MT Research: Seam Carving for OMSCS 6475

Author: Ryan Miller

Video Link: https://youtu.be/MWZhQtvoVr4

rmille14@alumni.nd.edu

Abstract

This report will cover an analysis of an individual implementation of the seam carving algorithms detailed in Avidan and Shamir's white paper.[1] The overall purpose will be to detail the methods used to implement, review the output files generated and compare and contrast them with the original output. Ultimately this will show the challenges to seam carving and also the unique benefits of attempting to implement a proven method for algorithmically manipulating photos to generate a new unique output.

Approach and Description of Algorithm

For my program I attempted to leverage numpy, opencv, and python together in a dynamic programming approach of implementation. Dynamic programming was a good first pass option as it allowed for a relatively decent overall execution time in comparison to full recursion.

The program consists of only two files to run: main.py, and seams1.py. It is invoked leveraging the python cli as follows: *python <image> <pixels>*, where <image> is the name of an image to perform seam carving on in the /images directory, and pixels is the amount of pixels vertically to remove/add to the generated output images. Once the cli has been invoked the main.py program passes the variables to the the seams1.py function where all of the calculations and operations are performed.

The order of operations for the function calls are as follows inside of seams1.py:

- *seam_carve(image, image_name, pixels)*
- *remove_seam(image, image_name, pixels)*
- *energy(image, image_name)*
- *vertical_map(image_energy*
- *calc_seams(value_map)*
- *seam_removal(new_image, list_seams, cost_seams)*
- *adj_seams(image, removed_seams, image_name)*
- *add_seams(image, adj_removed_seams, pixels, image_name)*

The algorithm begins by first loading in an image the end user specifies. It then loads in the image and converts it to gray scale. I did this to operate with a simpler image in one channel as opposed to managing three throughout. Also, I don't have metrics on it, but I believe this would also help performance as it allows for just one channel to be passed for each operation.

Next, the energy function is run. This calculates the differences in pixel intensity values between columns. If there is a large difference between a pixel and its neighbors in a 3 kernel cluster, then the energy value will be high. If it is smaller than it will be low. The idea at this stage is to capture the low value changes for possible seam locations. I tested between a Sobel and Scharr gradient at this stage, ultimately sticking with a Sobel in the end, as it handled seam placement slightly better in live testing. Up to this point everything is fairly simple, the next 3 operations are performed over and over again for the total pixel replacement:

- Calculated weighted cumulative matrix
- Find seams in matrix
- Remove seams and store removed seams

For the examples you'll see later, I ran these 100 times each, which results in 100 vertical seams being removed from the image. Here I tried to stick to dynamic python leveraging matrices, numpy, and iterative loops whenever possible. To calculate the weights I followed both the paper and the Wikipedia entry on seam carving[2]. First the program iterates through the input energy matrix top down comparing to the parent 3 neighboring pixels and summing the lowest values. This runs through the entire energy matrix to generate a new cumulative vertical matrix. Next it inverts and goes bottom up, analyzing the top three neighbors for each base row column pixel and storing the col index up to the

[1] Avidan, Shamir 2007. *Seam Carving for Content-Aware Image Resizing*

[2] Seam Carving. https://en.wikipedia.org/wiki/Seam_carving

top row. This index path is stored in a list and is what is referred to as the seams. In parallel to the path, the program also stores the total summed energy for each path, which is the seam cost. To pick the best seam to remove I choose the seam at the col index with the lowest overall summed energy.

Next I leverage the np.argsort[3] to pick the lowest valued column index in the seam list. There likely is a more efficient way to accomplish this, specifically a min function, but I've used argsort in the past and it functions nicely here. This entire time a thing to note is the program is always passing and storing column indexes. This allows for passing simpler smaller sections of data to do work, while only editing the memory values when needed. Finally, to remove the seam, the program creates a new numpy zeroes 3 channel matrix and removes all 3 channels in every row where the lowest energy seam col index is leveraging np.delete[4], which returns all of the values except the specified that was passed to remove. This is an extremely cost efficient way to remove the entire vertical seam. The seam col index positions are then stored in a list for use in the final two functions.

