Lab 9: Huff(man)ing and Puffing

Due April 18/19 (Implementation plans due 4/16, reports due 4/20)

The number of bits required to encode an image for digital storage or transmission can be quite large. Consumer quality digital cameras take pictures that are 2560 pixels wide and 1920 pixels tall or larger. Such an image contains a total of 4915200 pixels or just about 5 megapixels. Each pixel is represented by three 8-bit number encoding it redness, greenness and blueness. In raw form, therefore, it would take 117,964,800 bits to represent such an image. On a 10 megabit/second Ethernet it would take over 10 seconds to transmit such an image without even accounting for collisions or other overhead.

To make it possible to transmit images more quickly and to store large numbers of images on computer disks and camera memory cards, considerable effort has been devoted to devising techniques for compressing the information in digital images. File formats like GIF, JPEG, and PNG represent the implementation of some of the compression techniques that have been developed.

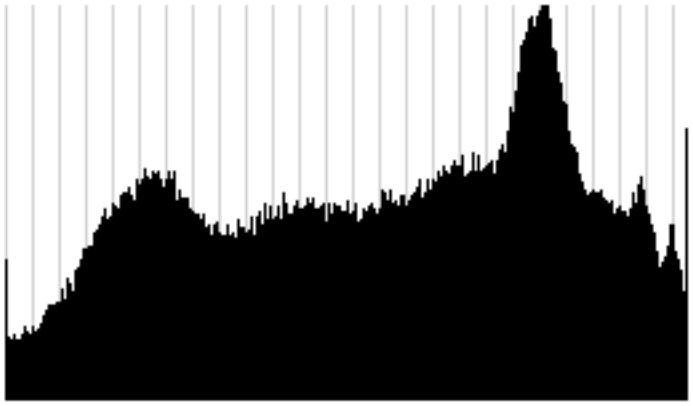
To help you appreciate both how one might go about compressing an image and how difficult it is to achieve high levels of compression, we would like you to implement components of several implementation techniques and then evaluate their effectiveness. Note: The "evaluate" aspect is a new feature of this lab. In addition to writing a Java program this week, we actually want you to write a lab report summarizing the data you collect using the program you have written. These reports will be collected in class after the programs are completed.

Image Simplification

Earlier in the semester, we discussed the use of Huffman codes to reduce the number of bits required to encode a text message. It is possible to use Huffman codes to compress image data. To do this, one would treat the 256 values that can appear in a pixel array as the letters of an alphabet containing 256 symbols. Based on Huffman's algorithm, short codewords would be assigned to the brightness values that appeared frequently in a pixel array and longer codewords would be assigned to the values that appeared less frequently. Then, the table of pixel values would all be translated from their original 8-bit binary codes to the Huffman codewords that had been assigned and the entire list of codewords would be saved or transmitted.[1] Unfortunately, using Huffman codes to compress image data in this way is not very effective.

Huffman codes exploit the fact that some symbols in an image occur more frequently than others. The more extreme the differences between the frequencies with which symbols occur in the data, the greater the degree of compression a Huffman code will provide. The frequencies with which brightness values occur within an image tend to be too uniform for effective Huffman coding. Consider the image shown on the right.

[1] In addition, one would have to encode the Huffman tree describing the code used and the width and height of the image. Since this information would require relatively few bits compared to those used to represent the pixel array values, we will not account for the cost of encoding it in this lab.



The graph shown on the left is a histogram of the frequencies with which various brightness values appear in this image. The histogram ranges from brightness 0 to 255. There is a significant peak around 200 and a smaller peak between 50 and 60. Within the range of 150 values between these peaks, the histogram is rather flat. The distribution of these brightness values is uniform. Huffman coding cannot do significantly better than a fixed length code when applied to such data. In fact, a Huffman encoding of the brightness values in this image would require 7.94 bits per pixel, less than 1% less than the obvious fixed length code. Fortunately, there are techniques we can use to encode the brightness values of an image that enable Huffman coding to work far more effectively. As an example, consider the following transformation.

