.1 s.

Now, we are going to consider other experimental methodology for KNNQ and ρDRQ. It consists of selecting 1000 points from one dataset and using these as *query set*. In this case, we base our experimental results on real datasets (*Cmax* = 204), since the uniform data follow similar trends. The 1000 points were chosen from CAS (since CAP corresponds to the R*-tree) using a uniform data distribution from 1 to 98,451. We then simply run 1000 KNNQ (using representative K values for real-life applications, $K = 1, 5, 10, 15, 20$ and 25 as in [20]) and ρDRQ ($\rho_2 = 0.1, 0.2, 0.3, 0.4$ and 0.5), and calculated the average (number of disk accesses and the response time (elapsed time)) of all queries in the workload. For KNNQ, the three search algorithms get the same average disk accesses ($K = 1, 5$ and 10: DA = 3; and K = 15, 20 and 25: DA = 4). For the average response time, Fig. 11a plots the number of milliseconds versus K, and BFS is slightly slower than the others (DFBnB and RBFS). It is due mainly to the creation and destruction of the global minimum binary heap (*GlobalMinDistHeap*) in each 1000 KNNQ of the query set. In the case of ρDRQ, Fig. 11b shows the average disk accesses when ρ_2 is varied, BFS and DFBnB obtain the same I/O cost and require fewer disk access than RBFS (as in Table 5, for one ρDRQ). For the average response time, BFS and DFBnB get very similar time costs, but RBFS is around 35% slower. This additional time (RBFS) is consumed in the accumulation of the internal node revisits during the execution of 1000 ρDRQ in the query set configuration.

4.3. Performance comparison of KCPQ and ρDJQ algorithms

The next experiment is to study the behavior of the search algorithms (DFBnB, BFS and RBFS) for KCPQ for uniform (*Unif*: UN1 and UN2) and real (*Real*: CAS and CAP) datasets, varying the *Cmax* and fixing

$K = 1000$. Note that we have used the *distance-based plane-sweep technique* [16] in the implementations of all search algorithms to accelerate the search process. Table 6 shows the number of R*-tree disk accesses (DA) of each search algorithm. Again, BFS obtains the best I/O performance, although it consumes many Kbytes (even Mbytes) of main memory for the global minimum binary heap (e.g. 2837.13 Kbytes for real datasets and $Cmax = 204$), because KCPQ is much more expensive than KNNQ. As for KNNQ, the number of disk accesses decreases as *Cmax* increases (e.g. for *Real* datasets, from $Cmax = 25$ to $Cmax = 204$ (8.16), the DA is 34046 and 3954 (8.61), respectively). The difference in disk accesses of BFS with respect to DFBnB and RBFS is reduced with the increase of *Cmax*. For example, DFBnB needs more disk accesses than BFS: 3.16% (25), 2.16% (50), 1.23% (102), 1.21% (204) and 0% (409); and RBFS: 70.70% (25), 54.45% (50), 1.18% (102), 0.51% (204) and 0% (409). RBFS is the worst search algorithm for low values of *Cmax*; although it becomes better as the fan-out increases (it gets the same number of disk accesses as BFS for $Cmax = 409$ and heights = 2). The internal R*-tree node revisit overhead is the main reason of this behavior for RBFS. To draw this conclusion, in Table 6 and for real datasets (CAS, CAP); when $Cmax = 25$ (height of the both R*-trees is 4 and the number of internal nodes is (327, 191)), RBFS is affected by internal node revisit (DA = 69,206, which is 2.1 times more expensive than BFS). On the other hand, when $Cmax = 204$, RBFS needs only a 0.7% of disk accesses more than BFS, and for $Cmax = 409$ and heights = 2, the number of disk accesses is exactly the same. The last performance measure to consider from the analysis of the results of Table 6 is the amount of the required main memory resources (MHS) for BFS to perform the KCPQ. It is very easy to observe that it grows with the increase of *Cmax* (except for $Cmax = 409$ and heights = 2, which MHS value is slightly smaller than for $Cmax = 204$), because of the larger the R*-tree node size is, the larger the number of pairs of MBRs from the combination of two R*-tree nodes is (although the number of disk accesses is smaller).

In this new experiment, for KCPQ, we have considered the same performance measures as KNNQ, using uniform and real datasets, varying the cardinality of the final result *K* (from 1 to 1,000,000) and fixing *Cmax* to 204 (see Fig. 12). Fig. 12a shows that BFS obtains the best performance for all *K* values using uniform data. The number of accesses to R*-tree nodes of each algorithm gets higher as *K* increases (as we expect according to [16]), and RBFS is better than DFBnB if we compare the recursive alternatives (as KNNQ) for this *Cmax*

Table 6
KCPQ results in terms of disk access and MHS (in Kbytes), using the three search algorithms, varying *Cmax* and fixing $K = 1000$ for uniform and real datasets

	$Cmax = 25$		$Cmax = 50$		$Cmax = 102$		$Cmax = 204$		$Cmax = 409$	
	Unif	Real	Unif	Real	Unif	Real	Unif	Real	Unif	Real
DFBnB	50344	34986	26822	23154	12994	9132	5510	3982	2632	1962
RBFS	83306	69206	40550	47596	12988	9152	5472	3982	2612	1946
BFS	48802	34046	26254	22784	12836	9028	5444	3954	2612	1946
MHS	454.19	316.25	546.56	386.75	1043.69	1046.01	3643.91	2837.13	3622.5	2289.75

