The Goblin Quadtree

R. WILLIAMS
Computer Laboratory, University of Cambridge, Corn Exchange Street, Cambridge CB2 3QG, UK

The Goblin quadtree is a new and simple data structure for representing spatial information. It stores a single pointer for each block of four nodes and average values at non-terminal nodes, enabling efficient depth-first traversal to any given level. The pointers are easy to generate and use, as demonstrated by algorithms for building and displaying Goblin quadtrees. The features of this new representation make it particularly suitable for geographic data. The concept of using a dominant value as the average value is explored, and is shown to be advantageous for quadtree display and storage.

Received November 1986, revised March 1987

1. INTRODUCTION

In recent years there has been considerable interest in using quadtrees to represent or index spatial data. The basic approach with these techniques is to recursively subdivide the object space into four equal-size quadrants until the quadrants are sufficiently simple to be described, or until the desired level of resolution is reached, as illustrated in Figs. 1 and 2. Fig 3 shows the quadtree structure produced by the regular subdivision into four. The terminal nodes, or leaves, of the quadtree contain, or reference, object descriptions. The non-terminal nodes point to the four descendant nodes, and may additionally contain a so-called average-object description.

There are numerous ways to represent quadtrees. These range from fully pointered quadtrees, in which each pointer from father to son is stored explicitly, to pointerless quadtrees, in which no pointer is stored. The pointerless quadtrees can be split into two categories: treecodes and leafcodes. Treecodes, such as the one suggested by Oliver and Wiseman, represent spatial data in the form of a traversal of the quadtree nodes. Leafcodes, such as the Garganti scheme, represent spatial data as a list of locational codes for the leaf nodes only. In Ref. 5, Mark and Lauzon compare the merits and space efficiency of several quadtree data structures. The fully pointered quadtrees offer maximum flexibility because the quadtrees can be traversed in any order, but they require the most storage space. Pointerless quadtrees are the most compact, but have to be traversed in the order of their creation, which reduces the speed of some algorithms. In between these two extremes of the time/space trade-off come the semi-pointered representations, such as the sextree and the one-to-four quadtree.

This paper introduces a new semi-pointered quadtree representation, known as the Goblin quadtree. It is designed principally for storing geographic data, but is also applicable to other fields. It is a very simple data structure, requiring the addition of only a single pointer for each block of four nodes. Average values are stored at the non-terminal nodes. These pointers and average values enable detail below any given node to be skipped, thereby allowing fast depth-first traversal to any given level.

Figure 1. Quadtree data.

Figure 2. Recursive decomposition.

Figure 3. Quadtree structure.
2. REQUIREMENTS

Quadtrees are used in a variety of applications, with different applications having different requirements. The major requirements considered necessary for a quadtree data structure that represents geographic data are: (1) to be able to store large amounts of data; (2) to be able to skip any amount of detail; (3) to be compact. In fact these are very general requirements and no doubt are equally applicable to other fields.

The first requirement implies a system that will perform well when the quadtree is stored on disc. Geographic data occupy considerable storage space, so a system that requires the whole quadtree to be stored in memory would be unsuitable. A disc-based system will perform best when the I/O buffering is exploited to its fullest (i.e. when there are a minimal number of actual disc reads or writes). This implies that for maximum efficiency a quadtree traversal should correspond to a sequential scan of the disc file.

The second requirement is a consequence of the first. If a large amount of data is being stored, it would be costly to traverse the whole quadtree for every operation, so it is essential to be able to skip detail. For instance, when displaying a quadtree it is unnecessary to retrieve data smaller than the pixel resolution of the device (e.g. a quadtree depth of 9 for a 512 by 512 pixel display). Similarly, if only a small window on the whole quadtree is being processed, the remainder of the quadtree should be discarded as quickly as possible. Therefore, what is needed is a capability to skip detail below any given level and a capability to store average values in non-terminal nodes. This would allow the data to be viewed at varying resolutions, and allow the cost of an inquiry to be tailored to the accuracy required.

The final requirement is also related to the first. If the quadtree is to be stored on disc, then some quadtree operations will be I/O-bound, with their performance directly proportional to the number of disc reads and writes. Therefore, the smaller the disc file, the faster these operations will perform. Compactness is also desirable is disc space is at a premium. However, there should be a suitable balance between compactness and simplicity. Simple structures are easier to program, and simple programs are easier to maintain.

