Project 1b: Image Processing

CS Y11

Saturday 2nd March, 2019

CLICK HERE FOR FILES NEEDED

In this homework, you'll add functionality to a photo editor! This will provide practice with 2D data structures, structures that use Objects of different types, and with adding code to a large-ish code base. These are all essential skills in programming and we hope that the assignment provides a fun way to practice with these topics.

The assignment also includes an algorithm for "seam carving" (i.e., removing a column of pixels), which is an example of using dynamic programming (or DP). DP is a problem solving strategy to take naturally recursive solutions and avoid duplicate calculations by working from the base case of a recursive algorithm and progressively building up to the solution you want. If you want to know more, I can give students who choose to do this assignment a quick presentation on it.

You will gradually extend the functionality of the program by writing several methods in the Picture class within the Picture.java file. Make sure to test the methods you write using the provided JUnit test cases along the way!

1 Setup for Photo Editor

1.1 Read about Seam Carving

Seam carving was first published in this paper by S. Avidan and A. Shamir in 2007. It's a relatively accessible paper – take a look! That said, all of the background needed should be available from lecture and this write-up.

Seam carving finds the "least-noticeable" seam from top-to-bottom or left-to-right in an image. Then, repeated seam removal allows for image-aware resizing to any smaller size. As with all of CS, this requires quantifying what is meant by "least-noticeable"! For us, "least-noticeable" refers to the shortest path through the image, where distance is defined to be the total strength of the image edges traversed along the way. The paper offers a seam-insertion algorithm for resizing to larger image sizes, too, though that won't be an official part of this assignment.

You'll have a number of steps you'll use and you'll likely find the following picture of the steps helpful! NOTE: This photo is Micro.bmp :-)

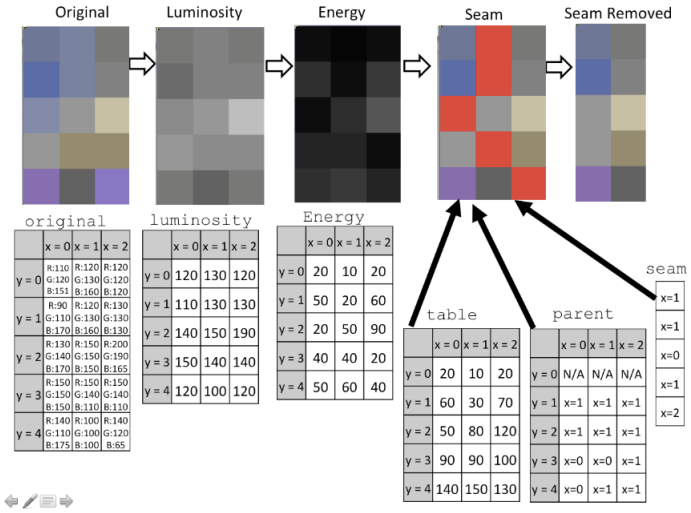

Figure 1: Micro.bmp

1.2 Get familiar with the structure of the files

In the provide starter files for this homework there are many images (used for testing) and a number of .java source files. The only .java file you'll need to edit is Picture.java. The other ones you'll use are PictureTest_ <something>.java (for running the JUnit tests) and Pixel.java (for accessing the images' pixels). The other files are listed here, but knowing their details are not important for completing this assignment. Here is an overview of all of them:

Files to edit:

- *Picture.java:* This class is a subclass of the SimplePicture class, and has placeholders where you'll implement various pixel-processing routines.

Files to use:

- *Pixel.java:* This class provides static and non-static accessor methods that allow you to obtain information regarding a pixel in a picture. This also uses Java's built-in Color class. Here is the online reference for that Color class.
- *PictureTest_ <Something>.java:*These classes contains the JUnit test cases that test the functions in the Picture class. You won't need to write additional tests, but you will want to use these tests.

Other files: (you don't need to look inside these; if you want to do so, see the extension...)

