A Practical Algorithm for Computing Neighbors in Quadtrees, Octrees, and Hyperoctrees

Robert Yoder, Department of Computer Science, Siena College, 515 Loudon Road
Loudonville, NY 12211.
Peter Bloniarz, Department of Computer Science, University at Albany, 1400 Washington Avenue,
Albany, NY 12222.

Abstract

A simple and general method for computing location codes of same-size neighboring nodes for any direction within quadtrees is developed and then extended for octrees and 4-D hyperoctrees. The effect of our algorithm is to ascend the tree, guided by navigational cues, until the nearest common ancestor of the target node and the computed neighbor node is reached. An advantage of our algorithm is that we can easily determine if the computed neighbor location is outside the boundaries of the quadtree or (hyper)octree.

An experimental analysis of the performance of this finite-state machine (FSM) based algorithm indicates that it runs fast in the average case by measuring the average distance to the closest common ancestor for each node and all of its neighbors. This average grows slowly as the quadtree or octree depth increases. An extension to the algorithm that uses a smaller FSM table with only primary (orthogonal) neighbor directions is given.

Keywords: Quadtree, Octree, Hyperoctree, Neighbor Finding, Finite State Machine

1 Introduction

Quadtrees and octrees are spatial data structures used for image processing, solid modeling, geographic information systems (GIS), and other applications. While quadtrees represent 2-D surfaces, octrees represent 3-D solid objects. Quadtrees and octrees share the property that individual elements (nodes) have a location code associated with it. The location code describes a unique path from the root of the structure to the node, and can be represented as an array of child node identifiers. The location code identifies a region in space enclosed by the node.

An important problem in many octree applications is finding adjacent regions within an octree. Given a target location code and a direction, how can we determine the properties of neighboring regions? The motivation for a fast and simple method for finding neighboring nodes came from research into geoprocessing operations that would be utilized by a 3-D octree-based GIS. Some of the operations studied were volume and surface area computation, connected component labeling, and Boolean (set) operations [4]. In particular, surface area computation and connected component labeling requires extensive neighbor searching. Other operations requiring extensive neighbor searching include region filling, boundary determination, and representation conversions [3].

An algorithm developed by the authors that uses a finite-state machine (FSM) to directly compute neighbor location codes for any direction (including corners) is presented in this paper. The finite-state machine approach is briefly contrasted with alternative methods that use bit manipulation of location codes to compute neighbors.

Since quadtrees are the easiest to visualize, an overview is first presented of how same-size edge neighbor location codes are computed for quadtrees using the "nearest common ancestor" method. Additional navigation information encoded in a finite-state machine solves the problem of computing quadtree corner neighbors. Generalization to octrees and hyperoctrees follows naturally. Neighbor finding can be divided into two phases: computing the neighbor node's location code; then probing (searching) the quadtree or octree to see if the node actually exists.

There are two major ways to represent quadtrees: pointer and linear. Each node in a pointer quadtree contains four pointers to child nodes (Figure 2). Each child node represents a quadrant of the region of space represented by its parent node. Linear quadtrees do not have the overhead of maintaining pointers; they store nodes in a (linear) list, indexed by a unique key (location code) that describes the path taken down the tree to reach the node. Techniques for probing linear and pointer octrees are independent of the algorithm and beyond the scope of this paper.

2 Quadtree terminology

Some quadtree terminology is defined here before delving into neighbor finding details. As shown in Figure 1 (left), each quadtree node has 4 edge neighbors in the Left, Right, Up, and Down directions, and 4 corner neighbors labeled by their position in the figure. Child nodes are labeled as shown in Figure 1 (right).

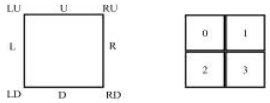

FIGURE 1. QUADTREE NEIGHBOR AND PARTITION NAMING CONVENTIONS

Quadrants are subdivided into smaller quadrants as needed to enclose regions. Figure 2 shows the regions and the tree diagram for nodes with location codes 2 and 32.