3. THE GOBLIN QUADTREE

Most of the existing quadtree representations cannot satisfactorily meet the requirements set out in the previous section. Pointerless quadtrees cannot skip detail, and fully pointed quadtrees are not compact because of the space needed to store the pointers. Note that the number of bits required for a pointer is generally greater than the number required for a leaf value. Semi-pointered quadtrees appear to be the most attractive of the methods, because they can be designed to skip detail without incurring the cost of a full set of pointers. However, the Autumnal quadtree recently described by Fabrini and Montani encodes a fully pointed quadtree with a 75% space saving. Their method will be discussed more fully in Section 5.

The Goblin quadtree is a semi-pointered quadtree. It consists of a root and a number of branches, as shown in Fig. 4. Each branch is stored as a record in a direct-access disc file. A branch contains five fields: the node values for the NW, NE, SW and SE quadrants and a pointer. The root contains a single node value, and is stored as a special record at the beginning of the file. The root contains the node value for the top-level quadrant encompassing the entire object space. In the degenerate case of a completely homogeneous object space, the root contains the object identifier and there are no branches.

There are a number of possible ways of encoding the node information, but for the purpose of this paper a node value will be assumed to be a signed integer. A positive value indicates a terminal node, with the value being either an attribute value or a pointer to a list of attribute values. A negative value indicates a non-terminal node, with the magnitude being some average value. A value of zero is not permitted.

If the node values are in a linear scale, then the average value could be a true average (i.e. the mean of the four descendant node values). For instance, with a black-and-white image the average values would represent grey-scale values and could be used for anti-aliasing. However, if the node values are pointers or discrete attribute values (e.g. land-use classes), the notion of average becomes meaningless. Ideally, in this case, the non-terminal nodes should hold information about all of their descendant values. Unfortunately such a scheme would be prohibitively expensive in storage space.

A more reasonable alternative is to store a single dominant value, which represents the most dominant descendant value (i.e. the one covering the greatest area). An estimate of such a dominant value for any given branch can be computed as a function of the four node values of that branch. The actual dominant value requires examination of all sub-detail, and although this information could be passed up when the quadtree is recursively built, a weighted comparison of the four node values was chosen to be the best method.
values was preferred for simplicity. A weight of three is awarded to terminal nodes and a weight of two is awarded to non-terminal nodes. The weights are summed for each node value and the node value with the largest score is chosen as the dominant value. Note that in the case of two or three non-terminal nodes of one value, versus a single terminal node of another value, this weighting will favour the non-terminal node value. In the case of a tie the choice is arbitrary, although the dominant value should be chosen randomly to avoid any locational bias. As an example, consider a branch that contains two terminal nodes — with values A and B, and two non-terminal nodes — with values B and C. The score would be A (3), B (5) and C (2), and B would be chosen as the dominant value, although is is actually possible that A covers the greater area. The dominance algorithm can be extended to allow different weights for different leaf values. This is particularly useful for cartographic applications, which require certain features to be given a prominence beyond their size. As will be shown in the following sections, dominant values are more than just an alternative to average values. They can be used to produce an improvement in the time taken to display a quadtree and in the space required to store a quadtree.

The order of the branches is depth-first. The first branch contains the first subdivision of the object space. The second branch contains the subdivision for the first non-homogeneous quadrant in the first branch (using the conventional NW, NE, SW, SE order). The third branch contains the subdivision for the first non-homogeneous quadrant in the second branch, or if there is none, then it contains the subdivision for the next non-homogeneous quadrant in the first branch, and so on. Therefore, a depth-first traversal can be accomplished by reading the branches sequentially and saving the partially processed branches on a stack.

The pointer field for each branch contains the number of the branch that would be processed next, if the detail for the current branch were to be skipped. When all four quadrants of a given branch are homogeneous, the pointer will point to the next branch (by definition). If skipping detail for a particular branch would complete the traversal, the value of the pointer is irrelevant, as it should never be used. In this paper, such a pointer is given a value one greater than the final branch number.

