# Content-Aware Image Resizing

Race Darwin Morel
*Bard College*

# Content-Aware Image Resizing

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by

Race D. Morel

Annandale-on-Hudson, New York

May, 2017

## Acknowledgements

My sincerest gratitude goes to my advisor Keith O’Hara, who always made time for me despite taking on a huge number of advisees. I know that advising so many students must feel like herding cats, but we couldn’t have asked for a better shepherd.

I am forever grateful to my parents for their unconditional love and support, for teaching me to be a free thinker and a problem solver, but mostly for the coffee maker they sent me that was instrumental to my completion of this project.

# Table of Contents

## Abstract

The purpose of this project is to implement and explore the use of seam carving — a tool used to select and remove “seams” of low-importance pixels from an image in order to reduce its height or width. I also cover the process of seam insertion, creating new seams of pixels to increase the image’s size rather than reducing it. Using these content-aware algorithms, I investigate the process of intelligently resizing an image. Using edge detection, dynamic programming, and pixel manipulation, I push the limits of seam carving and attempt to quantify the qualitative concept of salience.

# 1 Introduction

## *1.1 The Problem*

Digital images come in all shapes and sizes, as do the devices they are displayed on. A classic problem of image manipulation is to resize a still image in order to fit it onto a screen of a different size or aspect ratio. The simplest ways of resizing an image are *cropping* and *scaling*, each with their own limitations. In this section I will detail the uses and limitations of these basic methods of image resizing before introducing the alternative this project is centered around.

## *1.2 Cropping & Scaling*

Cropping an image will cut out a rectangular portion of the image at the desired size and aspect ratio. This method of image resizing cannot increase an image’s size. It will always reduce the vertical and/or horizontal dimensions. This means cropping will always result in the loss of rows of pixels from one or more edges of the image. If you are reducing an image’s width, cropping the image will invariably cut off the left or right side of the picture. For images that have important information on opposite sides of the image, this is obviously not an ideal solution.

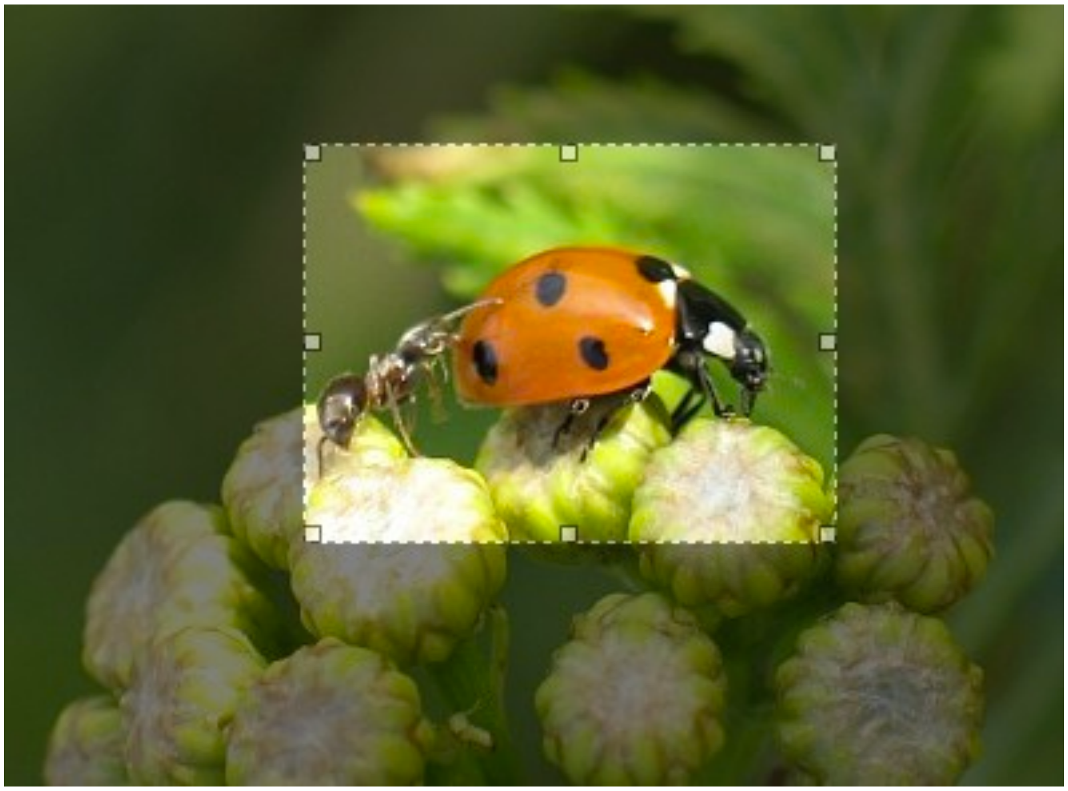


**Fig 1.1: Cropping an image**

Scaling can be used to either increase or decrease the image’s size, and is probably the most common method of image resizing. Scaling an image will change its size, but is limited by the image’s aspect ratio. The aspect ratio (i.e. the width of the image vs. the height of the image) must remain constant when scaling an image in order to avoid stretching the image content. This severely limits the ways that scaling can be used to resize images. For example, a 4x5” image can be scaled to an 8x10” image (or vice-versa) without problem, but a 4x5” image cannot be scaled to a 4x10” image without risk of severe warping.

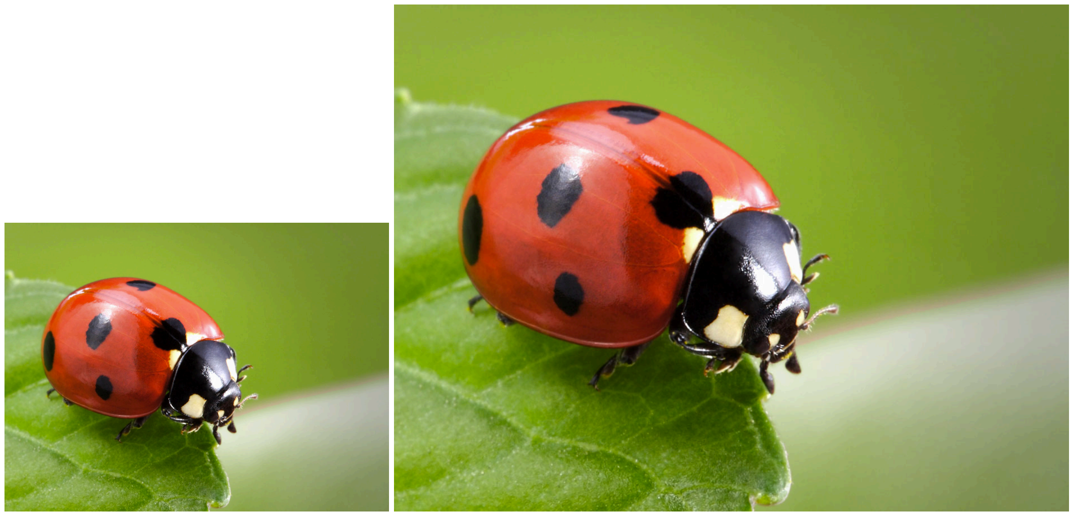


**Fig 1.2: Image Scaling**

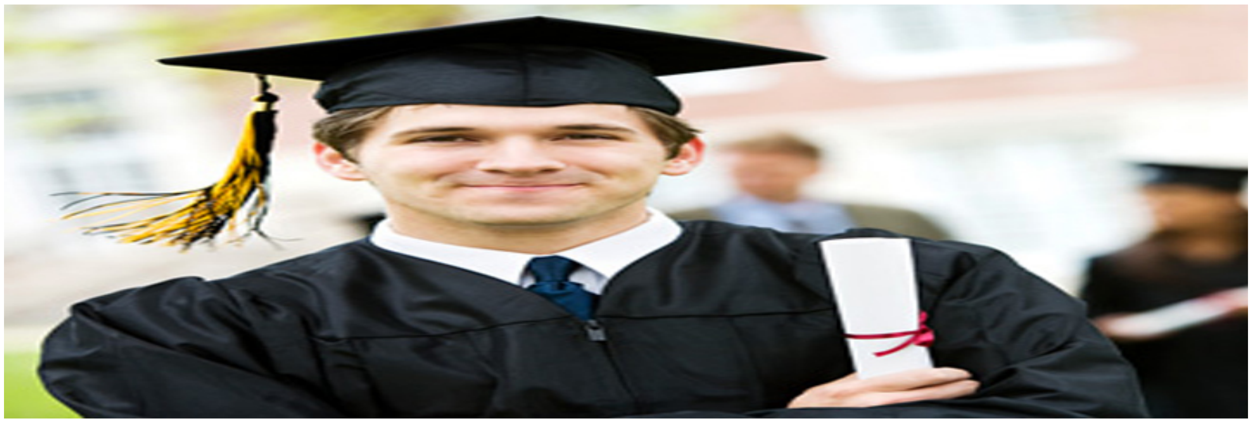


**Fig 1.3: Result of scaling without preserving Aspect Ratio**

Scaling an image without maintaining the aspect ratio often resulting in objects appearing either too skinny or too thick. When using scaling to change an image’s size by small amounts, this may not be noticeable. Therefore it is often perfectly acceptable to scale an image without locking the width/height ratio. However, an extreme change in the image’s aspect ratio (see **Fig 1.3** above) will result in the image content being visibly warped.

