CS 315 Data Structures **Fall 2008**

Project # 5 image compression

Due: December 14, 2008

Problem Statement

A given image in bmp format will be compressed by representing it using a quad-tree data structure. The details of this representation and the compression and decompression algorithms are described below. The input will be a bit map image, the outputs will be a compressed representation of the image (which is a string over the alphabet {0, 1, 2, #}) and the decompressed image. An integer parameter d, a tolerance parameter (d in [0, 255]) that characterizes the error tolerance will also be specified. This parameter is also related to the degree of compression – the larger value will produce a more compressed image, and with a lower resolution. Thus the algorithm is lossy – which means the output image will generally be not the same the input. You will also implement an algorithm to search an individual pixel in the compressed representation of the image. The search will be interactive and when the user specifies the row and column values of a pixel, the program will output the RGB value of that pixel (in the compressed image). **Decompression is extra-credit and will be given 5 additional points.** (The project will be weighted 10 points.)

Goals

- Learn about quad-tree representation – an important data structure for storing geometric data (e.g. maps, GIS data etc.) in addition to images.
- Learn to implement a pointer-based tree structure and a compression algorithm.
- Learn to implement a recursive algorithm to access individual pixels in a compressed image (stored as a string).
- Gain insight into the compression – quality trade-offs.

Quad-tree

We will describe a data structure called a ***quad-tree***. A node in this tree represents a rectangular region of an image. This node will be a leaf node if all the pixels in the region are identical (or nearly identical in the case of ***lossy*** representation). If not, the region is divided into four quadrants and each region is (recursively) represented by a tree and each of these will be a sub-tree of a tree. See Figure 1 for an example.

Notice that the node at the blue level of the tree corresponds to the cell of a 1x1 grid over the root square; the nodes at the purple level of the tree correspond to cells of a 2x2 grid over the root square; the nodes at the red level of the tree correspond to cells of a 4x4 grid over the root square; the nodes at the black level of the tree correspond to cells of an 8x8 grid over the root square; and so on.

In Figure 1, the 34 leaves of the tree correspond to the 34 unequal squares into which the root square is split by the diagram.

Compression algorithm

The basic idea behind the compression algorithm is as follows: It takes a parameter d which is an integer between 0 and 255. (In practice, d will take a small integer value, around 10.) The quad-tree representation of the image is obtained by examining all the pixels of the image. If the max and min values of each color component in the image differ by no more than d, then the whole image is represented by a single node whose color attributes are obtained by averaging the R, G, B values over the entire image. If this is not the case, then the tree will be created with a root node and 4 sub-trees, each of which is recursively built using one quadrant of the image. The compression algorithm builds this tree, then it uses two pieces of information to convert the tree into a string – (a) the dimension of the input image and (b) a pre-order traversal of the the tree. Details behind these are described below with an example.

