Homework07

Parallel and Sequential Algorithms COMP 312 (Spring '17)

1 Introduction

In this homework, you will also provide an implementation of a dynamic programming algorithm known as seam carving.

2 Files

After downloading the assignment tarball, extract the files by running:

```
tar -xf homework07-handout.tar
```

from a terminal window. Some of the files worth looking at are listed below. The files denoted by * will be submitted for grading.

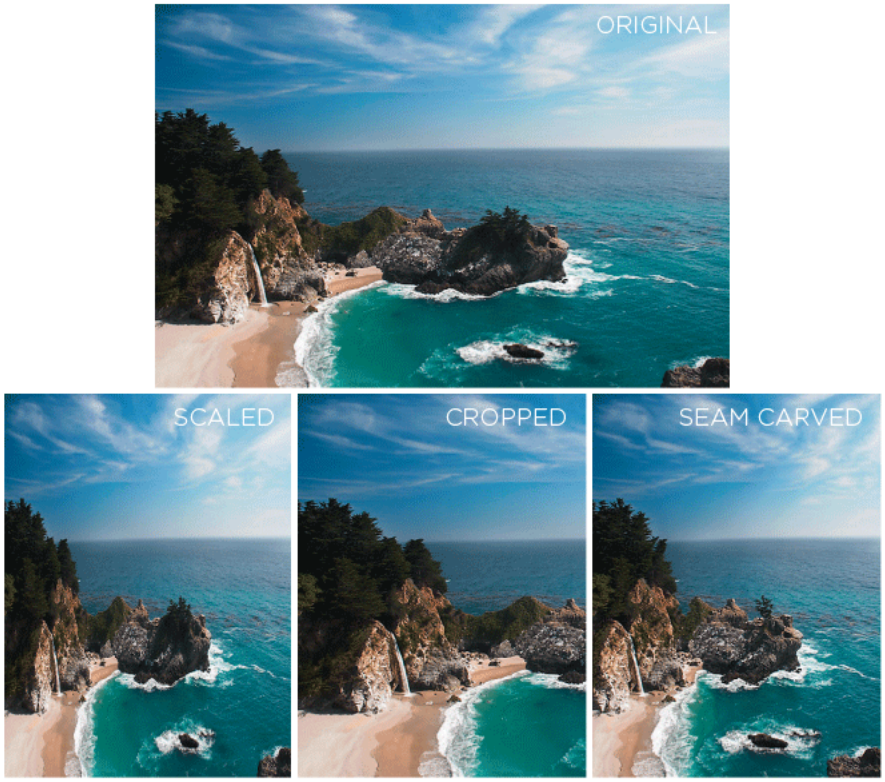
1. * `MkSeamFind.sml`
2. `support/images/`
3. `Sandbox.sml`
4. `Tests.sml`

3 Submission

To submit your assignment, copy `fMkSeamFind.sml` to your handin directory, and bring your answers to any written problems to the next course meeting.

4 Seam Carving

Seam Carving is a relatively new technique discovered for "content-aware image resizing" (Avidan & Sheridan, 2007). Traditional image resizing techniques involve either scaling, which results in distortion, or cropping, which results in a very limited field of vision.



The technique involves repeatedly finding 'seams' of least resistance to add or to remove, rather than straight columns of pixels. In this way the salient parts of the image are unaffected. It's a very simple idea, but very powerful. You will work through some simple written problems and then implement the algorithm yourself to use on real images. For simplicity, we will only be dealing with horizontal resizing.

To motivate this problem, watch the following *SIGGRAPH 2007* presentation video which demonstrates some surprising and almost magical uses of this technique:

`http://www.youtube.com/watch?v=6NcIJXTlugc`

4.1 Background on Pixels and Gradients

Images will be imported by a Python program that uses some image libraries, so you may need to test on the lab computers like last week. Each image is represented as a 2-dimensional grid of pixels, where each `pixel` is represented by its r (red), g (green), and b (blue) compositions.

The first step in the seam carving algorithm, which we have implemented for you, is to convert the image into a grid of *gradients*. Intuitively, the gradient is a measure of how different a pixel is from the pixels to the right and below it.