- *FileChooser.java*: This class controls and abstracts the file choosers that allow the user to select files to open and save.
- *ImageDisplay.java*: This class controls and abstracts the image display and the crosshair feature.
- *PictureExplorer.java*: This class brings together and controls the various elements of the graphical user interface.
- *PictureFrame.java*: This class controls and abstracts the frame containing the picture.
- *SimplePicture.java*: This class is the parent of the Picture class, and also stores variables and methods relating to a particular picture file, allowing the Picture class to deal only with the various features that can be implemented on that picture file.

1.3 Download and Run the starter files.

Find the Picture.java file. Right click on the file and run it. This program should show up on you screen.

- Try out loading a new image with File - Open. You'll have to navigate to the project1d_ imageProcessing folder the first time, but for additional images it should start wherever you left off the previous time. Okinawa.bmp and Camel.bmp and several others are available. Also, try your own images or ones you grab online and download to your Desktop/Downloads... the images do not have to come from the Java source folder.
- Also, try zooming in/out. If you use the 5000 % zoom on anything but a very tiny image, it will likely run out of memory - this is Ok.
- The only image-processing method that works within the starter code is the Change colors - Grayscale. Try that, too.
- File - Reset Picture tries to reload the file with the same filename as the last image displayed. If the last image displayed did not have a filename, you'll have to use File - Open... instead.

  The rest of the options are the rest of the project – or the extension!

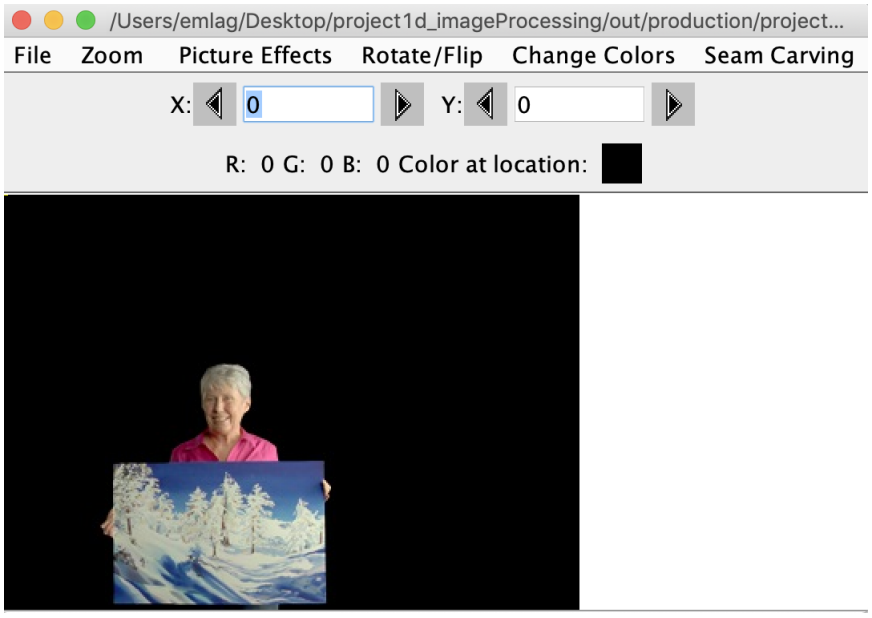

Figure 2: Initial program

2 Part I

2.1 Warm up: Find the Secret Message

We have two photos (Maria.bmp and SecretMessage.bmp) that look almost identical, but if we compare them - you can see a secret message!

- In Picture.java **main** method, we load in the photo Maria1.bmp:

```
Picture pic = Picture.loadPicture("Maria1.bmp");
```

- Add code to the **main** method to load in the photo SecretMessage.bmp.
- Create a new photo by calling the method **showDifferences**. Note: You need to look at the method to figure out how to call it.
- Look at the new photo you created in the previous step using the method **show()**.
- Write a description of the secret message in a comment at the top of Picture.java

Make sure to look at the FAQ section at the end of this document!

2.2 Write color-changing methods

Background: In all of our images, whether color or grayscale, each individual pixel has:

- a redness level (0-255, where 0=no redness and 255=maximum redness)
- a blueness level (0-255)
- a greenness level (0-255)
- an opacity or opaqueness (0-255, where 0=completely transparent and 255=totally opaque).

