The 'Cross' Rectangle Intersection Problem

V. KAPELIOS[1], G. PANAGOPOULOU[1,2], G. PAPAMICHAIL[1], S. SIRMAKESSIS[1,2], AND A. TSAKALIDIS[1,2]

[1]*Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece*
[2]*Computer Technology Institute, PO Box 1122, 26110, Greece*
Email: panag@cti.gr

In this paper we present a solution for a special case of the general rectangle intersection problem that has not been previously considered as a different case. This case, named the 'cross' intersection case, reports the set of these iso-oriented rectangles that intersect a query rectangle but do not enclose it and do not have one of their vertices inside it. We present solutions for unrestricted and restricted universe (grid) for the R^d space. In the case of unrestricted d-dimensional space, the problem is solved in time $O(\log^{2d-3} n \log\log n + K)$ using $O(n\log^{2d-3} n)$ space, where n is the number of rectangles and K is the size of the answer. In the case of restricted universe the same problem can be solved in $O(\log^{d-1} M + K)$ time and $O(n\sqrt{\log M}^{2d-3})$ space, where M is the upper limit of the grid coordinates. Update operation in the dynamized version of the problem for the unrestricted and grid case is performed in $O(\log^{2d-2} n)$ and $O(\log^{d-1} M)$ time, respectively.

1. INTRODUCTION

The rectangle intersection problem has been tackled by many scientists under various considerations, as it is one of the most common problems in computational geometry. With the term *rectangle intersection* problem in the plane, we mean the problem of determining the subset of a set of n iso-oriented rectangles that intersect a given query rectangle Q. Edelsbrunner and Maurer (1981) presented a solution to the general rectangle intersection problem with space $O(n\log^d n)$ [following Knuth in (Knuth, 1976) a function will be said to be $O(f)$ iff it is no more than proportional to f, $\Omega(f)$ iff it is at least proportional to f and $\Theta(f)$ iff it is both $\Omega(f)$ and $O(f)$ ($\log n$ stands for $\log_2 n$)] and time $O(\log^{d-1} n + K)$, where K is the size of the answer and d is the dimension of space, using layered structures. In a dynamic environment, the above time is increased to $O(\log^d n + K)$. Moreover, they also described how their solution can apply to the determination of all intersecting pairs of a given set of orthogonal objects. In their work, Edelsbrunner and Maurer adopted a unified approach for orthogonal intersection searching which can be applied to rectangle intersection searching particularly. Though, the method proposed can not be applied to give answers in special cases of intersection, such as rectangle containment or enclosure.

Furthermore, Lee and Wong (1981) have solved the intersection problem in $O(\log^{2d-1} n + K)$ time and $\theta(n\log^{2d-1} n)$ space when all inputs are not known before the preprocessing. They also gave a unified approach which solves the same problem when all inputs are known at preprocessing stage, in $O(n\log n + n\log^{2d-3} n + K)$ time. Both results hold for the containment and edge intersection problems as they stated in Lee and Wong (1981).

Both of these works solve the problem without dividing it into distinct subcases. However, individual problems that are actually parts of the rectangle intersection problem, such as rectangle containment or enclosure, are of special interest in computational geometry. The solutions of the individual cases of rectangle intersection are usually more complex and have worst time bounds than the general case, since the special characteristics of each distinct case (which are not crucial in the general case) should be now taken under consideration and handled separately. Taking advantage of known results in the two particular cases, we can also use them in order to solve a part of the general rectangle intersection problem. Moving in this direction we can divide the intersection problem into three different subcases.

The first case is the problem of determining the rectangles that have one or more of their vertices included in the query rectangle (rectangle containment). This case has been solved in d-dimensions using $O(\log^d n + K)$ query time and $O(n\log^{d-1} n)$ space executing a two-dimensional range search (Willard, 1985).

The second case is the problem of determining the rectangles that enclose the query rectangle (rectangle enclosure). This particular case has been solved by Bistiolas *et al.* (1993) using $O(\log^{2d-1} n + K)$ query time and $O(n\log^{2d-2} n)$ space (dynamic version).

The third case is the problem of determining the rectangles that intersect the query, but do not belong to the solutions of the cases previously described. In other words, the answer of this case-query are those rectangles that do belong to the overall solution of the rectangle intersection problem, but are excluded by the solutions of the other two cases. This particular case has not been previously considered as an individual problem and no known results have been presented. This case of intersection is of great importance in VLSI design and testing for intersected layers of silicon (testing for

Correspondence to S. Sirmakessis

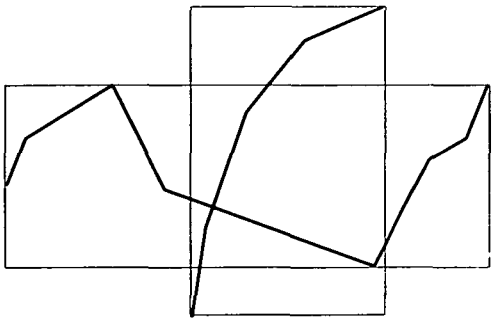

FIGURE 1. A curve intersection in the BSPR tree.

contacts of silicon). Moreover, this intersection can be found important for geographical information systems. For example, the BSPR tree (Burton, 1977), used to represent curves for GIS, answers curve intersection queries by transforming them to intersections of rectangles. More precisely, the BSPR tree stores curves as polygonal lines by using approximations based on rectangles (referred to as section rectangles). In the case where we want to check whether or not two curves intersect, we have to compute one clear intersection between their associated section rectangles. A clear intersection is defined as the case where two parallel edges of the first section rectangle intersect with the two parallel edges of the other rectangle (see Figure 1). For more details of the problem refer to Burton (1977).

We name this case the 'cross' intersection case. This name is due to the plane figure which is created by a rectangle of the solution and the query rectangle (see Figure 2).

More formally, the rectangles which belong to the set of answers are those whose horizontal (vertical) sides enclose the corresponding sides of the query and at least one of their vertical (horizontal) sides is enclosed by the corresponding sides of the query (see Figure 2, instances 1–3 and 4–6). The problem described above is solved, using a 2-fold structure in $O(\log^2 n)$ time and $O(n \log n)$ space.