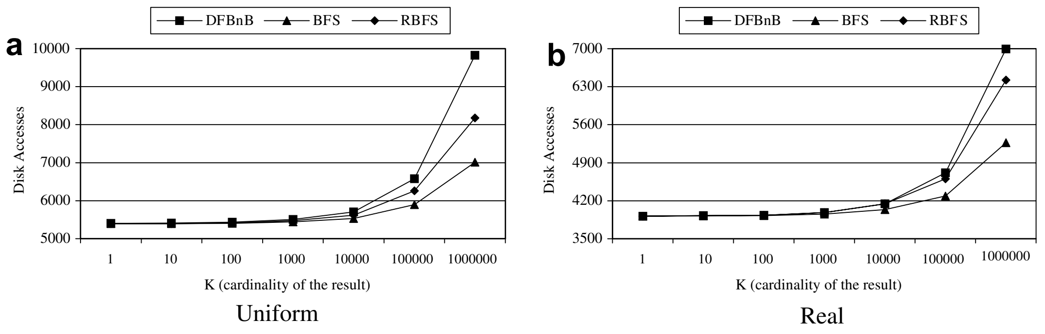

Fig. 12. KCPQ results in terms of disk accesses, using the three search algorithms (DFBnB, BFS and RBFS), varying *K* and fixing $Cmax = 204$ for (a) uniform and (b) real datasets.

value (204). For example, for $K = 100{,}000$, RBFS needs a 6% of disk accesses more than BFS (although it requires 3.5 Mbytes of main memory for the global minimum binary heap) to complete this kind of DBQ, and DFBnB a 12%, i.e. DFBnB is more expensive than RBFS in terms of I/O activity using uniform data. Moreover, the increase of DA does not depend directly on the increase of K, since it is not a parameter that affects to the structure of the R*-tree in its creation, as occurs with *Cmax*. For example, BFS for $K = 1000$ needs a 1% of disk accesses more than $K = 1$, and a 30% less than $K = 1{,}000{,}000$.

The same behavior as Fig. 12a, we can observe for KCPQ using real datasets in Fig. 12b. BFS obtains the best performance for all K values, the number of accesses to R*-tree nodes of each algorithm gets higher as K increases and RBFS is cheaper than DFBnB ($Cmax = 204$). For example, for $K = 1{,}000{,}000$, RBFS needs a 22% of disk accesses more than BFS to complete this kind of DBQ, and DFBnB a 33% more. Besides, as for uniform case, the increase in DA does not depend directly on the increase of K, e.g. BFS for $K = 1000$ needs a 1% of disk accesses more than $K = 1$, and a 35% less than $K = 1{,}000{,}000$. Remember that BFS needs additional main memory for store the global minimum binary heap (*GlobalMinDistHeap*), and it is around 3 Mbytes.

As for KCPQ, for ρDJQ, we have performed the same experiments ($\rho_1 = 0$ and $\rho_2 > 0$, i.e. Buffer Query [8]). The number of disk accesses for real datasets (the uniform data have similar trend), when the *Cmax* value is varied and $\rho_2 = 0.03$, are the following. DFBnB and BFS: 405,780 (25), 142,550 (50), 43,446 (102), 14,324 (204) and 5198 (409). RBFS: 27,394,044 (25), 13,105,374 (50), 76,604 (102), 22,910 (204) and 5198 (409). As for ρDRQ, DFBnB and BFS have the same number of disk accesses in all cases; and RBFS is much more expensive, except for large *Cmax* values. As for KCPQ, the number of disk accesses decreases as *Cmax* increases. We must highlight that RBFS is always the worst search algorithm, except for $Cmax = 409$ and heights = 2, where all search algorithms have the same performance. For example, when $Cmax = 204$, RBFS needs 60% more disk accesses than DFBnB and BFS. Again, the main reason of this behavior for the three search algorithms is that the distance pruning (ρ_2) remains always with the same value during the whole execution of the algorithm. For ρDJQ, BFS consumes more main memory than ρDRQ during its execution (*GlobalMinDistHeap*) mainly for large *Cmax* values and the Kbytes of main memory used by BFS are: 4787.8 (25), 1384.7 (50), 540.1 (102), 187.2 (204) and 81.2 (409). It is due to the combination of two R*-trees, and the height of the indexes affects negatively to this performance measure (i.e. the more levels the R*-tree has, the more main memory the BFS consumes).

As for KCPQ, we have also measured the I/O activity (number of disk accesses) and the main memory requirements (only for BFS) for ρDJQ, varying the distance threshold ρ_2 (0.001, 0.005, 0.01, 0.02, 0.03, 0.04 and 0.05) and fixing the maximum branching factor (*Cmax*) to 204. Table 7 show the number of disk accesses for real data (the same trends are for uniform), and it gets higher as ρ_2 increases for all search algorithms. DFBnB and BFS obtain the best (and the same) performance for all ρ_2 values; and RBFS is always the worst, because of the pruning distance (ρ_2) is always the same (it affects negatively to the internal node revisit overhead). From this table of results, for $\rho_2 = 0.01$, RBFS needs a 35% of disk accesses more than BFS (it only requires 63.81 Kbytes of main memory for *GlobalMinDistHeap*) and DFBnB to complete this kind of DBQ. Finally, and as ρDRQ, the demanded main memory for BFS (MHS) grows with the increase of ρ_2; and it is due to the fact that, the larger pruning distance is, the more nodes are necessary to complete the distance-based query (i.e. the number of internal R*-tree nodes involved in the query processing is larger).

