INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING, VOL. 31, 1–17 (1991)

AN ALTERNATING DIGITAL TREE (ADT) ALGORITHM FOR 3D GEOMETRIC SEARCHING AND INTERSECTION PROBLEMS

JAVIER BONET

Institute for Numerical Methods in Engineering, University of Wales, Swansea SA2 8PP, U.K.

JAIME PERAIRE

Department of Aeronautics, Imperial College of Science, Technology and Medicine, London SW7 2BY, U.K.

SUMMARY

A searching algorithm is presented for determining which members of a set of n points in an N dimensional space lie inside a prescribed space subregion. The algorithm is then extended to handle finite size objects as well as points. In this form it is capable of solving problems such as that of finding the objects from a given set which intersect with a prescribed object. The suitability of the algorithm is demonstrated for the problem of three dimensional unstructured mesh generation using the advancing front method.

INTRODUCTION

The problem of determining the members of a set of n points which lie inside a prescribed subregion of an N dimensional space is known as *geometric searching*. Devising algorithms for efficient solution of this problem has been the aim of much research in areas such as computational physics or geometric computer representation. Several algorithms have been proposed[1–4] which solve the above or equivalent problems with a computational expense proportional to $\log(n)$. The problem complexity increases considerably when, instead of considering points, one deals with finite size objects such as line segments, geometrical CAD components or computational cells. A common problem encountered here, namely *geometric intersection*, consists of finding the objects which overlap a certain subregion of the space being considered. To our knowledge, efficient algorithms for solving this problem exist only in two dimensions,[5] and have been applied in determining the intersection between geometrical objects in the plane. These algorithms cannot be applied to problems involving more than two dimensions.

Despite the similarity between the problems of geometric searching and geometric intersection, in the published literature they have received completely different treatments. It is the intention of this paper to present a unified approach to the solution of both problems.

In what follows, we shall describe an algorithm and associated data structure, called the *alternating digital tree* (ADT), which allows for the efficient solution of the geometric searching problem. It naturally offers the possibility of inserting and removing points and optimally searching for the points contained inside a given region. It is applicable to any number of dimensions, and is a natural extension of the so-called *digital tree search* technique which is exhaustively treated by Knuth[6] for one dimensional problems. A procedure which allows

0029–5981/91/010001–17$08.50

Received 25 October 1989
Revised 9 April 1990

treatment of any geometrical object in an N dimensional space as a point in a $2N$ dimensional space will be introduced, thereby allowing the proposed technique to be employed for the solution of geometric intersection problems.

Emphasis has been made on the FORTRAN 77 implementation, not because we consider it to be the most suitable language for this type of application, but because most scientific programming is still done using this language. The implementation is simpler in languages like Pascal or C which feature recursion and dynamic allocation.

The particular application which has motivated the development of this work is the automatic generation of tetrahedral finite element meshes by the advancing front technique.[7-9] The application of the proposed algorithm to this problem will be discussed in the last section of this paper.

BINARY TREE STRUCTURES

Binary trees are one of the most important non-sequential types of data structures. They provide the basis for several searching algorithms, including the one to be presented here. It is therefore necessary to introduce some basic concepts and terminology related to binary tree structures. More detailed expositions can be found in References 5 and 10.

Definition and terminology

Originally, tree structures were conceived as a systematic way of storing a collection of data items which would enable not only a quick access to the information stored, but also frequent insertions of new items as well as deletion of unwanted items. This degree of flexibility requires the storage of data items in non-sequential locations of the computer memory. As Figure 1(a) illustrates, to achieve this, each data item is extended by the addition of two integer values, known as the *left* and *right links*, and stored in what is known as *a node* of the tree. Each added link can either be equal to zero or equal to the position in memory where another node of the tree can be found. Hence, from one node of the tree it is possible to reach at most two other nodes. Moreover, in order to ensure that every node can be reached, these links must be such that for each node except one, known as the *root*, there is one and only one link pointing at it.

This definition establishes a hierarchy of nodes: the root at the top level of the hierarchy points at 0, 1 or 2 nodes at the next level; each of these in turn points at other 0, 1 or 2 nodes at the next level of the hierarchy; and so forth. This hierarchical structure inspires the graphical representation shown in Figure 1(b) for a simple tree comprising only eight nodes $\{A, B, C, D, E, F, G, H\}$.

Genealogical terms are normally used to describe the relative position of nodes in a tree: when a node points at a second node, the former is called the *father* of the latter, and this the *son* of the former node. A node without sons, that is, with both links blank, is called a *terminal node*, and the only node without a father is the root (node A in Figure 1(b)). Given a node, the set of nodes formed by itself together with all its descendants constitutes a *subtree* of the main tree. For instance, in Figure 1(b) the trees $\{C, D, E, F, G, H\}$ and $\{E, G, H\}$ are subtrees of the main tree rooted at C and E respectively.

Tree traversal

To retrieve information stored in a given node requires knowledge of its location in memory, which is kept by its father. Hence, a node in the tree can only be examined or *visited* if all its ancestors are visited first. However, it is possible to systematically examine each node in such a way that every node is visited exactly once. Such an operation is known as *traversing the tree* and provides the basis for the searching methods discussed below. Although several algorithms can be found in the literature to traverse a binary tree,[10] attention will be centred here on the so-called *preorder traversal* method. This technique is embodied in the following three steps:

1. *Visit the root of the current subtree.*
2. *If the left link of the root is not zero then traverse the left subtree.*
3. *If the right link of the root is not zero then traverse the right subtree.*

The procedure determined by these three steps is clearly recursive, that is, steps 2 and 3 invoke again the algorithm which they define. In order to illustrate this process, consider again the tree shown in Figure 1(b); for this tree, the repeated application of the above algorithm yields the following sequence:

```
1. Traverse the tree {A, B, C, D, E, F, G, H}
    1.1. Visit A
    1.2. Traverse the tree {B}
        1.2.1. Visit B
        1.2.2. Skip
        1.2.3. Skip
    1.3. Traverse the tree {C, D, E, F, G, H}
        1.3.1. Visit C
        1.3.2. Traverse the tree {D, F}
            1.3.2.1. Visit D
            1.3.2.2. Traverse the tree {F}
                1.3.2.2.1 Visit F
                1.3.2.2.2 Skip
                1.3.2.2.3 Skip
            1.3.2.3 Skip
        1.3.3. Traverse the tree {E, G, H}
            1.3.3.1. Visit E
            1.3.3.2. Traverse the tree {G}
                1.3.3.2.1 Visit G
                1.3.3.2.2 Skip
                1.3.3.2.3 Skip
            1.3.3.3. Traverse the tree {H}
                1.3.3.3.1 Visit H
                1.3.3.3.2 Skip
                1.3.3.3.3 Skip
```

Thus, the nodes of the tree in Figure 1(b) in preorder are A, B, C, D, F, E, G and H.