In this paper we present a solution to the cross intersection problem for unrestricted and restricted universes in d-dimensional space. This approach solves the on-line problem (meaning that we find the set of rectangles that intersect a query one) in time $O(\log^{2d-3} n \log\log n + K)$ using $O(n \log^{2d-3} n)$ space in

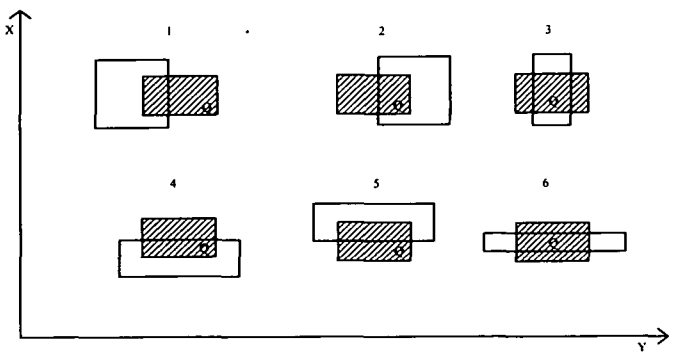

FIGURE 2. The query rectangle is shadowed.

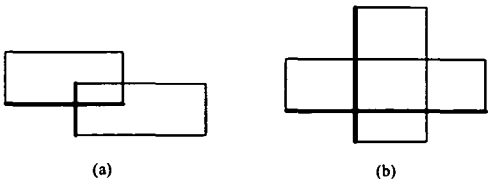

FIGURE 3. Two similar cases for the solution of Edelsbrunner and Maurer.

the unrestricted universe, where n is the number of rectangles and K the size of the answer. In the case of a restricted universe the same problem can be solved in $O(\log^{d-1} M + K)$ time and $O(n\sqrt{\log M}^{2d-3})$ space, where M is the upper limit of the grid coordinates. The update operation in the dynamized version of the problem for the unrestricted and grid case is performed in $O(\log^{2d-2} n)$ and $O(\log^{d-1} M)$ time, respectively. The solution of Lee and Wong (1981), applied in the cross intersection case, gives worse results for the time and space bounds in an unrestricted universe than our solution and no reference is done for the restricted universe.

Our result is different from the result of Edelsbrunner and Maurer (1981). They have presented a unified way to deal with intersection and enclosure without considering them as separate problems. They approach every intersection in a common way without knowing what kind of intersection occurs. For example, using the solution of Edelsbrunner and Maurer the two cases presented in Figure 3 are treated similarly. If someone wants to report only rectangles that have a cross intersection (perhaps in VLSI design and testing) which is the case (b) of Figure 3, he should execute a more complicated query than the solution described in Edelsbrunner and Maurer (1981). The solution of Edelsbrunner and Maurer cannot report only cross intersections since this case is more complex than the cases presented in Edelsbrunner and Maurer (1981). Instead of this, the results presented in this paper can be used. That is the reason why the time and space bounds of the cross intersection are worse than the bounds in Edelsbrunner and Maurer (1981). Moreover the equivalent problem for the grid case is not mentioned in their work. Our solution can be used in this area.

This paper is organized as follows. Section 2 briefly presents some known data structures used in our solution. Section 3 introduces the basic 2-fold structure which answers the cross intersection query in arbitrary space. Section 4 introduces the corresponding two-layered structure which answers the same query on a grid. Problems, appearing in the appropriate combination of the various basic structures, are also tackled. Section 5 shows how these structures can be modified in order to answer the query in a dynamic environment. Finally, Section 6 summarizes the results of this paper.

2. PRELIMINARIES

This section briefly reviews the major outlines of the data structures used in our method. The first structure is the

priority-search tree introduced by McCreight (1985). It is a balanced binary tree used for representing a dynamic set D of ordered pairs $[x, y]$ over the set $0,1,\ldots,k-1$ of integers and it supports algorithms for the following operations:

- **InsertPair (x,y):** Insert a pair $[x, y]$ into D.
- **DeletePair (x,y):** Delete a pair $[x, y]$ from D.
- **MixXInRectangle (x_0, x_1, y_1):** Given test integers x_0, x_1 and y_1, among all pairs $[x, y]$ in D such that $x_0 \leqslant x \leqslant x_1$ and $y \leqslant y_1$, find a pair whose x is minimal.
- **MaxXInRectangle (x_0, x_1, y_1):** Given test integers x_0, x_1 and y_1, among all pairs $[x, y]$ in D such that $x_0 \leqslant x \leqslant x_1$ and $y \leqslant y_1$, find a pair whose x is maximal.
- **MinYInXRange (x_0, x_1):** Given test integers x_0 and x_1, among all pairs $[x, y]$ in D such that $x_0 \leqslant x \leqslant x_1$, find a pair whose y is minimal.
- **EnumarateRectangle (x_0, x_1, y_1):** Given test integers x_0, x_1 and y_1, enumerate those pairs $[x, y]$ in D such that $x_0 \leqslant x \leqslant x_1$ and $y \leqslant y_1$.

This searching is actually described as 1.5-dimensional. The data has two independent dimensions, but the priority search tree does not allow equally powerful searching operations on both. There is a major dimension (x) permitting arbitrary range queries and a minor one (y) permitting only enumeration in increasing order. The tree stores points of the plane, where the leaves correspond to coordinates according to one axis, while the internal nodes are associated with a point.

More precisely, the point with the largest coordinate in the other axis is stored in the root. Internal nodes store the point with the largest coordinate that has not been stored higher up in the tree. The tree occupies $O(n)$ space and answers three-sided (half-infinite) range queries in two-dimensional plane in time $O(\log n + K)$, where K is the size of the answer.

Another basic structure used in our method is the range tree. It was introduced in Willard and Lucker (1985) solving the range searching problem, in two dimensions, in $O(\log^2 n + K)$ time using $O(n \log n)$ space. A range query is the problem that fits the following general description:

> Given a set of objects X in R^d, store X in a suitable data structure so that for any query object y (regarded as 'range'), those $x \in X$ such that $x \cap y \neq 0$ can be identified quickly.