The final two steps of the program are then to adjust the seams and then expand the image leveraging the seams. To expand an image, unlike reducing one, the program creates an offset where each seam was calculated on a smaller and smaller image. This creates an issue with the column indexes in the list of seams, as each one is calculated at k-1 columns from the last. To compensate for this the program iterates through the list and checks the column index from the first seam to last, if the seam at a later position is to the right of the selected it will shift it to the right one, to place it in the proper location. Finally this seam is then inserted back into the image to enlarge it by leverage array slicing[5].

A new output np.zeros matrix is created for each row added iteration by iteration. Increase by 1 column each time. Each iteration has an operation where the seam col index is used to calculate the average rgb value of the pixels to the right and left of the insert location to generate the pixel to be added, and finally insert the newly created pixel into it's position. Array slicing is then used again to copy the original pixel values at all other positions in the row. This yields and image 1 column wider every iteration through.

Results

Multiple sample tests were run to compute different outputs and review results. The two main outputs for comparioson with the original paper are presented in Figure 1, 2, and 3. The larger image is the original with the seam values calculated and displayed over the image. The majority of the seams are selected in the trees and the ocean as the image consistency here is the highest, i.e. it has the lowest overall change between pixel values and this the lowest energy value. The top right image in figure 1 is the seam

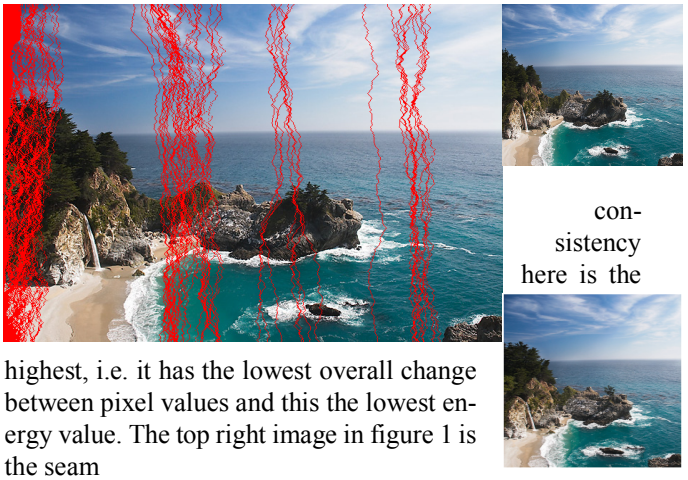

Figure 1: Seam carving with aspect ratio change as a result of vertical column removal. Image on left is calculated seam values. Top right is output from algorithm, and bottom left is Avida and Shamir output.

Figure 2: Seam calculations for seams1.py on left and Avida and Shamir on right.

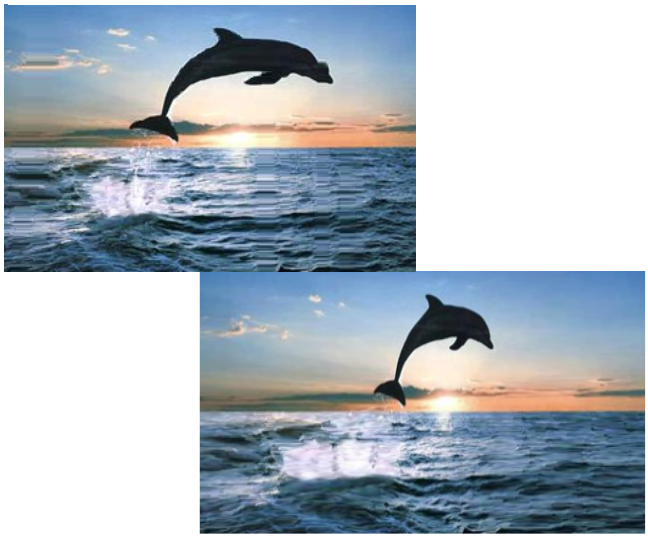

Figure 3: Outputs of expansion functions, seams1.py on top, Avida and Shamir on bottom.