This traversing algorithm can easily be implemented if a programming language which allows for recursive routines is used. Unfortunately, this capability is not provided by the standard FORTRAN 77 language, which makes the implementation substantially more complicated. In particular, before moving on to traverse the left subtree—step 2 in the previous algorithm—it is necessary to store the value of the right link, that is, the address of the right son, in order to enable the subsequent traversal of the right subtree. Moreover, whilst traversing the left subtree it is likely that additional right links will have to be stored. In fact, a list containing the addresses of all right subtrees encountered along the way which are yet to be traversed must be kept and has to be continuously updated as follows. After visiting each node, the right link, if different from zero, is

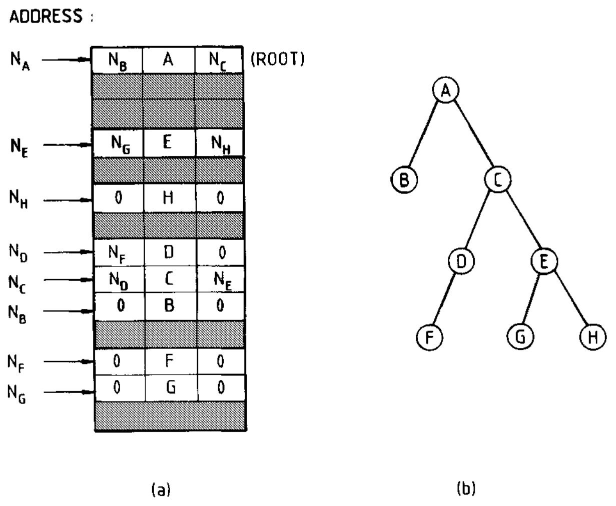


Figure 1. A simple binary tree and its storage in computer memory

added to the list and if the left link is not zero the left subtree is traversed. When a zero left link is encountered, the last right link inserted in the list is retrieved , as well as removed, from the list and the subtree rooted at this address is traversed.

This type of list, in which items are inserted one by one and extracted, also one at a time, in the reverse order, is known as a *stack*.[10] A stack can be implemented in FORTRAN 77 by means of a linear array, or vector, together with an integer variable to record the number of items in the array. This variable, being initially zero, is increased by one every time an item is added to the stack and decreased by one when an item is extracted from it.

With the help of a stack, any recursive algorithm can be implemented without the need to use recursive routines. For instance, a non-recursive implementation of the traversal algorithm given above can be symbolically expressed as:

```
0.a Set root_address = address of the root node
0.b Set stack_size = 0
    1.  Visit the node stored at root_address
    2.  If right_link ≠ 0 then:
          Set stack_size = stack_size +1
          Set stack(stack_size) = right_link
        endif
    3a. If left_link ≠ 0 then:
          Set root_address = left_link
          go to 1
        endif
    3b. If left_link = 0 then:
          If stack_size ≠ 0 then:
            Set root_address = stack(stack_size)
            Set stack_size = stack_size - 1
             go to 1
           endif
          endif
        If stack_size = 0    terminate the process
```

Inserting and deleting nodes

In order to add a new data item to a binary tree, a node containing the new item of information must be created and stored in a convenient memory location. The left and right links of this node are set to zero. If the current tree is empty, the new node becomes the root of the tree, otherwise the node must be inserted or linked to the existing tree. To achieve this, the tree is followed downwards, starting from the root and jumping from father to son, until a blank link is found. This link is then set to the memory position of the new node. When moving down the tree, a criterion must be provided at each node to choose between the left or right branches. This criterion determines the final position in the tree of the new node and, consequently, the shape of the tree itself. Different criteria—each defining a particular tree structure—are available in the literature to suit a wide variety of applications (see for example Reference 6 for one dimensional search applications and Reference 11 for N dimensional search applications). The specific criterion employed to construct the alternating digital tree is described in the next section.

Deleting a node from a binary tree is a straightforward operation if the undesired node is a terminal node; changing to zero the corresponding link of its father effectively 'prunes' the node from the tree and renders the memory occupied by it available for future uses. In the case of an intermediate node, the process becomes slightly more complicated since a gap can not be left in the tree. To overcome this problem, the unwanted node is replaced by a terminal node chosen from among its descendants. This operation can be carried out by modifying the links to suit the new structure of the tree and without moving the nodes from their memory positions. Figures 2(a) and 2(b) illustrate the deletion of node C from the tree shown in Figures 1(a) and 1(b) and its replacement by node H. Some types of tree structures impose certain restrictions as to which terminal node can be chosen to replace an obsolete intermediate node (see for instance Reference 11). However, the ADT structure will introduce no such limitations and terminal nodes are normally chosen with the tree being descended in a random manner until a terminal node is reached.