## *1.3 Seam Carving Overview*

Seam carving is a more intelligent method of image resizing. It was originally developed by Shai Avidan and Ariel Shamir who published a paper about it in 2007 [1]. Since then it has been implemented in Adobe Photoshop starting with CS4 under the name "Content Aware Scaling." The algorithm works by finding the path of least importance across the image, then removing all the pixels in that path.

The core concept that seam carving revolves around is *saliency*. Saliency refers to one member of a group standing out relative to its neighbors. In the case of seam carving, we are referring to groups of pixels. A high-saliency pixel is one that stands out from its neighbors, whereas a low-saliency pixel would be close to identical to its neighbors. Seam carving works by finding low-saliency *seams* — paths across the image. A seam can traverse the image either vertically from bottom to top, or horizontally from left to right. The idea is to find the path across the image with the lowest saliency and then remove it in order to make the image one pixel shorter or thinner, depending on whether it is a vertical or horizontal seam. Removing vertical seam from an image will result in the image's width being reduced by one pixel, and removing a horizontal seam will reduce the image's height.

In theory, removing only the lowest saliency seam or seams will reduce the image's size without disrupting the actual content of the image. Since the seams are comprised of pixels that ideally are quite similar to their neighbors, it should be difficult to tell that they have been removed. The content of the image should remain mostly undisturbed, hence this algorithm being labeled as "content-aware." However, the removal of numerous seams can begin to disrupt the image content, since each subsequent seam that is removed will naturally tend to have a higher saliency than those preceding it.

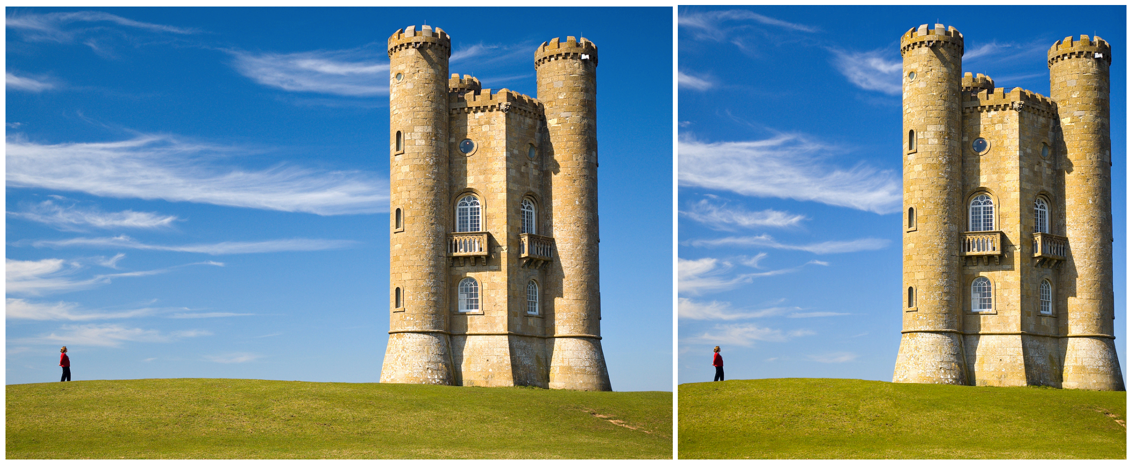


**Fig 1.4: Using Seam Carving to reduce an image's width**

## *1.4: Motivation & Evolution of Project Goals*

It took me a long time to decide that I wanted to implement this particular project. At first I spent some time studying cellular automata, thinking I could do something interesting involving Langton's Ants or Conway's Game of Life. As it turns out, the Game of Life is Turing complete, making it difficult to write an original paper about.

I realized that the part of the appeal I found in cellular automata was that it was visually stimulating. I decided that I wanted to create something visual for my senior project. One of my hobbies involves making "glitch art," created by manipulating image files on a computer to break down the image's coherency and reveal visual artifacts.

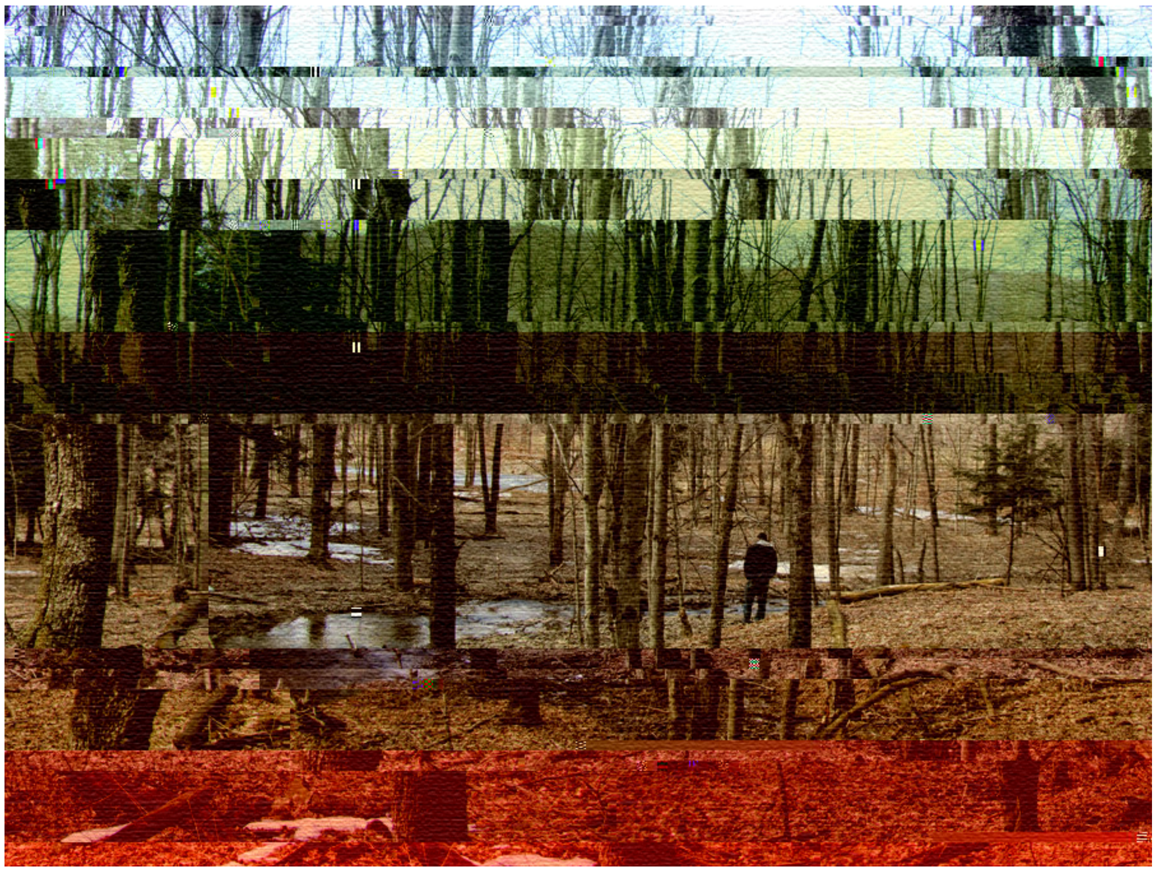


**Fig 1.5: "Solitude" — Some original glitch art by yours truly**

After that revelation, I began to think about image manipulation. Initially, I had the idea to create a project not just about resizing images, but reshaping them — changing rectangular images to fit irregularly shaped borders, be they ovals, triangles, or any strange shape in-between.

In my research into this topic, I was trying to find a way to resize the image without disrupting the content of the image. This is when I stumbled across the concept of seam carving. Initially I was planning to use seam carving to reduce the image size to within the bounds of the irregular border, and then using a form of texture synthesis called *image quilting* to fill the rest of the border [2]. However, after some consideration, I realized that there was a very slim chance that this would result in a cohesive or natural-looking image. Software gore, while sometimes visually stimulating, probably shouldn't be the goal of my project.

Instead, I chose to focus solely on seam carving, which I was already in the process of implementing. I noticed that while there exists a good deal of documentation on the implementation of seam carving, I hadn't seen any detailed analysis on the cases in which it fails or how it affects the saliency of the image. I decided to test what specific quantitative elements, if any, cause the algorithm to destroy image coherency.

# 2 Implementation

## *2.1 Saliency Map*

In order to find low-saliency seams across the image, the saliency (or *energy*) of each pixel must be evaluated. The most common way to go about calculating the saliency of a pixel is through the *image gradient* — the directional change in color/intensity of an image. The magnitude of the gradient for each pixel corresponds to differences in color or intensity between that pixel and its neighbors. A high gradient magnitude is typically associated with the edges of objects in the image — probably not something we want to remove if we wish the image to remain cohesive. Really it's the low magnitude pixels that are of interest to us; the pixels that blend in with their neighbors are less noticeable and therefore much easier to remove.

