CS245
Assignment 1: Image Mosaics (15 points)
Due: March 3 (Thurs) by start of class

This is a solo assignment – you are to work by yourself.

Image Mosaics

In this assignment you will be implementing a program to create image mosaics. It will read in an image and make a mosaic from it (see Fig1). It will also read in a set of tile images (see Fig2). And it will use the tiles to create and write out a mosaic of the original image. I will provide the photo I used in Figure 1. Use it for testing purposes to make sure your program is generating the same output images (but only for testing purposes – you will also need to select your own image to use in this assignment).

Fig1: Original image and Mosaic image

Fig2: Four (out of 115) tile images (Feb and March 2006 DVD releases)

Images
You will need an image for your original image as well as a set of tile images. You may pick whatever images you wish for this assignment. I suggest that you pick a fairly large image for your original image. The final image will be resized so the exact size of the original image will not matter that much. For your tile set, you should have a fairly large number of images to pick from (over 100) in order to make a good mosaic. The set I used is available for download. It is a set of 115 DVD releases for February and March 2006. Feel free to use this set of tiles or to use your own.

Coding
You will organize your code into a single package named edu.uwec.cs.yourusername.program1. Within this package you will create several classes (discussed below). Your code will be further divided into methods within these classes to make the development easier and more modular.
Main Class

Main method

The only class with a “main” method is the one named “Main.” Within Main, your main method will initialize variables and coordinate calling methods in other classes. This is what will be in your Main class (nothing else – however you will need to change the originalFilename and mosaicFilename paths to reflect the name of your package and files):

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // Define the names of the files/folders and the number of tiles to use in the mosaic
        String originalFilename = "NYCcentralPark.jpg";
        String mosaicFilename = "NYCmosaic.jpg";
        String tileFolder = "dvdReleases";
        int horzMosaicSize = 640;
        int vertMosaicSize = 480;
        int horzNumTiles = 64;
        int vertNumTiles = 48;

        // Load in the tiles, resize them, and compute their average colors
        MosaicMaker mm = new MosaicMaker(tileFolder, horzMosaicSize, vertMosaicSize, horzNumTiles, vertNumTiles);

        // Read the original file which we will turn into a mosaic
        UWECImage originalImage = new UWECImage(originalFilename);

        // Form a mosaic of the original image
        UWECImage mosaicImage = mm.createMosaic(originalImage);

        // Write out the mosaic version
        mosaicImage.write(mosaicFilename);

        // At this point we could mosaic another image to the same size as the one above if we wished...
        System.out.println("Finished");
    }
}
```

Main is in charge of the overall steps involved in creating a mosaic. I will describe what those steps are below, but keep in mind that you will probably want to code and test these steps one at a time to be sure that each sub-step is working properly before trying to combine them together to form the full mosaic.

UWECImage Class

Note from the main that the UWECImage class is used to represent both the original and mosaic images. It will also be used internally to hold all the tiles images. Start with the UWECImage class from the convolution lab. It already handles loading and saving an image to a file, getting the size of the image, getting/settting the colors at pixel locations in the image, and creating a new blank image. Additionally, it can also draw the image onto an ImagePanel as you saw in the convolution lab.

This class give you basic image processing ability. However, you will eventually need to add several methods to this class in order to give it a bit more functionality. These methods will be discussed below.

MosaicMaker Class

Constructor

This class is in charge of making the actual mosaic from a set of tiles. It will obviously need a constructor. As can be seen from the main, this constructor should take in the folder name that contains the tile image. It also needs the size of the resulting mosaic as well as the number of tiles requested for the mosaic.
Since you are going to need to read in all the tiles images for any mosaic you make (with createMosaic used later), the constructor is a great place to load in all these images. UWECImage already contains a method that can be used to read in a single image. What we need for this constructor is to read in every image in the given tileFolder. In order to discover the names of all the files in the folder you should use the File object. You can make a new File object that points to any given File or Folder. Then you can ask it various questions about the file or folder. For example you can ask it to list all the files in the folder and it will hand back an array of Strings containing their names. Please check out the API for File to see how this works.

Once the images have been read in they need to be rescaled to the proper size. Actually, it makes the most sense if you scale them one at a time as you read them in. You know the requested size of the final mosaic as well as how many tiles need to go on that mosaic. From that you can calculate the size of the individual tiles. They should all be the same fixed size.

**UWECImage Scaling**

Currently UWECImage does not have the ability to scale. This will be something you will need to add to the UWECImage class:

```java
public void scaleImage(int newWidth, newHeight)
```

This method should scale the internal BufferedImage of the UWECImage so that it has the new width and height given. Since it is a mutating operation, it does not need to return anything.

So how exactly does one scale an image? The first step is to create a new BufferedImage where your scaled copy will go. UWECImage already has an example of doing this in one of its constructors. At this point you will have two BufferedImages -- the original and the new blank one.

Next you need to calculate the scaleX and scaleY factors which can be used to figure out the mapping between the original image and the new image.

