Quadtrees and Image Processing

CS 151 - Introduction to Data Structures

Assignment 9 - due Friday 5/5

This is a group project to be done in pairs. Image processing with quadtree is more complex than the AVL-tree choice, which is why you are given more time and doing it in groups. This project howerver should also be more fun as it is definitely more visual and will produce very nice results. Students completing this version of the final project will be given extra credit as well for challenging themselves.

1 Digital Images

A digital image is simply a rectangular grid of dots, often rendered as squares. Each dot is of a single solid color and is known as a pixel (short for picture element). The reason why we do not see individual pixels as squares is because they are very small. Resolution of a display device is measured by pixel density, or number of pixels per inch (PPI). Digital displays marketed from 2009 onwards have at least 100 PPI, with the newer and sharper ones reaching 200–300 PPI. Pixels are commonly arranged in a two-dimensional grid, the dimensions of which are specified by the image's size in pixel resolution. For example, when we have a JPEG image of 1920x1080, it means the image has 1920 pixels in its width and 1080 pixels in its height. It has 1920x1080 = 2,073,600 total pixels and thus can also be referred to as a 2-megapixel image. Zooming into an image repeatedly will allow you to reach low enough resolution to see individual pixels, at which point you also lose most recognizable details in the image. This is called pixelation, also known as pixelization when the resolution lowering is done deliberately, a technique often employed in censorship.

Figure 1: An image showing individual pixels rendered as squares

In Computer Graphics, an image is also known as a bitmap, and is a data structure as well as an image representation corresponding to a spatially mapped array of bits, a natural representation of a pixel grid. The color representation of a pixel typically requires more than a single bit (which can only represent two colors, black and white), and the number of bits used per pixel is known as color depth. Colors are encoded as integers from 0 to 255, where 0 is black (no color) and 255 is full color and use the RGB color model in which red, green and blue are (added) blended together to produce a broad spectrum of colors. Consult the wikipedia page `https://en.wikipedia.org/wiki/RGB_color_model` for more details on RGB colors. Figure 2 shows three pixels with their individual RGB color values and indices at which they are stored in the pixel grid. For example, the leftmost pixel can be found at row 81, column 123 and is colored with a red value of 137, green value of 196, and blue value of 138. Note that an RGB value of (137, 137, and 138) would be light gray, and thus the combination of (137, 196, 138) comes out light green.

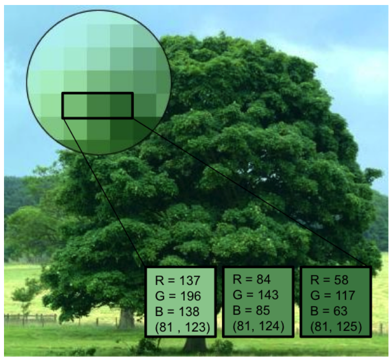

Figure 2: Three pixels

2 Image Processing

Now that you see an image as an array of colors (given as integer triples), image processing is as simple as looping over this array and changing the numbers! This is known as "filtering". Some simple image filters are for example:

1. negative - change each pixel `(r, g, b)` to `(255-r, 255-g, 255-b)`
2. grayscale - change each pixel `(r, g, b)` to `(c, c, c)` where `c = r*0.3+g*0.59+b*0.11`
3. tint - given a tint color `(R, G, B)`, scale each pixel color `(r, g, b)` to `(r/255*R, g/255*G, b/255*B)`. Note that full white `(255, 255, 255)` will become exactly `(R, G, B)`, and everything else will be scaled proportionally between 0 and R/G/B.

2.1 Convolution Filters

A another popular technique is to compute a pixel's color based on its immediate neighbors, including itself. We will base our discussions of these filters on a 3x3 neighborhood, the smallest and simplest; however, in practice neighborhoods can be larger, as well as differently shaped than a square (box). A 3x3 box filter computes the pixel color via a weighted average of all 9 pixels using some predetermined weights. If we consider an input pixel $input[i, j]$, then the weighted average can be defined as:

$$\begin{aligned} output[i,j] = {} & w_1 \times input[i-1, j-1] + w_2 \times input[i, j-1] + w_3 \times input[i+1, j-1] \\ & + w_4 \times input[i-1, j] + w_5 \times input[i, j] + w_6 \times input[i+1, j] \\ & + w_7 \times input[i-1, j+1] + w_8 \times input[i, j+1] + w_9 \times input[i+1, j+1] \end{aligned}$$

The choice of the weights (kernel) has dramatic effect on the resulting image. One popular and useful kernel is edge detection, with the weights set as:

$$\begin{matrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{matrix}$$

That is:

$$\begin{aligned} output[i,j] = &-input[i-1,j-1] - input[i,j-1] - input[i+1,j-1] \\ &- input[i-1,j] + 8 \times input[i,j] - input[i+1,j] \\ &- input[i-1,j+1] - input[i,j+1] - input[i+1,j+1] \end{aligned}$$

Edge detection works by enhancing the differences between the center pixel and those that surround it. When you perceive an edge in an image, you are simply noticing that there is a (sharper) change in color/brightness. Thus, edge detection works by setting a pixel to black (0) if it's not very different from its neighbors, and trend towards white (255) the more different it is. Consider what happens when the filter above is applied to an area of similar colors. The sum will trend towards zero. If the pixel is high contrast and bright, the sum will become more positive.