Once the saliency of each pixel has been determined, a *saliency map* of the image is created. This map, usually a grayscale representation of the original image, will brighten the high-saliency pixels and darken the low-saliency pixels. Hopefully this results in the focus points of the image appearing as bright white objects, while the background is mostly black.

For this project I created a saliency map using a *sobel filter* [3]. The sobel filter (or "sobel operator") is an edge detection algorithm that calculates the approximate image gradient of each pixel. It does this by convolving each pixel with a pair of 3x3 kernels designed to calculate the approximate difference in intensity between the pixel in question and its neighbors.

One kernel will approximate the horizontal derivative, while the other calculates the vertical derivative.

| | | |
|---|---|---|
| **-1** | **0** | **+1** |
| **-2** | **0** | **+2** |
| **-1** | **0** | **+1** |

**X**

| | | |
|---|---|---|
| **+1** | **+2** | **+1** |
| **0** | **0** | **0** |
| **-1** | **-2** | **-1** |

**Y**

**Fig 2.1: X and Y kernels used in Sobel Filters**

When applied to every pixel in the image, these kernels will produce two gradient values for each pixel — *Gx* and *Gy*. The final gradient value of each pixel of the new image is then:

$$G = \surd(G_x^2 + G_y^2)$$

The result of this filter will be a grayscale version of the original image with all the edges and more salient content highlighted in white, while the lower energy content is darker.

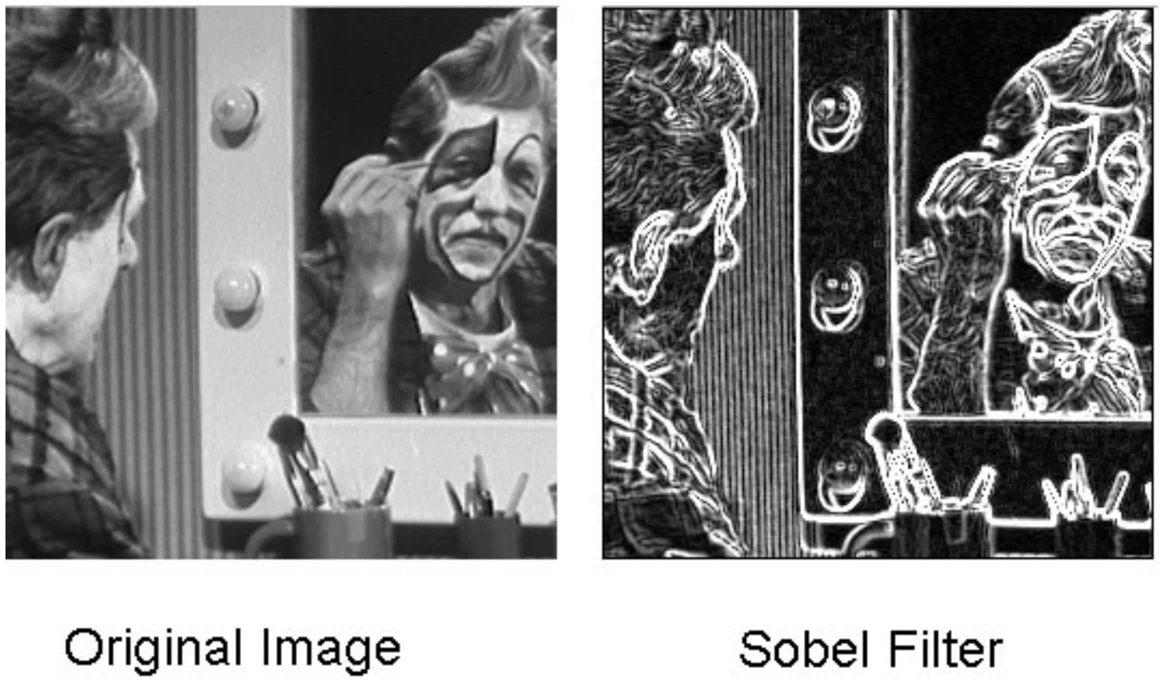


**Fig 2.2: Black & white image put through a Sobel Filter**

It should be noted that this is a grayscale image. In order to create a color saliency map, I took the same sobel filter and applied it to each color value (RGB) separately before adding them together.

```
// For this pixel in the new image, set the RGB values
// based on the sums from the kernel
magnituder = Math.sqrt((sumxr * sumxr) + (sumyr * sumyr));
magintr = (int)Math.round(magnituder);
magnitudeg = Math.sqrt((sumxg * sumxg) + (sumyg * sumyg));
magintg = (int)Math.round(magnitudeg);
magnitudeb = Math.sqrt((sumxb * sumxb) + (sumyb * sumyb));
magintb = (int)Math.round(magnitudeb);
edgeImg.pixels[y*img.width + x] = color(magintr, magintg,
magintb);
```

**Fig 2.3: Calculating gradient magnitudes for red, green, and blue values**

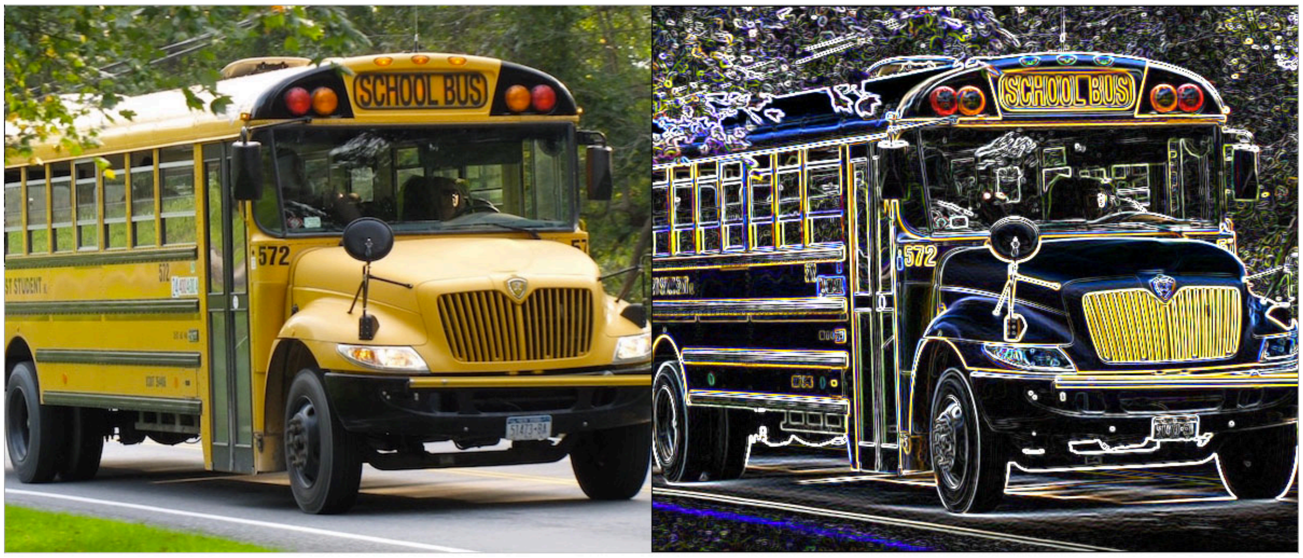


**Fig 2.4: Color image put through the Color Sobel Filter**

Now that the sobel filter has been modified to work on color images, it can be used as a saliency map to find the lowest energy seams across the image. However, because the act of carving a seam changes the image, this saliency map can only be used to carve a single seam. To reduce the image size by multiple pixels, a new saliency map must be computed for each seam that is to be carved from the image.

## *2.2 Finding Lowest Saliency Seams*

Once the saliency map is complete, it's time to find the seams. A seam is a vertical or horizontal path, one pixel in width, which connects one side of the image to the other (either top to bottom or left to right). This path is computed to minimize the total energy of the pixels that it crosses. The intention is then to remove all the pixels in the path from the image, resulting in the image being either one pixel shorter or thinner. Because this removes the lowest-energy pixels possible, this process hopefully does not disrupt the focal points of the image.

The computation of seams is not as complex as it may seem, and is achieved using dynamic programming. When attempting to compute a vertical seam, the algorithm works something like this:

Each square in the **Fig 2.5** (below) represents a pixel with its energy value displayed as the red number in the top left corner. The black number represents the sum of each pixel's energy added to the sum of the energy of all the pixels in the seam above it. For the top row of pixels, of course, this sum will be equal to the energy of that pixel. For the row of pixels below that, however, we must first determine which "parent" pixel we are going to use. The pixels in a seam must all be touching, so each pixel could be part of a seam with the pixel above and to the left, directly above, or above and to the right of it. Thus, for each pixel in the second row, we examine the three pixels above it and find the one that has the least sum. (For pixels on the edge of the image, we only examine two potential parent pixels.)