We will only use the red, green, and blue components; opacity will always be 255. Shades of gray arise when you have equal amounts of red, green, and blue, with black containing 0 for all three and white containing 255 for all three.

The grayscale2 and grayscale methods convert from RGB to gray by computing the average of the three. They also show how to use many of the methods in the Picture class (for example, the

constructor to copy an image) and in the Pixel class, e.g., the way to access individual components of a Pixel or to access a whole Color object that, in turn, has all of the color components needed.

You will implement the methods whose signatures appear below. To implement the first six methods, adapt one of the two approaches to computing grayscale, either **grayscale2** or **grayscale** (with helper methods). Be sure to try out your image transformations in the graphical user interface (GUI). Also, be sure to test your implementations with PictureTest_ ColorChange.java

It is really important within this set of methods to reuse as much code as possible! If you find yourself with many copy and pasted versions of the same code - you should look for a way to avoid this duplicated code (Note - because these methods - other than negate()- are very similar to grayscale, almost all of the points within this section are allocated to avoiding this duplication of code using a helper method! For **lighten**, **darken**, **addBlue**, **addGreen**, and **addRed**, you shouldn't need to copy and paste the for loops!)

Here are the signatures of these methods that you 'll implement.

- **public Picture negate()**
  should return the "photo inverse" of this Picture
- **public Picture lighten(int lightenAmount)**
  returns a new Picture that is lighter in red, green, and blue by lightenAmount (maxing at 255)
- **public Picture darken(int darkenAmount)**
  similar for creating a darker version of thisPicture
- **public Picture addBlue(int amount)**
  should return a new Picture with only the blue channel changed (keeping r,g,b values between 0 and 255, inclusive)
- **public Picture addGreen(int amount)**
  should return a new Picture with only the green channel changed (keeping r,g,b values between 0 and 255, inclusive)
- **public Picture addRed(int amount)**
  should return a new Picture with only the red channel changed (keeping r,g,b values between 0 and 255, inclusive)
- **private int luminosityOfPixel(int x, int y)**
  returns the luminosity of this Picture at Pixel(x,y). Implement this before luminosity(). Here is a visual example showing the difference between uniform averaging and luminosity. Luminosity is a weighted average of red, green, and blue in which the weights take into account that human vision has much better visual acuity within some color frequencies (e.g., greens) than in others (e.g., blues): luminosity = (int)(0.21 * redness + 0.72 * greenness + 0.07 * blueness);

  Warning! The order in which you perform the luminosity calculation, above, matters! Be sure to use the order above (adding red, then green, then blue) with the coefficients shown. Small floating-point errors can accumulate and, once in awhile, they will change a pixel value by 1 gray level – not enough to see, but enough to cause a test to fail...
- **public Picture luminosity()**
  should return a different grayscale image, using the luminosity calculation described above.

When you complete this part, all of the tests from PictureTest_ ColorChange.java should pass. You might find the method printLuminosity() helpful because you can compare the output to the correct output shown in class.

2.3 Do backflips with 2D Structures

Implement the following rotation method:

- **public Picture rotateRight()**
  returns a brand-new Picture that holds this Picture, only rotated right by 90 degrees. Thus, if the original Picture was 50x70, the the new image will be 70x50.

For the rotateRight method, you will need to use a different one of the Picture class's constructors in order to create a brand-new image of a different size. Here is an example call:
Picture myPicture = new Picture(newWidth, newHeight);

Of course, this presumes that you have defined newHeight and newWidth beforehand! From there, you will want to loop through all of the pixels, copying their color from the old image (this) to the appropriate place in the new image (whatever you've named it). Be sure to return the new image in the end! Personally, I looped through the destination picture's pixels and - for each one - computed the location from which to grab the source pixel that corresponded to it. The reverse works equally well.