In the next experiment, we are going to study the response time (in seconds) for KCPQ. First of all, we have observed in our experiments that this performance measure is not significantly affected by the increase of

Table 7
ρDJQ results in terms of disk accesses and MHS (in Kbytes), using the three search algorithms, varying ρ_2 (*Buffer Query*, $\rho_1 = 0$) and fixing $Cmax = 204$ for real datasets

	$\rho_2 = 0.001$	$\rho_2 = 0.005$	$\rho_2 = 0.01$	$\rho_2 = 0.02$	$\rho_2 = 0.03$	$\rho_2 = 0.04$	$\rho_2 = 0.05$
DFBnB	4152	5224	6712	10196	14324	18990	24196
RBFS	4348	6336	9040	15388	22910	31464	41066
BFS	4152	5224	6712	10196	14324	18990	24196
MHS	34.97	44.28	63.81	119.28	87.28	257.16	341.13

Table 8
KCPQ results in terms of response time (in seconds), using the three search algorithms, varying *Cmax* and fixing $K = 1000$ for uniform and real datasets

	$Cmax = 25$		$Cmax = 50$		$Cmax = 102$		$Cmax = 204$		$Cmax = 409$	
	Unif	Real	Unif	Real	Unif	Real	Unif	Real	Unif	Real
DFBnB	0.97	0.63	0.84	0.58	0.78	0.53	0.81	0.53	0.77	0.56
RBFS	2.44	2.05	3.18	0.93	0.85	0.63	0.83	0.61	0.81	0.61
BFS	0.84	0.59	0.77	0.56	0.73	0.53	0.81	0.53	0.81	0.61

Cmax, i.e. for a fixed *K* value ($K = 1000$), the response time of the query is almost the same for all *Cmax* values (25, 50, 102, 204 and 409), although RBFS is slower for low *Cmax* values as we can observe in Table 8.

For this reason, we are going to focus on this time-based metric when *K* is incremented for a given *Cmax* value, using the three search algorithms (DFBnB, BFS and RBFS) and the *distance-based plane-sweep technique* to speed up the query. Fig. 13 shows the total response time (in seconds) for KCPQ using uniform (6.a) and real (6.b) datasets, varying the cardinality of the final query result (*K*) from 1 to 1,000,000 and fixing *Cmax* to 204. For small and medium *K* values ($K \leqslant 10{,}000$), DFBnB was slightly faster than BFS and RBFS as in [16]. But for large *K* values ($K \geqslant 100{,}000$) the fastest was BFS. The explanation of this behavior is owing to the fact that DFBnB traverses the R*-trees using a depth-first search and it can deviate to the subtrees where no optimal solutions are located; and RBFS can suffer from internal R*-tree node re-expansion overhead, although it traverses the nodes in best-first order. For example, for $K = 1{,}000{,}000$ and uniform data (Fig. 13a), RBFS is only a 9% slower (2.61 s) than BFS, and DFBnB 2.42 times slower (38.64 s), this difference can be easily checked in such chart. Another comparative example is shown in Fig. 13b for real datasets (right chart) and $K = 1{,}000{,}000$, where RBFS is only a 1.25% (0.30 s) slower than BFS, and DFBnB a 63% (15.18 s). This behavior is conditioned by the number of *MinDist* computations needed to complete the query, which is a measure that affects notably to the response time [16]. For uniform datasets, $K = 1{,}000{,}000$ and $Cmax = 204$, DFBnB and RBFS need a 98% and 8.5% more *MinDist* computations than BFS, respectively. And for real datasets, DFBnB and RBFS demand a 45.5% and 10% more *MinDist* computations than BFS, respectively. As in [16], BFS minimizes the number of distance computations in our experiments.

As for KCPQ, for ρDJQ, we have performed similar experiments to study its behavior with respect to the response time. First of all, as KCPQ, we have observed the response time is not significantly affected by the increase of *Cmax*. But the increase of ρ_2 (distance threshold) has got influence over this performance metric as we can observe in Fig. 14. It shows the total response time (in seconds) for ρDJQ using uniform (14.a) and real (14.b) datasets, varying the distance threshold ρ_2 (0.001, 0.005, 0.01, 0.02, 0.03, 0.04 and 0.05) and fixing *Cmax* to 204. For all ρ_2 values, DFBnB and BFS are faster than RBFS, and both of them have very similar times (although BFS is slightly faster for large ρ_2 values). The reason of this similar trends is owing to the fact that DFBnB does not traverse subtrees where no desired solutions are located (the pruning distance is always ρ_2 and there is no additional disk accesses in the processing of the algorithm). We must highlight that RBFS is

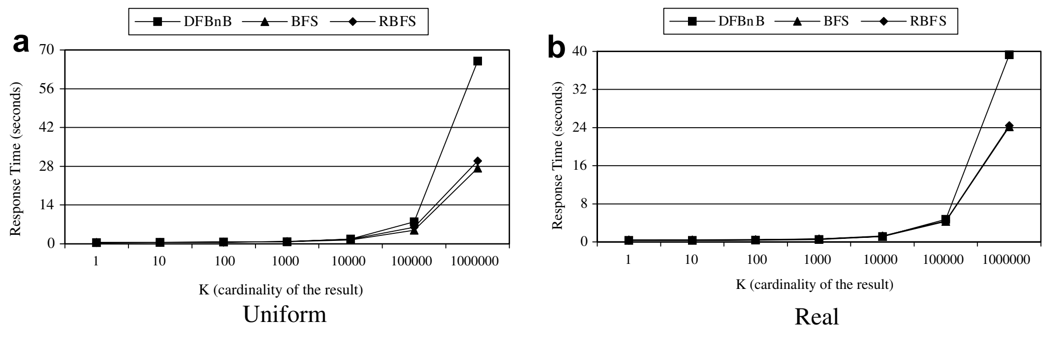

Fig. 13. KCPQ results in terms of response time, using the three search algorithms (DFBnB, BFS and RBFS), varying *K* and fixing $Cmax = 204$ for (a) uniform and (b) real datasets.