A one-dimensional range tree can be considered as a leaf-oriented balanced binary search tree, for the coordinates of points in one dimension. A d-dimensional one is a $(d-1)$-dimensional range tree, where each internal node corresponds to a one-dimensional range tree that organizes the points descending from that node according to the last coordinate.

For a set of n points in R^d, we would like to retrieve quickly all points in any specified orthogonal box. Range trees allow us to solve orthogonal range queries in time $O(\log^d n)$ and space $O(\log^{d-1} n)$. Consider the case $d = 2$. For a query rectangle $[a, b] \times [c, d]$, we decompose the x-range $[a, b]$ into $\log n$ intervals and perform in each interval a one-dimensional search in an auxiliary y-tree. The query time is $O(\log^2 n + K)$ if k points are retrieved. The same method applies for the generalization in d-dimensions.

Other data structures used in the solution of the problem in restricted universe (grid) are the p- and q-fast tries. By the term grid we refer to a d-dimensional space, where each coordinate can take only distinct, discrete values under a highest limit M. We say that all points have coordinates in any axis from the finite set $[1,..,M]$.

The trie, as a hybrid data structure that combines arrays and pointers, was proposed by Fredkin (1962). It is a simple way to structure a file by using the digital representation of its elements. The general condition is as follows: the universe U consists of all strings L over some alphabet of say k elements, i.e. $U = \{0, 1, \ldots, k-1\}^L$. A set $S \subseteq U$ is represented as the k-ary tree consisting of all prefixes of elements of S. An implementation which immediately comes in mind is to use an array of length k for every internal node of the tree. Then operations access, insert and delete are very fast and are very simple to program. The algorithms take time $O(L) = O(\log_k N)$ where $N = |U|$. Unfortunately, the space requirements of a trie as described above can be horrendous: $O(n \cdot L \cdot k)$. For each element of set S, $|S| = n$, we might have to store an entire path of L nodes, all of which have degree 1 and use up to space $O(k)$.

There is a simple method to reduce the storage requirements to $O(n \cdot k)$. We only store internal nodes which are at least binary. Since a trie for a set S of size n has n leaves there will be at most $n-1$ internal nodes of degree 2 or more. Chains of internal nodes of degree 1 are replaced by a single number; the number of nodes in the chain. A trie can support the following three operations:

- **Successor(x):** Find the least element in the set S with key value greater than x.
- **Predecessor(x):** Find the greatest element in the set S with key value less than x.
- **Subset(x1, x2):** Find the list of those elements of S whose key value lies between x_1 and x_2.

The structures p- and q-fast tries are two modified versions of tries presented by Willard (1984). These kind of tries use additional fields (structures) in each of their internal nodes, performing retrieval and update operations in $O(\sqrt{\log M})$ time. Their only difference is the space requirements, which is $O(n\sqrt{\log M} 2^{\sqrt{\log M}})$ for the p-fast trie and declines in linear $O(n)$ for the q-fast trie (where n is the number of elements stored in them).

The p-fast trie is a trie where a new internal node is stored if and only if it is the ancestor of some elements in the set of integer keys stored in the structure. Moreover each leaf of a p-fast trie contains a pointer to the leaf that

lies to its immediate left and another pointer to the leaf at its right. A q-fast trie is a data structure which represents a set S by employing two substructures, called the upper and lower parts. Its upper section is a p-fast trie that represents a partition of S. Its lower part is a forest of two to three trees whose ith tree represents the sub-set S_i of each partition of S. The reduction in space of q-fast trie is achieved after prunning of the bottom of the p-fast trie in order to conserve memory. A detailed analysis of p- and q-fast tries can be found in Willard (1984).

Another type of fast tries are the x- and y-fast tries (Willard, 1983). These kinds of tries use as the basic structure a binary trie and are augmented by a level-search structure (Fredman *et al.*, 1982) for each level of the trie (the total number of levels is h). Using these structures, the overall retrieval complexity is $\Theta(\log\log M)$ for both tries [the retrieve operation is based on perfect hashing (Mehlhorn, 1984; Jacobs and van Emde Boas, 1986) using additional information in each internal node]. They only differ in the occupied memory space; the x-fast trie needs $O(n\log M)$ space, while the y-fast trie only $\Theta(n)$, as the pruning technique is applied.

In a few words, comparing the two kinds of fast tries (p and q with x and y) we can mention that the latter achieves better time bounds for the retrieval operation in the same space, but the preprocessing time is significantly higher (because of perfect hashing structures). In addition only p- and q-fast tries support update operations in time proportional to the retrieval time.

3. THE CROSS INTERSECTION CASE

First of all we solve the two-dimensional case of the rectangle intersection assuming that we have n rectangles in the plane, where a subset of them intersect a query rectangle Q in the way we previously described. For the sake of simplicity, we try to find the rectangles that enclose the query rectangle according to the y-axis and one at least of its sides is partly included in the query (except its vertices) according to the x-axis. (Figure 2, instances 1–3). The other symmetric case (Figure 2, instances 4–6) is tackled in an analogous way.

We assume that the query rectangle Q is represented by its coordinates in the two axes, forming the four-element tuple $[x_{Ql}, x_{Qr}, y_{Qb}, y_{Qt}]$. A random rectangle is represented by the tuple $[x_{il}, x_{ir}, y_{ib}, y_{it}]$, $(1 \leqslant i \leqslant n)$.

The pairs $[x_{il}, y_{ib}], [x_{ir}, y_{ib}]$ represent a two-dimensional point (more precisely vertices of a rectangle). We construct our 2-fold structure as a two-dimensional range tree for the $2n$ points:

$$[x_{1l}, y_{1b}], [x_{1r}, y_{1b}], \ldots, [x_{nl}, y_{nb}], [x_{nr}, y_{nb}]$$

modified in the second dimension in order to be combined in a priority search tree. In other words, every node v of the range tree in the first layer points to a priority search tree for the $[y_{ib}, y_{it}]$ pseudopoints, which correspond to leaves of the subtree rooted at v. In this way, the coordinates of rectangles are stored in our structure. Finally, the priority trees are organized having the largest value y_{it} stored in the root.