[3]https://docs.scipy.org/doc/numpy/reference/generated/numpy.argsort.html

[4] https://docs.scipy.org/doc/numpy/reference/generated/numpy.delete.html

[5] https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html

carved outputs from the program, and the bottom right is from Avidan and Shamir[6]. The results are very close. Both images take away from the tress and ocean; however, the program in Avidan and Shamir is a little more efficient at identifying the similarities in the ocean. Likely with more pixel removals and a better energy function the results may align more closely. This example and output shows the importance the energy function has on the generated seam carve. The kernel and algorithm create the base for the rest of the functional operation and thusly, tailoring the right gradient to the right picture is critical.

A clearer example of this can be seen in the expansion function outputs in Figure 2 and 3. The seams1.py program seams outline the dolphin heavily, similar to Avida's; howeber, it also grabbed extra artifacts in the dolphin towards the head and tail, while not capturing down the center. This resulted in an expanded dolphin that looks enlarged slightly whereas Avida's output is more closely aligned to the correct size of the original dolphin. This can also be seen in the fact that the seam offset algorithm is more efficient in Avida's than in the seams1.py program outlined in this paper. Each seam is clearly separate and offset on its own, whereas seams1.py has a overlap throughout. Ultimately, the final result is fairly close, but with a better energy function tied to the input image, and proper offset/overlap of seam handling, a better output would be generated.

Issues/Reflection

Overall this project took a lot of trial and error and rewriting code from base up a few times to understand concepts and implementation. Correct matrix manipulation in the cumulative weighted function was fairly straightforward; however, calculating and storing the seam required much trial and error. Simplifying the scope and generating a small 6x5 random matrix and working through logic repeatedly allowed for this hurdle to be overcome. Ultimately it came down to better handling of indexes as well as array slicing that assisted in this area.

The second major pain point was around seam offset. Initially my program's seams were all over the image for the addition step. Evenly distributed throughout and through the dolphin as well. Conceptually it was difficult to work through how to handle the offset. Taking the time to whiteboard the intended output here led to the solution of list indexing and checking the column position shifting by 1 if the value was larger for the index. This achieved a result much closer to the intended output; however, it still isn't an exact match. Playing with different gradients and kernel sizes led to a closer output, but the ambiguity of the energy function and it's implementation created difficulties.

Third, The final resultant is focused on optimal retargeting. Seams1.py doesn't have this implemented due to time constraints and errors that occurred through development. The main pain point here was felt in testing in run time as the base performance of seams1.py for seam carving alone took 6 min for images generated in figure 1, and 40 seconds in figures 2 and 3. Early tests of optimal retargeting and transport map creation were taking 30 min plus. This is likely due to the amount of loop iterations performed. More efficient use of linear algebra libraries, array slicing, and less iterative loops could assist in this area. That being said the concept and flow is relevant for this paper. By leveraging the cumulative matrix function and shifting to run on both the row and column for each energy iteration yields and overall sum of seams. Taking the min comparison here can generate a transport map for the image with lighter colors indicating less energy, and higher colors i.e. red in a heat map, indicating higher energy. Stepping through this matrix for each pixel can yield the best output image as it will select the optimal seam either horizontally or vertically to add or remove, rather than just entirely in one direction.

The final and biggest hurdle was the difficulty of analyzing a pure whitepaper and grasping the concepts presented in it. I don't have the best depth in summation/series calculus combined with very little experience converting research to practical implementation. Scouring Wikipedia and taking time to draw out the problem at hand was hugely beneficial, my initial strategy of attacking the code immediately led to heavy frustration, whereas taking the time to plan sped up output significantly. Also leveraging array slicing and libraries was a huge accelerator. Simplifying complex logic to single lines of code. That being said my programmatic implementation is far too reliant on looping and iterations. I spent too much time focusing on the functional behavior with known approaches than researching the functions numpy affords to simplify the linear algebra. More time to attempt broader uses of array slicing and numpy itself could likely yield far better performance.

References

Seam Carving. https://en.wikipedia.org/wiki/Seam_carving

Avidan, Shamir 2007. *Seam Carving for Content-Aware Image Resizing* http://www.faculty.idc.ac.il/arik/SCWeb/imret/imret.pdf

https://docs.scipy.org/

https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html

[6] [6] Avidan, Shamir 2007. *Seam Carving for Content-Aware Image Resizing*