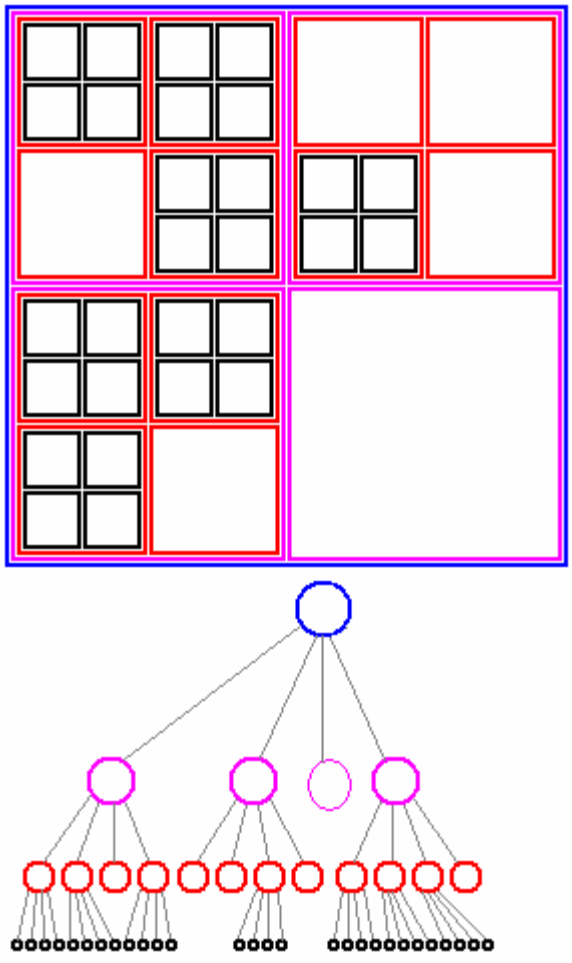

Figure 1. Quad-tree, with pointers ordered
NW, NE, SE, SW.

The internal nodes are assigned a value of 2. Leaf nodes are assigned the color value associated with that region – represented as a 24-bit binary string.

The encoding of the quad-tree would be ***size#traversal-string*** where size is the size of the image, followed by the symbol #, followed by the **preorder** traversal of the tree.

Shown below is an example. On the left is a simple black and white image and on the right is the quad-tree representation. The image is 128 by 128. Its dimension is thus represented by the string 1000000. The quad-tree is represented by the string $21^{24}21^{24}0^{24}1^{24}0^{24}1^{24}0^{24}$ by pre-order traversal. Thus the overall encoding is $10^{7}\#21^{24}21^{24}0^{24}1^{24}0^{24}1^{24}0^{24}$. In the above, a^{b} represents aaaa … a (repeated b times).

(This notation is only for reading convenience. The actual encoding is 1000000#211111111111111111111111112111111111111111111111111 …) ***We will assume that the dimension of the input image is of the form 2^k x 2^k for some k.***

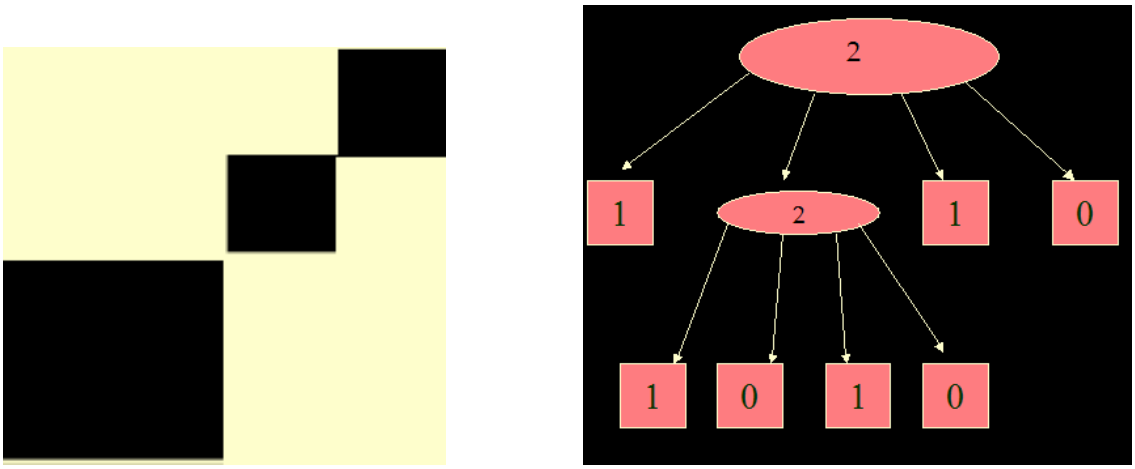

Figure 2: An image and its tree representation. In the figure on the right, 0 represents [0,0,0] and 1 represents [255,255,255].

A more detailed description of the compression algorithm:

Input: An image I, and a parameter d. Assume: width(I) = height(I) is a power of 2.

Output: Compressed representation of the image.

Step 1: If the image is small (say has size 4 by 4) represent it by a single node whose label is the average color of the 16 pixels. Return a pointer T to this node.