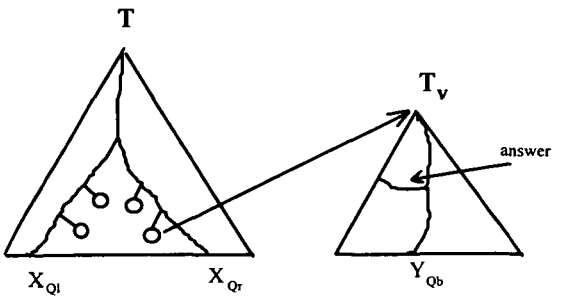

FIGURE 4. The 2-fold structure.

3.1. The algorithm

Let T be the first layer of our structure and T_v the second layer which is associated with each internal node v of T. The algorithm that answers the query is described below.

Step 1. Search for x_{Ql} and x_{Qr} in tree T. In this way two paths are defined, as illustrated in Figure 4. Let $v_1, v_2, \ldots, v_i$ be the sons of the nodes forming the paths to x_{Ql} and x_{Qr}. without belonging to the path themselves and they also lie between the two paths. Observe that the trees $T_{v1}, T_{v2}, \ldots, T_{vi}$ of the second layer store all the rectangles, having at least one of their vertical sides included between the lines $x_1 = x_{Ql}$ and $x_2 = x_{Qr}$.

Step 2. Search for y_{Qb} in every second layer tree T_{v_i} (Figure 4 shows only one of them). The answer is contained between the search paths to y_{Qb} and the leftmost leaf of the tree. We traverse top-down all T_{v_i} trees and report all pairs $[y_{ib}, y_{it}]$ (corresponding rectangles) until the condition $y_{Qt} < y_{it}$ holds (the largest y_{it} values are pushed to the root).

Additional inspection is needed for the path leading to y_{Qb}, the rightmost of the examined paths. This is so because in this path pseudopoints can be stored (corresponding rectangles) with $y_{ib} > y_{Qb}$ (while all the paths on the left have surely stored points with $y_{ib} < y_{Qb}$). So, for each possible answer from this path, we must examine if the above condition holds too. This additional traversal of the path does not bring any changes to the overall time complexity.

LEMMA 1. The presented algorithm reports all the rectangles that belong to the cross intersection case once and only once. Exceptions to the rule above are the ones whose sides are included partly in the query according to one axis; these rectangles are reported twice.

Proof. The fact that we find all the answers of the query follows from the definition of the range and priority trees. Besides, the rectangles that belong to the answer and have one coordinate [e.g. in (x_{Ql}, x_{Qr})] are reported once and only once, while those with both of their x-coordinates in the range above are reported exactly twice. At step 1 we can find the second layer trees

$T_{v1}, T_{v2}, .., T_{vi}$ that store points $[x_{il}, y_{ib}]$ and $[x_{ir'}, y_{ib'}]$ that satisfy the conditions $x_{Ql} < x_{il} < x_{Qr}$ and $x_{Ql} < x_{ir'} < x_{Qr}$ accordingly. The points $[x_{il}, y_{ib}]$ and $[x_{ir'}, y_{ib'}]$ can belong to different rectangles (in this case the corresponding rectangles are reported once) or to the same one (thus $[x_{ir'}, y_{ib'}] = [x_{ir}, y_{ib}]$) so that is reported twice (once for the $[x_{il}, y_{ib}]$ and once for $[x_{ir}, y_{ib}]$).

Comment. The attribute of the algorithm described above does not stand as a disadvantage. We can keep a vector of bits where each bit corresponds to a rectangle and set it at a predefined value every time we find an answer. Finally we can check the vector and report only the rectangles whose entry has the predefined value.

3.2. Space and time analysis

THEOREM 1. The space occupied of the structure is $O(n\log n)$ and the answer is calculated in time $O(\log n \log\log n + K)$, where K is the cardinality of the answer and n the number of rectangles.

Proof. The total space occupied by our two layered structure is $O(n\log n)$, because every point $[x_{il}, y_{ib}]$ or $[x_{ir'}, y_{ib'}]$ is stored in $O(\log 2n)$ nodes of range tree (first layer) and every priority tree occupies $O(n_i)$ space ($n_i \leqslant n$) for every one of the n_i pseudopoints $[y_{ib}, y_{it}]$, that are stored in it.

As the time bound in static case is concerned, we consume $O(\log^2 n + K)$ time, since in Step 1 we traverse two paths of length logn and in Step 2 we answer the enclosure query using a priority tree in $O(\log n_i + K_i)$ with $n_i \leqslant n$ (K_i is the answer taken by every second layer tree T_{v_i}) for every node $v_1, v_2, ..., v_i$ found in Step 1.

The time complexity can be reduced to $O(\log n \log\log n + K)$ by using extended priority trees (Fries *et al.*, 1987) as the second layer of our structure, instead of a simple priority search tree. □

3.3. The *d*-dimensional case

Considering the problem in d dimensions, we can state that the d-dimensional rectangles that belong to the set of answers are those that enclose the query according to $d-1$ dimensions and at least one of their sides is partly included (without the vertices) in the query one according to one dimension. More precisely, let $[x_{il}^1, x_{ir}^1], [x_{il}^2, x_{ir}^2], ..., [x_{il}^d, x_{ir}^d]$ be the pair of coordinates in each of d axis of the rectangle and $[x_{Ql}^1, x_{Qr}^1], ..., [x_{Ql}^d, x_{Qr}^d]$ the corresponding pairs of the query. We must now find all the rectangles whose $d-1$ pairs enclose (as intervals) the corresponding pairs (intervals) of query and at least one of the two coordinates of the last pair is included in the corresponding pair of the query.

The above statements are true because if, for example, a rectangle encloses the query according only to $d-2$ dimensions, then the vertex which is formed by the coordinates in two other dimensions (axis) is surely included in the query. Then, the intersection of these two rectangles is reported by range searching, as described in Section 1 (first case of intersection).

In order to enlarge our structure for more than two layers to answer the query in d dimensions, we take into account that the layers added must solve the problem of segment enclosure. Having these observations in mind, we simply add two layers of range trees for every dimension above two, where the problem is set. Each one of the two layers of range tree stores rectangles according to the one of the two coordinates $[x_{il}^j, x_{ir}^j]$ in the j dimension, and we simply search in the first tree for rectangles having $x_{il}^j < x_{Ql}^j$ and in the second tree for the ones having $x_{ir}^j > x_{Qr}^j$.