4. ALGORITHMS

Standard quadtree algorithms can easily be modified to use Goblin quadtrees, as demonstrated by the build-and-display procedures outlined below. The only extra effort is in maintaining a global variable next, which keeps track of the next branch number. The procedures GetRoot, PutRoot, GetBranch and PutBranch make the interface to the file system transparent. The programming language is Modula-2.

```
TYPE
  QUAD  = (NW, NE, SW, SE);
  BRANCH = RECORD
    value: ARRAY QUAD OP INTEGER;
    pointer: INTEGER
  END;

PROCEDURE build (x, y, size: INTEGER): INTEGER;
VAR
  quadtree: FILE;
  next: INTEGER;
BEGIN
  PROCEDURE build (x, y, size: INTEGER): INTEGER;
  VAR
    last, node: INTEGER;
    branch: BRANCH;
  BEGIN
    IF homogeneous (x, y, size) OR (size = 1) THEN
      RETURN description(x, y, size);
    END
    size := size DIV 2;
    last := next;
    next := next + 1;
    branch.value [NW] :=
      build(x , y + size, size);
    branch.value [NE] :=
      build(x + size, y , size);
    branch.value [SW] :=
      build(x , y , size);
    branch.value [SE] :=
      build(x + size, y + size);
    branch.pointer := next;
    node := dominant(branch.value);
    IF node < 0 THEN
      PutBranch(quadtree, last, branch);
    ELSE
      next := next—1;
    END;
    RETURN node;
  END build;
  (* main program *)
  next := 1;
  PutRoot(quadtree, build(0, 0, 4096));
  RETURN description(0, 0, 4096);
END;
```

The build procedure calls the generic procedures homogeneous and description to obtain information about the object space. The dimension of the object space is assumed to be 4096 by 4096 in this example. The procedure dominant returns the dominant value of a branch as described in the previous section. If all four quadrants are homogeneous and have the same value, dominant will return a positive value indicating a terminal node. This situation can arise when a node value has been estimated as a result of the recursive reaching pixel level. The redundant branch is pruned by simply decrementing the global variable next, which will cause the branch to be overwritten. Thus the four quadrants are merged into a single quadrant, and the resulting quadtree is minimal.

The order in which the branches are written is not sequential, but tends to be close to sequential, with occasional backward jumps to write the ancestor branches once all their non-terminal nodes are known. The first branch will of course always be written last. For the quadtree in Fig. 4, the order in which the branches are written is 3, 4, 2, 6, 8, 5, 10, 9, 1. This ordering suggests that multi-buffering the disc reads would be beneficial.
PROCEDURE display(x, y, size, node,
    oldpen: INTEGER);
VAR pen: INTEGER;
    branch: BRANCH;
BEGIN
pen := colour(ABS(node));
    IF pen # oldpen THEN
fill(x, y, size, pen);
END;
    IF (node < 0) AND (size > 1) THEN
size := size DIV 2;
GetBranch(quadtree; next, branch);
next := next + 1;
display (x, y + size, size,
    branch.value[NW], pen);
display (x, y + size, size,
    branch.value[NE], pen);
display (x + size, y, size,
    branch.value[SW], pen);
display (x + size, y, size,
    branch.value[SE], pen);
IF size = 1 THEN
    next := branch.pointer;
END;
END;
END display;

(* main program *)
next := 1;
display (0, 0, 512, GetRoot(quadtree),
    -1);

The display procedure calls the generic procedures fill and colour to draw the quadtree. The resolution of the display device is assumed to be 512 by 512 in this example. When the size of a quadrant is smaller than a pixel, the pointer is used to skip detail. The order in which the branches are read is sequential, except for the forward jumps made when detail is skipped.

The time taken to display a quadtree can be reduced by taking advantage of the dominant values. Instead of only calling fill for the terminal nodes, it is also called for the non-terminal nodes. The top-level quadrant is always drawn (using the dominant value), and thereafter quadrants are only drawn if they are a different colour from their father. The definition of dominance implies that at least one of the four quadrants must be the same colour as its father, and it is likely that others will also be the same colour. Therefore drawing a large quadrant, and then over-drawing the smaller quadrants that are different, results in fewer calls to fill—generally about half the usual number. If the time taken by a call to fill is roughly constant, this technique will double the speed in which a quadtree can be drawn.

Many display devices can indeed fill a large rectangle almost as quickly as a small rectangle. This is because the bottleneck is in the procedure call overhead or the bandwidth over which the drawing instructions are sent. Also, raster devices often have to write a whole word of pixels to the graphics memory, so the time taken to draw a rectangle smaller than a word wide (most quadrants will be small for a complex quadtree) is only proportional to the rectangle’s height and not its area, and each time a rectangle is drawn there is a constant overhead for address calculations, etc. Obviously the success of this method is device-dependent, and the method may even be detrimental on some devices. Since the drawing instructions describe the quadtree, this suggests that the technique of not storing node values that are the same as their father could be used to produce a more compact quadtree representation. This possibility will be explored in the next section.