If the application at hand demands frequent deletion and insertion, a memory book-keeping system is necessary for the efficient implementation of tree structures. This is required so that new nodes can be placed in the memory space released by the deletion of previous nodes. With

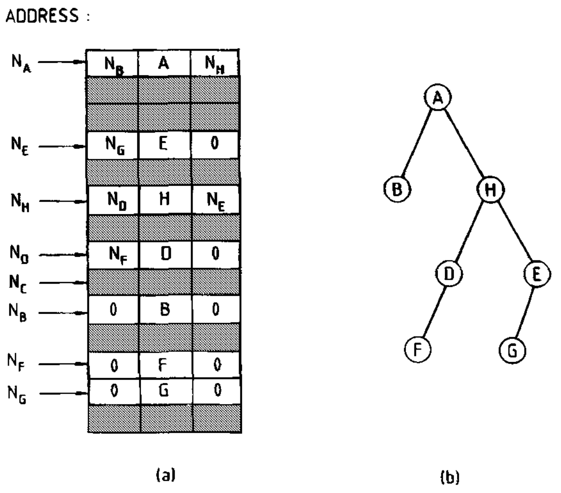


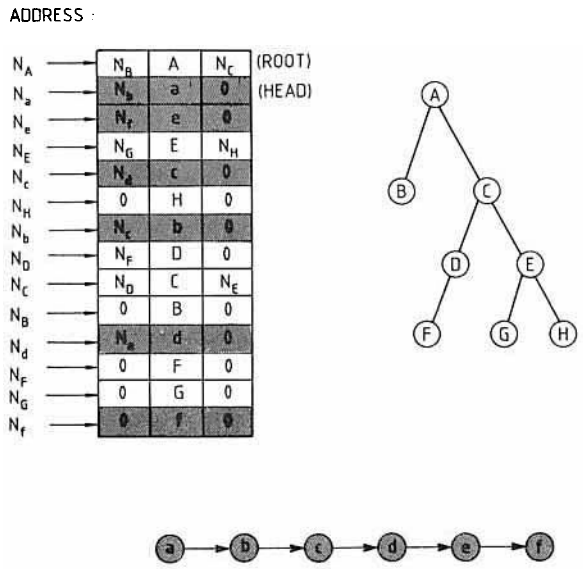
Figure 2. Deletion process

programming languages featuring dynamic allocation, this is automatically provided. However, FORTRAN 77 does not offer this capability and in this case the problem can be solved by using a *linked list* structure to record all the available memory spaces. A linked list is a data structure that differs from the binary tree data structure described above in that every node has always only one link pointing at another node, and every node has always one link pointing at it. There are two exceptions, which are the *head* and the *end* nodes. The head is a node with no link pointing at it—the address of which needs to be kept separately—and the end is a node with a blank link.

As shown in Figure 3 the two data structures, binary tree and linked list, are updated simultaneously. Initially, the available memory is partitioned into cells of the correct size to store tree nodes. These cells, which contain no relevant information other than a single link, are then joined together to form a linked list. Every time a node needs to be inserted into the tree, the memory space required by this new tree node is generated by removing a node from the list (see Figure 3(b)). Similarly, when a node is deleted from the tree it is added to the list (see Figure 3(c)). Inserting and deleting nodes in the list always takes place at the head. To insert a node into the list, the link of the new node is set equal to the address of the head and the inserted node becomes the new head of the list. The deletion of the head node can be done by simply allowing its link to be the new head.

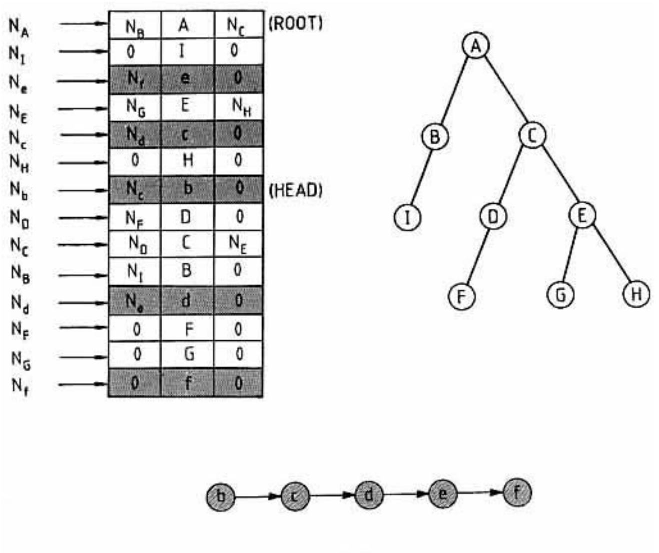
THE ALTERNATING DIGITAL TREE

Consider a set of n points in a N dimensional space (R^N) and assume for simplicity that the co-ordinate values of their position vectors $\{\mathbf{x}_1, \mathbf{x}_2 \ldots \mathbf{x}_n\}$, after adequate scaling, vary within the interval [0, 1). The aim of geometric searching algorithms is to select from this set those points

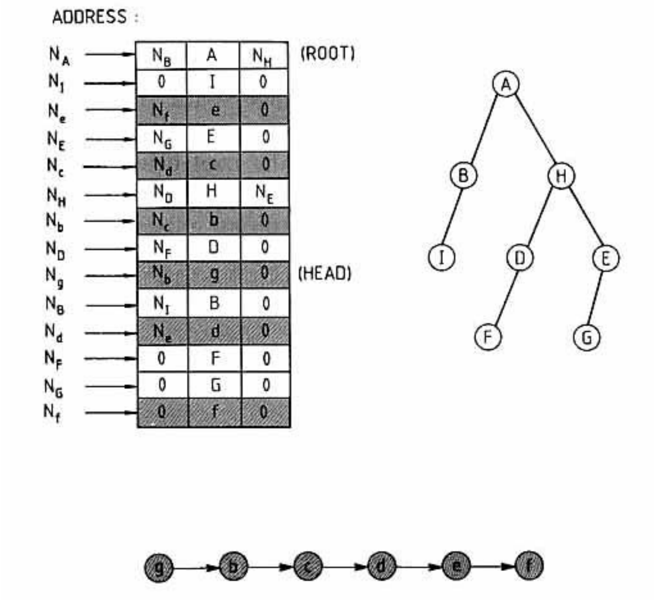


(a)

Figure 3(a)



(b)



(c)

Figure 3. Binary tree/linked list configurations: (a) tree shown in Figure 1; (b) after inserting node I using storage released by node a; and (c) after deleting node C

that lie inside a given subregion of the space. To facilitate their representation, only rectangular—or 'hyper-rectangular'—regions will be considered, thereby allowing their definition in terms of the scaled co-ordinates of the lower and upper vertices as (**a**, **b**).