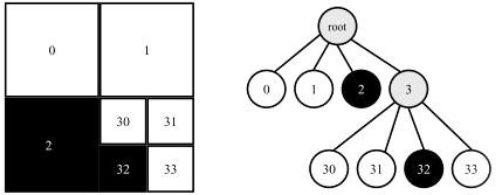

FIGURE 2. SAMPLE QUADTREE REGION AND TREE REPRESENTATION

3 Computing the location code of a quadtree edge neighbor

Any two nodes in a quadtree will have a common ancestor. In Figure 2, node 32 and its neighbor node 2 have a common parent at the root node. For expository purposes, the node we are computing neighbors for is called the *target* node. The terms "location code" and "path" can be used interchangeably, since the location code describes the path taken down a quadtree or octree to reach a given node representing a region of space.

3.1 Computing neighbors using the quadtree finite state machine (FSM)

How do we navigate between same-size neighboring nodes in a quadtree? Given a quadrant and a direction, the FSM returns the quadrant number encountered when moving from the given quadrant to an adjacent same-size quadrant for the given direction. Another way to describe this query is: if there were a same-size node adjacent to the current node in a given direction, what would its quadrant number be? By examining a "tapestry" of repeating nodes of the same size (Figure 3 left), we can determine the adjacent quadrant numbers at any quadtree level, and encode this adjacency information in the quadtree FSM (Table 1).

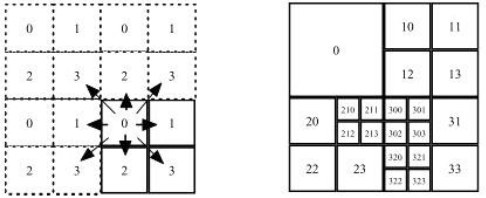

FIGURE 3. TAPESTRY OF REPEATING NODES AT THE SAME LEVEL, AND SAMPLE QUADTREE

For example, Table 1 indicates the quadrant to the RIGHT (R) of any quadrant zero is quadrant one; and the LEFT (L) neighbor of any quadrant zero is also quadrant one. Note, however, that the RIGHT neighbor of any quadrant zero is also a sibling while the LEFT neighbor (L) of quadrant zero is also a quadrant one but is not a sibling. For sibling nodes we add a "halt" command that tells us when two adjacent same-size nodes are siblings so that we can terminate the search for a common ancestor.

TABLE 1. FSM FOR QUADTREE NEIGHBORS IN ANY DIRECTION

Direction	Quadrant 0	Quadrant 1	Quadrant 2	Quadrant 3
R	1,halt	0,R	3,halt	2,R
L	1,L	0,halt	3,L	2,halt
D	2,halt	3,halt	0,D	1,D
U	2,U	3,U	0,halt	1,halt
RU	3,U	2,RU	1,halt	0,R
RD	3,halt	2,R	1,D	0,RD
LD	3,L	2,halt	1,LD	0,D
LU	3,LU	2,U	1,L	0,halt

If the nodes are not siblings, we must go up a level in the quadtree. We need to encode additional navigation information to supplement the basic "halt" action code by including the direction of the *parent node* (one level up) of the same-size neighbor node for the given direction. This additional navigation information has the effect of selecting the correct unique path to the nearest common ancestor. Once the FSM is created, neighbor finding is simple: the location code of a same-size quadtree edge neighbor is computed by replacing each digit (from right to left) of the target node's location code with the digit from the quadtree FSM table for the desired direction. This has the effect of ascending the quadtree, starting from the target node, until the nearest common ancestor node of the target and its neighbor is reached. Each entry in the FSM described by Table 1 has an action code consisting of "halt" or a (new) direction. The new direction is used to look up the next digit of the neighbor. The ascension is complete when one of the following conditions is true:

1) The neighbor is a sibling (same size and same parent, "halt" encountered)
2) The root is encountered without condition (1) becoming true

Example:
Compute the edge neighbor of node with tesseral address 320 in the LU direction. (see Figure 3, right).
Look up the extended FSM table entry for quadrant 0 and (initial) direction LU.
It is 3. Replace the (rightmost) node address digit 0 with 3.
The result is 323. The direction remains LU.
Look up the entry for (middle digit) 2 and direction LU. It is 1.
Replace digit 2 with 1. The result is 313. The new direction becomes L.
Finally, look up (leftmost) digit 3 for direction L. It is a 2. Replace digit 3 with 2.
The result is 213. The action code is "halt," computation stops.