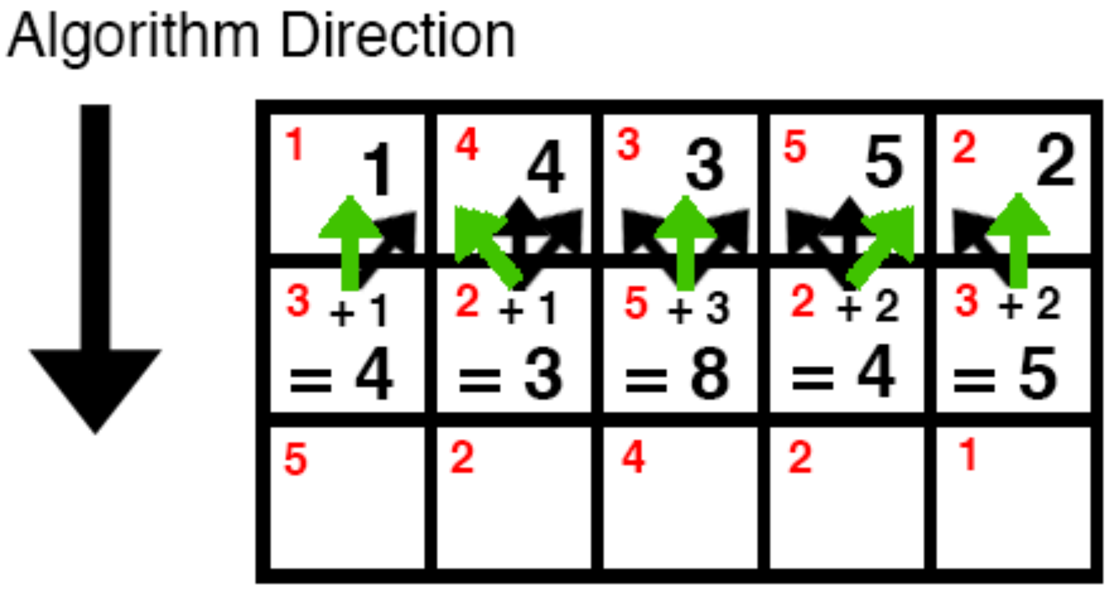


**Fig 2.5: First step of seam-finding algorithm**

For example, the middle pixel in the second row has energy value 5. It checks the sums of the three pixels above it (4,3, and 5) and chooses the pixel with the least sum as its parent — namely the pixel with the sum of 3. Because it has energy 5 and the seam's current sum is 3, it adds itself to that seam and the seam's sum becomes 8. This process is then repeated for the third row, and so on until the bottom of the image is reached.

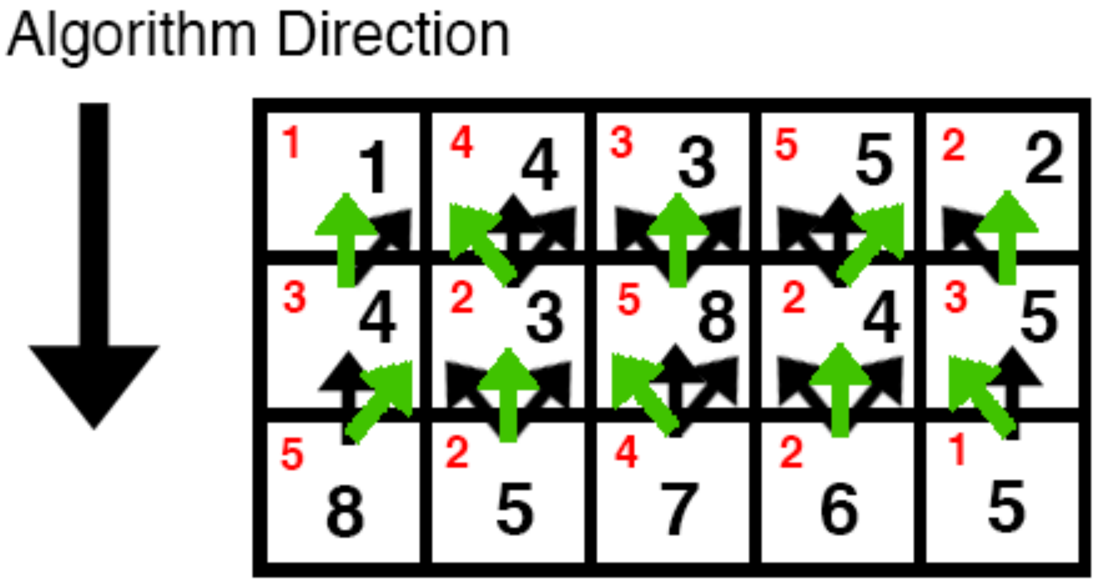


**Fig 2.6: Second step of seam-finding algorithm**

Once all the seams have been computed, we can easily find the lowest energy seams by looking at the sums of the pixels in the bottom row. In this case we have a tie — there are two pixels in the bottom row that have the sum of 5.

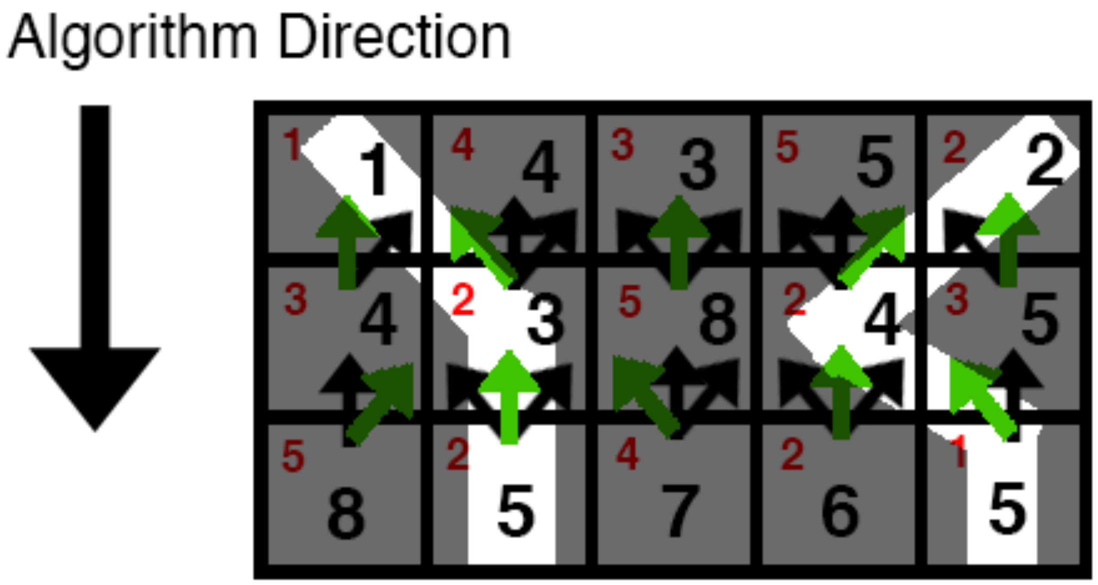


**Fig 2.5: Final step of seam-finding algorithm**

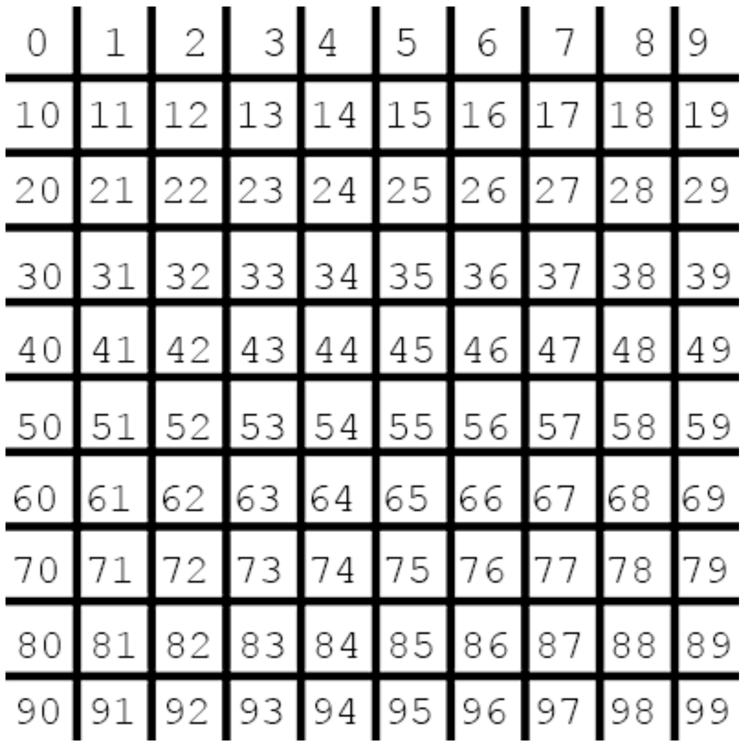
We can trace the seam upwards by looking at the “parent” of each pixel to highlight these two seams. Since both of these seams are tied for lowest energy, it doesn’t really matter which one we remove. Neither seam contains high-saliency content, and removing either will make the image one pixel thinner.