As a result, our structure in d dimensions will be a $(2d-2)$-dimensional range tree modified in the last layer, as we described previously, such that every node of layer $2d-3$ points to a priority search tree.

Let us find the rectangles that have at least one side partly included in the query according to the first dimension and enclose the query according to the other $d-1$ dimensions. The other $d-1$ symmetric instances of the problem are solved in an analogous way. In the first layer of our structure we store the $2n$ $(2d-2)$-dimensional points of the $[x_{il}^1, x_{il}^2, x_{il}^3, ..., x_{il}^d], [x_{ir}^1, x_{ir}^2, x_{ir}^3 ..., x_{ir}^d]$ in order to find the rectangles that have at least one coordinate according to dimension 1 in the range $[x_{Ql}^1, x_{Qr}^1]$. The subsequent $2d-4$ layers are range trees that were used to solve the enclosure problem according to $d-2$ dimensions (each pair of layers solve the problem in one dimension, see Figure 5). At layer $2d-3$ we have already solved the problem according to $d-1$ dimensions and we have to select from those rectangles found that also enclose the query according to the d dimension. Using the priority tree in the same way as we did in two dimensions, we can locate the answers in only one layer.

THEOREM 2. The space bound in the solution for the d-dimension case is $O(n\log^{2d-3} n)$ and the answer can be achieved in $O(\log^{2d-3} n \log\log n + K)$ time, where K is the cardinality of the answer and n the number of rectangles.

Proof. From the discussion above, it follows that the overall query time in d dimensions is $O(\log^{2d-2} n + K)$. A time reduction in $O(\log^{2d-3} n \log\log n + K)$ can be achieved if we replace the ordinary priority tree of the last layer with an extended priority search tree. The space bound is $O(n \log^{2d-3} n)$, since each one of the n rectangles is stored in $2d-3$ layers of the range tree and the priority tree in the last layer occupies linear space. □

4. ANSWERING THE QUERY IN A RESTRICTED UNIVERSE

So far, we have assumed that the rectangles involved in our problem are set in two-dimensional or generally in d-dimensional space R^d. This assumption requires that the values of the rectangles' coordinates are real numbers.

This accuracy in values may be impossible or even

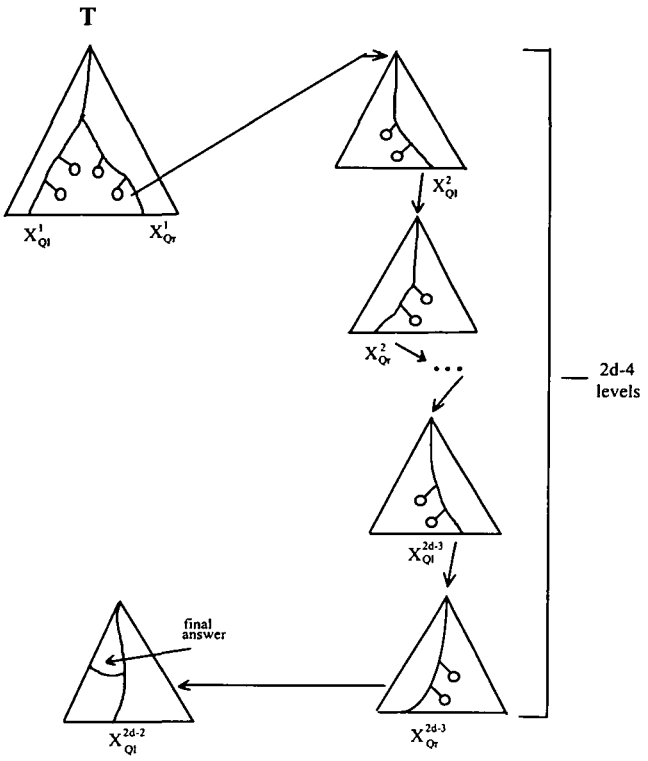

FIGURE 5. The multi-layered structure.

undesired when algorithms of computational geometry are used in real applications (VLSI design, computer graphics, etc.). For this reason, solutions to known geometrical problems on a grid have received an increasing amount of attention. As we mentioned earlier, the grid is the geometrical space, where points have integer coordinates from a finite set [1,...,M].

Many optimal solutions in arbitrary space cannot be proved on a grid. Innovated data structures, such as those presented in the Introduction, can solve efficiently various geometrical problems.

4.1. The modified data structure

We present only the solution of the problem reporting the two-dimensional rectangles that enclose the query according to the y-axis and have one vertical side partially included (without the vertices) in query according to the x-axis. The symmetric situation is handled in analogous way.

The first layer of our structure is a p-fast trie which stores the $2n$ points $[x_{l1}, y_{lb}][x_{lr}, y_{lb}], ..., [x_{nl}, y_{nb}]$ according to the x-coordinates in its leaves. This trie is augmented in the same way as proposed by Overmars (1988) for efficient searching. More precisely, every leaf δ of the p-fast trie is associated with a set of lists $R_\delta^1, ..., R_\delta^{\lceil\sqrt{\log M}\rceil}$, where every list R_δ, holds pointers to internal nodes which are right sons of the nodes consisting the path to δ, but they do not belong to the path and in addition they are in a level less than i (the length of every path in the p-fast trie is $\lceil\sqrt{\log M}\rceil$ and the level is counted bottom to top).

Similarly every leaf δ is associated with a set $L_\delta^1, ..., L_\delta^{\lceil\sqrt{\log M}\rceil}$ of lists. Every list has pointers to left sons of nodes which belong to the path to δ and do not form the path themselves in a level less than i (Figure 6 depicts the first layer of the structure and associated lists of leaves δ_1 and δ_2). Every list needs $O(\sqrt{\log M})$ space. We have $O(\sqrt{\log M})$ lists (the cardinality of levels) for every leaf, leading to $O(\log M)$ space complexity for each leaf.