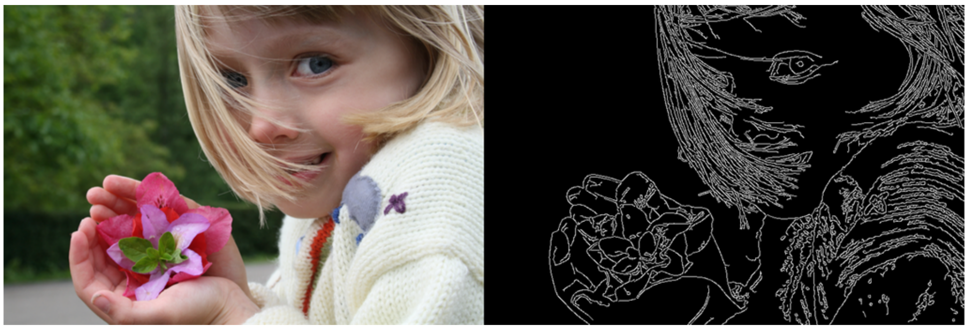
Figure 3: edge detection

3 Quadtree

A quadtree is a tree data structure in which each internal node has exactly four children and is most often used to partition a two-dimensional space by recursively subdividing it into four quadrants. On an image, a quadtree recursively divides the image into four subimages and stops when some criteria are met or reaching a single pixel. Each node either stores the color of the pixel (if it's a single pixel), or the average of the colors of all pixels in the quadrant.

3.1 Quadtree Image Compression

A quadtree decomposition can be used to compress an image in stages as shown in Figure 5. This computes a hierachy of images that represent a high quality image with increasingly more details (in the right places). The idea is to split only in those quadrants where the colors of the children differ greatly (according to some threshold) from that of the parent. Note the quadtree is selective and regions we choose to not split on

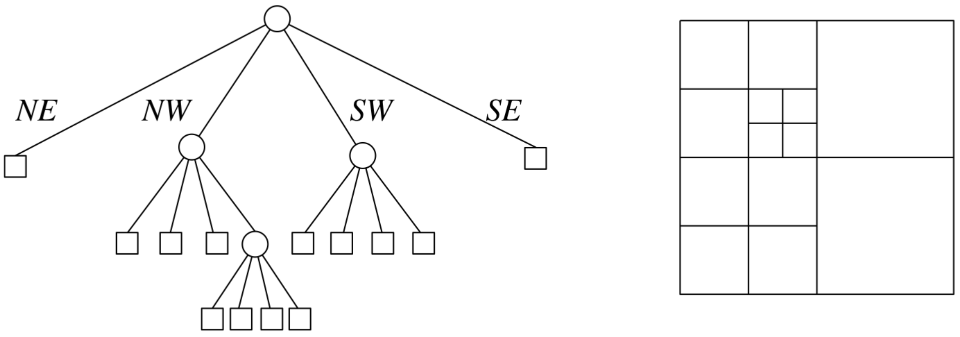

Figure 4: Quadtree

presumably already have more or less the same colors and can afford to be blurred into the mean color. When the subdivion stops, the compressed image can be recovered from the leaves of the quadtree. Different settings of the difference threshold will generate compressions at different resolutions.

Figure 5: Animated quadtree compression of an image step by step

Besides compression, there are many applications for image hierachies - a form of level-of-detail techniques in Computer Graphics. In computer games for example, if a player is far away, only low-res textures are applied on objects and as the player gets closer, textures of higher and higher resolutions are swapped in.

The algorithm is: Start with the entire image as the root node. For each current node n_i, calculate the mean color C_i and mean squared error $E_i = \frac{1}{N^2}\Sigma_{x=1}^{N}\Sigma_{y=1}^{N}|n_i(x,y) - C_i|^2$. The error E_i is the average of the cumulative squared error between the compressed representation C_i and the original, which is computed as the squared Euclidean distance between the mean color C_i and all original pixel colors in the quadrant n_i of size $(N \times N)$. That is, for a pixel with color (r, g, b), the squared error should be computed as $((r - C_i.r)^2 + (g - C_i.g)^2 + (b - C_i.b)^2$. The usual square root is not needed. If a node has error greater than some threshold, split into four subimages and recurse on each child. Recursion stops when error is at or below threshold or child becomes a single pixel.