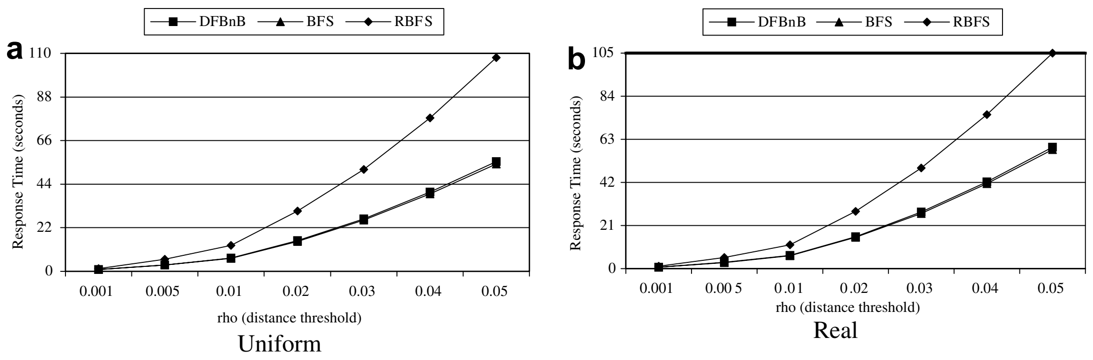

Fig. 14. ρDJQ results in terms of response time, using the three search algorithms (DFBnB, BFS and RBFS), varying ρ_2 and fixing $Cmax = 204$ for (a) uniform and (b) real datasets.

always the worst search algorithm, except for small ρ_2 values (e.g. $\rho_2 = 0.001$), where all of them have almost the same performance. The explanation of this behavior is because of RBFS traverses the nodes in best-first order regardless of the pruning distance value (ρ_2) and it can suffer from internal R*-tree node re-expansion overhead. For example, in the most expensive case (for $\rho_2 = 0.05$) and real data in Fig. 14b (the trend for uniform data is very similar), DFBnB is only a 2.11% (1.23 s) slower than BFS, and RBFS an 80% (46.93 s). As for KCPQ, this behavior of ρDJQ is conditioned by the number of *MinDist* computations needed to complete the query, which is a measure that affects to the response time [16].

To speed up query processing, SDBMSs use spatial access methods that may partially reside in *buffers* if main memory is available. The buffering effect should be studied, since even a small number of buffer pages can drastically improve the overall performance [32]. Here, we will use the buffer architecture proposed in [16] for KCPQ:*global LRU buffer*.

For these new experiments, we have coded a LRU trace-drive buffer, reporting the real number of node accesses. The buffer is initially empty and its size (B) is given in R*-tree nodes (pages) as input ($B = \{0, 4, 8, 12, 16, 32, 64, 128, 256, 512 \text{ and } 1024\}$). Fig. 15 plots the number of disk accesses versus buffer size for the *query set* configuration and we can observe how buffer size affects the relative performance of the three search algorithms for KNNQ and ρDRQ using real data ($Cmax = 204$). The chart of the left is for KNNQ $K = 10$; the one of the right for ρDRQ ($\rho_2 = 0.3$). In Fig. 15a for KNNQ, the trend (average performance) of the three search algorithms is almost the same (for larger K values, the trend remains). This similar behavior is due to the small number of disk accesses needed for this query using 2-dimensional data and small K values. In Fig. 15b for ρDRQ, for small buffer sizes, RBFS requires more disk accesses than BFS and DFBnB. At a buffer size of 32, the performance of the three search algorithms coincides, and all of them have the same trend. It is very interesting to highlight the difference with respect to the buffering impact from $B = 128$

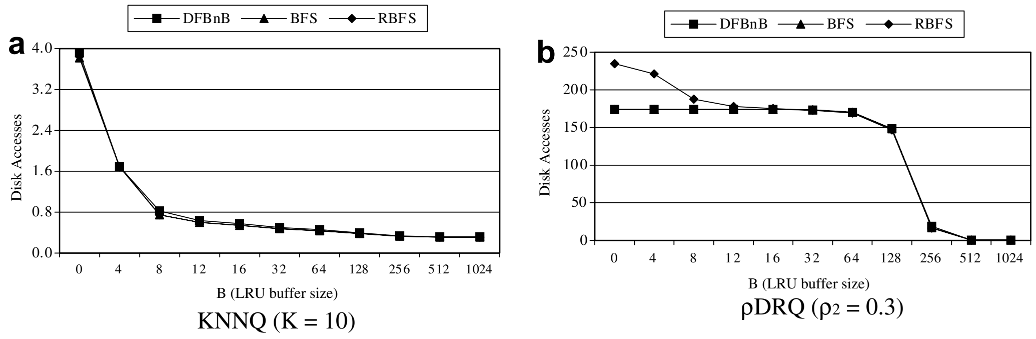

Fig. 15. Sensitivity of buffer size for (a) KNNQ ($K = 10$) and (b) ρDRQ ($\rho_2 = 0.3$), using the search algorithms, real data and $Cmax = 204$.