For any two pixels $p_1 = (r_1, g_1, b_1)$ and $p_2 = (r_2, g_2, b_2)$, we define their *pixel difference* $\delta(p_1, p_2)$ to be the sum of the differences squared for each RGB value:

$$\delta(p_1, p_2) = (r_2 - r_1)^2 + (g_2 - g_1)^2 + (b_2 - b_1)^2$$

Then, for a given pixel $p_{i,j} = I[i, j]$ (i.e. the pixel at row i and column j of image I), the *gradient* $g(i, j)$ is defined as the square root of the sum of its *pixel difference* with the pixel on the right and its *pixel difference* with the pixel below it:

$$g(i, j) = \sqrt{\delta(p_{i,j}, p_{i,j+1}) + \delta(p_{i,j}, p_{i+1,j})}$$

Note that this leaves the right-most column and bottom row undefined. For an image with height n and width m, we define $g(n-1, j) = 0.0$ for any j, and $g(i, m-1) = \infty$ for any i. Using the gradient as a cost function, we can now develop an algorithm to compute a lowest-cost seam for any given image.

4.2 Seam Finding Algorithm

The part of the algorithm that you will implement is given a 2-dimension grid of gradients G, represented as a pixel sequence sequence. For example, suppose we have a 4×4 image with the following table of gradient values.[1]

$i \downarrow\ j \rightarrow$	**0**	**1**	**2**	**3**
0	5	**4**	1	3
1	7	2	**3**	5
2	6	**5**	6	1
3	**3**	2	7	8

A valid vertical seam S must consist of m pixels, where m is the height of the image, and each pixel must be *adjacent* in the sense that if the seam at row i is at column j, then the seam at rows $i-1$ and $i+1$ must be from columns $j-1$, j, or $j+1$. That is, for each row of a seam, its chosen column must be either directly below, or next to, the column of the row above it. For example, the bold entries in the above table are a valid vertical seam. The cost of a seam C_S is:

$$C_S = \sum_{p_{i,j} \in S} G(i, j)$$

where $p_{i,j}$ is a pixel in the original image, and $G(i, j)$ is the gradient given in the above table.

[1] For this example, we ignore the fact that the right-most column will be all ∞s and the bottom-most row will be all 0s.

The overall structure of seam carving is to repeatedly find the minimum cost seam (i.e. the one for which the pixels differ as little as possible) and remove it, with the idea that this removes as little "interesting" information from the picture as possible. Your job is to compute the minimum cost seam, given a gradient table G as above.

An idea for a dynamic programming algorithm for this is to compute a new table M, where $M(i, j)$ is the minimum cost valid vertical seam that ends at $G(i, j)$. For the above gradients, we have the following table:

$i \downarrow\ j \rightarrow$	**0**	**1**	**2**	**3**
0	5	4	**1**	3
1	11	**3**	4	6
2	9	**8**	9	5
3	11	**10**	12	13

We represent a seam by the sequence of column indices as the rows go from 0 to m. The minimum cost vertical seam is the above example is the bolded one, $S = \langle 2, 1, 1, 1 \rangle$, and its cost is $C_S = 10$.

Task 4.1 (75 pts). Using this idea, in `MkSeamFind.sml`, implement the function

```
val findSeam : gradient seq seq -> int seq
```

which finds the lowest-cost seam in a given image, **which is preprocessed into its gradient values**, as described in Section 4.1. The resulting seam is represented as an ordered sequence of column indices going from the top row of the image to the bottom row. For full credit, your implementation must have $O(mn)$ work and span.

Hints:

- For each cell in the above table M, what data is necessary to compute it?
- Which row(s) of M do you need to answer the overall question of the best seam for the entire image? You do not necessarily need to store the entire table M in memory at once.
- The above table M describes how to compute the *cost* of the lowest cost seam. What extra data do you need to store to compute the seam itself?

4.3 Testing

We've given you a few sample images in the `support/images/` directory with which you can run your Seam Carving implementation on. After completing the above task, you should be able to perform the following in the terminal:

```
- CM.make "autograder.cm";
- Autograder.removeSeamsFile("images/cove.jpg", "images/my-cove-50.jpg", 50);
```

where `Autograder.removeSeamsFile` (P_i, P_o, n) takes the image at path P_i, removes n seams from it, and stores the resulting image at path P_o. You should compare your results with the given `sample-50.jpg`, `sample-100.jpg`, and `sample-200.jpg` files in `support/images/`.

For reference, our solution removes 100 seams from `images/cove.jpg` in < 10 seconds. Removing more seams from higher-resolution images will naturally take longer.