Compression of N-Tree Structures

George Buyanovsky and David Salomon

Objects used in real life are normally three dimensional, although many objects are close to being two- or even one dimensional. In geometry, objects can have any number of dimensions, although we cannot visualize objects in more than three dimensions. An N-tree data structure is a special tree that stores an N-dimensional object. The most common example is a quadtree, a popular structure for storing a two-dimensional object, normally an image.

The root of a quadtree corresponds to the entire image. If the image is completely uniform (i.e., all the pixels are the same color) the color is stored in the root, and the root becomes the entire tree. Otherwise, a new level is added to the tree, with four nodes, each corresponding to one quadrant of the image. If a quadrant is uniform, its color is stored in its node and the node becomes a leaf of the quadtree. A nonuniform quadrant adds a new level to the quadtree, with four children nodes that correspond to the four subquadrants of the nonuniform quadrant. This process continues until the subsubquadrants reach the size of a pixel. Figure 1 shows a simple example:

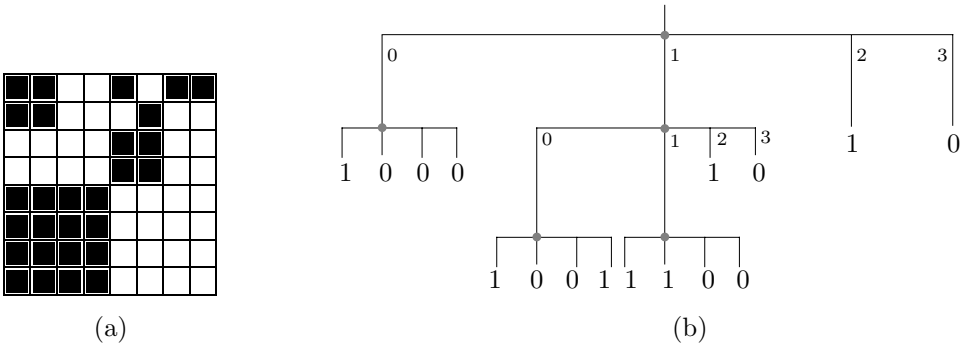

Figure 1: A Quadtree.

The 8×8 image of Figure 1a produces the 21-node quadtree of Figure 1b. Sixteen nodes are leaves (each containing the color of one quadrant, 0 for white, 1 for black), and the other five (the gray circles) are interior nodes containing four pointers each. The quadrant numbering used is $\binom{0\,1}{2\,3}$.

In a similar way, an *octree* is a data structure where a 3-dimensional object can be stored. In an octree, a node is either a leaf or has exactly eight children. The object is divided into eight octants, each nonuniform octant is recursively divided into eight suboctants, and the process continues until the subsuboctants reach a certain minimal size. Similar trees can, in principle, be constructed for n-dimensional objects.

The technique described here is a simple, fast method to compress an N-tree. It has been developed as part of a software package to handle medical data such as

NMR, ultra sound, and CT. Such data consists of a set of 3-dimensional medical images taken over a short period of time, and is therefore four dimensional. It can be stored in a hextree, where each node is either a leaf or has exactly 16 children. This document describes the data compression aspect of the project and uses a quadtree as an example.

Imagine a quadtree containing the pixels of an image. For the purpose of compression we assume that any node A of the quadtree contains (in addition to pointers to the four children) two values, the minimum and maximum pixel values of the nodes in the subtree whose root A is. For example, the quadtree of Figure 2 contains pixels with values between 0 and 255, so the root of the tree contains the pair $0, 255$. Without compression, pixel values in this quadtree are 8-bit numbers, but the method described here makes it possible to write many of those values on the compressed file encoded with fewer bits. The leftmost child of the root, node ①, is itself the root of a subtree where pixel values are in the range $[15, 255]$. Obviously, this range cannot be wider than the range $[0, 255]$ for the entire quadtree, so pixel values in this subtree may, in principle, be encoded in fewer than eight bits. The number of bits required to arithmetically encode those values is $\log_2(255 - 15 + 1) \approx 7.91$, so there is a small gain for the four immediate children of node ①. Similarly, pixel values in the subtree defined by node ② are in the range $[101, 255]$ and therefore require $\log_2(255 - 101 + 1) \approx 7.276$ bits each when arithmetically encoded. Pixel values in the subtree defined by node ③ are in the narrow range $[172, 216]$ and therefore require only $\log_2(216 - 172 + 1) \approx 5.49$ bits each. This is achieved by encoding these four numbers on the compressed file relative to 172. The numbers actually encoded are 44, 0, 9, and 25.