In order to find horizontal seams whose removal will make the image shorter, one only has to rotate this algorithm 90 degrees.

This process can be repeated until the image has reached the desired aspect ratio. However, since the seams that are removed first are the lowest energy paths across the image, the subsequent seams that are removed will tend to be of a higher total energy. Thus, removing too many seams will inevitably result in the removal of high saliency pixels and the noticeable distortion of prominent objects in the image.

## *2.3: Removing Seams*

The actual removal of the seams is dependent on the software used to edit the images. For this project, I implemented seam carving using Processing 2.2.1. Processing technically has its own programming language, but it is heavily based on Java and uses the same syntax. Processing handles images by using a datatype called a *PImage*. This datatype has attributes width and height, as well as an array called `pixels[]` which (unsurprisingly) contains all the pixels in the image. This is not a two-dimensional array, so pixels cannot be directly referenced by their x and y values alone. The pixels are placed in the array starting at the top left corner of the image as `pixels[0]` and continuing from left to right.



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 |
| 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 |

**Fig 2.6: Graph of pixels in a 10x10 image by index number**

However, since the PImage must have a value for width and height, a pixel's x and y coordinates can be easily found with the following formulae:

```
x = i - (y * PImage.width);
y = i/PImage.width;
```

where `i` is the index of the pixel within `pixels[]`.

The index of the pixel can also easily be found using x and y values.

```
i = y * PImage.width + x;
```

Removing seams from a PImage must be done differently depending on whether the seam is a horizontal or vertical one. In both cases, a new PImage is created and pixels from the original image are copied into it, with the exception of the pixels in the seam. A vertical seam is simpler to remove, so let's start with that one.

To remove a vertical seam from an image, we create a new PImage with the same height as the original, but one less pixel in width. Then all one must do to remove the seam is to iterate through the original array of pixels, copying the ones that are not part of the seam into the `pixels[]` array of the new image.

```
PImage post = createImage(img.width - 1 , img.height, RGB);
                      //final image
post.loadPixels();
int j = 0;
boolean inSeam = false;
for (int i = 0; i < img.pixels.length; i++){
  for (int x = 0; x < seamIndex.length; x++){
    if (i == seamIndex[x]){ // check to see if pixel is in the
                                   seam
       inSeam = true;
    }
  }
  if (inSeam == false) { // if pixel is not in seam, copy it into
                              new image
     post.pixels[j] = original.pixels[i];
     j++;
  }
  else { inSeam = false; } // if pixel is in seam, skip it and
                              reset inSeam variable
   }
post.updatePixels();
image(post,0,0);
return(post);
```

**Fig 2.7: Removing a vertical seam from an image in Processing**

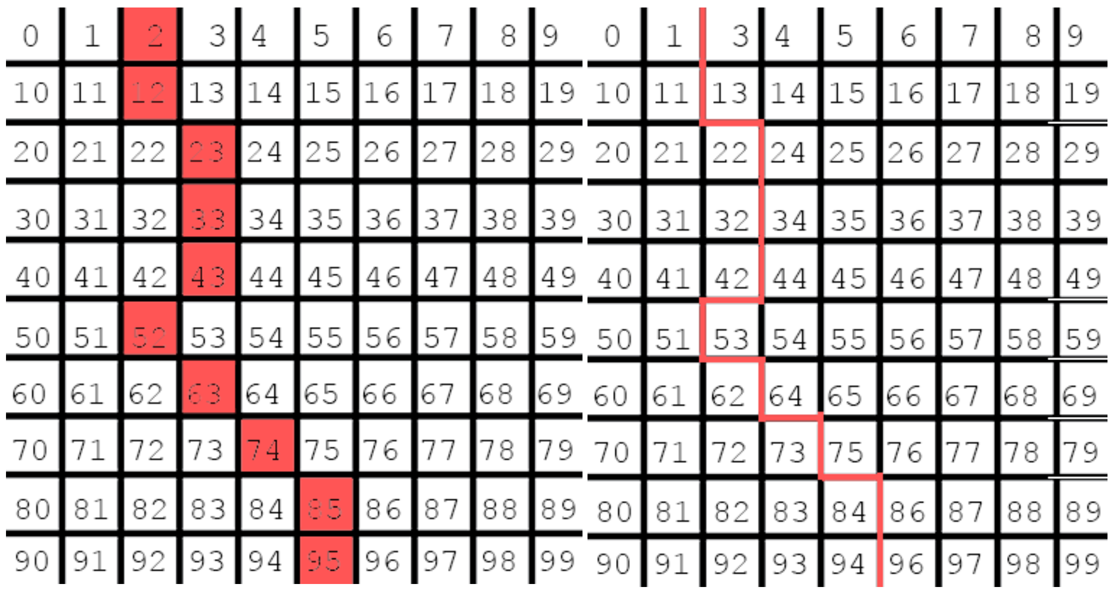


**Fig 2.8: Graph of pixels before and after removing seam (in red)**

As you can see from **Fig 2.8**, the result of removing a vertical seam from a 10x10 image is a 9x10 image in which all pixels except those in the seam retain their relative position. Notice that all the pixels to the right of the seam in each row seem to have moved one square to the left to fill the void that the seam previously occupied. Thus, pixel 3, which was at position (3,0) in the original image, is now at position (2,0) in the resulting image. It has also moved in the pixel array from `pixels[3]` to `pixels[2]`.

Similarly, pixel 13, which was at position (3,1) is now at (2,1) and has moved from `pixels[13]` to `pixels[11]`. It has moved two spaces in the pixel array because two pixels have been removed before it in the array — pixel 2 and pixel 12. Note this discrepancy: graphically, the pixel has only moved one space, but in the array of pixels it has moved more.

Let's see what happens when we try to apply the same algorithm to a horizontal seam:

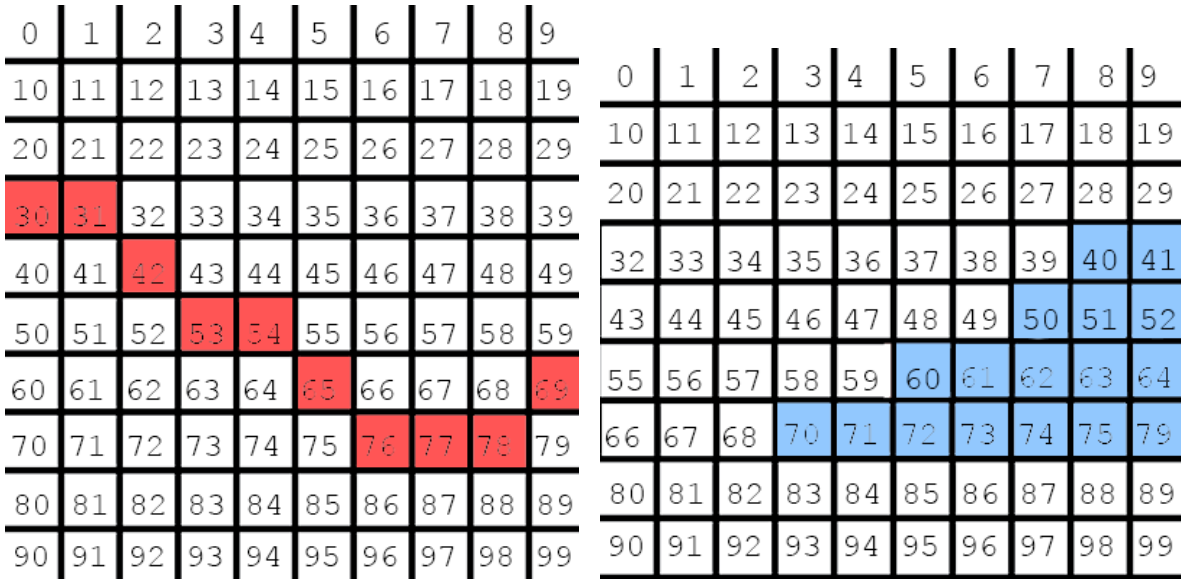


**Fig 2.9: Results of using vertical seam removal algorithm for horizontal seam**

As you can see, the image is indeed one pixel shorter. However, a large number of pixels that were originally on the left side of the image have wrapped around and wound up on the right side of the image (highlighted in blue) in order to fill the space left over in that row once the seam has been removed. And since those pixels wrap around, they leave space on the left side to be filled. So some pixels from the right side will be shifted to the left. This will obviously be highly disruptive to the content of the image.