The second layer of our structure (see Figure 6) is a set of priority search tries; everyone of them is pointed by an internal node of a first layer structure. A priority search trie (in Figure 6 only one of them is showed) is a q-fast trie that stores the pseudopoints $[y_{ib}, y_{it}]$ $(1 \leqslant i \leqslant n)$ sorted by y_{ib} coordinate to its leaves and every internal node has a priority field identical to the corresponding priority tree (McCreight, 1985) forming a heap structure in the trie. The organization of a priority search trie is based on the placement of the largest y_{it} value at the root.

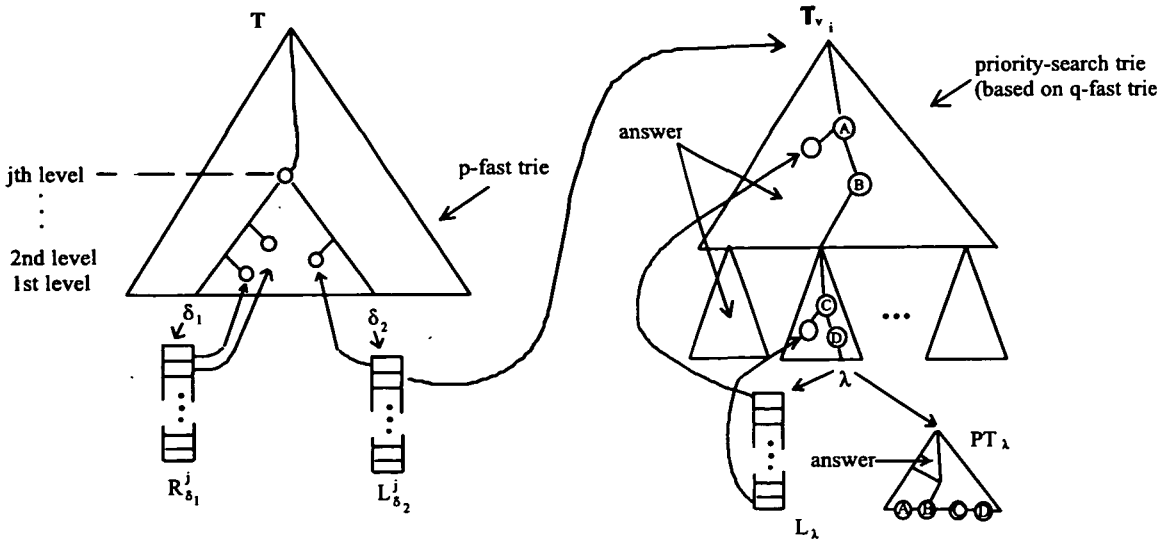

FIGURE 6. The 2-fold structure for a restricted universe (grid).

The q-fast trie, which is the basic structure of priority search tries, is also augmented in order to support efficient searching. More precisely, every leaf λ of a priority trie is associated with a priority trie PT_λ, which stores the $O(\sqrt{\log M})$ points $[y_{ib}, y_{it}]$, found in the path towards λ. Every path of PT_λ has $O(\log\sqrt{\log M})$ nodes.

Furthermore, every leaf λ is associated to a list L_λ consisting of points to nodes that are left sons of nodes belonging to the path towards λ, but do not belong to the path themselves. These nodes are stored in descendant order of y_{it} values of points which are associated with their priority fields.

Because of the pruning used to q-fast tries' construction (Willard, 1984), a number of pseudopoints $[y_{ib}, y_{it}]$ are not 'pushed' to internal nodes' priority fields and accordingly stored in list L_λ of their leaf. Consequently, we construct a list $LIST_\lambda$ (as in Overmars, 1988) of pseudopoints similar to those previously mentioned. The pseudopoints are stored according to largest y_{it} values. Every list $LIST_\lambda$ occupies $O(\sqrt{\log M})$ space.

Let $[x_{Ql}, x_{Qr}, y_{Qb}, y_{Qt}]$ be the discrete integer coordinates of query rectangle, T the p- fast trie of the first layer and T_v the priority trie of the second layer associated with the v node of T.

Step 1. First, we search for x_{Ql} and x_{Qr} in T. Let δ_1 and δ_2 be, respectively, the leaves where the search ends. The nodes $v_1, v_2, ..., v_k$ are the sons of the nodes forming the paths to δ_1 and δ_2 without belonging to the path themselves and they also lie between the two paths. These nodes store the rectangles that have at least one of their x-coordinates in the interval $[x_{Ql}, x_{Qr}]$.

In order to locate these nodes and consequently the second layer tries that they are associated with, we firstly find the common ancestor of leaves δ_1 and δ_2. This can be done in only $O(\sqrt{\log M})$ time. Let j be the level of the common ancestor. Then the lists $R^j_{\delta_1}$ and $R^j_{\delta_2}$ (see Figure 6) have pointers to the nodes and to the tries $T_{v_1}, T_{v_2}, .., T_{v_k}$ of the second layer which must be further searched. So, in time $O(\sqrt{\log M})$ we can find all the rectangles that have at least one x-coordinate (corresponding vertical side) in interval (x_{Ql}, x_{Qr}).

Step 2. In every priority trie $T_{v1}, ..., T_{v_k}$ we search for y_{Qb} in $O(\sqrt{\log M})$ time for each priority trie. Let λ be the leaf where the search for y_{Qb} ends. The rectangles that belong to the answer must satisfy the conditions $y_{ib} \leqslant y_{Qb}$ and $y_{it} \leqslant y_{Qt}$.

First we examine the rectangles stored in the path towards λ. In other words, we perform the enclosure query in priority tree PT_λ as it was described in Step 2 of the algorithm for the arbitrary universe. This can be done in $O(\log\sqrt{\log M + K_i})$ time, where K_i are the answers reported from the path.

We traverse top-down each subtrie rooted from nodes pointed by list L_λ, reporting all rectangles having $y_{it} \geqslant y_{Qt}$. The traversal may be continued to the elements of $LIST_\kappa$, where κ is any leaf between the leftmost leaf of priority trie and λ, but even in that case we report answers. If we find a node with priority field $y_{it} < y_{Qt}$, the traversal in this subtrie is immediately stopped. For each priority trie, $O(\sqrt{\log M} + K_i)$ time is consumed in order to report k_i answers.