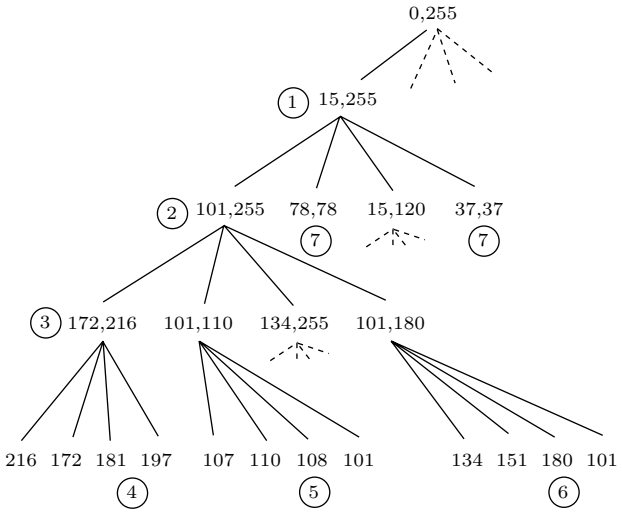

Figure 2: A quadtree with pixels in the range $[0, 255]$.

The two nodes marked ⑦ have identical minimum and maximum pixel values, which indicates that these nodes correspond to uniform quadrants of the image and

are therefore leaves of the quadtree.

The following point helps to understand how pixel values are encoded. When the decoder starts decoding a subtree, it already knows the values of the maximum and minimum pixels in the subtree because it has already decoded the parent tree of that subtree. For example, when the decoder starts decoding the subtree whose root is ②, it knows that the maximum and minimum pixel values in that subtree are 101 and 255, because it has already decoded the four children of node ①. This knowledge is utilized to further compress a subtree, such as ③, whose four children are pixels.

A group of four pixels is encoded by first encoding the values of the first (i.e., leftmost) two pixels using the required number of bits. For the four children of node ③, these are 44 and 0, encoded in 5.49 bits each. The remaining two pixel values may be encoded with fewer bits, and this in done by distinguishing three cases as follows:

Case 1: The first two pixel values are the minimum and maximum of the four. Once the decoder reads and decodes those two values, all it knows about the following two values is that they are between the minimum and the maximum. As a result, the last two values have to be encoded by the encoder with the required number of bits and there is no gain. In our example, the first two values are 216 and 172, so the next two values 181 and 197 (marked by ④), have to be written as the numbers 9 and 25. The encoder encodes the four values $44, 0, 9, 25$ arithmetically in 5.49 bits each. The decoder reads and decodes the first two values, finds out that they are the minimum and maximum, so it knows that this is case 1 and two more values remain to be read and decoded.

Case 2: One of the first two pixel values is the minimum or the maximum. In this case, one of the remaining two values is the maximum or the minimum, and this is indicated by a 1-bit indicator that's written by the encoder, in encoded form, on the compressed file following the second pixel. Consider the four pixel values 107, 110, 108, and 101. They should be encoded in 3.32 bits each (because $\log_2(110-101+1) \approx 3.32$) relative to 101. The first two values encoded are 6 and 9. After the decoder reads and decodes these values, it knows that the maximum (110) is one of them but the minimum (101) is not. The decoder therefore knows that this is case 2 and an indicator, followed by one more value, remain to be read. Once the indicator is read, the decoder knows that the minimum is the fourth value and therefore the next value is the third of the four pixels. That value (108, marked by ⑤) is then read and decoded. Compression is increased because the encoder does not need to encode the minimum pixel, 101. Without the use of an indicator, the four values would reqire $4{\times}3.32 = 13.28$ bits. With the indicator, the number of bits required is $2{\times}3.32 + 1 + 3.32 = 10.96$, a savings of 2.32 bits or 17.5% of 13.28.

Case 3: None of the first two pixel values is the minimum or the maximum. Thus, one of the remaining two values is the maximum and the other one is the minimum. The encoder writes the first two values (encoded) on the compressed file, followed by a 1-bit indicator that indicates which of the two remaining values is the maximum. Compression is enhanced, since the encoder does not have to write the actual values of the minimum and maximum pixels. After reading and decoding the first two values, the decoder finds out that they are not the minimum and

maximum, so this is case 3, and only an indicator remains to be read and decoded. The four pixel values 134, 151, 180, and 101 serve as an example. The first two values are written in 6.32 bits each relative to 101. The 180 (marked by ⑥) is the maximum, so a 1-bit indicator is encoded to indicate that the maximum is the third value. Instead of using $4\times 6.32 = 25.28$ bits, the encoder uses $2\times 6.32 + 1 = 13.64$ bits.

The case max = min + 1 is especially interesting. In this case $\log_2(\max - \min + 1) = 1$, so each pixel value is encoded in one bit on average. The algorithm described here does just that and does not treat this case in any special way. However, in principle, it is possible to handle this case separately, and to encode the four pixel values in fewer than four bits (in 3.8073 bits, to be precise). Here is how.

We denote min and max by n and x, respectively and explore all the possible combinations of n and x.

If one of the first two pixel values is n and the other one is x, then both encoder and decoder know that this is case 1 above. No indicator is used. Notice that the remaining two pixels can be one of the four pairs (n,n), (n,x), (x,n), and (x,x), so a 1-bit indicator wouldn't be enough to distinguish between them. Thus, the encoder encodes each of the four pixel values with one bit, for a total of four bits.