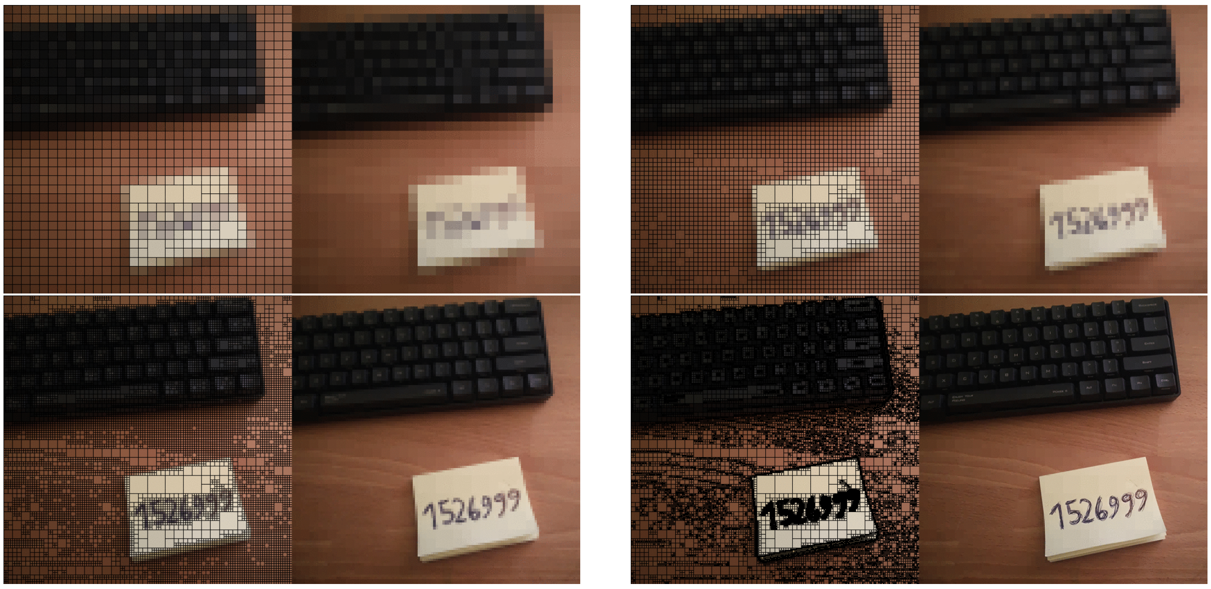

Figure 6: Quadtree image compression, different thresholds

4 Requirements

1. Given an input image, create a quadtree decomposition of the image and compute 8 compressed images of increasing resolutions as explained in 3.1. These 8 images should be generated at compression levels 0.002, 0.004, 0.01, 0.033, 0.077, 0.2, 0.5, 0.75. The compression level is defined as the number of leaves in the quadtree divided by the number of the original pixel count. Note lower levels represent higher compression ratio. The computed fractions do not have to match the numbers exactly, approximately is fine. Also note that in images with large number of pixels of similiar color (a lot of sky or ground, etc), some compression-levels may not be achievable and in that case, only output those that are.

2. Edge detection is expensive, since the filters require multiple (9 for 3x3, 25 for 5x5, etc) operations per pixel. With high resolution digital images having pixel counts in the 10s of millions and results needed in real time, it is often preferred to only apply the filter in important areas, which is where the quadtree comes in. Based on the same quadtree decompostion of an image, apply the edge detection filtering as explained in 2.1 only to those nodes of a sufficiently small size and replace the color of the larger nodes with black.

3. Extra credit: design your own filter based on the quadtree decomposition and do something interesting to the image. Explain your design in your README.

4. For debugging purposes, implement the ability to show the outline of the quadtree cells as shown on the left half in Figure 6. This is a required functionality requested by the commandline flag `-t`. You should implement this BEFORE your program is fully functional, as it is designed to help you debug. Programming it just to satisfy the requirement is the wrong order!

5. It is acceptable to only work on square images. If the input image is not square, print appropriate error message and exit. On the other hand, it's not that much

work to adapt to rectangular images and if yours does not need the restriction to square images, please note clearly in README. Extra credit will be given.

5 Images Formats

In this assignment, our input will be images given in the Portable Pixel Map (PPM) format, which is a simple text file listing colors of each pixel in an image, as explained here: `https://en.wikipedia.org/wiki/Netpbm_format` Your output will be image(s) in the same format. Note that there are two versions of PPM, P3 (also known as plain or ascii) and P6 (binary). P6 is much more widely found because it is more space efficient and less prone to parsing difficulties. On the other hand P3 is plain text and thus easier to debug. We will be using the P3 format. Note that P3 doesn't specify the number of pixel colors per line, thus you should be lenient when parsing P3 format and accept anything that looks remotely like a P3. Most image readers will convert to/from ppm to any other standard image format. On Linux, the command is called `convert`, which is part of the `imagemagick` software suite. An image called `test.png` can be converted using the following command: `convert test.png -compress none test.ppm`. You can substitute `png` for any other popular image format extensions such as `jpg` and `tif`. For more information, consult the manual pages using `man convert`.

