Algorithms Seam Carver Write-Up

David Ericson

**Description**

Vertical seam and horizontal seam are basically mirrors of each other so I’ll explain the find and remove for vertical and any of the major changes that occurred to alter the algorithm to work for the respective horizontal counterpart.

For the find I did a topological sort of the image. I created a node class that held the node’s energy, parent, and total energy of the path. The total energies all start at Double.Max so that I can ensure that a less expensive path can be found. First, I create an array of indices at the top, which we will return later, as well as an array of my node class that is width\*height large. For vertical I did a single for loop that loops through all values from 0 to width\*height. I obtain the x and y of the current indexed pixel. To get x I mod the index by the width and to get y I divide the index by the width. I calculate the nodes energy using the x and y values. If the current index is less than the width then we set that node’s total energy to 0, since we can start there. Next I check to make sure we aren’t on the last row, index < width\*height – width. I then check to see if any of the possible next adjacent nodes would have a lower total energy path if I added the current node’s energy to the current node’s total energy, if they do I also change their parent to my current node’s index. I grab the next node to the left by first checking to make sure the index mod the width isn’t 0, to make sure we don’t wrap across the image. Next, I simply add the width to the index and subtract 1. For the next node down I just add the width to the current index. For the right adjacent node I make sure that the index + 1 mod width isn’t 0. To get the node’s index I add the width + 1 to the current index. When going through the very last row I check to find the lowest total energy by taking the node’s total energy plus its own energy.

Now I create my set of indices by backtracking from my least expensive indexed node and following his parents all the way up. I create the array of indices starting at the back and heading forward so that I don’t need to reverse the indices. I then return the indices.

Finding a horizontal seam is fairly similar. Calculating the indices at the end is exactly the same. I found it more helpful to do a double for loop for this horizontal seam business. I wanted to go down the rows before advancing to the next column so that I could guarantee that nodes would have their lowest energies by the time I reach the last column. This made most of the math easier. So I made a variable nodeIndex which equaled j\*height+I, this let me recycle a decent chunk of my vertical seam math. If i = 0 then we set the total energy to 0, since these are the starts. If i+1 < width then we aren’t on the last column, which means we can calculate the adjacent node’s total energy. This is done in the same fashion. If we aren’t on the top row then grab the nodeIndex-width+1 and do the energy checks. We always check the nodeIndex+1 node. Last, if we aren’t on the bottom row we check the nodeIndex+width+1 node. Calculating the lowest energy is the same.

For removing a vertical seam I loop through all the indices length and then loop through all of the indices[i]-width\*i. The inner for loop will give me the number of times the pixel needs to be swapped upward before it reaches the left side of the image. I then do a pic.set(indices[i]-width\*i-j, I, pic.get(indices[i]-width\*i-j-1, i)) which looks pretty rough at first so let’s step through it a bit. The x value of the set is indices[i]-width\*i-j, which will give us the current x of the given index subtracted by one each time through the loop since we are moving the pixel’s color to the left one. The y is always i since we iterate down with a vertical seam. We set that pixel’s color to the pixel directly to the left of it, this means the y value will stay the same. The x is simply the same equation with a -1 at the end.

Now that our picture has all of the low-energy pixels on the left side of the image we can crop it. I make a new picture of width-1 and height dimensions. I have a double for-loop through the width and height where I start the width at 1 and height at 0. I set the new picture’s i-1 and j to the old picture’s i and j pixel. Since i starts at 1 we are always going to want one less for the x value of our new picture. I then set the old picture equal to the newly cropped picture.

Again, removing a horizontal seam is fairly close to a vertical seam removal. The outer for loop stays the same. The inner for loop changes to have the following conditional statement j<indices[i]/width which means it will only run the number of times that the value at the current index divides by the picture’s width correctly, giving us the row it is on. This time we can use the i for the x value since we always move to the right. The y value, on the other hand, is indices[i]/width-j. This gives us the row number minus the number of times we have already swapped. The color we are switching the current pixel with is the color of the pixel directly above us, which means a -1 on the y equation, x stays the same. The creation of the new picture is exactly the same except we start the height at 1 and the y is j-1.

My Ө analysis of my algorithm is Ө(width\*height + (width or height)), the addition is dependent on which find you are calling. Either way it simplifies down to Ө(width\*height). My removes are Ө(2\*width\*height), which simplifies down to Ө(width\*height). My energy, getWidth, and getHeight functions are all constant time.

For this lab I worked with Steven Razim and Clarissa Garza and also received a little bit of clarification help from Drew Hurdle.