3.2 Reducing the size of FSM tables – only orthogonal directions needed

Note that listing the corner neighbors in Figure 1 is unnecessary: once we know the neighbors for the primary (orthogonal) directions {R,L,D,U}, we can easily compute the corner neighbor directions of Table 1 {RU, RD, LD, LU} by using a double look-up of the primary directions. For example, to calculate the RU neighbor of quadrant 1, we first look up the Right neighbor of quadrant 1 in table 1, then use that entry to lookup its Upper neighbor quadrant.

The new direction (action code) can also be determined simply; just "collect" the directions for each child involved in the calculation. If only HALTs are found, then the new action is HALT. Otherwise the action code is the set of directions encountered during the calculation. In this example, the RU neighbor of child 1, we encounter table entries {0,R} and {2,U}. Thus the result is quadrant 2 with collected directions RU. The new entry is {2,RU}. We will use this same technique to reduce the table sizes for (hyper)octrees later in this paper. This algorithm extension introduces a trade-off between table space and computation time for neighbor calculation, and serves to reduce the length of this paper by reducing the size of the (hyper)octree FSM tables.

4 Computing octree neighbor location codes

Octree face, edge and corner directions are named for their location on each cube-shaped node (see Figure 4, left). There are six faces, each is denoted by a single letter indicating its position on the node {Front, Back, Left, Right, Up, Down}. Each of the twelve edges is denoted by two letters indicating its position; for example RB is the Right and Back vertical edge on the cube. The eight octree corners have three letters indicating its position; for example RDB indicates the Right, Down, Back corner. There are a total of 26 octree neighbor directions. The octant naming convention for this paper is also shown (see Figure 4, right).

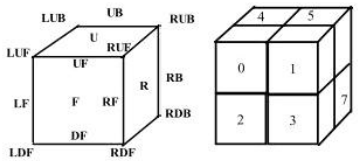

FIGURE 4. OCTREE DIRECTIONS AND OCTANT NUMBERING

4.1 Computing same-size octree neighbors using the octree FSM

The "nearest common ancestor" neighbor finding method described earlier for quadtrees works in the same manner for octree neighbors. The quadtree FSM can be extended to contain neighbor adjacency information for octrees. Then the octree FSM is used to compute the location code of a neighboring node by looking up the path to the desired neighbor, starting from the rightmost octal digit of the path (deepest in tree) and proceeding leftward (higher in tree), replacing the original location code octal digit found in the FSM.

The nearest common ancestor is found when the target node and the neighbor node (for a given direction) are siblings, as indicated by a "halt" in the octree FSM. If the root of the octree is encountered without a sibling being found, then no neighbor exists for that node and direction in the octree.

The octree FSM for the 6 primary directions is shown in Table 2. It is interpreted as follows: for a given axis direction (first column) and octant number (top row), the intersecting FSM table entry is the adjacent octant number. When a sibling is found ("halt" encountered) we have reached the nearest common ancestor of both nodes. Otherwise, we go up a level by using the next octal digit of the location code as the top row index in the direction table, and using the new direction as indicated in the table entry.

Example: find the octree face neighbor of node address 301 in the Right direction. Look up the table entry (in Table 2) for octant 1 (rightmost octal digit in node address) for the Right direction. It is 0. Replace the 1 in the node address with a 0. The result is 300. The direction remains Right. Now examine the middle digit in the node address. It is 0. Look up the table entry for octant 0 in the Right direction. It is 1. Replace the 0 with 1 in the node address and the resulting neighbor node address becomes 310. This entry has a "halt" command; computation stops.

Listing the full 8 by 26 octree table is unnecessary because we can use the technique described earlier to compute edge and corner entries in the table. After we derive the neighbor octants for the orthogonal directions {D,U,R,L,B,F}, we can easily compute the 12 octree edge neighbor directions {LD,LU,etc} and the 8 corner directions {LUF,RUF,etc} using the a double or triple lookup in Table 2. For example, to calculate the RUF neighbor of octant 3, we look up the entry for octant 3 and direction R. It is {2,R}. We then look up the entry for octant 2 and direction U. It is {0, halt}. Lastly, we look up octant 0 and direction F. It is {4,F}. Thus the RUF neighbor of octant 3 is octant 4, with additional collected action code (navigation information) RF.