Comparing the co-ordinates of each point k with the vertex co-ordinates of a given subregion to check whether the condition $a^i \leqslant x_k^i \leqslant b^i$ is satisfied for $i = 1, 2 \ldots N$ would render the cost of the searching operation proportional to the number of points n. This computational expense, however, can be substantially reduced by storing the points in a binary tree in such a way that the structure of the tree reflects the positions of the points in space. There exist several well-known algorithms that will accomplish this effect for one dimensional problems; the most popular are the binary search tree and digital tree methods.[5,6] Binary search trees have been extended to N dimensional problems,[11] but the resulting tree structures, known as *N-d trees*, do not allow the efficient deletion of nodes. The algorithm presented here, referred to as the alternating digital tree method, is the natural extension of the 1-d digital tree algorithm and overcomes the difficulties encountered in N-d trees.

Definition and node insertion

Broadly speaking, an alternating digital tree can be defined as a binary tree in which a set of n points is stored following certain geometrical criteria. These criteria are based on the similarities arising between the hierarchical and parental structure of a binary tree and a recursive bisection process: each node in the tree has two sons, likewise a bisection process divides a given region into two smaller subregions. Consequently, it is possible to establish an association between tree nodes and subregions of the unit hypercube as follows: the root represents the unit hypercube itself; this region is now bisected across the x^1 axis and the region for which $0 \leqslant x^1 < 0{\cdot}5$ is assigned to the left son and the region for which $0{\cdot}5 \leqslant x^1 < 1$ is assigned to the right son; at each of these nodes the process is repeated across the x^2 direction, as shown in Figure 4. In a two dimensional space this process can be repeated indefinitely by choosing x^1 and x^2 directions in alternating order; similarly, in a general N dimensional space, the process can be continued by choosing directions $x^1, x^2, \ldots, x^N$ in cyclic order.

Generally, if a node k at the hierarchy level l—the root being level 0—represents a region $(\mathbf{c}_k, \mathbf{d}_k)$, the subregions associated with its left and right sons, $(\mathbf{c}_{kl}, \mathbf{d}_{kl})$ and $(\mathbf{c}_{kr}, \mathbf{d}_{kr})$, result from the bisection of $(\mathbf{c}_k, \mathbf{d}_k)$ by a plane normal to the jth co-ordinate axis, where j is chosen cyclically from the N space directions as

$$j = 1 + \operatorname{mod}(l, N) \tag{1}$$

and $\operatorname{mod}(l, N)$ denotes the remainder of the quotient of l over N. Hence $(\mathbf{c}_{kl}, \mathbf{d}_{kl})$ and $(\mathbf{c}_{kr}, \mathbf{d}_{kr})$ are obtained as

$$c^i_{kl} = c^i_k, \quad d^i_{kl} = d^i_k \quad \text{for} \quad i \neq j \quad \text{and} \quad c^j_{kl} = c^j_k, \quad d^j_{kl} = \tfrac{1}{2}(c^j_k + d^j_k) \tag{2a}$$

$$c^i_{kr} = c^i_k, \quad d^i_{kr} = d^i_k \quad \text{for} \quad i \neq j \quad \text{and} \quad c^j_{kl} = \tfrac{1}{2}(c^j_k + d^j_k), \quad d^j_{kl} = d^j_k \tag{2b}$$

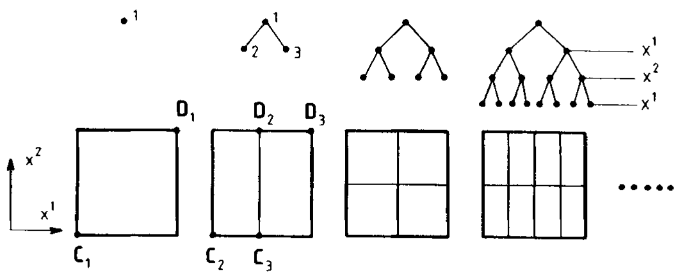


Figure 4. Relation between a binary tree and a bisection process

This correlation between nodes and subdivisions of the unit hypercube allows an ADT to be further defined by imposing that each point in the tree should lie inside the region corresponding to the node where it is stored. Consequently, if node k of an ADT structure contains a point with co-ordinates $\mathbf{x}_k$, the following condition must be satisfied:

$$c_k^i \leqslant x_k^i < d_k^i \quad \text{for} \quad i = 1, 2, \ldots, N \tag{3}$$

Owing to this additional requirement there exists only one possible way in which a new point can be inserted in the tree. As discussed in the previous section, the tree is followed downwards until an unfilled position where the node can be placed is found. During this process, however, left or right branches are now chosen according to whether the new point lies inside the region related to the left or right sons, thereby ensuring that condition (3) is satisfied.

Given a predetermined set of n points, an ADT structure can be built by placing any one point at the root and then inserting the remaining points in consecutive order according to the algorithm described above. This is illustrated in Figure 5 for a set of 5 points $\{A, B, C, D, E\}$. The