In the last lab, you constructed a filter that replaced blocks of pixel values with their average brightness. Suppose we instead processed blocks by leaving the value in the upper left corner unchanged and replacing each of the other brightness values in the block with the difference between the original value and the value found in the upper left corner. For example, if we started with the 2x2 block of pixels:

A	B
C	D

We would replace its values with the values:

A	B - A
C - A	D - A

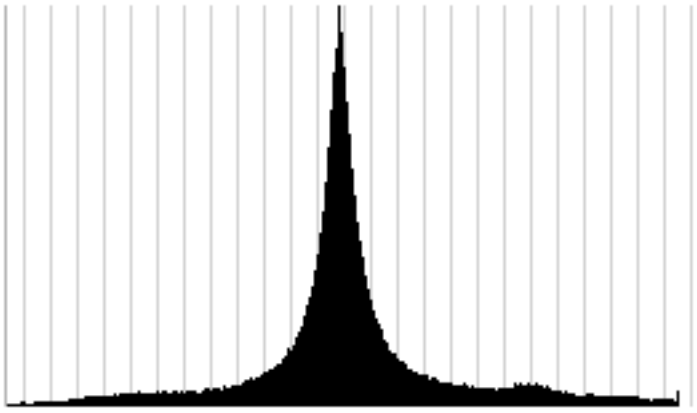
The pixels that are changed by this transformation are likely to have fairly small values. In most images, the shades of pixels that are adjacent are very similar. Therefore the differences we compute while performing this transformation are likely to be small values. These pixels account for 75% of the pixels in the transformed image. Accordingly, this transformation will change the distribution of values in the pixel array significantly. A significant number of the pixel values will be close to 0.

Unfortunately, this also means that a significant number of the values we computed will be negative and therefore fall outside the original range of pixels. We can adjust for this by adding 128 to each difference, centering the new values in the range of existing values.

The result of applying this transformation to all 2x2 blocks in the image presented earlier is shown on the right. Since one quarter of the pixel values are unchanged, the original image remains visible. The changed pixels give the new image a dull gray look.

Note that we can recover all of the pixel values from the original image given just the transformed image. If we process each block within the transformed image by adding the value in the corner of the block to all of the

other values, the original image will reappear.



The histogram of the transformed image is shown on the right. The distribution of brightness values is no longer uniform. Instead, there is a significant peak at 128, reflecting the fact that most of the differences computed while making the transformation were near 0. Huffman coding the collection of pixel values is therefore much more effective. The average number of bits per pixel is roughly 7, reflecting a 12% savings overall. As a result, this transformation provides a way to compress an image for transmission. We first apply the transformation. Then we Huffman encode the resulting pixel values and transmit them. When this transmission is received, the receiving computer can first decode the Huffman codewords to restore the brightness values of the transformed image. Finally, the corner values can be added to the other values in each block to restore the original image.

The savings obtained using this technique will vary from image to image depending on how the brightness value in each image are distributed. Changing from 2 x 2 blocks to larger blocks would also effect the saving. As a result, the only way to really evaluate such a technique is to try several variants on a selection of typical (and atypical) images and analyze the results. For this lab, we want you to conduct such an experiment.

The process of replacing most of the pixels in a block with their differences from the pixel in the corner of the block is just one way one might transform an image to increase the effectiveness of Huffman coding. Any transformation that will a) lead to a less uniform set of values in the image's encoding and b) provide the means to restore the original brightness value or something that closely approximates them can be used. We will call such a transformation an *image simplification*.

For this lab, we want you to implement four image simplification algorithms described below. In addition, we want you to implement an algorithm that computes the bits per pixel required to encode a set of brightness values using a Huffman code. Then, you will use these tools together to evaluate the effectiveness of five image simplification algorithms including the four you have implemented.

The Algorithms

Range Reduction

The first algorithm we want you to evaluate is the simple technique of reducing the range of brightness values used to encode the image by dividing all of the brightness values by a fixed constant. Because integer division will be used to do this range reduction, pixels that had different but similar values in the original image will be represented by a single value after this transformation is applied. This does not actually change the shape of the histogram associated with an image, but by reducing the number of distinct values used, it makes it possible to encode the values with fewer bits. On the other hand, given the reduced brightness values, it is not possible to restore the original image exactly. As long as the value used to divide the pixel values is not too large, however, multiplying all of the values in the transformed image will produce a close approximation of the original image. Such a transformation is said to be *lossy*. By contrast, the block differencing simplification described in our introduction is said to be *lossless*.