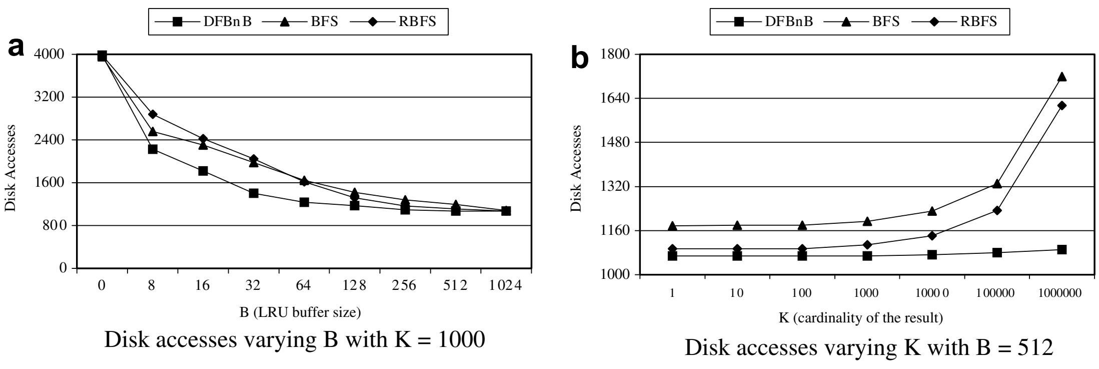

Fig. 16. KCPQ results in terms of disk accesses, using the three search algorithms (DFBnB, BFS and RBFS) and real data: (a) varying the buffer size ($K = 1000$), (b) varying K ($B = 512$).

and $B = 256$ for ρDRQ, it means if there is sufficient memory to store the entire R*-tree in main memory, we can bound the buffer size to around 50% of the R*-tree size (in term of nodes).

In these new experiments for studying the buffer impact, we are going to consider KCPQ for the workspace configuration (CAS, CAP) with different buffer sizes (B), varying from 0 to 1024 pages (R*-tree nodes) as in [16]. Fig. 16a shows that BFS and RBFS presents an average excess of I/O activity around 24% and 23%, respectively, for $K = 1000$ and $Cmax = 204$ with respect to DFBnB, as can be noticed by the gap between the lines (mainly for $B \leqslant 256$, since for B values larger than 512 the difference is negligible, less than 1%). This behavior is owing to the fact that recursion with a depth-first order (DFBnB) favors the most recently used pages (LRU) in the backtracking phase and this effect is preserved in case of large buffer sizes. Moreover, the best-first order (BFS and RBFS) is penalized when the LRU replacement algorithm is adopted, even when the recursion is used (RBFS), since LRU cannot avoid the node re-expansion overhead of RBFS for low and medium buffer sizes.

On the other hand, Fig. 16b illustrates that the gap for KCPQ algorithms (BFS and RBFS with respect to DFBnB) remains when the K value is incremented (and it grows when K is very large) and $B = 512$ pages. For instance, the average I/O saving between DFBnB and RBFS for medium K values ($K = 10{,}000$) is 6%, and DFBnB with respect to BFS is 15%. For large K values, this gap is even greater, for example $K = 100{,}000$ the I/O saving for DFBnB with respect to RBFS and BFS is 14% and 23%, respectively. But, special attention deserves $K = 1{,}000{,}000$ (very large K values), where the DFBnB saving is around 48% with respect to RBFS and 58% for BFS. Again, this interesting effect for DFBnB is owing to the combination of recursion, depth-first order and LRU page replacement policy. Moreover, we can observe that K does not radically affect the relative performance with respect to the number of disk accesses for DFBnB, due to the important saving of including a global LRU buffer. But this relative performance is more affected by the increase of K in RBFS and BFS. For example, from $K = 1$ to $K = 1{,}000{,}000$ for DFBnB there is only a 2% extra cost, whereas for RBFS and BFS this increase of I/O cost is around 46% for both of them.

In the last experiment, we are going to study the behavior of ρDJQ in presence of a global LRU buffer, as for KCPQ. We will use the same workspace configuration (CAS, CAP) with different buffer sizes (B), varying from 0 to 1024 pages. Fig. 17a shows that RBFS and BFS present a strange behavior for $\rho_2 = 0.03$ and $Cmax = 204$ with respect to DFBnB, and we can notice the huge gap (much larger than for KCPQ) between the lines (except to $B = 1024$). DFBnB, for ρDJQ, has a similar behavior in presence of buffer that for KCPQ, because this search algorithm favors the LRU replacement policy in the backtracking phase. With DFBnB, we can save many disk accesses, e.g. if we have a buffer $B = 512$, we obtain 13 times less disk accesses that when there is no buffer ($B = 0$). The algorithms following a best-first order (BFS and RBFS) are severely penalized when LRU is used in comparison with DFBnB. RBFS has a difference of around 30% in average with respect to BFS when $B \leqslant 32$ (due to the internal node re-expansion in order to follow the best-first order, when the buffer size is small; i.e. the combination of recursion and LRU replacement policy is not enough for small buffer sizes), and for $B \geqslant 64$ both lines have the same trend.