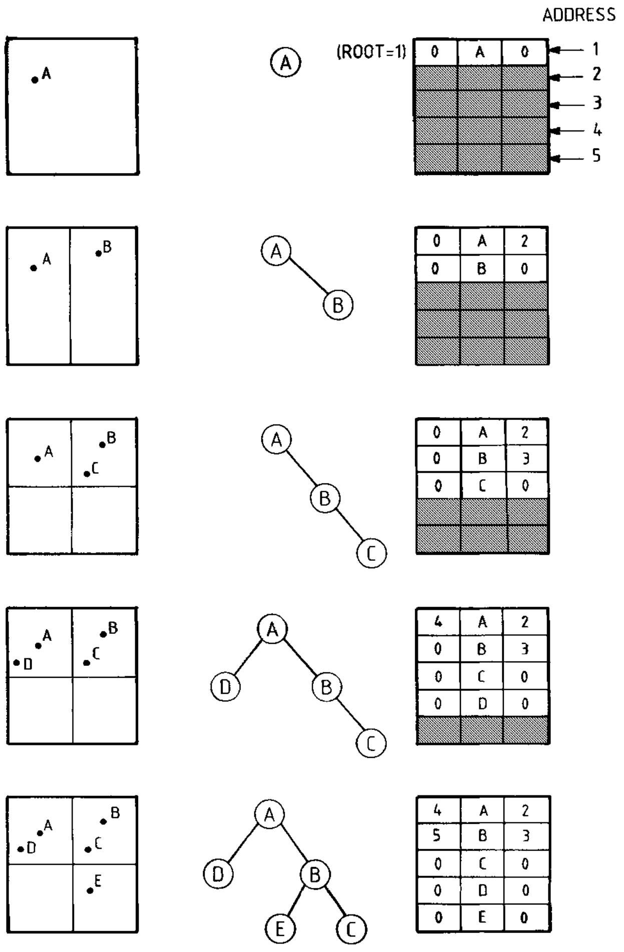


Figure 5. Building an ADT by successive insertion

shape of the tree obtained in this way depends mainly on the spatial distribution of the points and somewhat on the order in which the points were inserted. The cost of operations like node insertion/deletion and geometric searching depends strongly on the shape of the tree; generally, poor performances are to be expected from highly degenerated trees (see Figure 6), whereas well balanced trees (see Figure 7), as those obtained for fairly uniform distributions of points, will result in substantial reductions of the searching cost. In these cases the average number of levels in the tree, and therefore the average cost of inserting a new point, becomes proportional to $\log(n)$, clearly a considerable cost if compared with the cost of storing the points in a sequential list, but fully justifiable in view of the reduction in searching costs that ADT structures will provide.

Geometric searching

Consider now a set of points stored in an ADT structure. The fact that condition (3) is satisfied by every point provides the key to the efficient solution of a geometric searching problem. To illustrate this, note first that the recursive structure of the bisection process described above implies that the region related to a given node k contains all the subregions related to nodes descending from k; consequently, all points stored in these nodes must also lie inside the region represented by node k. For instance, all points in the ADT structure are stored in nodes descended from the root and, clearly, all of them lie inside the unit hypercube—the region associated with the root. Analogously, the complete set of points stored in any subtree is inside the region represented by the root of the subtree.

This feature can be effectively used to reduce the cost of a geometric searching process by checking, at any node k, the intersection between the searching range $(\mathbf{a}, \mathbf{b})$ and the region represented by node k, namely $(\mathbf{c}_k, \mathbf{d}_k)$. If these two regions fail to overlap, then the complete set of

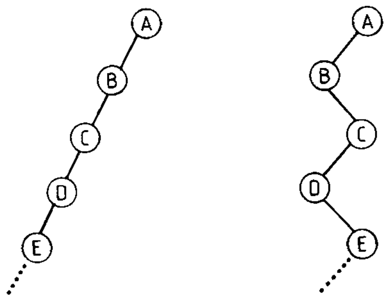


Figure 6. Degenerated trees

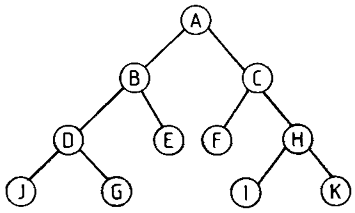


Figure 7. Well balanced tree

points stored in the subtree rooted at k can be disregarded from the search, thus avoiding the need to examine the co-ordinates of every single point.

Consequently, a systematic procedure to select the points that lie inside a given searching range $(\mathbf{a}, \mathbf{b})$ can be derived from the traversal algorithm previously presented. Now the generic operation 'visit the root' can be re-interpreted as checking whether the point stored in the root falls inside the searching range. Additionally, the left and right subtrees need to be traversed only if the regions associated with their respective root nodes intersect with the range. Accordingly, a geometric searching algorithm emerges in a recursive form as:

1. Check whether the co-ordinates of the node stored in the root, say $\mathbf{x}_k$, are inside $(\mathbf{a}, \mathbf{b})$, i.e. check whether $a^i \leqslant x_k^i < b^i$ for $i = 1, 2, \ldots, N$.
2. If the left link of the root is not zero and the region $(\mathbf{c}_{kl}, \mathbf{d}_{kl})$ overlaps with $(\mathbf{a}, \mathbf{b})$, i.e. if $d_{kl}^i \geqslant a^i$ and $c_{kl}^i \leqslant b^i$ for $i = 1, 2, \ldots, N$, search the left subtree.*
3. If the right link of the root is not zero and the region $(\mathbf{c}_{kr}, \mathbf{d}_{kr})$ overlaps with $(\mathbf{a}, \mathbf{b})$, i.e. if $d_{kr}^i \geqslant a^i$ and $c_{kr}^i \leqslant b^i$ for $i = 1, 2, \ldots, N$, search the right subtree.*

In order to illustrate this process, consider the set of points and the searching range shown in Figure 8(a) and the corresponding alternating digital tree depicted in Figure 8(b). For this simple example, the algorithm given above results in the following sequence of steps:

```
Search the tree {A, B, C, D, E, F, G, H}:
    1. Check if $a^i \leqslant x_A^i \leqslant b^i$ for $i = 1, 2$
    2. Since $d_B^i \geqslant a^i$ and $c_B^i \leqslant b^i$ search the tree {B, C, D, E}:
        2.1. Check if $a^i \leqslant x_B^i \leqslant b^i$
        2.2. Since $d_C^i \geqslant a^i$ and $c_C^i \leqslant b^i$ search the tree {C, E}:
            2.2.1. Check if $a^i \leqslant x_C^i \leqslant b^i$
            2.2.2. Skip (left link is zero)
            2.2.3. Skip ($c_E^1 > b^1$)
        2.3. Skip ($c_D^2 > b^2$)
    3. Skip ($c_F^1 > b^1$)
```

Again a 'non-recursive' implementation of this algorithm can be achieved using a stack in a very similar way to that previously described for the traversal algorithm.

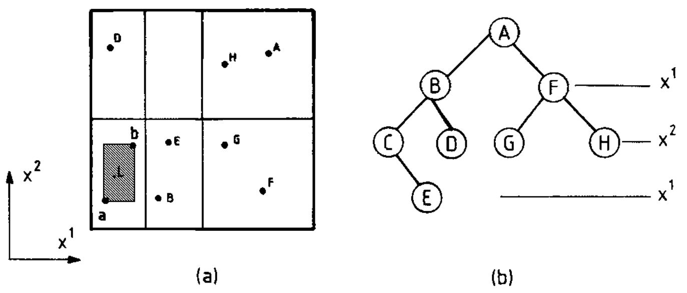


Figure 8. Searching problem in R^2

* In fact many of these conditions are redundant since from father to son only the jth component of $\mathbf{c}_k$ and $\mathbf{d}_k$ change where j is given in (1); hence it would suffice to compare only for $i = j$

Note that, with this technique, only the co-ordinates of points A, B and C are actually examined, the rest being immediately disregarded in view of their position in the tree. In general, only those points stored in nodes with associated regions overlapping (**a**, **b**) will be checked during the searching process.