TABLE 2. REDUCED FSM FOR FINDING OCTREE NEIGHBORS

Direction	0	1	2	3	4	5	6	7
D	2,halt	3,halt	0,D	1,D	6,halt	7,halt	4,D	5,D
U	2,U	3,U	0,halt	1,halt	6,U	7,U	4,halt	5,halt
R	1,halt	0,R	3,halt	2,R	5,halt	4,R	7,halt	6,R
L	1,L	0,halt	3,L	2,halt	5,L	4,halt	7,L	6,halt
B	4,halt	5,halt	6,halt	7,halt	0,B	1,B	2,B	3,B
F	4,F	5,F	6,F	7,F	0,halt	1,halt	2,halt	3,halt

5 Computing hyperoctree neighbors

How can we visualize an additional dimension in a 4-D hyperoctree? Time is commonly used as the fourth dimension, but the additional dimension could be something like temperature as well. A 4-D hyperoctree node contains sixteen child nodes labeled in hexadecimal fashion, and can be envisioned as a pair of sibling octrees. The left octree, with child nodes labeled 0..7, can represent the object at time = 0 and the right octree, with child nodes labeled 8..F, represents the object at time =1. Sub-partitioning each node adds a time demarcation.

For the purposes of this paper we will use time as the 4th dimension, with additional neighbor directions of Plus and Minus. Finding an adjacent neighbor node in the time dimension operates as follows: the neighbor in the Plus direction for nodes 0..7 is the corresponding (congruent) node in the sibling octree. Thus, the neighbor of node 0 is node 8, the Plus neighbor of node 1 is 9, and so on. The Plus neighbor of nodes 8..F will be a node 0..7, but we must go up a level in the hyperoctree in the Plus direction and continue the search for the neighbor. The time Minus direction operates in a similar way: the Minus neighbor of node 8 is node 0 in the Minus direction, and so on. The time minus neighbor of node 0 will be a node 8 in the Minus direction at a higher level.

For the 4-D FSM adjacency table we know there are 16 child nodes, but how many neighbor directions are there? For the 1-D case there are two directions (left and right), for quadtrees there are 8 directions, and for octrees, 26. The number of directions for the next higher dimension $d+1$ is the number of directions in dimension d times 3, then add 2. Thus, the number of directions in a 4-D hyperoctree is (26*3) +2 = 80. Some of the new directions include LUP, LUM, RDBP, RDBM, etc. Listing a 16 by 80 table is unnecessary since we can generalize our hyperoctree adjacency tables by using the additional lookups described earlier for quadtrees and octrees. Thus, we only need to list a 16 by 8 table (Table 3) with new primary time directions Plus and Minus, and calculate any corner directions as needed. Note that to avoid confusion it must be made clear that in Table 3, the first entry in each cell is the hexadecimal child number, the second entry is the direction; thus {D,D} indicates child D (13) and direction DOWN. H indicates Halt. Computing neighbor locations operates in the same manner as octrees.

TABLE 3. REDUCED FSM FOR COMPUTING NEIGHBORS IN HYPEROCTREES

Child number →																
Dir	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D	2,H	3,H	0,D	1,D	6,H	7,H	4,D	5,D	A,H	B,H	8,D	9,D	E,H	F,H	C,D	D,D
U	2,U	3,U	0,H	1,H	6,u	7,U	4,H	5,H	A,U	B,U	8,H	9,H	E,U	F,U	C,H	D,H
R	1,H	0,R	3,H	2,R	5,H	4,R	7,H	6,R	9,H	8,R	B,H	A,R	D,H	C,R	F,H	E,R
L	1,L	0,H	3,L	2,H	5,L	4,H	7,L	6,H	9,L	8,H	B,L	A,H	D,L	C,H	F,L	E,H
B	4,H	5,H	6,H	7,H	0,B	1,B	2,B	3,B	C,H	D,H	E,H	F,H	8,B	9,B	A,B	B,B
F	4,F	5,F	6,F	7,F	0,H	1,H	2,H	3,H	C,F	D,F	E,F	F,F	8,H	9,H	A,H	B,H
P	8,H	9,H	A,H	B,H	C,H	D,H	E,H	F,H	0,P	1,P	2,P	3,P	4,P	5,P	6,P	7,P
M	8,M	9,M	A,M	B,M	C,M	D,M	E,M	F,M	0,H	1,H	2,H	3,H	4,H	5,H	6,H	7,H

6 Comparison of neighbor-finding techniques

The FSM algorithm presented in this paper is a simple, general method for computing the location code for neighbors in any direction for quadtrees and (hyper)octrees. Its execution time in the worst case is proportional to the length of the location code, but we will show in the next subsection that the average execution time is fairly fast. This was done by computing the average distance (number of intervening parent nodes) to the common ancestor of the target node and all of its neighbors. This distance is the number of transitions (table lookups) required to find their common ancestor. A comparison of the FSM method with bit-manipulation techniques follows.