A collection of test images can be found at `~dxu/handouts/cs151/data/a9/ppms`.

6 Command Line Input

You will receive an image file on the command line following the `-i` flag as your input, for example: `java Main -i test.ppm`. In addition, support the following flags:

1. `-o <filename>` indicates the root name of the output file that your program should write to
2. `-c` indicates that you should perform image compression
3. `-e` for edge detection
4. `-x` for running your own filter (if doing EC). Note that you can also add arguments to your `-x` flag if your filter design calls for it (typically to set some level or default). If so, explain how to supply these arguments in your README.
5. `-t` indicates that output images should have the quadtree outlined

For example:
`java Main -c -i test.ppm -o out` will generate 8 compressed images of `test.ppm` named `out-1.ppm`, `out-2.ppm`, ..., `out-8.ppm`, where `out-1.ppm` is the image with the lowest resolution/highest compression and so on. `java Main -e -t -i test.ppm -o out` will generate one output image called `out.ppm` which is the result of applying edge detection to `test.ppm`, with the quadtree outlined. You may assume that only one of `-c`, `-e` or `-x` will be given. However, `-t` may or may not be present on any filter. Order of the flags should not matter, i.e. `java Main -o out -e -i test.ppm -t` is equivalent to `java Main -e -t -i test.ppm -o out`

7 Where to Start

At this point, you should be able to design class structures on your own, so we do not give you specific suggestions here. However, try not to do too much all at once, because it's really easy to lose control and end up in debugging hell. Here are some hints about tasks that you should make sure you can perform. These tasks are independent of each other and should be tested individually.

1. Be able to read in an image from a given file name and write out an image to a given file name.
2. Apply filters to an image, try the easy ones in section 2, then write the result back out.
3. Generate a quadtree representation of an image (first uncompressed then compressed).

In addition, a good discussion to divide work between the partners is also highly recommended, so that you do not end up duplicating work, or worse, impeding each other.

8 Debugging Tips

There are three types of errors that are common on this assignment:

Recursion stack overflow This happens during the quadtree decomposition. It means your recursion depth is too deep, in other words, your recursive subdivision is not stopping where it should. The advice is to use a variable to count recursion depth, thereby control recursion depth exactly so that you can find out where the error occurs. Another method is to rig a very small image, say 16 pixels by 16 pixels so that you can predict exactly how and where the recursions will happen.

The other two common exceptions:

NullPointerException This occurs typically when you are reading from your pixel array (to write to output file for example), and it happens because some cells of your array aren't filled in. Any error that leads to misalignments of your quadtree nodes will do this.

ArrayIndexOutofBound This occurs when you are writing into the pixel array using an invalid index in some loop. Suggested debugging strategy is the same for both above errors: Assuming that you are keeping these instance variables (or similar) per node:

```
int starti; //row index of top left corner
int startj; //column index of top left corner
int height; // height of quadrant, number of rows
int width; // width of quadrant, number of columns
```

Your pixel reading/writing looping should be completely controlled by these variables. That is, you should be going over the pixels in each node/quadrant like this:

```
for (int i = node.starti; i < node.starti+node.height; i++)
  for (int j = node.startj; j < node.startj+node.width; j++)
    img[i][j] ...
```

Thus, the only way you could have an invalid index or under coverage is if one of the 4 variables in one of the nodes is wrong. Thus I suggest you print out all 4 of these variables, and in particular, check the sums `starti+height` and `startj+width` in all nodes. None should ever exceed the width or height of the input image, or become negative. And all possible pixels between `(0,0)` to `(width-1, height-1)` should be covered by the combinations of all quadrants from all leaf nodes.

9 Electronic Submissions

1. README: The usual plain text file `README`

> **Your names: make sure both names are here!**
>
> **How to compile:** Leave empty if it's just `javac Main.java`
>
> **How to run it:** Leave empty if it's just `java Main`
>
> **Known Bugs and Limitations:** List any known bugs, deficiencies, or limitations with respect to the project specifications. Documented bugs will receive less deduction versus uncaught ones.
>
> **Discussion:** Design of your own filter

2. Source files: all `.java` files

DO NOT INCLUDE: Please delete all executable bytecode (`.class`) files prior to submission.

To submit, store everything (README and source files) in a directory called `A9`. Then follow the directions here:
`https://cs.brynmawr.edu/systems/submit_assignments.html`