GEOMETRIC INTERSECTION

Geometrical intersection problems can be found in many finite element applications; for instance, a common problem that may emerge in contact algorithms,[12] hidden line removal applications or in the advancing front mesh generation algorithm[7-9] is to determine from a set of three noded triangular elements those which intersect with a given line segment. Similar problems, involving other geometrical objects, are encountered in a wide range of geometrical applications. In general, a geometric intersection problem consists of finding from a set of geometrical objects those which intersect with a given object. If every one-to-one intersection is investigated, the solution of these problems can become very expensive, especially when complex objects such as curves or surfaces are involved. Fortunately, many of these one-to-one intersections can be quickly discarded by means of a simple comparison between the co-ordinate limits of every given pair of objects. For instance, a triangle with x-co-ordinate varying from 0·5 to 0·7 cannot intersect with a segment with x-co-ordinate ranging from 0·1 to 0·3. Generally, the intersection between two objects in the N dimensional Euclidean space requires each of the N pairs of co-ordinate ranges to overlap. Consider, for instance, the intersection problem between triangular facets and a target straight line segment in R^3; then, if $(\mathbf{x}_{k,\min}, \mathbf{x}_{k,\max})$ are the co-ordinate limits of element k and $(\mathbf{x}_{0,\min}, \mathbf{x}_{0,\max})$ are the lower and upper limits of the target segment (see Figure 9), an important step towards the solution of a geometric intersection problem is to select those which satisfy the inequality

$$\begin{matrix} x^i_{k,\min} \leqslant x^i_{0,\max} \\ x^i_{k,\max} \geqslant x^i_{0,\min} \end{matrix} \quad \text{for } i = 1, 2, \ldots, N \tag{4}$$

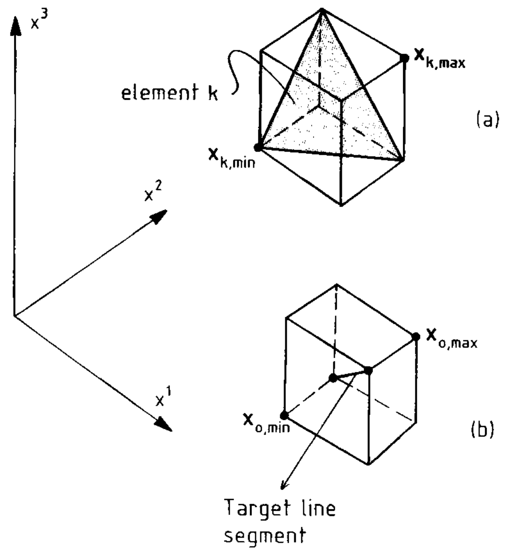


Figure 9. Definition of co-ordinate limits for triangular elements and straight line segments

The cost of checking condition (4) for every element grows proportionally to n, and for very numerous sets may become prohibitive. This cost, however, can be substantially reduced by using a simple device whereby the process of selecting those elements which satisfy condition (4) can be interpreted as a geometric searching problem. Additionally, since the number of elements that satisfy condition (4) will normally be much smaller than n, the cost of determining which of these intersects with the target segment becomes affordable.

In order to interpret condition (4) as a geometric searching problem, it is first convenient to assume that all the elements to be considered lie inside a unit hypercube—a requirement that can be easily satisfied through adequate scaling of the co-ordinate values. Consequently, condition (4) can be re-written as

$$\begin{array}{c} 0 \leqslant x^1_{k,\min} \leqslant x^1_{0,\max} \\ \vdots \\ 0 \leqslant x^N_{k,\min} \leqslant x^N_{0,\max} \\ x^1_{0,\min} \leqslant x^1_{k,\max} \leqslant 1 \\ \vdots \\ x^N_{0,\min} \leqslant x^N_{k,\max} \leqslant 1 \end{array} \tag{5}$$

Consider now a given object k in R^N with co-ordinate limits $\mathbf{x}_{k,\min}$ and $\mathbf{x}_{k,\max}$; combining these two sets of co-ordinate values, it is possible to view an object k in R^N as a point in R^{2N} with co-ordinates $\mathbf{x}^i_k$ for $i = 1, 2, \ldots, 2N$ defined as (see Figure 10)

$$\mathbf{x}_k = [x^1_{k,\min}, \ldots x^N_{k,\min}, x^1_{k,\max}, \ldots x^N_{k,\max}]^{\mathrm{T}} \tag{6}$$

Using this representation of a given object k, condition (5) becomes simply

$$a^i \leqslant x^i_k \leqslant b^i \quad \text{for} \quad i = 1, 2, \ldots, 2N \tag{7}$$

where $\mathbf{a}$ and $\mathbf{b}$ can be interpreted as the lower and upper vertices of a 'hyper-rectangular' region in R^{2N} and, recalling (5), their components can be obtained in terms of the co-ordinate limits of the target object (see Figure 11) as

$$\mathbf{a} = [0, \ldots, 0, x^1_{0,\max}, \ldots, x^N_{0,\max}]^{\mathrm{T}} \tag{8a}$$

$$\mathbf{b} = [x^1_{0,\min}, \ldots, x^N_{0,\min}, 1, \ldots, 1]^{\mathrm{T}} \tag{8b}$$

Consequently, the problem of finding which objects in R^N satisfy condition (4) becomes equivalent to a geometric searching problem in R^{2N}, i.e. obtaining the points $\mathbf{x}_k$ which lie inside the region