Note that objects of type Pixel have methods that will get and set all of the components of a color at once! They use Java's Color class, e.g.,
Color c = some_ pixel.getColor();

some_ other_ pixel.setColor(c);

or even the equivalent

some_ other_ pixel.setColor(some_ pixel.getColor());

When you complete this part, all of the tests from PictureTest_ Rotate.java should pass.

3 Part 2

3.1 Compute Energy

Implement the following methods for energy:

- **private int getEnergy(int x, int y)**
  is similar to luminosityOfPixel but returns the edge-energy of this Picture at Pixel (x,y).
  Implement this **before** energy(). A lot of people have the problem that they call luminosity() H*W times, but luminosity() already requires H*W steps!

- **public Picture energy()**
  should return the edge-energy image, as described below

The key idea in seam-carving is finding the edge-energy-minimizing path through the image. But what is this "energy"? Intuitively, the strength of the edges around a particular pixel is that pixel's "energy." High-energy pixels are ones at a strong edge in the image – those are pixels we're most likely to notice if they disappear. Low-energy pixels are those in the middle of uniform patches of color or gray – those are least likely to be missed! To quantify this idea of "energy," we use simple approximations to the image derivative, which is simply how fast the luminosity is changing between adjacent pixels. If we abbreviate the luminosity at point (x,y) as i(xy). Moving horizontally, we can numerically approximate the **horizontal derivative** (rate of horizontal change in intensity) at (x,y) with any of three methods:

- The *forward difference*

$$i(x+1, y) - i(x, y) \tag{1}$$

- The *backwards difference*

$$i(x, y) - i(x-1, y) \tag{2}$$

- The *central difference*

$$2i(x+1, y) - i(x-1, y) \tag{3}$$

Vertical derivatives are similar. For example the vertical central difference would be computed as

$$2i(x, y+1) - i(x, y-1) \tag{4}$$

Although the central difference is the most accurate approximation numerically, Avidan and Shamir (the paper authors) simply used the forward energy, which is what we will do. The Seam Carving paper defines the energy at each pixel to be the sum of the absolute value (Math.abs) of two derivatives: the horizontal and vertical rates of change in luminosity. Specifically, you should use the forward difference for both the horizontal and vertical directions:

$$energy = abs(i(x+1,y) - i(x,y)) + abs(i(x,y+1) - i(x,y)) \quad (5)$$

except when the pixels are on the edge of the image so that they do not have a "forward" neighbor, either x+1 and/or y+1. Whenever one of those cases happens, your implementation should use the backwards difference. For one pixel (the lower right corner), you'll need to use two backwards differences.

When you complete this part, all of the tests from **PictureTest_Energy.java** should pass.You might find the method **printEnergy()** helpful because you can compare the output to the correct output shown in class.

You might find the method **showDifferences(Picture p)** helpful. It makes differences between two pictures red. You can view that picture on the screen using **explore()**.

```
   public static void main(String[] args)
{
   Picture picTiny = Picture.loadPicture("Tiny.bmp");
   Picture picCorrect = Picture.loadPicture("TinyEnergy.bmp");
   Picture picTest = picTiny.energy();

   Picture picWithDifferences = picCorrect.showDifferences(picTest);
   picWithDifferences.explore();
}
```

3.2 Compute Seam

Implement the following method to find a seam:

- **private int[] computeSeam()**
  should return an array of integers representing the location of the least-noticeable seam in this Picture. This is the crucial subroutine and the DP algorithm this week! See below for a pretty extensive explanation.

  You might find the debugging methods **printArray(int[][] A)** helpful.

This is the dynamic-programming part of the assignment. A vertical seam is a sequence of pixels, one per row of pixels, such that pixels in adjacent rows are no more than one column apart. Since there's exactly one pixel per row, we only have to remember the columns of each pixel; this can be represented as a single array of integers.