The pointer in the Goblin quadtree is also useful in other quadtree operations. If the display procedure was modified to draw only that part of the quadtree inside a given window, the pointer could be used to skip detail for quadrants entirely outside the window. For the quadtree set operations of union and intersection, detail can be skipped for one quadtree if the other quadtree node determines the result of the operation. For instance, in intersection, if the node for quadtree A is WHITE the result must be WHITE, so any detail at the corresponding quadrant in quadtree B can be skipped. The pointer and dominant value can also be used to give views of the data at limited resolutions, and the user can tailor any traversal operation to the accuracy required.

5. SPACE EFFICIENCY

The previous two sections introduced the basic Goblin quadtree. This section discusses space efficiency and suggests some improvements which reduce the size of Goblin quadtree files. To obtain a reasonable range of spatial coherence, two types of data were used in analysing space requirements: geographic regions and pictures from a paint system. The geographic regions have a high degree of spatial coherence, whereas the paint-system pictures lack coherence because of the airbrush effects used in creating them.

At first glance the Goblin quadtree may appear quite compact, as only one pointer is required for each block of four nodes, but if the Autumnal quadtree is extended to include an average or dominant value, the resulting structure has the same storage space requirements as the Goblin quadtree. This is shown in Fig. 5. In the Autumnal quadtree the leaf nodes are dropped (hence autumnal), and the leaf values are stored in place of the pointers to them. The sign bit is used to distinguish pointers from leaf values, with the convention being that positive numbers are pointers. Because this quadtree is fully pointered, it can be arranged to have the same ordering as the Goblin quadtree, and consequently achieve the same benefits gained by sequential access. However, the Goblin quadtree has two important advantages. First, the average values are stored one level higher in the

<table>
<thead>
<tr>
<th>Node</th>
<th>NW</th>
<th>NE</th>
<th>SW</th>
<th>SE</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>-1</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>3</td>
<td>-1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>-3</td>
<td>-3</td>
<td>1</td>
<td>-2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>-1</td>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>-3</td>
<td>-1</td>
<td>-3</td>
<td>-3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>-3</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>-1</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>-2</td>
<td>-2</td>
<td>-1</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 5. Extended Autumnal quadtree.
Goblin quadtree, so a truncated traversal requires far fewer records to be read. Secondly, the Goblin quadtree does not mix pointers and leaf values, so space is not wasted when the pointer requires more bits than the leaf value.

The ratio of pointer size to leaf value size depends on the spatial resolution and the number of leaf values in the data, but in general a pointer requires more space. If pointers are assumed to require two bytes and leaf values one byte, then the extended Autumnal quadtree requires nine bytes per record, as opposed to six bytes per record for the Goblin quadtree. If space is only allocated as required and the pointers store byte addresses, the space requirements for the extended Autumnal quadtree are reduced to an average of six bytes per record. However, by treating pointers and space allocation similarly in the Goblin quadtree, the space requirement can be reduced by removing the pointer for branches containing only leaf values (this pointer always points to the next branch). Using the data described above, the pointer is redundant for 43–69% of the branches, resulting in an average of about five bytes per record. In fairness, the Goblin quadtree is less versatile than a fully pointered quadtree, because additional branches must be read if the quadrants are to be visited in a non-standard order.

In essence, the major difference between the Goblin quadtree and the extended Autumnal quadtree is the amount of space taken up by the pointer information. The relative advantage the Goblin quadtree has in this respect can be increased by reducing the amount of space required for node information (this is the same for both quadtrees). This can be done by adopting a coding scheme similar to that proposed by Woodwark. In his scheme, called a compressed traversal code, only distinct leaf values are stored for a set of four quadrants, and a code is used to indicate which quadrants have which values. If the average values are not stored, but replaced by the codes, this scheme requires only 63–67% as much space (ignoring pointers). This agrees with the figure of 65% reported by Woodwark. The space requirement can be further reduced, to 44–52%, by omitting sons that have the same value as their father, as in the display time. The blocks of four nodes and their pointers, which are termed branches, are stored in depth-first order. This means that when the quadtree is traversed, the branches are accessed sequentially, which is likely to be the most efficient order if the quadtree is stored on disc (or any other buffered medium).

The basic Goblin quadtree can be compacted to give representations that compare favourably with other representations in many respects. The Goblin method requires less space and fewer records to be read (for truncated traversals) than an Autumnal quadtree supplemented with average values. A pointerless version of the Goblin quadtree is more compact and stores more information than Woodwark's compressed traversal code, although the Goblin quadtree is harder to build. The nature of the data and the type of processing being performed will determine whether the advantages of Goblin quadtrees outweigh any disadvantages. Goblin quadtrees can of course be extended to handle three-dimensional data.