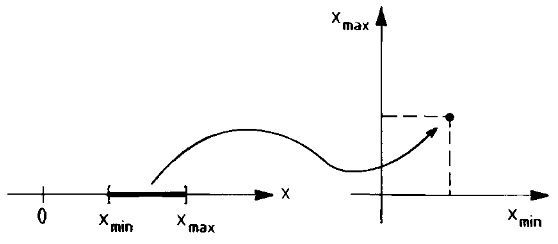


Figure 10. Representation of a region in R^1 as a point in R^2

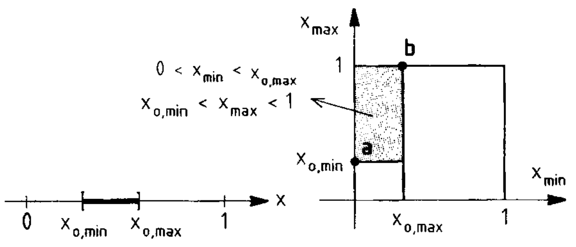


Figure 11. Intersection problem in R^1 as a searching problem in R^2

limited by **a** and **b**. Once this subgroup of elements has been selected, the intersection of each one of them with the target object must be checked to complete the solution of the geometric intersection problem.

APPLICATIONS

We describe in this section the application of the ADT data structure described above to the problem of mesh generation using the *advancing front technique*. The mesh generation procedure itself has been extensively documented[7–9] and only a general description is given here.

The basic underlying concept in the advancing front technique is illustrated in Figure 12 for the generation of a uniform size triangular mesh over a two dimensional domain. The boundary curves of the domain to be meshed are discretized first. Points are placed on the boundary curves in such a way that the distance between them is as close as possible to the desired mesh spacing. Contiguous points on the boundary curves are joined by straight line segments and assembled to form the initial *generation front*. At this stage the triangulation loop begins. A *side* from the front is chosen and a triangle is generated that will have this selected side as one edge. In generating this new triangle an interior node may be created or an existing node in the front may be chosen. At this stage it is necessary to ensure that the element generated does not intersect with any existing side in the front. After generating the new element the front is conveniently updated in such a way that it always contains the sides which are available to form a new triangle. The generation is completed when no sides are left in the front.

The mesh generation strategy described above can be directly extended into three dimensions. The initial generation front consists of the assembly of the *triangular faces* which result from discretizing the boundary surfaces. The advancing front approach is used to discretize the domain into tetrahedral elements. Every time an element is generated, three new triangular faces and three new straight sided segments will be created (see Figure 13). To guarantee the consistency of the generated mesh it is necessary to ensure that, for every generated tetrahedron, the triangular faces created are not crossed by any of the existing sides and that the sides created do not cross any of the existing faces.

It is obvious from the application described that operations such as searching for the points inside a certain region of the space or determining intersections between geometrical objects—in this case sides and faces—will be performed very frequently. The complexity of the problem is increased by the fact that the set of faces forming the generation front changes continuously as new faces need to be inserted and deleted during the process. Clearly, for meshes consisting of a large number of tetrahedra the cost of performing these operations can be very large.

A successful implementation of the above mesh generator has been accomplished by making extensive use of the ADT data structure. Two trees are employed for the basic mesh generator;

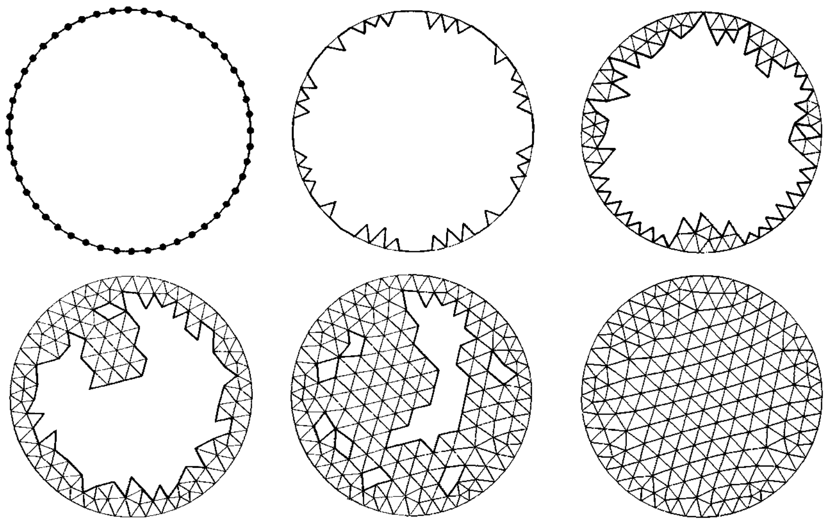


Figure 12. Advancing front technique in 2D

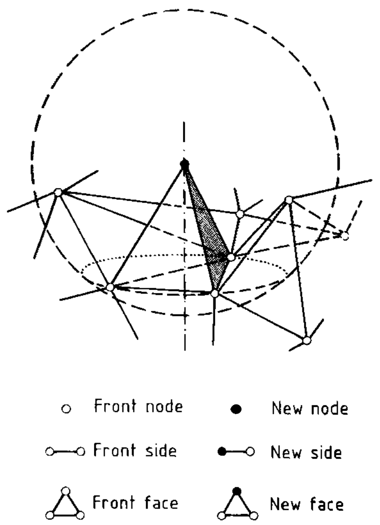


Figure 13. Generation of a tetrahedral element in 3D

one for the faces in the front and the other for the sides defined by the intersection between each pair of faces in the front (see Figure 13). This combination allows a high degree of flexibility and the operations of insertion, deletion, geometric searching and geometric intersection can be performed optimally. The overall computational performance of the algorithm is demonstrated by generating tetrahedral meshes, using the above method, for a unit cube (see Figure 14). Different numbers of elements have been obtained by varying the mesh size. In Figure 14 the