4.2. Space and time analysis

THEOREM 3. The grid case of the cross rectangle intersection problem can be solved in $O(\log M + K)$ time using $O(n\sqrt{\log M})$ space, where K is the cardinality of the answer and M is the size of the grid.

Proof. From the description of the algorithm and the observation that $O(\sqrt{\log M})$ priority tries at the second layer are examined, it can be concluded that $O(\log M + K)$ overall time is needed to answer the query on a two-dimensional grid.

As the space complexity is concerned, it is obvious that each of $2n$ points of p-fast trie T is stored in $\sqrt{\log M}$ internal nodes which occupies $\Theta(2^{\sqrt{\log M}})$ space. Furthermore, each leaf of T occupies $O(\log M)$ space due to the R_{δ^j} and L_{δ^j} lists ($1 \leqslant j \leqslant \sqrt{\log M}$). As a result, the first layer needs totally $O(n\sqrt{\log M}2^{\sqrt{\log M}})$ space. The priority trie uses linear space on the number of points stored at it. However, every leaf λ of such a trie occupies $O(\sqrt{\log M})$ space because of the lists L_λ and $LIST_\lambda$ associated with it, as well as the priority tree PT_λ. Consequently every T_{v_i} priority trie occupies:

$$O\left(\frac{n}{\sqrt{\log M}2^{\sqrt{\log M}}}\sqrt{\log M}\right)$$

$= O(\text{n})$ space

$$\left(\frac{n}{\sqrt{\log M}2^{\sqrt{\log M}}} \text{ are the number of leaves of } q\text{-fast trie}\right).$$

From the discussion above follows that the overall space complexity of our structure on a grid is $O(n\sqrt{\log M}2^{\sqrt{\log M}})$.

The space complexity can be reduced to $O(n\sqrt{\log M})$ provided that we use a q- instead of a p-fast trie as the basic structure of the first layer, taking advantage of its linear space. More specifically the augmentation of this trie with the set of lists R^j_δ and L^j_δ only increases the space to $O(n\sqrt{\log M})$ from the $O(n)$ bound of the q-fast trie. However, the complexity above is exactly the overall one needed by our 2-fold structure since the height of first layer is $O(\sqrt{\log M})$ and every one of the $2n$ elements (corresponding rectangles) are stored in a path of the first layer. □

4.3. Generalization to d-dimensions

The extension of our structure in the d-dimensional grid follows the ideas stated at Section 3.3 for an unrestricted universe using the q-fast trie as the basic structure. The first and last layer are constructed in exactly the same

way as we described in the two-dimensional case. In order to achieve efficient searching in the intermediate $2d-2$ layers (depicted in Figure 5), we have to construct a list R_δ or L_δ in each leaf of these tries, aiming to find the rectangles which have $x^j_{ir} > x^j_{Qr}$ or $x^j_{il} < x^j_{Ql}$ $(1 \leqslant i \leqslant N$ and $1 \leqslant j \leqslant d)$, respectively.

Finally, we answer cross intersection queries in a d-dimensional grid in $O((\sqrt{\log M})^{2d-2} + K) = O(\log)^{d-1} M + K)$ time and $O(n\sqrt{\log M}^{2d-3})$ space.

4.4. An alternative solution

Although the method described above gives a satisfying solution to the intersection query on a grid, there are other structures (tries) which can achieve even better performance in retrieval operations. The x- and y-fast tries (Willard, 1983) have overall retrieval complexity $O(\log\log M)$. As the condition $\log\log M < \sqrt{\log M}$ holds in general, one could expect that using these tries, in order to build our structure, might have better performance results. We have already mentioned that a disadvantage of these tries compared with the p- and q-fast tries is the high preprocessing time and the fact that they do not support a $\log\log M$ time for update operations in a dynamic environment.

From the above observations, we can construct the 2-fold structure which answers the query in the plane, following exactly the same way as before, replacing only the q-fast trie in the two layers by a y-fast trie . We augment the y-fast trie of the first layer with lists L^j_δ and R^j_δ $(1 \leqslant j \leqslant \log M)$ for each of the $2n$ leaves. In the second layer each priority trie, associated with a node of the first layer, is constructed using a y-fast trie as the basic structure. The corresponding structures (L_λ, PT_λ, $LIST_\lambda$) are added in the same way as in q-fast tries. The priority tries based on y-fast tries are organized by placing the largest y_{it} value at the root.

The algorithm works similarly in this structure by searching for x_{Ql} and x_{Qr} in the first layer structure (y-fast trie) and locating the priority tries which must be further searched using the lists $L^j_{\delta 1}$ and $R^j_{\delta 2}$. (We remember that δ and δ_2 are the leaves where the search operation for x_{Ql} and x_{Qr} terminates and j is the level where these two paths split). The actions above cost only $O(\log\log M)$, but the priority tries which must be further examined are $O(\log M)$ (proportional to the height of the binary y-fast trie). The search for the pseudopoints (corresponding rectangles) which satisfy both of the conditions $y_{ib} \leqslant y_{Qb}$ and $y_{it} \leqslant y_{Qt}$ in each of the priority tries found, consumes $O(\log M \log\log M + K)$ time (where K is the size of the answer). The space occupied by this new 2-fold structure is $O(n\log^2 M)$ since each one of the $2N$ points in the first layer is stored in $\log M$ nodes of the x-fast trie and everyone of its leaves occupies $O(\log^2 M)$ due to the lists R^j_δ and L^j_δ. Furthermore, each one of the priority tries of the second layer occupies $O(n\log M)$ space. However, we can save some space if we prune the y-fast trie of the first layer so that every leaf corresponds to $\log^2 M$ elements (thus the trie has $O(\frac{n}{\log^2 M})$ leaves). Consequently, the overall space complexity falls to $O(n\log M)$. In a d-dimensional grid, the method described above answers the query in $O(\log^{2d-3} M \log\log M + K)$ time and uses $O(n\log^{2d-3} M)$ space.