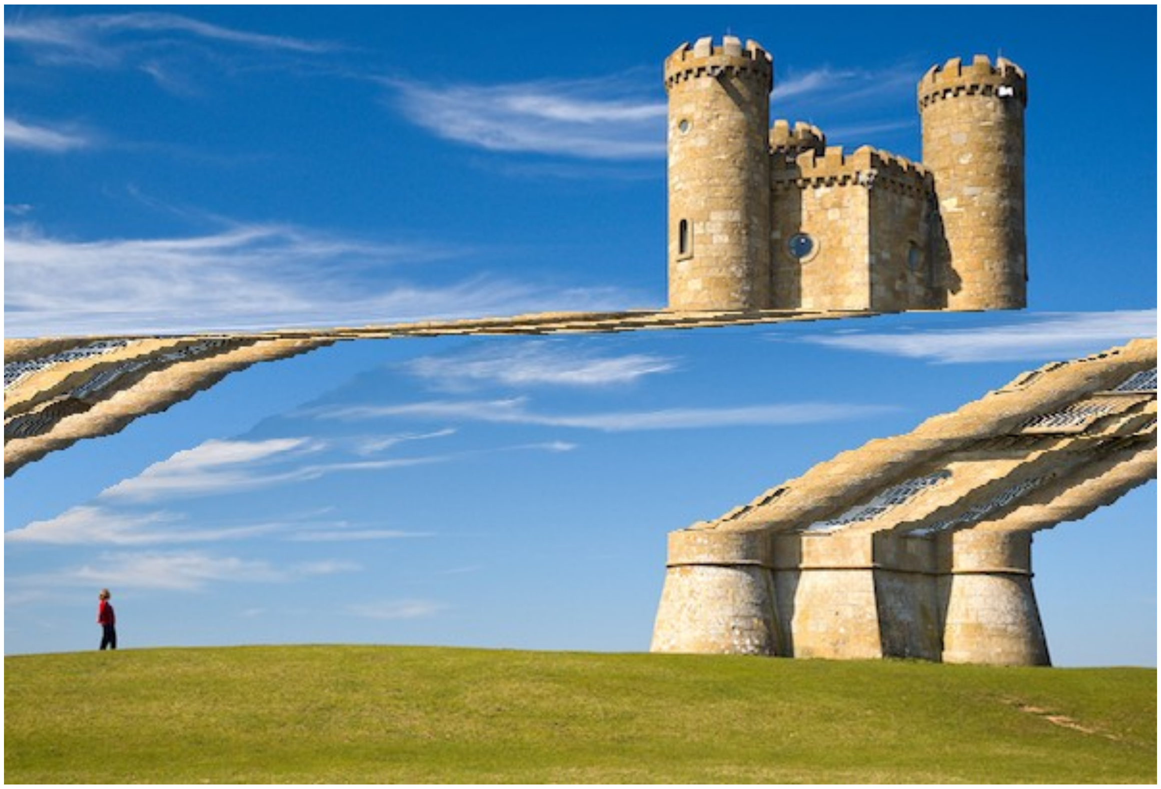


**Fig 2.10: The disastrous results of removing a horizontal seam incorrectly**

Obviously this doesn’t meet our standards for leaving the image content undisturbed. So what *should* the removal of the horizontal seam look like?

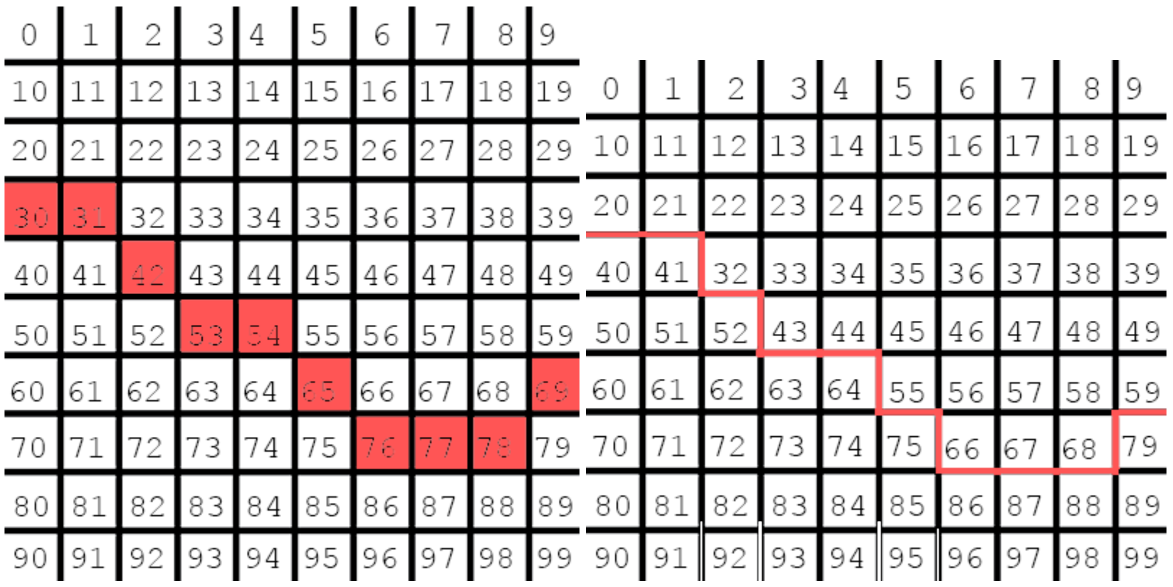


**Fig 2.11: Horizontal seam carving done right**

This is how we want seam removal to work. In this version of horizontal seam carving, we can see that every pixel that is below the seam has been moved up one space. In order to perform this operation we must iterate through the vertical columns of the image until we hit the seam, rather than iterating through the horizontal rows of the image as we did when removing the vertical seam. (This is more difficult to do in Processing because the array of pixels is one dimensional, but since we know the dimensions of the image, we can calculate a pixel's x and y coordinates to work around this.) Before hitting the seam, we are simply copying each pixel in the column as we iterate down it into the new image. When we hit a pixel in the seam we skip it, and continue iterating through the column. But now that we've passed the seam, we copy every pixel we find into the space directly above its original location. Thus, the resulting image will be one pixel shorter than the original image.

```
PImage post = createImage(img.width, img.height -1, RGB);
                                                  //finalimage
post.loadPixels();
   for (int x = 0; x < original.width; x++){ //for each x
     boolean carved = false;
     for (int y = 0; y < original.height-1; y++){ //iterate
                                                  through the column
       if (carved == false){
         for (int i = 0; i < seamIndex.length; i++){
           if (y*img.width + x == seamIndex[i]){ // check to see
                                                  if pixel is in seam
             carved = true;
             int y2 = y;
             while (y2 < post.height){ // copy pixels into space
                                             above original location
                post.pixels[y2*original.width + x] =
                     original.pixels[(y2+1)*img.width + x];
                y2++;
             }
           }
           else{ // if above seam, just copy pixel normally
                post.pixels[y*original.width + x] =
                     original.pixels[y*original.width + x];
           }
         }
       }
     }
   }

post.updatePixels();
image(post,0,0);
return(post);
```

**Fig 2.12: Removing a horizontal seam from an image in Processing**

## *2.4 Seam Insertion*

The process of seam insertion is nearly identical to that of seam carving. It follows all the same steps up until the actual removal of the seam. The image is put through a sobel filter in order to create a saliency map, and a seam of lowest saliency is found using the dynamic programming algorithm detailed in **Section 2.2**. However, instead of removing this seam, a new one is created adjacent to it. The pixels in this new seam are created by finding the average color of the pixels neighboring them. For a vertical seam, the average is taken from the pixels directly to the left and right of the seam in each row. Once the average has been found for that row, a pixel is inserted directly adjacent to the seam, pushing the rest of the row over. This happens for each row in the image until a new vertical seam of pixels has been added and the image's width has increased by one pixel. The horizontal version of this is similar, the average being found from the pixels directly above and below the seam.

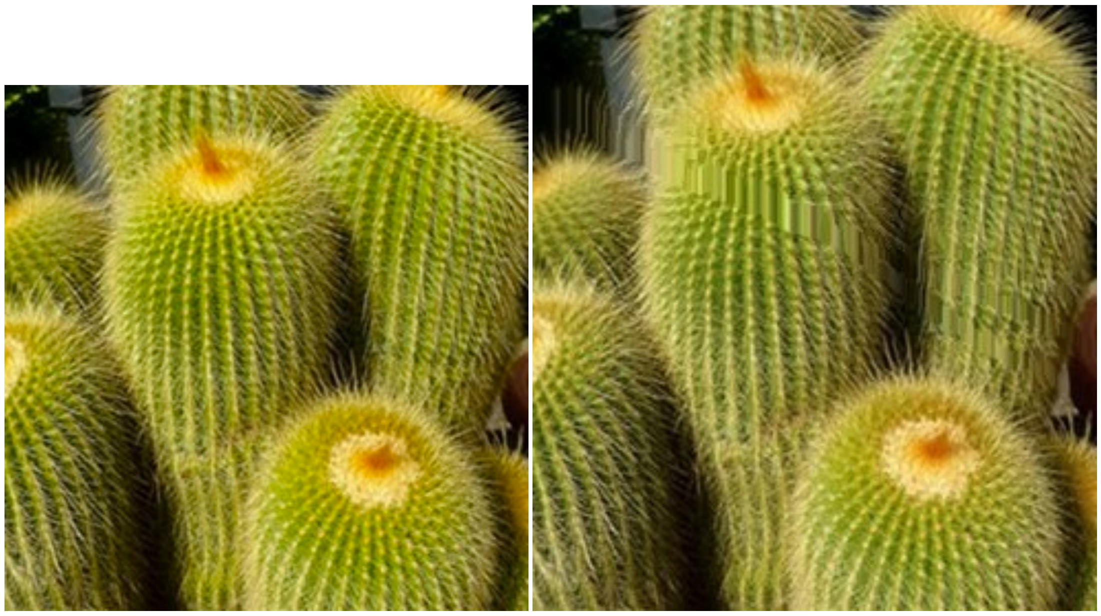


**Fig 2.13: Inserting Horizontal Seams to increase image height**

The seam insertion process for both vertical and horizontal seam insertion is nearly identical to the seam removal process for horizontal seams outlined in **Section 2.3**. We still iterate through the pixels array (either vertically or horizontally), but instead of pulling the pixels we encounter after the seam over the hole left from the seam's removal, instead we push them away from the seam to create a gap that is filled with the new pixels we generate. For details, see the `addvert()` and `addhoriz()` methods within the full code in the appendix.