We will not actually make you implement this transformation. Instead we will provide you with a completed implementation in a starter project for the lab. We include this simplification for two reasons. First, when you are collecting data on the simplification schemes you implement, it can serve as a base-

line. Second, just as we had you implement filters by extending an `ImageFilter` class last week, in this week's lab, you will implement simplification schemes by extending an `ImageSimplifier` class provided in the starter project. The two implementations of range reduction included in the starter project will serve as an examples of how you should define your simplifiers by extendend `ImageSimplifier`.

Waterfall

The first simplification scheme we want you to implement involves computing the difference between pixel values much like the block corners scheme described in the introduction. The idea is very simple. Leave the pixels along the topmost row of the image unchanged. Replace every other value in each pixel array with the difference between its original value and the original value of the pixel directly above it.

The name of this algorithm comes from the process used to restore the original image given this collection of transformed values. Starting at the top of each column you will add the first pixel value (which will be unchanged) to the value below it (which will be a difference). The sum of these two value will be the original value of the lower pixel. You then repeat this process "falling" down from the top of the column of pixels to the bottom. When you are done, all of the pixels values will be restored to those of the original image.

The waterfall algorithm should be implemented by defining a `WaterfallSimplifier` class that extends the `ImageSimplifier` class included in the project starter folder. Its implementation should mimic the `RangeSimplier1` class that we have also included in the starter.

Wavelet

While the waterfall algorithm processes an image's pixels in pairs from top to bottom, the Wavelet simplification process works with pairs from left to right. It also is more like the corner block scheme in that it works on small groups of pixels independently rather than processing all of the pixels in an entire row or column together.

The wavelet simplifier works with pairs of pixel (i.e., 2 x 1 blocks). The leftmost pixel in each pair is replaced by the average of the two values in the pair and the rightmost pixel is replaced by half of the difference. For example, given the pair:

A	B

We would replace its values with the values:

(A + B)/2	(A - B)/2

Rather than leave the transformed values next to one another, the wavelet transformation moves all of the averages toward the left side of the image and all of the differences to the right. Thus, if a single row of an image that was eight pixels wide contained the values:

A	B	C	D	E	F	G	H

then after the transformation was complete the values stored in the row would be:

(A+B)/2	(C+D)/2	(E+F)/2	(G+H)/2	(A-B)/2	(C-D)/2	(E-F)/2	(G-H)/2

Of course, if an image's width is odd, there will be one pixel that has no partner to form a pair with. We will handle this by simply placing the value of the last pixel in such a row between the averages and the differences. That is, given a row like:

A	B	C	D	E	F	G	H	I

the wavelet simplifier will produce the transformed row:

(A+B)/2	(C+D)/2	(E+F)/2	(G+H)/2	I	(A-B)/2	(C-D)/2	(E-F)/2	(G-H)/2

The resulting image will look like a horizontally contracted copy of the original with a similarly sized dark region to its right. The result of applying this transformation to the image we have been using as our example is shown on the right. If you look carefully, you can see that the dark rectangle actually contains bright regions that correspond to the edges of objects in the original image. The edges are where the differences between adjacent pixels tend to be largest.

The procedure for restoring the original image given the values produced by the wavelet transformation is simple. If the initial values of two adjacent pixels were A and B, then the values stored for these pixels in the transformed image will be E = (A+B)/2 and F = (A-B)/2. If you evaluate the expression E + F, the result will be the value of A. If you evaluate E - F, the result will be B.

Actually, while our claims about E, F, A, and B are true in normal mathematics, in Java, things won't quite work out. Since we will be working with integer values, when we compute (A+B)/2, Java will throw away any remainder from the division. As a result, E + F and E - F will only produce close approximations of A and B. Wavelet simplification is therefore another example of a lossy approach.

Your implementation of wavelet will mimic our `RangeSimplifier1` much like your implementation of waterfall.

Recursive Wavelet

If you look at the sample image shown above to illustrate the result of applying the wavelet transformation, you should notice that while it is a bit squished, the left half of that image looks a lot like a normal picture. Wavelet is a transformation designed to process pictures. Suppose we applied it again to just the left half of the image shown above. The result would look like the image on the right. The left quarter of the image is a very compressed version of the original. The right half of the image is the difference values from the original image. The dark quarter between these two is a collection of difference values from the first compressed version of the image. As such, it is very close to a compressed

version of the right half of the image.