Recently there has been a growing interest in the use of quadtrees in geographic information systems (GISs). Traditional vector-based GISs have always struggled to handle locational algorithms, such as overlaying layers of information. Quadtrees can handle this type of operation very easily, needing just a single quadtree traversal for each layer. Due to the rapid increase in the availability of geographic data, there is an increased demand for systems that can efficiently answer queries requiring information from several sources. A quadtree-based GIS may well be the solution, and the Goblin method is not as great if only six bits are used for Woodwark's code.

It may seem an anomaly that storing extra information (dominant values) actually produces a more compact representation. The reason this is so can be explained informally as follows. The Goblin method always saves one value at the leaf level, since one value must be the same as the father. For coherent data there will generally only be two values at this level (i.e. the values of two adjacent regions), so in these cases only half the number of values are stored. However, this saving should be offset by the extra dominant values stored further up the quadtree. Nevertheless, an overall saving is achieved because the number of intermediate nodes diminishes rapidly (by a factor of four at every level), and because the father/son rule applies recursively all the way up the quadtree (i.e. fathers are not stored if they are the same as grandfathers, and so on). Unfortunately there is a drawback with this method. The compaction cannot be applied until the dominant values are known, so it seems that two passes are needed to generate the compact quadtree. This problem is currently being investigated.

6. CONCLUSION

A simple new quadtree data structure, the Goblin quadtree, has been described. It is a semi-pointered structure, containing a pointer for each block of four nodes. The pointer points over any descendants of the four nodes, so algorithms can use the pointer to skip detail. Average or dominant values are stored in the non-terminal nodes, and provide an approximation to the detail, and can be used to reduce quadtree display time. The blocks of four nodes and their pointers, which are termed branches, are stored in depth-first order. This means that when the quadtree is traversed, the branches are accessed sequentially, which is likely to be the most efficient order if the quadtree is stored on disc (or any other buffered medium).

The basic Goblin quadtree can be compacted to give representations that compare favourably with other representations in many respects. The Goblin method requires less space and fewer records to be read (for truncated traversals) than an Autumnal quadtree supplemented with average values. A pointerless version of the Goblin quadtree is more compact and stores more information than Woodwark's compressed traversal code, although the Goblin quadtree is harder to build. The nature of the data and the type of processing being performed will determine whether the advantages of Goblin quadtrees outweigh any disadvantages. Goblin quadtrees can of course be extended to handle three-dimensional data.
suitability of the Goblin quadtree as the underlying data structure for such a system is being studied.

Acknowledgements

The author gratefully acknowledges the helpful comments and advice received from Dr N. E. Wiseman, Dr T. H. Glauert and Mr I. P. Stewart and the financial support given by the ICL Research Studentship and the ORS award scheme. The author would also like to thank Dr J. R. Woodwark for his extensive comments, and for suggesting the extended Autumnal quadtree scheme.

REFERENCES


Announcement

26-28 September 1988

BCS-FACS, Term Rewriting Workshop and Tutorial, Bristol, UK.

Term rewriting is becoming increasingly important as it finds applications in specification, logic programming and theorem proving. Interest is growing throughout the world and in the UK. This workshop is organised by BCS-FACS in conjunction with the UK Term Rewriting Group and aims at stimulating interest in the subject.

Tutorial

The workshop will be preceded by a Tutorial given by Prof. Jean-Pierre Jouannaud (Université de Paris-Sud). The tutorial will provide a broad introduction to term rewriting for the non-specialist.

Workshop

The Workshop will provide a forum for presenting and discussing current UK and European research work. Presentations will be organised into sessions, with each talk taking 30 to 45 minutes. Areas of interest for the workshop include

- Unification, Pattern Matching, Narrowing,
- Termination of Rewrite Systems, Knuth-Bendix Completion,
- Theorem Proving, Inductionless Induction,
- Term Rewriting and Specification, Equational Reasoning,
- Applications of Term Rewriting,
- Term Rewriting and Logic Programming.

Demonstrations

In addition to the tutorial and workshop sessions it is hoped to organise demonstrations of state-of-the-art systems.

Accommodation and fees

The workshop will be held at the University of Bristol. Accommodation will be in University Halls of Residence.

Registration, including lunch, tea and coffee, £70:

There will be a reduced registration fee of £45 for BCS-FACS members.

Accommodation and evening meals (including conference dinner), £75.

These figures are provisional.

For further information please contact:

Derek Coleman, Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6QZ, Tel: (0272) 799910.