5. ANSWERING THE QUERY IN A DYNAMIC ENVIRONMENT

So far, we have consider ways of answering the query in a static environment, i.e. the set of rectangles (their coordinates) are given as an input once and for all. Of course, the only element changed every time we perform an on-line query is the coordinates of the query rectangle. If we consider our problem in an environment where the given set of rectangles is modified through time, then we say that the query is set in a dynamic environment.

The dynamization of our structure in arbitrary space is achieved by using BB[a] and BB trees as the basic structures for implementing the first and the second layer, respectively. In this way, the amortized time for each update is $O(\log^2 n)$ for the plane and $O(\log^{2d-2} n)$ for a d-dimensional space. By applying the method proposed by Willard and Luecker (1985) we can have $O(\log^{2d-2} n)$ time for each update operation.

As far as the update operation on a grid is concerned, we must keep in mind that p- and q-fast tries support update operations in time analogous to retrieve operations, while x- and y-fast tries do not have this property. Furthermore, except for other modifications, an insert operation adds two more leaves in the first layer ($[x_{il}, y_{ib}]$, $[x_{ir}, y_{ib}]$) and one more ($[y_{ib}, y_{it}]$) in the priority trie of the second layer. Because each update operation affects $O(\sqrt{\log M})$ priority tries in the second layer and the time needed for update in q-fast trie is $O(\sqrt{\log M})$, the amount of time spent for update in our structure is $O(\log M)$. However, after an insertion in our 2-fold structure two more sets of R^j_δ and L^j_δ in new leaves δ of the first structure must be created, as well as L_λ, PT_λ in new leaf λ of the second structure. The construction of the $\sqrt{\log M}$ lists R_δ and L_δ of $O(\sqrt{\log M})$ elements each consumes $O(\log M)$ time totally for each added leaf. Besides, the construction of a list L_λ, having length $O(\sqrt{\log M})$, and the priority tree PT_λ, storing $\sqrt{\log M}$ points of the path leading to new added leaf λ of the second layer, costs $O(\sqrt{\log M})$ totally. The deletion does not add any further complexity in update execution, if we assume that the release of space costs constant time.

From the discussion above, it is obvious that no matter what modifications are made in the structure the overall time complexity remains $O(\log M)$ (analogous to the retrieval time of our two-dimensional structure on a grid). Consequently, an update in a d-dimensional structure on a grid costs $O(\log^{2d-2} M)$ time.

6. CONCLUSION

Using the results presented in Section 1 with the results presented here we have solutions for each individual case

TABLE 1. The results of this paper

Answer of cross intersection query		*Time complexity*	*Space complexity*	*Update of dynamized structure*
Arbitrary (unrestricted) universe		$O(\log^{2d-3} n \log\log n + K)$	$O(n \log^{2d-3} n)$	$O(\log^{2d-2} n)$
Restricted universe (grid)	Case A[a]	$O(\log^{d-1} M + K)$	$O(n\sqrt{\log M}^{2d-3}$	$O(\log^{d-1} M)$
	Case B[b]	$O(\log^{2d-3} M \log\log M + K)$	$O(n \log^{2d-3} M)$	—

[a] Combination of augmented q-fast trie with priority search trie (based on q-fast trie).
[b] Combination of augmented y-fast trie with priority search trie (based on y-fast trie).

of the rectangle intersection problem. More precisely, we can answer on-line the cross intersection case in $O(\log^{2d-3} n \log\log n + K)$ time, while the enclosure query is answered (Bistiolas *et al.*, 1993) in $O(\log^{2d-1} n + K)$ time and the range query (Willard, 1985) in $O(\log^{d} n + K)$ time. These results give an overall and complete consideration of the various instances of the problem.

The solution approach on grid can give some hints of the way that this and similar problems may be handled, taking advantage of already proposed structures and the particular cases that occur in restricted universe. Table 1 summarizes the results for the cross intersection on-line queries previously proved.

ACKNOWLEDGEMENTS

This work was partially supported by ESPRIT Basic Research Action Program (ESPRIT), p 4171 (ALCOM II).

REFERENCES

Bistiolas, V., Sofotassios, D. and Tsakalidis A. (1993) Computing Rectangle Enclosures. *Comp. Geometry J.: Theory and Applications*, **2,** 303–308.

Burton, W. (1977) Representation of many-sided polygons and polygonal lines for rapid processing. *Commun. ACM*, **20,** 166–171.

Edelsbrunner, H. and Maurer, H. A. (1981) On the intersection of Orthogonal objects. *Inform. Process. Lett.*, **13,** 74–79.

Fredman, M. L., Komolos, J. and Sremeredi E. (1982) Storing a space table with O(1) worst case access times. In *Proc. 23rd IEEE Symp. on Foundations of Computer science*, 165–169.

Fredkin, E. (1962) Trie memory. *Commun. ACM*, **3,** 420–429.

Fries, O., Mehlhorn, K., Naeher, S. and Tsakalidis, A. (1987) A loglogn data structure for three-sided range queries. *Inform. Process. Lett.*, **25,** 269–273.

Jacobs, T. M. and van Emde Boas, P. (1986) Two results on tables. *Inform. Process. Lett.*, **22,** 43–48.

Knuth, D. E. (1976) Big omicron, big omega and big theta. *SIGACT News*, **8,** 18–24.

Lee, D. T. and Wong, C. K. (1981) Finding intersection of rectangles by range search. *J. Algorithms*, **2,** 337–347.

McCreight, E. M. (1985) Priority search Trees. *SIAM J. Comput.*, **14,** 257–276.

Mehlhorn, K. (1984). Data structures and algorithms I: sorting and searching. *EATCS Monographs on Theoretical Computer Science*, Springer, Berlin.

Overmars, M. H. (1988) Efficient data structures for range searching on a grid. *J. Algorithms*, **9,** 274–295.

Willard, D. E. (1983) Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inform. Process. Lett.*, **17,** 81–84.

Willard, D. E. (1984) New trie data structures which support very fast search operations. *J. Comp. Syst. Sci.*, **28,** 379–394.

Willard, D. E. (1985) New data structures for orthogonal range queries. *SIAM J. Comput.*, **14,** 232–253.

Willard, D.E. and Luecker, G.S. (1985) Adding range restriction capability to dynamic data structures. *J. Association for Computing Machinery*, **32,** 597–617.