Ibaroudene [1] presented separate bit-manipulation algorithms for each octree face direction, and suggested using a double calculation to compute octree edge nodes and a triple calculation to compute corner adjacent nodes. For example, the Right Upper (RU) octree edge neighbor can be found by first computing the Right face neighbor and using the result as input to the Upper face neighbor calculation:

```
Right_neighbor = FACE-RIGHT(Target_node)
RU_neighbor = FACE-UPPER(Right_neighbor)
```

We have employed Ibaroudene's idea to calculate edge and corner direction entries in our quadtree and (hyper)octree FSM's. Schrack [3] described a constant-time bit-manipulation algorithm for computing same-sized quadtree and octree neighbors by recognizing the interleaved nature of the x and y coordinates in each location code. Schrack's octree neighbor calculation requires 14 arithmetic and logical operations. Recent work by Lee and Samet [2] describes a constant-time algorithm for computing neighbors in triangle meshes implemented as linear quadtrees.

In the worst case, the FSM technique will execute n table lookups when the target node is at depth n and the nearest common ancestor is the root node. The other worst case is when the neighbor node is not in the tree. On the other hand, a single lookup will suffice when the nodes are siblings. What about the average case? Often we will be processing an entire quadtree or octree to perform an operation such as connected component labeling; thus the average performance of the algorithm is important. Limitations of bit-manipulation approaches include word-length restrictions and the extra work needed to determine if the computed neighbor is within the boundaries of the tree.

What is the average distance to the common ancestor of two neighboring nodes of the same size? Programs that counted the number of state transitions needed to compute neighbor location codes (in all directions) for every possible node in a quadtree or (hyper)octree were run for level 1 through level 7 trees. The key result is that the average distance grows slowly and asymptotically as the depth increases (see Figure 5).

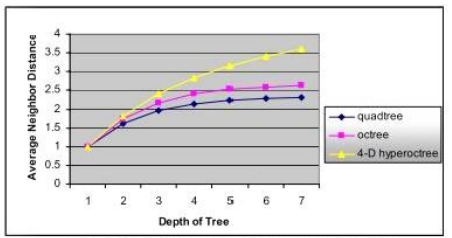

FIGURE 5. AVERAGE TRANSITIONS NEEDED TO COMPUTE NEIGHBOR

7 Summary

A simple and general multidimensional neighbor-finding algorithm using a Finite State Machine was proposed. Using a Finite State Machine demonstrates the essence of bottom-up neighbor finding in hierarchical data structures. The effect of our algorithm is to ascend the quadtree or octree, guided by navigational cues, until the nearest common ancestor of the target node and the computed neighbor node is reached. By using a "halt" code in the FSM we can quickly determine if the computed neighbor location is within the quadtree or octree. A generalization to hyperoctrees was presented, along with a way to reduce the FSM table size by calculating corner direction entries using a lookup operation on primary directions. This algorithm runs fast in the average case even as the depth of quadtrees or octrees increase.

8 References

[1] IBAROUDENE, D. 1991. Algorithms and Parallel Architecture for Multi-Dimensional Image Representation. Ph.D. dissertation, SUNY Buffalo, Buffalo, NY.

[2] LEE, M., and SAMET, H. 2000. Navigating through Triangle Meshes Implemented as Linear Quadtrees. *ACM Transactions on Graphics*, Vol. 19, No. 2, pp 79-121.

[3] SCHRACK, G. 1992. Finding Neighbors of equal size in linear quadtrees and octrees in constant time, *CVGIP: Image Understanding*, 55, 3 (May 1992), pp. 121-230.

[4] YODER, R. C. 1999. Design Considerations for Implementing Octree Based 3-D GIS, Ph.D dissertation, University at Albany, Albany, NY.