Step one is to create an array table of integers, of the same size as the image. Each entry table[x][y] is the total energy of all pixels in the least-energy seam starting on the top row and moving down to the ending at pixel (x,y). It is sometimes called the cumulative-energy table (or cost table): it does not contain single energies, but the smallest cumulative energy starting from anywhere at the top and ending at pixel (x,y). As discussed in the paper, we can define the entries in table recursively by at is, the least-energy seam ending at (x,y) extends the best seam ending

$$\begin{aligned} \text{table[x][0]} &= \text{getEnergy[x][0]} \\ \text{table[x][y]} &= \text{getEnergy[x][y]} + \min \begin{cases} \text{table[x - 1][y - 1]} \\ \text{table[x][y - 1]} \\ \text{table[x + 1][y - 1]} \end{cases} \end{aligned}$$

either directly above-and-to-the-left, directly above, or directly above-and-to-the-right. (Note that for pixels on the far left and far right borders, only two of these three possibilities exist.)

The dynamic programming approach is to fill out this table in order of increasing y: first all the entries where y=0, then the entries where y=1, etc.

We also need code to keep track, at each pixel, which of the 3 possible parent seams was chosen. This means creating a second 2D array, perhaps named parent. The idea is that parent[x][y] will hold the column number (x-value) of the previous-row's pixel in the best seam ending at (x,y).

For Example, if we had a 3-by-3 image whose energies were

1	2	3
8	6	4
5	6	5

then `table` would be

1	2	3
9	7	6
12	12	11

and `parent` would be

X	*X*	*X*
0	0	1
1	2	2

The top row of the parent array doesn't matter, since there are no more pixels above. Slightly more generally, if while computing table[x][y] we decided the smallest of the three possible parent locations was table[x+1][y-1], then parent[x][y] should be set to x+1. If the smallest of the three possible parent locations was table[x][y-1], then parent[x][y] should be set to x. And so on.

Finally, when we have filled out the table and parent arrays, we can find the best seam. You won't be surprised that several loops are involved in this!

First, we loop across the bottom row of the table array. As we do so, we find the smallest table value in that bottom row (where y == height-1). The location you find has the minimum cumulative energy and will be where the seam ends. Then, using the parent array, you can determine whether the seam extends up-and-left, up, or up-and-right. Then, you continue working your way up, until reaching the top row.

As this process happens, you can collect each of the column (x) values for the pixels in the seam at each row. A one-dimensional array whose length is the height of the picture is ideal for this: the index into the array is the row number and the contents of the array is the column location of the seam at that row. (Think of it as a "column" array, though it doesn't really have any preferred direction.) My implementation called this array **seam**.

Continuing from the 3x3 example above, the best seam ends in the lower right corner, in column 2, because that's the smallest value of table in the bottom row. Thus, seam[2] = 2. Working backward through the parent array, we find that the best north-neighboring pixel in row 1 (the one with lowest cumulative energy) is also in column 2, so seam[1] = 2. From there, the best north-neighboring pixel in row 0 is in column 1, so that seam[0] = 1. Thus we get the seam (from top to bottom) 1,2,2, corresponding to original pixels that had intensities 2, 4, and 5.

Tiebreaking! We need a rule for breaking ties when defining the seam: more than one of the neighbors in the next-row-up may share the same minimum value!

If there is a tie when you're identifying which Pixel to select on the minimum-cost path, please select the Pixel with the same x value as the first choice (if it is the min or shares the min). Choose the Pixel to the left (x-1) as the second choice (if it's the min but the previous one is not). Finally, choose the pixel to the right (x+1) - if it's the only one with the minimum value. If there are multiple paths that have same, minimum cost: Select the path that has the minimum X value on the bottom row (i.e. the largest Y value)

Please make sure you understand what table, parent, and seam are before you start coding!

When you complete this part, you can test your code by:

- making computeSeam public (so it can be tested with JUnit)
- downloading the file PictureTest_ ComputeSeam.java
- Note - once youve got the tests passing - make computeSeam private again and remove the test file, which will no longer compile!

You might find it helpful to write a helper method so that your code isn't more than 50 lines of code. Here's an example of how you can pass an array as an argument and that method can write content in that array.

3.3 Show Seam

Implement the following method to show a seam on a Picture:

- **public Picture showSeam()**
  should return a new Picture, a modified copy of this, which includes the vertical seam (found from computeSeam) drawn in red.