Step 2. Else (i.e., width(I) = height(I) >= 8) if a single color c dominates I (i.e., all the pixels are in the interval [c, c + d – 1]) for some c for each color component, then create a single node and assign the average color component to this node and return a pointer T to the node.

Step 3: Else, divide the image into four quadrants NW(I), NE(I), SE(I) and SW(I).

Recursively apply the algorithm on each quadrant to get the subtrees T1, T2, T3 and T4. Create a new node N, and make T1, … , T4 its children. Return a pointer T to N.

Step 4: Perform a preorder traversal on T to get string y. Let the width (and the height) of the image be 2^k. ***Output 10^k#y.***

Decompression algorithm

This is the inverse operation in which the string is converted back to the image. It will be assumed that the input contains a valid encoding. The details of this algorithm are as follows. The algorithm takes as input a string and constructs the quadtree representation.

Input: string s[0..n-1]
Output: T = quadtree representation of s.
Step 1: let s = s1#s2.
Step 2: Return T = convert(s2, 0, s1).
Convert is a recursive procedure described below:

```
qtree convert(string s, int& posn, int size)
if (s[posn] != 2) {
  create a node N with dimension = size, R = s[posn .. posn + 7], G = s[posn + 8 .. posn +
15] and B = s[posn + 16 .. posn + 23], set the children links to null. posn += 24; Return N }
else {
  T1 = convert(s, posn+1, size/4);
  T2 = convert(s, posn, size/4);
  T3 = convert(s, posn, size/4);
  T4 = convert(s, posn, size/4);
 Create a node N. Set N->dimension = size, N->NW = T1, N->NE = T2, N->SE = T3 and
 N->SW = T4. Return N. }
}
```

Search for individual pixels

The problem is to access the individual pixel of an image from the compressed representation. One way to do this is will be to decompress the image and use the standard ***EasyBMP*** function to access the pixel. However, here we will present an algorithm that directly obtains the pixel data from the compressed representation.

Input: string s that represents the image in compressed format, p, q where p represents the column number and q represents the row number of the pixel. Recall that the (0,0) represents the top left corner pixel.

Output: RGB representation of the pixel, e.g., [212, 0, 34].

Algorithm:

Step 1: convert s into a quad-tree representation T.

Step 2: If s is a leaf, then output the label of s and stop.

Step 3: Using the values (p,q), determine which of the four children contains the pixel (p,q) and recursively continue the search on that subtree.

Quadtree class

A quad-tree class with the data members and member functions is suggested below. However, you need not follow this suggestion exactly.

```
class qtree {

 int dimension; RGBApixel data;

 qtree* NW, NE, SE, SW;

qtree(RGBApixel p); // constructor creates a leaf node with p as pixel.

 RGBApixel check(BMP& image, int xval, int yval, int width, int d, bool& check1) ;

 // this procedure returns true (false) via the variable check1 if maxC – minC <= d for c = R, G
//and B in the subimage whose upper-left corner is (xval, yval) and R as width and height. d =
//tolerance. If check1 is true, then the average pixel is returned.
```

```
build(BMP& image, int xval, int yval, int width, int d);

  // returns a quadtree representation of  the subimage whose upper-left corner is

// (xval, yval) and R = width and height, d = tolerance

RGBApixel search(int xval, int yval) // outputs the color values of the specified pixel.

string compress( ); // string representation of the quad-tree

 void uncompress(string comp_img);

// converts the string representation comp_img into quad-tree

 }
```

Details of submission

When your program is run, a dialog similar to the one shown below should occur. The last integer parameter is the d value. In the following, string_out is the name of the file that contains the compressed string.

```
% project5 image_in string_out 12
Enter 1 to access a pixel, enter 0 to quit
1
Enter the x and y values of the pixel you want to access
134 67
Pixel inaccessible, image size is 128 x 128
Enter 1 to access a pixel, enter 0 to quit
112 67
The RGB values of the pixel at (112, 67) are: R = 120, G = 200, B = 20
Enter 1 to access a pixel, enter 0 to quit
0
%
```