If the first two pixel values are both n or both x, then this is case 2 above. There can be two subcases as follows:

Case 2.1: The first two values are n and n. These can be followed by one of the three pairs (n,x), (x,n), or (x,x).

Case 2.2: The first two values are x and x. These can be followed by one of the three pairs (n,x), (x,n), or (n,n).

Thus, once the decoder has read the first two values, it has to identify one of three alternatives. In order to achieve this, the encoder can, in principle, follow the first two values (which are encoded in one bit each) with a special code that indicates one of three alternatives. Such a code can, in principle, be encoded in $-\log_2 3 \approx 1.585$ bits. The total number of bits required, in principle, to encode case 2 is thus $2 + 1.585 = 3.585$.

In case 3 above, none of the first two pixel values is n or x. This, of course, cannot happen when max = min + 1, since in this case each of the four values is either n or x. Case 3 is therefore impossible.

Thus, we conclude that in the special case max = min + 1, the four pixel values can be encoded either in four bits or in 3.585 bits. On average, the four values can be encoded in fewer than four bits, which seems magical! The explanation is that two of the 16 possible combinations never occur. We can think of the four pixel values as a 4-tuple where the four elements are either n or x. In general, there are 16 such 4-tuples, but the two cases (n,n,n,n) and (x,x,x,x) cannot occur, which leaves just 14 4-tuples to be encoded. It takes $-\log_2(14) \approx 3.8073$ bits to encode one of 14 binary 4-tuples. This is a minor saving that does not justify a special treatment of the case max = min + 1.

The pseudo-code that follows shows how pixel values and indicators are sent to a procedure `encode(value, min, max)` to be arithmetically encoded. This procedure encodes its first parameter in $\log_2(\max - \min + 1)$ bits on average. An indicator is encoded by setting min = 0, max = 1, and a value of 0 or 1. The indicator bit

is therefore encoded in $\log_2(1-0+1) = 1$ bit on average. (This means that some indicators are encoded in more than 1 bit, but others are encoded in as few as zero bits! In general, the number of bits spent on encoding an indicator is distributed normally around 1.)

The method is illustrated by the following C-style code

```
/* Definitions:
encode( value , min , max ); - function of encoding of value
 (output bits stream),
 min<=value<=max, the length of code is Log2(max-min+1) bits;
Log2(N) - depth of pixel level of quadtree.
struct knot_descriptor
{
int min,max; //min,max of the whole sub-plane
int pix ; // value of pixel (in case of pixel level)
int depth ; // depth of knot.
knot_descriptor *square[4];//children's sub-planes
} ;
Compact_quadtree (...) - recursive procedure of quadtree compression.
*/
Compact_quadtree (knot_descriptor *knot, int min, int max)
{
encode( knot->min , min , max ) ;
encode( knot->max , min , max ) ;
if ( knot->min == knot->max ) return ;
min = knot->min ;
max= knot->max ;
if ( knot->depth < Log2(N) )
{
Compact_quadtree(knot->square[ 0 ],min,max) ;
Compact_quadtree(knot->square[ 1 ],min,max) ;
Compact_quadtree(knot->square[ 2 ],min,max) ;
Compact_quadtree(knot->square[ 3 ],min,max) ;
}
else
{ // knot->depth == Log2(N) e pixel level
int slc = 0 ;
encode( (knot->square[ 0 ])->pix , min , max );
if ((knot->square[ 0 ])->pix == min) slc=1;
else if ((knot->square[ 0 ])->pix==max) slc=2;
encode( (knot->square[ 1 ])->pix , min , max );
if ((knot->square[ 1 ])->pix == min) slc |= 1;
else if ((knot->square[ 1 ])->pix==max) slc |= 2;
switch( slc )
{
case 0:
```

```
encode(((knot->square[2])->pix==max),0,1);
return ;
case 1:
if ((knot->square[2])->pix==max)
{
encode(1,0,1);
encode((knot->square[ 3 ])->pix,min,max);
}
else
{
encode(0,0,1);
encode((knot->square[ 2 ])->pix,min,max);
}
return ;
case 2:
if ((knot->square[2])->pix==min)
{
encode(1,0,1);
encode((knot->square[ 3 ])->pix,min,max);
}
else
{
encode(0,0,1);
encode((knot->square[ 2 ])->pix,min,max);
}
return ;
case 3:
encode((knot->square[ 2 ])->pix,min,max);
encode((knot->square[ 3 ])->pix,min,max);
}
}
}
```

Bibliography:

Samet, Hanan (1990a) *Applications of spatial data structures: computer graphics, image processing, and GIS*, Reading, Mass. Addison-Wesley.

Samet, Hanan (1990b) *The design and analysis of spatial data structures*, Reading, Mass. Addison-Wesley.

Questions should be directed to George Buyanovsky at `buyanovsky@home.com` or to David Salomon at `david.salomon@csun.edu`.