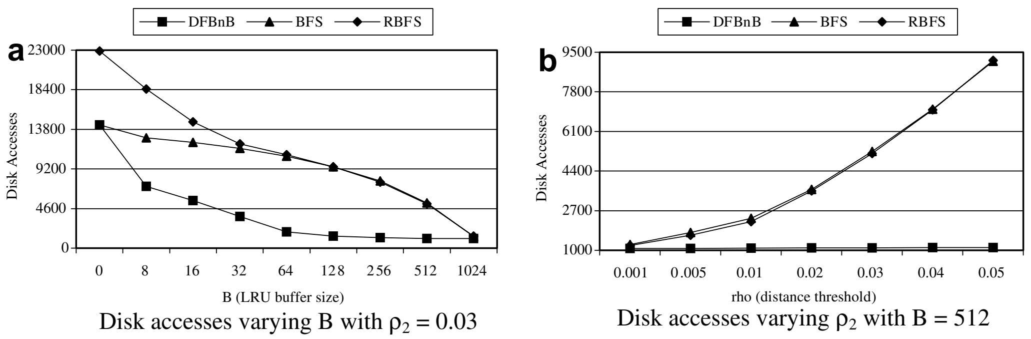

Fig. 17. ρDJQ results in terms of disk accesses, using the three search algorithms (DFBnB, BFS and RBFS) and real data: (a) varying the buffer size ($\rho_2 = 0.03$), (b) varying ρ_2 ($B = 512$).

As we can observe in Fig. 17b, the performance lines of BFS and RBFS grow when the ρ_2 value is incremented ($B = 512$ pages); and such line for DFBnB is almost constant. For instance, the average I/O saving between DFBnB and RBFS (and BFS) for medium ρ_2 values ($\rho_2 = 0.02$) is more than three times (DFBnB: 1100 and RBFS: 3551), and for large ρ_2 values ($\rho_2 = 0.05$), this gap is much greater and the I/O saving is more than 8 times (DFBnB: 1112 and RBFS: 9154). As for KCPQ, DFBnB is the best and this interesting behavior is owing to the combination of recursion, depth-first order and LRU page replacement policy. Moreover, we can also observe that the increase of ρ_2 does not affect the relative performance with respect to the number of disk accesses for DFBnB, due to the important saving of including a global LRU buffer. But this relative performance is more affected by large ρ_2 values in DFBnB (for RBFS and BFS, it is smaller). For example, from $\rho_2 = 0.01$ to $\rho_2 = 0.05$ for DFBnB there is only a 2% extra cost (the same as KCPQ), whereas for RBFS and BFS, this increase of I/O cost is around three times for both of them.

4.4. Conclusions from the experiments

The main conclusions from this experimental section, taking into account the previous performance measures, are summarized in Table 9.

Table 9
Comparison of three search algorithms (relative merits) with respect to the DBQs

	KNNQ–KCPQ	ρDRQ–ρDJQ
DFBnB	It is linear-space consuming with respect to the height of the R*-trees It is the worst in the number of disk accesses, but it is the best in presence of an LRU buffer It is appropriate in systems with space (memory) limitations It is the slowest for large K values in KCPQ	It has the same behavior as BFS with respect to DA and RT It is owing to the fact that the pruning distance is always ρ_2, and there is no possibility to follow subtrees where no desired solutions are located It is the best in presence of an LRU buffer
RBFS	It is linear-space consuming with respect to the height of the R*-trees It is better than DFBnB in disk accesses when the node size is large enough (small node sizes and large K values, it is affected by the internal node revisits) It is appropriate in systems with space memory limitations	It is the worst alternative in terms of DA and RT It is mainly owing to the internal node revisits in order to follow a best-first order (regardless of the ρ_2 value and the node size)
BFS	It is the fastest It minimizes the number of disk accesses without LRU buffer It is space-consuming, and it is preferable when we do not have memory limitations	It is the fastest It minimizes the number of disk accesses It is space-consuming

1. BFS is the best for all DBQs, since BFS is optimal in terms of the number of disk accesses for KNNQ [3] and KCPQ [16]; and this search algorithm needs the minimum number of distance computations [16]. But it can consume several Mbytes of main memory to keep the global minimum binary heap, even more main memory requirements are needed if K is very large for KCPQ. The reason of this high memory requirements is because of BFS has a space complexity of $O(n)$ in the worst case (n is the number of MBRs in internal R-tree nodes) for KNNQ [3], since if all elements of the tree are stored in the global minimum binary heap, it can have a length of $O(n)$. The formula of n can be estimated for R-trees as follows, where $f = Cmax * \mathrm{Uavg}$ is the effective R-tree node capacity (Uavg is the average R-tree node utilization), N is the number of points indexed by the R-tree and h is its height (the leaf nodes are located at level $l = h - 1$, the root at level $l = 0$, and internal nodes $0 \leqslant l \leqslant h - 1$).

   $$n = f * \sum_{l=0}^{h-2} \frac{N}{f^{h-l}}$$

   However, for KNNQ and KCPQ, RBFS obtains similar behaviors with negligible differences in most cases; and for ρDRQ and ρDJQ, DFBnB has the same performance. This equivalent behavior with respect to DFBnB is owing to the pruning distance is always ρ_2 and there is no possibility to follow subtrees where no desired solutions are located.
2. RBFS is competitive for KNNQ and KCPQ (very similar behaviors to BFS) where the maximum branching factor ($Cmax$) is large enough in terms of disk accesses and response times (for large K values the trends remain), even better than DFBnB. We have to highlight that RBFS does not have good performance when $Cmax$ (node size) is small, since it leads to an increment in the number of nodes of the R*-tree, and hence an increment of the internal node revisits during the processing of the algorithm. RBFS is a good alternative when we have main memory limitations in our computer, since it is linear space consuming with respect to the height of the R*-trees, for this reason RBFS is preferable to BFS. Nevertheless, RBFS is the worst alternative for ρDRQ and ρDJQ, since it is penalized by the internal node re-expansion overhead, to follow a best-first order.
3. DFBnB is a good alternative for KNNQ, but for KCPQ consumes many time to report the final result when K is very large; because, in this case, it can follow many subtrees where no desired solutions are located (unnecessary node visits). It has the same performance as BFS for ρDRQ and ρDJQ, due to the searching characteristics for such DBQs. Moreover, it is also linear space consuming with respect to the height of the R*-tree (as RBFS), and it is suitable to install in systems with memory limitations by high process overload (e.g. Web Server in a Web GIS architecture [32]). The reason of this small memory requirements (as RBFS) is because of DFBnB has a worst case space complexity of $O(\log n)$ for KNNQ [3], because the recursion depth is at most equal to the height of the R-tree. The height of the R-tree, h, can be estimated by the following formula:

   $$h = 1 + \left\lceil \log_f \left(\frac{N}{C\max} \right) \right\rceil$$

4. When a global LRU buffer is included, the behavior of the algorithms is very interesting. For KCPQ and ρDJQ, DFBnB is the best in terms of disk accesses, due to the combination of recursion, depth-first order and LRU page replacement policy (K and ρ_2 do not affect the I/O performance). On the other hand, RBFS has an interesting behavior for KCPQ, which is better than BFS because of the use of recursion in its implementation, but LRU replacement algorithm cannot avoid the node re-expansion overhead for low and medium buffer sizes. But for ρDJQ, RBFS and BFS have almost the same behavior, and it is far away from DFBnB.

5. Conclusions and ideas for future work

Efficient processing of DBQs (e.g KNNQ, ρDRQ, KCPQ and ρDJQ) is of great importance in spatial databases due to the wide area of applications that may address such queries. To the best of the authors' knowledge, it is the first attempt to adapt the RBFS algorithm to this type of spatial queries. RBFS is a general search algorithm that runs in linear space and traverses nodes in best-first order, but it can suffer from node re-expansion overhead (i.e. to expand nodes in best-first order, some nodes can be read more than one). In this paper, based on the properties of distance functions between two MBRs in the multidimensional Euclidean space, we propose new pruning mechanisms to apply them in the design of new non-incremental RBFS algorithms for KNNQ (KNNQ_RBFS), ρDRQ (ρDRQ_RBFS), KCPQ (KCPQ_RBFS) and ρDJQ (ρDJQ_RBFS) between spatial objects indexed in R-trees. And we have compared experimentally these algorithms with respect to BFS and DFBnB traversal policies for the same DBQs.

In our experimental study, we have used an R-tree variant (R*-tree) in which the spatial objects (points) are stored directly in the tree leaves. Moreover, an exhaustive performance study was also included, which resulted to several conclusions about the efficiency of each search algorithm (BFS, RBFS and DFBnB), in terms of disk accesses, response time and main memory requirements (only for BFS), with respect to maximum branching factor (*Cmax*), cardinality of the final query result (K), distance threshold (ρ) and the size of a global LRU buffer (B) for KCPQ and ρDJQ. The most important conclusions for the experimental executions of the search algorithms are the following: (1) BFS is the best for all DBQs, but it can consume many main memory resources to keep the global minimum binary heap. (2) RBFS is competitive for KNNQ and KCPQ where the maximum branching factor (*Cmax*) is large enough in terms of disk accesses and response times (for all K values the trends remain), even better than DFBnB. RBFS is a good alternative when we have main memory limitations in our computer because of high process overload in our system, since it is linear space consuming with respect to the height of the R-trees. Nevertheless, RBFS is the worst alternative for ρDRQ and ρDJQ, due to the high number of internal node revisits during the processing of the algorithms. (3) DFBnB is appropriate for KNNQ, but for KCPQ consumes long time to report the final result when K is very large. For ρDRQ and ρDJQ, it has the same performance as BFS. Moreover, as RBFS, it is also linear space consuming with respect to the height of the R-tree. (4) When a global LRU buffer is included for KCPQ and ρDJQ, DFBnB is the best in terms of disk accesses (K and ρ_2 do not radically affect the I/O performance), and RBFS has an interesting behavior for KCPQ, which is better than BFS owing to the use of recursion in its implementation.

Future work may include:

1. Extensions of RBFS algorithms to other complex DBQs as: *iceberg distance join query* [34] and *K Nearest Neighbor Join query* [4].
2. Approximate implementations of RBFS algorithms by using ε-approximate and α-allowance techniques that are distance-based approximate techniques [14].
3. Design the *incremental* versions of the RBFS algorithms for KNNQ and KCPQ according to [20,19]. For example, in the case of KNNQ, the algorithm has to traverse the R-tree in best-first order (in ascending order of distances), reporting one by one the elements stored in the leaf nodes, according to its distance.

Acknowledgements

The authors wish to thank *INDALOG* (TIC2002-03968) and *Almacenes de Datos Espacio-Temporales basados en Ontologias* (TIN2005-09098-C05-03), projects of the Spanish Ministry of Science and Technology. Moreover, we owe our special thanks to the anonymous reviewers whose comments were critical to enhancing the quality of the paper.

References

[1] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD), Atlantic City, NJ, 1990, pp. 322–331.

[2] S. Berchtold, C. Böhm, D.A. Keim, H.P. Kriegel, A cost model for nearest neighbor search in high-dimensional data space, in: Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Tucson, AZ, 1997, pp. 78–86.