One thing is obvious. More of the pixels in this image are nearly black. Therefore, it will have even a bigger peak in its histogram and should compress better. Of course, if it is good to apply wavelet twice, it must be better to do it three times, or four, or...

The third technique we would like you to implement is a recursive version of wavelet. It will begin by applying the simple version of wavelet described above to an image. Then, if the original image is wider than two pixels, it should extract the left half of the result as a new image and recursively apply itself to the result. Finally, it should paste the result of this recursive call back into the left half of the image from which it extracted the left half.

While you have written recursive code before, the implementation of this simplification process will illustrate a slightly different from of recursion. There will be no recursive class involved. That is, you won't define a class that has an instance variable that refers to another instance of the same class. Instead, you will simply define an image processing method within a class that invokes itself on a smaller image. As a result, there will also be no `empty` boolean to tell you when to stop recursing. Instead, as suggested above, this recursive process will terminate when the image has been reduced to a single column.

You should now recognize the purpose of one of the `ImageFilters` we had you use last week. The Paster filter is just the tool you need to insert the result of the recursive application of the wavelet algorithm back in as the left half of the final image to be returned. In addition, as part of the implementation of this algorithm, you should implement a `CopyLeft` filter that extracts the left half of an image as an independent `SImage`. The definition of "half" here is critical. If an image's width is 2N, then the width of a half is obviously N. If the width is 2N+1, then your filter much return a "half" whose width is N+1.

Once these filters are written, it will be possible to implement the recursive wavelet class much more concisely than the waterfall and non-recursive filters. Rather than working with pixel arrays, you should be able to describe all of the steps of the process directly in terms of `SImages`. First you will simplify an `SImage`. Then you will apply a filter to extract the left half of an `SImage`. Then you will recursively simplify the half image, and so on. As a result, rather than mimicking our `RangeSimplifier1` class as you did in your earlier simplifiers, you should imitate the implementation of our `RangeSimplifier2` when you write a class to implement the recursive wavelet transformation.

The Kitchen Sink
Well, if waterfall is good and wavelet is good, what if we did both? We deliberately described wavelet working top to bottom and waterfall working left to right so that this would be possible. For your last simplifier, implement a class that first applies the recursive wavelet transformation to an image and then applies wavelet to the result. When you are all done, all but one pixel in the result will be a difference value. To reverse the process, simply apply the reversing transformations in the opposite order. That is, first apply the waterfall unsimplifier and then the wavelet unsimplifier.

Huffman Code Size Computation

This week's homework focuses on computing the cost of a Huffman code without actually building the Huffman tree. In particular, we describe an algorithm that returns the total size (in bits) of the encoded document. In the case of this lab our document is an image and our encoded document is a compressed image. We would like you to develop a small variant of the algorithm in the homework. Instead of computing the number of bits in the compressed image, we would like you to compute the average number of bits per pixel in the compressed image. In other words, just divide the total number of bits in the compressed image by the number of pixels. Call this new method `huffmanSize`.

Seeing `double`
Until now, all the division we've performed has been integer division; we always drop the remainder. However, when computing the average number of bits per pixel, we care about the remainder. In fact, we'd like to display the average bits per pixel as is conventional --- with a decimal point. To do this, we'll use a primitive type in Java called `double`. The `double` type represents (as best it can) a real number. For example,

```
double half = 5.0/2.0;
```

mens `half` now has value 2.5. To convert from integers to doubles, one has to *cast* the integer to a double and then divide. For example, suppose `x` has type `int` and value 5. Then the following would assign 2.5 to the `double half`

```
double half = (double) x / 2;
```

Notice that even though the constant 2 is an integer, the resulting type is a `double`. When dividing integers and doubles, Java always converts the integer to a double. As a result, we could have written the division without doing the cast by initially writing 2 as 2.0.

```
double half = x / 2.0;
```

Back to Histograms
An appropriate place to add the `huffmanSize` method is the `Histogram` class. This is because the array storing the number of pixels for each of the 256 brightness values is exactly the input to Huffman's algorithm --- a set of weights or counts. To implement the algorithm, you'll want to do the following:

- Copy all of the non-zero entries in the histogram array into a new `weights` array. The `weights` array should be created to have the same size as the histogram array.
- Because computing the Huffman cost involves working with decreasingly smaller lists of weights, we will maintain a variable called `distinct` that reflects the current length of our array of weights. Why do this? Isn't it the case that weights has its own length member variable? That's true, but it will become clear in the next few bullets.
- Computing the Huffman cost involves both finding the position of and extracting a minimum value from an array of weights. It's easy to find the position of a minimum value, but to extract the minimum, you'll need to use a little trick. Suppose your weight array contains 10 weights (i.e., the value of `distinct` is 10) and the minimum weight appears at index 3. The idea is to move the last weight at index 9 to index 3 and then later decrement `distinct`. This means the array still has its original length, but you can simulate shrinking the array size by using `distinct` as the *simulated* length in your iteration steps.
- Call the new methods from above `findMin` and `extractMin` respectively. `findMin` takes an array of integers `weights` and the integer `distinct` and returns the *index of a minimum value* in `weights`. `extractMin` takes an array of integers `weights`, and the integer `distinct` and returns an integer representing the minimum weight. `extractMin` calls `findMin` as a subroutine. In addition, `extractMin` performs the little trick above. Notice that `extractMin` does not itself decrement `distinct` --- you'll have to do that yourself after calling `extractMin`.
- `huffmanSize` should compute the Huffman cost by repeatedly calling `extractMin` until only a single weight remains in the array of weights. As should be clear by now, `huffmanSize` returns a `double`.

Using the Starter Project

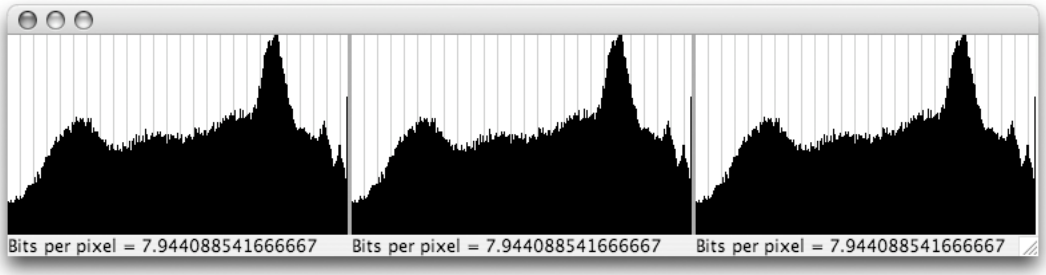
We want you to incorporate your implementations of the simplification algorithms described above into a program that will allow you to compare their behavior by systematically applying the algorithms to a variety of images. The interface for this program will resemble the interface of the program you wrote last

week in many ways. Therefore, rather than having you repeat much of the work you did last week by writing the code to implement the interface from scratch, we will provide a starter file containing all of the code needed for the interface, together with several of the image filter classes you used or implemented last week. To prepare you to work with our starter project, we will first describe the interface the program provides. Then, we will provide a brief tour of the classes included in the starter project.

Overview of the Program's User Interface

The the main class is in the program provided in this week's starter project is named DualImageViewer just like the class you defined last week. When you create a new instance of this class, it will display a window very similar to last week's program. The window will provide room to display two images side by side, but it won't provide quite as many controls for modifying the images as last week. The sliders are gone. Instead of having a separate "Load Button" image for each half of the window, there will only be one button that will load a new image into the left side of the window.

Although the "Show Histogram" button is still included it will work a bit differently. When you press this button it will display a window containing separate histograms for the red, green, and blue components of the image as shown below. In addition, below each histogram the program will display the bits required per pixel to encode the corresponding pixel array using a Huffman code.