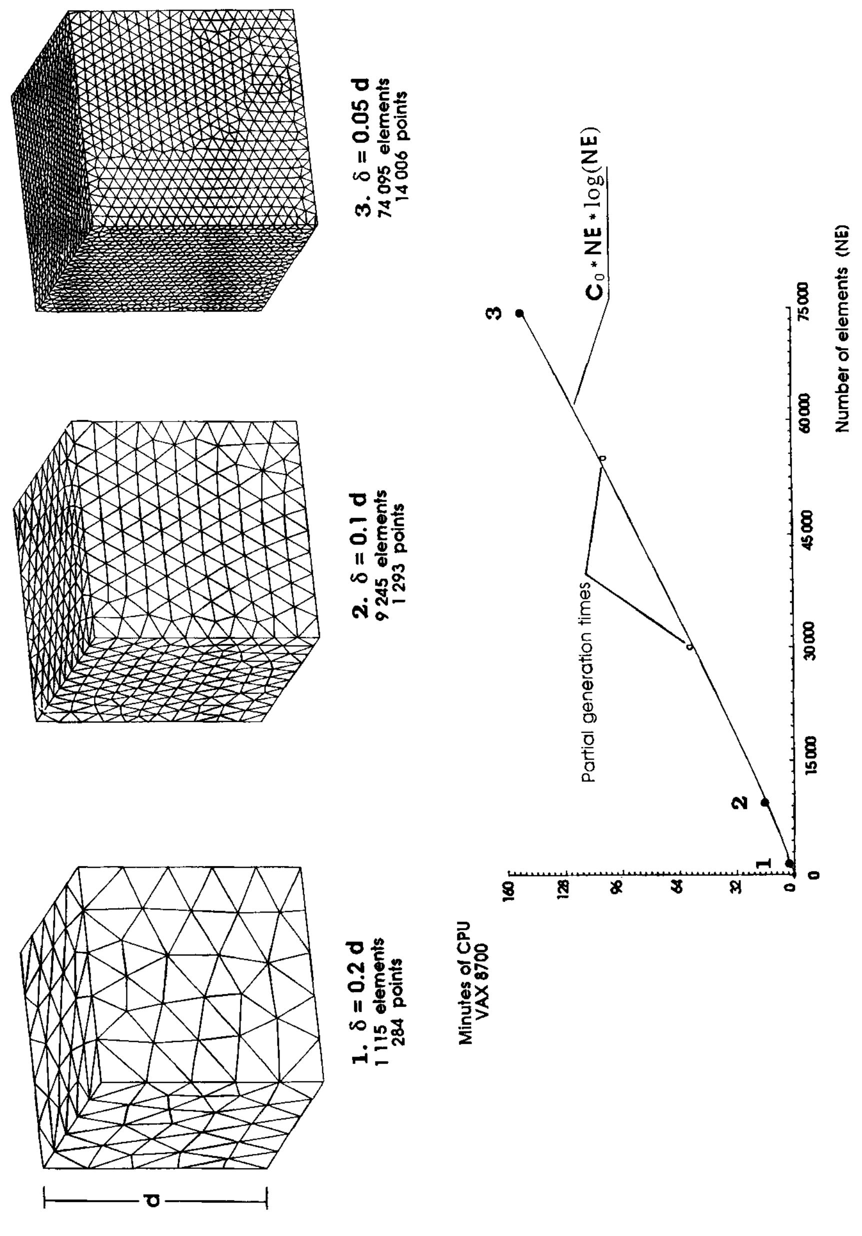


Figure 14. Mesh generation CPU times

computer time required on a VAX 8700 machine has been plotted against the number NE of elements generated. It can be observed that a typical NE*log(NE) behaviour is attained. Using this approach meshes containing up to one million elements have been generated[9] and no degradation in the performance has been detected.

CONCLUSIONS

The technique described can be applied to a wide variety of problems in multi-dimensions. When the distribution of points/elements to be treated is 'reasonably' uniform, the necessary cost to solve the geometric searching and intersection problems is found to be proportional to $n^*\log(n)$, where n is the number of items. To our knowledge this is the first time that the geometric intersection problem has been solved in more than two dimensions with this level of efficiency.

When compared to the linear array, the ADT data structure requires only two extra storage locations per item, i.e. left and right links, and provides a much greater degree of flexibility.

Unfortunately, the procedures described rely heavily on indirect addressing and, for this reason, it is virtually impossible to vectorize them to any significant degree. However, we think they offer interesting possibilities for parallel computers as work on disjoint subtrees can, in principle, be carried out independently.

ACKNOWLEDGEMENTS

The authors wish to thank K. Morgan, J. Peiró, N. P. Weatherill and R. D. Wood from the Institute for Numerical Methods in Engineering, University College Swansea, U.K. for the many valuable discussions and suggestions.

REFERENCES

1. J. L. Bentley and J. H. Friedman, 'Data structures for range searching', *Comp. Surveys*, **11,** 4 (1979).
2. M. I. Shamos and D. Hoey, 'Geometric intersection problems', *17th Annual Symposium on Foundations of Computer Science*, IEEE, New York, 1976.
3. *Fundamental Algorithms for Computer Graphics*, R. A. Earnshaw (ed.), *NATO ASI Series F, Vol. 17*, Springer-Verlag, Berlin, 1985.
4. J. Boris, 'A vectorised algorithm for determining the nearest neighbors', *J. Comp. Phys.*, **66,** 1–20 (1986).
5. Sedgewick, *Algorithms*, 2nd edn, Addison-Wesley, Reading, MA, 1988.
6. D. Knuth, *The Art of Computer Programming—Sorting and Searching, Vol. 3*, Addison-Wesley, Reading, MA, 1973.
7. J. Peraire, M. Vahdati, K. Morgan and O. C. Zienkiewicz, 'Adaptive remeshing for compressible flow computations', *J. Comp. Phys.*, **72,** 449–466 (1987).
8. J. Peraire, J. Peiro, L. Formaggia, K. Morgan and O. C. Zienkiewicz, 'Finite element Euler computations in three dimensions', *Int. j. numer. methods eng.*, **26,** 2135–2159 (1988).
9. J. Peraire, K. Morgan and J. Peiro, 'Unstructured finite element mesh generation and adaptive procedures for CFD', *Proc. AGARD FDP: Specialist's Meeting*, Loen, Norway, 1989.
10. D. Knuth, *The Art of Computer Programming—Fundamental Algorithms, Vol. 1*, Addison-Wesley, Reading, MA, 1969.
11. J. L. Bentley, 'Multidimensional binary search trees used for associative searching', *Commun. ACM*, **18,** 1 (1975).
12. J. Bonet, 'Finite element analysis of thin sheet superplastic forming processes', University of Wales, *Ph.D. Thesis*, C/PhD/128/89, 1989.