[3] C. Böhm, S. Berchtold, D.A. Keim, Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases, ACM Computing Surveys 33 (3) (2001) 322–373.
[4] C. Böhm, F. Krebs, High performance data mining using the nearest neighbor join, in: Proceedings of the IEEE International Conference on Data Mining (ICDM), Maebashi, Japan, 2002, pp. 43–50.
[5] D.P. Bovet, M. Cesati, Understanding the Linux Kernel, O'Reilly, Sebastopol, 2003.
[6] T. Brinkhoff, H.P. Kriegel, B. Seeger, Efficient processing of spatial joins using R-trees, in: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD), Washington, WA, 1993, pp. 237–246.
[7] P. Brown, Object-Relational Database Development: A Plumber's Guide, Prentice-Hall, Indianapolis, 2001.
[8] E.P.F. Chan, Buffer queries, IEEE Transactions Knowledge Data Engineering (TKDE) 15 (4) (2003) 895–910.
[9] J.K. Chen, Y.H. Chin, A concurrency control algorithm for nearest neighbor query, Information Sciences 114 (1–4) (1999) 187–204.
[10] K.L. Cheung, A.W. Fu, Enhanced nearest neighbour search on the R-tree, ACM SIGMOD Record 27 (3) (1998) 16–21.
[11] N. Chrisman, Exploring Geographic Information Systems, John Wiley and Sons, New York, 2002.
[12] D. Comer, The ubiquitous B-tree, ACM Computing Surveys 11 (2) (1979) 121–137.
[13] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, Cambridge, 2003.
[14] A. Corral, J. Cañadas, M. Vassilakopoulos, Approximate algorithms for distance-based queries in high-dimensional data spaces using R-trees, in: Proceedings of the 6th East European Conference on Advances in Databases and Information Systems (ADBIS), Bratislava, Slovakia, 2002, pp. 163–176.
[15] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Closest pair queries in spatial databases, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD), Dallas, TX, 2000, pp. 189–200.
[16] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Algorithms for processing *K*-closest-pair queries in spatial databases, Data and Knowledge Engineering (DKE) 49 (1) (2004) 67–104.
[17] V. Gaede, O. Günther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998) 170–231.
[18] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD), Boston, MA, 1984, pp. 47–57.
[19] G.R. Hjaltason, H. Samet, Incremental distance join algorithms for spatial databases, in: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD), Seattle, WA, 1998, pp. 237–248.
[20] G.R. Hjaltason, H. Samet, Distance browsing in spatial databases, ACM Transactions on Database Systems (TODS) 24 (2) (1999) 265–318.
[21] Y.W. Huang, N. Jing, E.A. Rundensteiner, Spatial joins using R-trees: breadth-first traversal with global optimizations, in: Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB), Athens, Greece, 1997, pp. 396–405.
[22] R.E. Korf, Linear-space best-first search, Artificial Intelligence 66 (1) (1993) 41–78.
[23] N. Koudas, K.C. Sevcik, Size separation spatial join, in: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD), Tucson, AZ, 1997, pp. 324–335.
[24] N. Koudas, K.C. Sevcik, High dimensional similarity joins: algorithms and performance evaluation, IEEE Transactions Knowledge Data Engineering (TKDE) 12 (1) (2000) 3–18.
[25] Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos, Y. Theodoridis, *R*-Trees: Theory and Applications, Advanced Information and Knowledge Processing Series, Springer, London, 2005.
[26] N. Nilsson, Artificial Intelligence: A New Synthesis, Morgan Kaufmann, San Francisco, 1998.
[27] Oracle Technology Network Oracle Spatial User's Guide and Reference, 2001. Downloadable from: <http://technet.oracle.com/doc/Oracle8i_816/inter.816/a77132.pdf>.
[28] A. Papadopoulos, Y. Manolopoulos, Performance of nearest neighbor queries in R-Trees, in: Proceedings of the 6th International Conference on Database Theory (ICDT), Delphi, Greece, 1997, pp. 394–408.
[29] A.N. Papadopoulos, A. Nanopoulos, Y. Manolopoulos, Processing distance join queries with constraints, The Computer Journal 49 (3) (2006) 281–296.
[30] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, in: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD), San Jose, CA, 1995, pp. 71–79.
[31] T. Seidl, H.P. Kriegel, Optimal multi-step *k*-nearest neighbor search, in: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD), Seattle, WA, 1998, pp. 154–165.
[32] S. Shekhar, S. Chawla, Spatial Databases: A Tour, Prentice-Hall, New Jersey, 2003.
[33] H. Shin, B. Moon, S. Lee, Adaptive multi-stage distance join processing, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD), Dallas, TX, 2000, pp. 343–354.
[34] Y. Shou, N. Mamoulis, H. Cao, D. Papadias, D.W. Cheung, Evaluation of iceberg distance joins, in: Proceedings of the 8th International Symposium on Spatial and Temporal Databases (SSTD), Santorini Island, Greece, 2003, pp. 270–288.
[35] M. Stonebraker, J. Frew, K. Gardels, J. Meredith, The sequoia 2000 benchmark, in: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD), Washington, WA, 1993, pp. 2–11.
[36] The R-tree Portal (TIGER/Line® files, Geography Division, US Census Bureau, <http://www.census.gov/geo/www/tiger/>) <http://www.rtreeportal.org/>, 2004.
[37] Z. Yang, G. Yang, A near-optimal similarity join algorithm and performance evaluation, Information Sciences 167 (1–4) (2004) 87–108.
[38] W. Zhang, R.E. Korf, Performance of linear-space search algorithms, Artificial Intelligence 79 (2) (1995) 241–292.