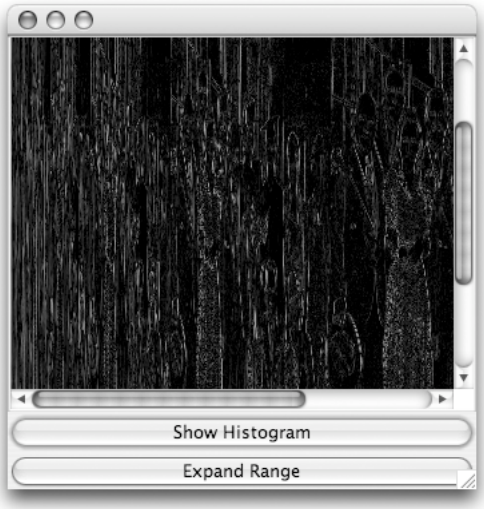


(Well.... Actually, while we will provide the code to display the three histograms and the bits per pixel, you will have to write the code to calculate the number of bits required per pixel.)

The buttons to copy or paste images from the left side to the right will be replaced by a button that will apply a simplification algorithm to the image on the left. In addition, there is a menu in the lower right corner of the window that can be used to select the simplification algorithm that you wish to apply. The

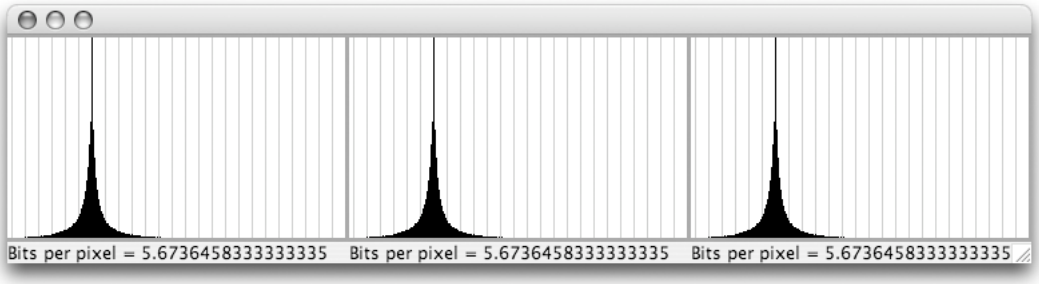
image shown above shows a menu that includes all of the algorithms we want you to compare. The version of the program provided in the starter project will only include the "Range Reducer" algorithm in the menu. You need to implement and add the others.

When the "Simplify/Unsimplify" button is pressed, the program will display two new images. It will create a new window like the one shown on the right containing the "simplified" version of the original image. In this case, we applied the recursive wavelet algorithm, so you may be able to see a series of increasing narrow drawings of the edges of the objects in the original image in the simplified version.



The program will also display the result of reversing the process by "unsimplifying" the simplified image on the right-hand side of the main window as shown below. With this particular image, it will probably be difficult for you to see any differences between the images in the sample window below, but since the algorithm is lossy, with some example images there will be noticeable "artifacts".

Note that the window in which the simplified image is displayed contains its own "Show Histogram" button. Pressing this button provides a way to see how effective the algorithm has been. In particular, it provides a way to determine how many bits/pixel are required to encode the simplified representation of the image. A sample of the histograms for the simplified image shown above is shown below.



Classes Provided

Image Filter Classes:

We provide all but one of the class you will use for filtering `SImage` objects.

- `ImageFilter`
- `Expander`
- `Paster`
- `Scaler`
- `Differencer`

Most of these are identical to the ones you implemented or received last week --- we're just giving you our versions. The only new filter provided is `Scalar`. It rescales the pixel values of an `SImage` by multiplying them by some rational number. As an example, `RangeSimplifier2` uses `Scalar` to scale the brightness bands of an image down for compression and then scale the brightness bands up (i.e. expand the brightness bands) during decompression.

Histogram Classes:

`Histogram` and `DisplayHistorgams` are the same as last week except that `DisplayHistograms` now displays three histogram images --- one for each color band --- along with the average bits per pixel when the data that generates such a histogram is compressed using Huffman's algorithm. You will need to add code to `Histogram` to compute the compressed image cost.

Image Simplifier Classes:

- `ImageSimplifier` is the base class of all simplifiers.
- `RangeSimplifier1` implements a very basic simplifier that decreases the bands of brightness available to an image. As you might expect, this results in some compression. The basic range simplifier also decompresses the image by expanding the range back out.
- `RangeSimplifier2` is identical in functionality to `RangeSimplifier1` except it uses `Scaler` to decrease the bands of brightness when compressing and expand the range of brightness when decompressing. This is a good example of how your recursive wavelet simplifier might look.