```java
scaleX = originalWidth / newWidth;
the scaleY is similar.
```

You will then need a double loop to walk through the image and move all the pixel colors over to their new location. This is easiest if you loop through the new image. That way each time through the loop you can set the color of a single pixel in the new image. You can use BufferedImage’s getRGB and setRGB to do this. There are examples of these methods in action in the given UWECImage class. The only thing left to do is to calculate where the color sample should come from in the original image. This is where the scaleX and scaleY come into play. You can simply take the x location in the new image and multiply it by scaleX to get the x location in the original image. This is called “nearest neighbor” sampling. It is quite fast, but does not produce the best results if doing large scales, especially upwards in size. It will be acceptable for our purposes.

Once you have read in and scaled a single tile image you will want to save it in an ArrayList so you can access it later when making the actual mosaic.

**UWECImage AverageColor**

Additionally, you will also want to compute this tile’s “average color” since this will be necessary for creating a mosaic as well and it makes sense to only compute the “average color” for a tile one time. Computing the average color of an image (or section of an image) is also something that UWECImage should be able to do for us. Currently it does not have this ability so you will need to add it.
Thinking ahead a bit, when the mosaic is created we will need to find the average color of a sub-section of a UWECImage. This is the general case. The current case of finding the average color for the entire image is just a special case where the sub-section is (0,0) through (width,height). Thus, we will need two `averageImageColor` methods:

```java
public Color averageImageColor()
public Color averageImageColor(int startX, int startY, int width, int height)
```

The first one should simply call the second one with the correct parameters to capture the entire image as the sub-section.

In the multiple parameter version, you will loop through all the pixels in the given region of the image and compute an average color for that region. The region starts at pixel (startX, startY) as its upper-left corner and extends for ‘width’ pixels horizontally and ‘height’ pixels vertically. If the specified region lies outside the bounds of the given image you should tell the user with a `System.out.println` and perform a `System.exit`.

Computing the average image color is simple. You loop through all the pixels in the region and compute an average of their colors per channel. That is, you add up all the red values, green values, and blue values separately. Then you divide by the total number of pixels in the region. This gives you average red, green, and blue values. Then a new average color can be formed by creating a new `Color` and passing the red, green, and blue averages to this constructor.

**Creating the Mosaic**

At this point, you have added some additional functionality to the UWECImage class. You have also written the constructor for the MosaicMaker class. Next comes the actual mosaic creation. As you can see from the main MosaicMaker also has a method called `createMosaic` which takes in the original image to mosaic.

The first step is creating the mosaic will be to scale the original image to the requested size (which you should have saved as instance variables from the constructor). Once you have resized the original, you also need to create a new blank UWECImage to build the mosaic from with this same size.

If you like to see what is happening visually you can now create an `ImagePanel` with the new blank UWECImage so it will show up in a GUI window just like in the convolution lab. You will also have to call `repaint()` on this GUI object each time you make changes (below) to the mosaic you are building.

Next write loops to go through the entire original image and examine it one rectangular section at a time. For each sub-section of the original image you need to compute its average color. Now you have what color the section wants to be. You also have a list of all the colors that each tile represents. You then need to find the best matching tile color to this color (more on this in a minute). Once the best tile has been selected you need to use nested loops to copy the pixels of that tile over to the section of the new mosaic image you are building. UWECImage can’t do this for us, so we will need to do it one pixel at a time in a loop.

**Finding the Best Tile**

The idea for finding the best match is to compute a measurement of the “distance” between the mosaic color and each tile color. The tile with the shortest distance is the best match.

You already know how to compute the distance between two 1D points. Subtract them and take the absolute value. Or put another way, subtract them, square them, then take the sqrt. That is, `sqrt((x1 – x2)^2)`. You also know how to compute the distance between 2D points. Just think of Pythagorean theorem. If `p1 = (x1, y1)` and `p2 = (x2, y2)`, then the distance between them is simply `sqrt((x1 – x2)^2 + (y1 – y2)^2)`. Note that this is...
exactly the 1D formula with the addition of the y component. And this expands the same way for high order distances such as 3D or 4D in the obvious way. Why is this important? Because we need to compute the “distance” between colors which have 3 components.

However, distance in the RGB color space is not the ideal way to do this. RGB is a great color space for displaying colors to the screen since monitors have red, green, and blue color “guns”. But for other purposes it isn’t so great. For example, when printing we convert to CYM (cyan, yellow, magenta) color space because those are the colors of the ink cartridges. As a side note, screens use RGB because they start out black and you “spray” color on them to turn them white, whereas printers use CYM because the paper starts out white and you “spray” color on it to turn it black. If printers used red, green, and blue ink then we would need to feed in black paper!

The human eye uses yet another color space. Since we are trying to make matches that will look similar to human eyes we should probably switch to different color space for this project as well. We will use the HSB (hue, saturation, brightness) color space. This still isn’t exactly what the human eye uses, but it is much closer than RGB and there is an automatic converter built into Java you can use.

- **Hue**: This is "color" how we see it. This value ranges from 0 to 359, starting with Red (0) and going through Orange, Yellow, Green, Blue, Purple and coming back to Red again (359). As you can see, this puts the color in a rainbow-like pattern that is very easy to use.

- **Saturation**: A colored spot can have a variation on the amount of color applied to it, that is, how vivid the color is. This is given by a percentage, where 0% means no color (White) and 100% means "normal" (very vivid).

- **Brightness**: Finally, the color spot can be classified as how dark or bright it is. This is also a percentage, where 0% means it is very dark (Black) and 100% means it's "normal".

Check out the method RGBtoHSB in the Color class to switch to this color space. Once in this color space, we can now perform our 3D distance calculation – with hue, saturation, and brightness being the 3 channels.

Of course, the above only gives us the distance between the mosaic color and the color for one tile. I suggest starting out by finding the index to the single best matching tile. Note that this may select the same tile again and again. For example, the sky in the example image might all end up being the same tile again and again. This doesn’t look particularly pleasing to the eye, but it is a good starting point.

I would suggest writing a helper method called findBestTile which takes in a Color (the mosaic color) and hands you back the index of the tile that is the best match:

```java
public int findBestTile(Color mosaicColor)
```

Once you have the single best tile working you can upgrade to the following more complex method. Although you should try to pick the tile with the best overall match to the sub-section being replaced in the original image, you also need to consider the overall picture effect. Mosaics sometimes avoid placing two identical tiles next to each other. So if the best match for a sub-section is a tile that is identical to one of its four neighbors
(left, right, up, down) then you will need to go down the best list to the next choice and try that one. This process continues until you have found the best matching tile that doesn’t conflict with any of its neighbors. Additional restrictions could also be made on tile placement, but you will stick with implementing the neighbor rule for this assignment. As before, loop through all of the tiles computing their distances from the subsection of the original image being considered. In the loop, store each distance and index in parallel ArrayLists. Something like this:

```java
    tileDistance.add(new Double(distHSB));
    tileDistIndex.add(new Integer(i));
```

When the loop finishes, sort the arrays by sorting the tileDistance ArrayList and swapping the entries in the tileDistIndex each time you swap entries in tileDistance.

Return the tileDistIndex ArrayList.

That is, instead of just finding the best match, you return a sorted list of best match to worst match.

You will also need to make modifications to the createMosaic method. My suggestion on how to do this is to create a separate array (2D) that holds the indexes of all the placed tiles. As you add a new section to the new mosaic image, add the index of the tile in the tiles ArrayList used for it. For example:

```java
    int[][] usedTiles = new int[horzNumTiles][vertNumTiles];
```

This should make it easy to determine if the tile you are about to place is in conflict with previously placed tiles. If it is, select the next best tile for the new mosaic image. Keep in mind, if you’re in the top row, you can’t compare to anything ‘up’, if you’re in the left column you can’t compare to anything to the ‘left”, etc.

### Adding Threads

The last step in this project is to add threads to increase the speed of the solving. In particular, the determination of which tile to select for each region in the image is almost completely independent of the selection of tiles for other sections. It is completely independent in the easier ‘pick the best match’ version. The ‘no similar tiles next to each other’ restriction adds a bit of a wrinkle to this independence.

For this assignment we will not worry about trying to match the number of threads we have to the physical number of processors on our system. Instead we will simply create a new thread for each tile that needs to be placed. Doing this requires a new class called `TileFinderThread` which implements `Runnable`.

The `createMosaic` method should now fork off a new thread to handle the finding of the correct tile for each sub-section of the mosaic. Of course, this method needs to return the final mosaic when done so it will need to wait for all those threads to finish doing their work before it can proceed with the return. This is very similar to your lab.

Much of the code you wrote previously can still be used. For example, you have a method called `findBestTile` in your `MosaicMaker` class. This does all the HSB and distance calculations. You can either move this method to the `TileFinderThread` class or you could pass a reference to the `MosaicMaker` (passing ‘this’) to the `TileFinderThread` and then simply leave the method in the `MosaicMaker` since the `TileFinderThread` will now have the ability to call those method. This second option is probably the best since it requires the least amount of work.

The other issue is that once the `TileFinderThread` has determined the sorted list of tiles that should be selected, it needs to somehow integrate this information with the results from the other threads so that no two tiles that sit
next to each other are the same. To do this, create a method in MosaicMaker called postTileChoice. It is this method that should have the code which takes the list of possible tile candidates and decides by looking at the usedTiles grid which one to pick. Note that all the threads will be calling this postTileChoice method (possibly at the same time) since it is a shared resource. This isn’t an issue if all it does is read values. However, postTileChoice needs to change values in both the mosaic image and in the usedTiles grid. Thus you need to make sure it is properly synchronized.

**Submission**

You are to write a 1 page (max) cover page with your name on it and a paragraph or two explaining any problems your program has or (if it is working correctly) anything you had difficulties with or might want to do to improve it in the future. **Stapled** to your cover page will be prints of your source code, of your original image, and your mosaic version with both of your names on all of them.

Additionally, you **must** copy a JAR file with *only* your program1 jared package to your W: drive class folder. I will give more specific direction pertaining to the submission as the due date approaches.