This method should return a copy of this with the lowest-cost seam, as computed by computeSeam, shown in red. To do this, you'll probably want to start by (1) computing the best seam and (2) making a new copy of **this**. For instance, these lines will accomplish these two things:

```
int[] seam = this.computeSeam();
Picture newPicture = new Picture(this);
```

From there, you'll need to loop through and change some of the pixels of newPicture to be red – namely those on the seam! Here is an example from the okinawa.jpg image:

When you complete this part, all of the tests from PictureTest_ ShowSeam.java should pass.

3.4 Carve and Carve Many

Implement the following method to carve seams from a Picture:

- public Picture carve()
  should return a new Picture of a size one pixel less in width than this. The new Picture should be the same as this except with the seam (found from computeSeam) removed.
- public Picture carveMany(int numSeams)
  should return a new image similar to carve, but it should have removed numSeams number of seams. You'll want to call carve in a loop to implement carveMany.

The carve method is similar in spirit to showSeam, except that instead of coloring the seam red, it removed those pixels altogether. Because of this, you won't be able to use the Picture constructor that copies this, because your newPicture has to have a different size! In this case, use the brand-new Picture constructor, e.g., starting with:

```
Picture newPicture = new Picture(newWidth, newHeight);
```

where you'll need to define newWidth and newHeight based on the height and width of this! From there, you'll need to copy pixel colors from this to newPicture. Notice that we always copy the colors of Pixels, not the Pixels themselves... . However, in doing this, your code will need to be careful to skip the seam's pixels and to shift over all of the pixels to the right of the seam by one pixel place to the left. For the carveMany method, you will want to loop numSeams times, carving once each time! Note that you can write a line such as this:

```
Picture p = this;
```

and then re-assign p to the result of p.carve() the correct number of times!

When you complete this part, all of the tests from PictureTest_ Carve.java and PictureTest_ CarveMany.java should pass.

4 Frequently Asked Questions (FAQs)

Take a look at these FAQs before you get started. We hope that this will encourage you to check these when you get stuck!

Q I keep getting the following error: *Uncaught error fetching image: java.lang.NullPointerException.*

A This error message arises because some, or all, of the images are not in the same folder as the result of your code. Please place any additional images files in the same place!

Q I keep getting the following error: *Exception in thread "main" java.lang.OutOfMemoryError: Java heap space.*

A This error message arises when you are trying to load too big a file. Given the current framework, the project works best with moderately-sized images.

Q I tried to compare two images together using the equals method in the *Picture* class, and even though they are obviously visually the same, the equals method returns false.

A Check to see whether one of the image files is a JPEG file. The equals method fails on JPEG files because JPEG files use lossy compression to store images: in other words, they compress an image to store it, but doing so causes information to be lost. Since the equals method attempts a perfect information match, it does not work well with JPEG files. This problem will not arise with GIF, BMP, and PNG files, since these file types involve lossless compression.

Q I run my program many times and after a while eclipse starts to get really slow.

A Click on the red square to stop each program individually.

Q Any tips on these two dimensional arrays?

A Here is a helpful resource for Arrays: Here is a quick-summary reference on Java Arrays and here is Java's official (longer) tutorial on arrays.

Q If there is a tie for a minimum path, which path should we pick?

A By default, please pick the cell above in a tie, if there is only a tie between the left and the right pixels, please pick the left pixel.

Q My energy is off by just a few pixels! What is wrong?

A Inside the *energy()* method you'll be working with two different *Picture* objects. The object the method was called on (*this*) and the one you plan to return. When you call the method *getEnergy(x, y)* make sure you call it on this and not the *Picture* object you plan to return because the contents of the object *this* won't be changing...

Q My *showEdges* method does not pass the tests, even though my resultant picture and the test picture look exactly the same. What can I do?

A If you are using the *Pixel.colorDistance* method, cast its result to an *int* before comparing it with the threshold.

Acknowledgments to Harvey Mudd College for this lab.