GUI Interface Classes:

`ImageViewer` and `DualImageViewer` are similar to the classes with the same names from last week's implementation, but the layout has changed slightly. You should read through these classes and note the changes. You should not need to change ImageViewer at all during the lab. The only change you will make to DualImageViewer is to add additional entries to its menu of simplification techniques as you implement them. All you will have to do to add a new entry to this menu is add a line of the form

```
addSimplifier( name-of-simplification-technique, simplifier );
```

to the DualImageViewer constructor. For example, the line we have included in the constructor to place our range simplifier in the menu looks like:

```
addSimplifier( "Range Simplifier", new RangeSimplifier1( 32 ) );
```

The `addSimplifier` method both adds an entry to the menu and places the `ImageSimplifier` provided in a collection of simplifiers implemented using a recursively defined class named `SimplifierList`. When a new menu item is selected, our code searches this list to determine which simplifier should be applied.

Getting Started

Download the starter file `Lab9Starter.zip` from the course website http://www.cs.williams.edu/~cs134/s07/labs and unpack it in your `Documents` folder.

Report and Experiments

In addition to the source code for a working program, you will also submit a written lab report. This lab report need not be long (1-2 pages will be sufficient if you are concise), however it needs to be clear and address the questions listed in this section. As with any piece of writing, you should use clear formatting and follow normal English grammar and spelling. However, this is not great literature--you can use repetitive sentence structure, have minimal transitions between sections, and use footnotes to clarify your points.

Because this is a technical document, it is important to be precise, using mathematics, data tables, and diagrams to support your claims. It is also important to be objective. In arguing the merits of one compression method over another you must stick strictly to the facts. However, you are welcome to speculate as long as your speculation is clearly delineated and follows a clear line of reasoning.

The centerpiece of the report will be your data table. It should list the compression ratios that you observed for each method on each image found in the `Compression` folder of the `AllImages` folder, the relative complexity of each method, and the subjective quality of the compressed image. Note any artifacts (distortions) that you observe in the compressed images. You are invited but not required to construct your own test images that may reveal weaknesses and strengths of the algorithms.

Specifically address the following questions and topics in the report:

- Which algorithm is best?
- Describe other test images that would be good for testing the properties and limitations of simplification algorithms.
- What constants are used in the algorithms that affect the quality and efficiency?
- What kinds of images compress well? What kinds of images compress poorly?
- Sketch out a new image simplification algorithms (you don't have to write code; just describe an idea). How do you think this will perform compared to the ones you experimented on?

Even if you are unable to complete the code for the lab (or have a few errors left), you should still submit the report. In this case, either discuss only the algorithms that you completed, or borrow a friend's completed implementation.

Submitting Your Work

As usual, make sure you include your name and lab section in a comment in each class definition. Find the folder for your project. Its names should be something like `FloydLab7`.

- Click on the Desktop, then go to the "Go" menu and "Connect to Server."
- Type "cortland" in for the Server Address and click "Connect."
- Select Guest, then click "Connect."
- Select the volume "Courses" to mount and then click "OK." (and then click "OK" again)
- A Finder window will appear where you should double-click on "cs134",
- Drag your project's folder into either "Dropoff-Monday" or "Dropoff-Tuesday".

You can submit your work up to 11 p.m. two days after your lab (11 p.m. Wednesday for those in the Monday Lab, and 11 p.m. Thursday for those in the Tuesday Lab). If you submit and later discover that your submission was flawed, you can submit again. The Mac will not let you submit again unless you change the name of your folder slightly. Just add something to the folder name (like the word "revised") and the re-submission will work fine.

Grading

Completeness (14 points) / Correctness (6 points)

- Image simplifier algorithms are correctly implemented
- Simplifying / Unsimpliying an image creates a new compressed image in a new image viewer and displays the uncompressed image in the right image viewer
- Histograms are properly displayed for each of the three color bands
- Histograms give the correct bits per pixel for each color band
- The `huffmanSize` method is correctly implemented

Style (10 points)

- Commenting
- Good variable names
- Good, consistent formatting
- Correct use of instance variables and local variables
- Good use of blank lines
- Uses names for constants