# 3 Results

## *3.1 Vertical Seam Carving*

For the results section, I will be including examples of seam carving and seam insertion, as well as some tables/graphs that show the relationship between the image's saliency and the number of seams carved. Saliency does not have a specific unit of measurement, as there are many ways to measure it. For our purposes, the saliency value of pixels are calculated by taking the sum of their red, green, and blue values after having been put through the color sobel filter detailed in **Section 2.1**.

I will be examining the relationship between the number of seams carved and the total saliency of the pixels making up the lowest-saliency seam, as well as the relationship between the number of seams carved and the total saliency of all pixels in the image divided by the number of pixels in the image (i.e. the average saliency of each pixel). My goal is to better understand the effect that carving out a large number of seams has on the image's salient content.

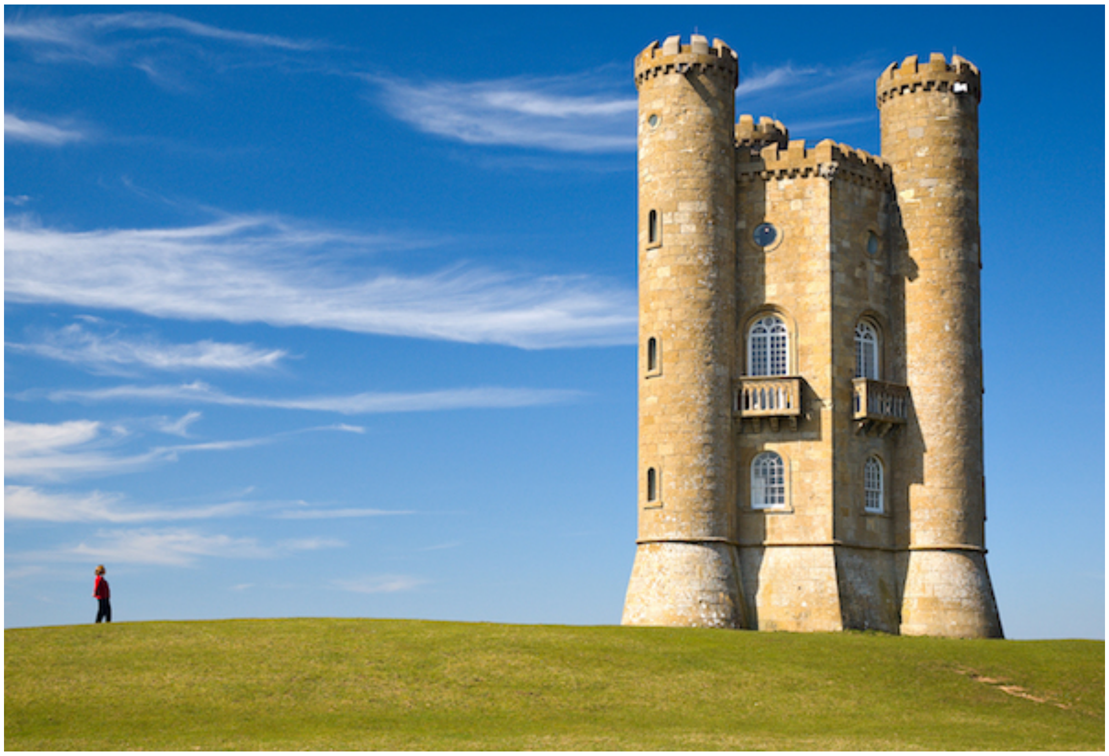


**Fig 3.1.1: Original Image “Tower.jpg” — Dimensions: 600x407 pixels**

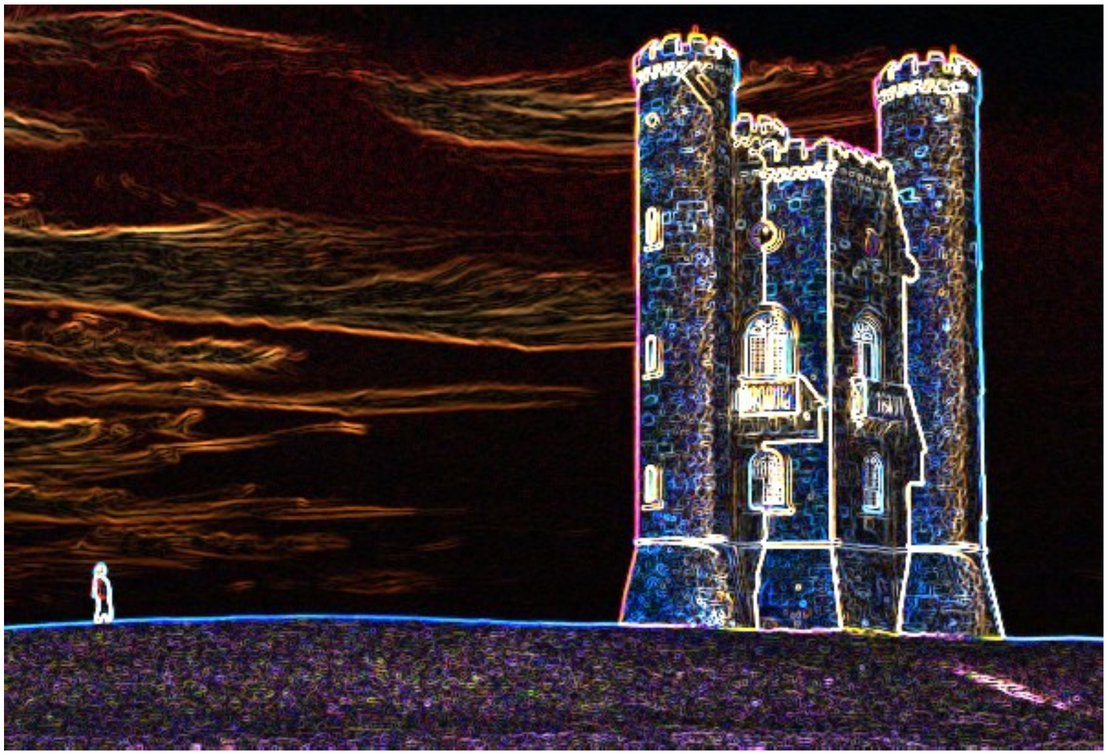


**Fig 3.1.2: Color Saliency Map**

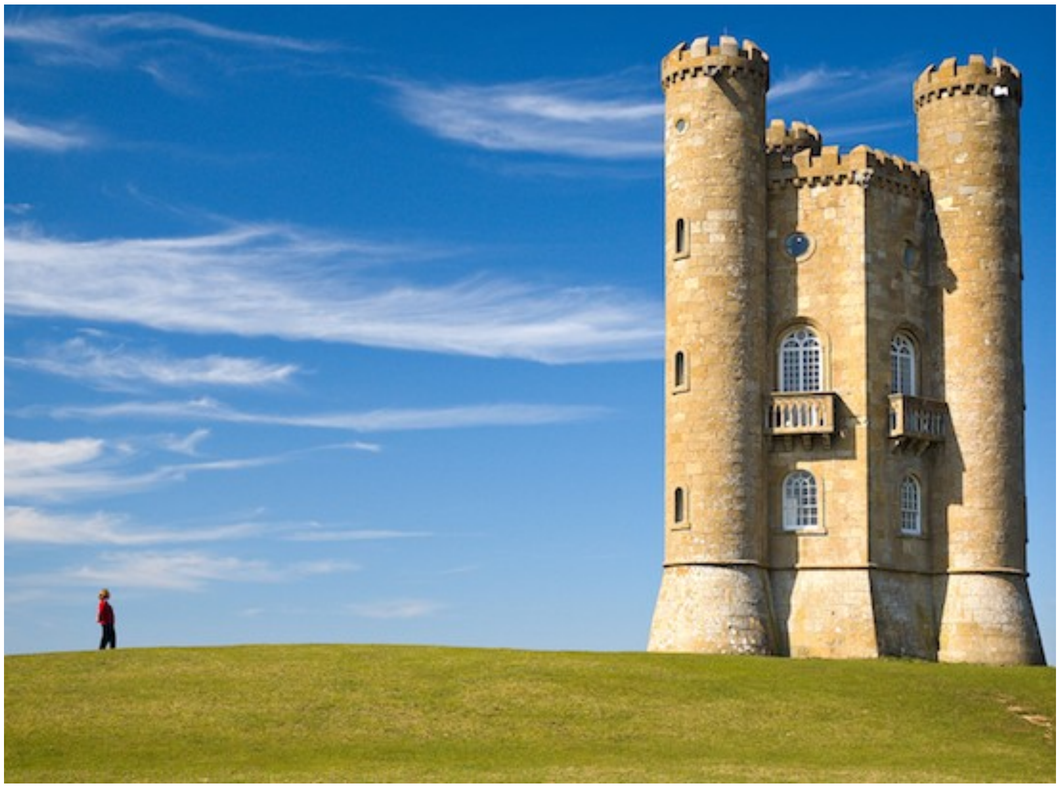


**Fig 3.1.3: Seams Carved: 50, vSeam Saliency: 16752, Saliency/Pixel: 159**

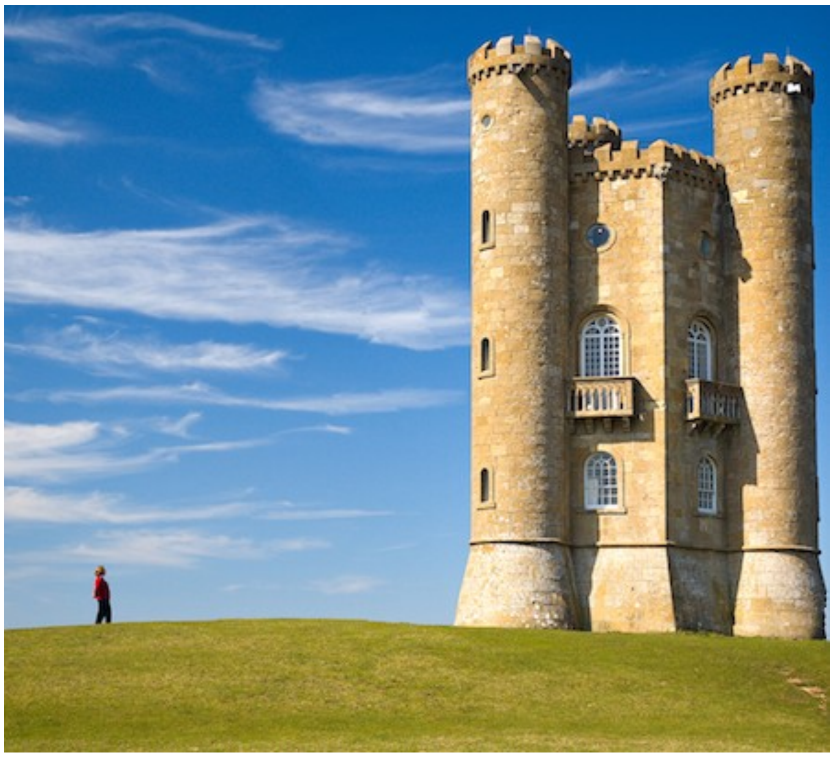


**Fig 3.1.4: Seams Carved: 150, vSeam Saliency: 22851, Saliency/Pixel: 180**

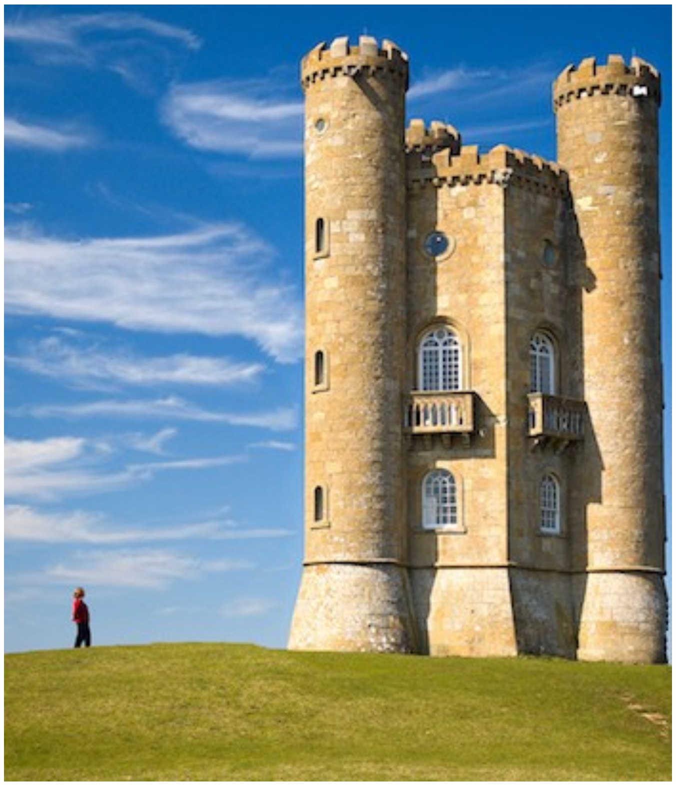


**Fig 3.1.5: Seams Carved: 250, vSeam Saliency: 27172, Saliency/Pixel: 210**

| Seams Carved | vSeam Saliency |
|---|---|
| 0 | 11919 |
| 50 | 16752 |
| 100 | 21251 |
| 150 | 22851 |
| 200 | 24719 |
| 250 | 27172 |
| 300 | 30049 |

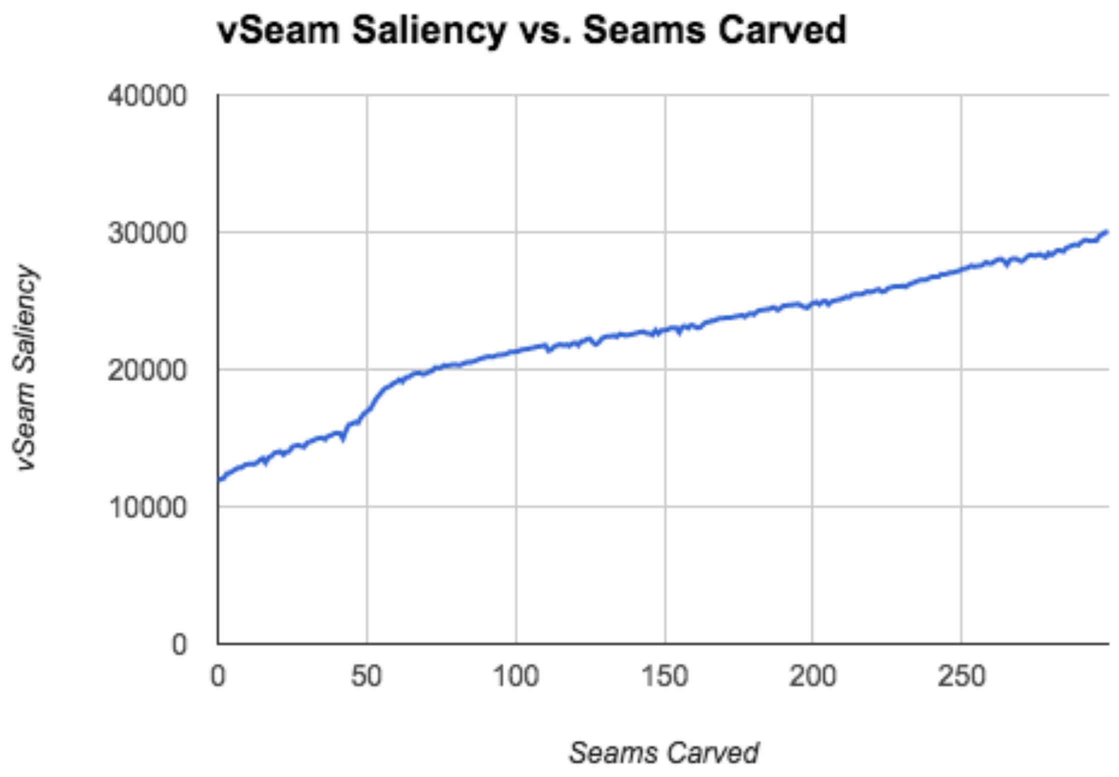


**Fig 3.1.6: Total saliency of pixels in vertical seam vs. Number of seams carved for "Tower.jpg"**

| vSeams Carved | Saliency/Pixel |
|---|---|
| 0 | 149 |
| 50 | 159 |
| 100 | 169 |
| 150 | 180 |
| 200 | 194 |
| 250 | 210 |
| 300 | 231 |
| 350 | 258 |
| 400 | 295 |

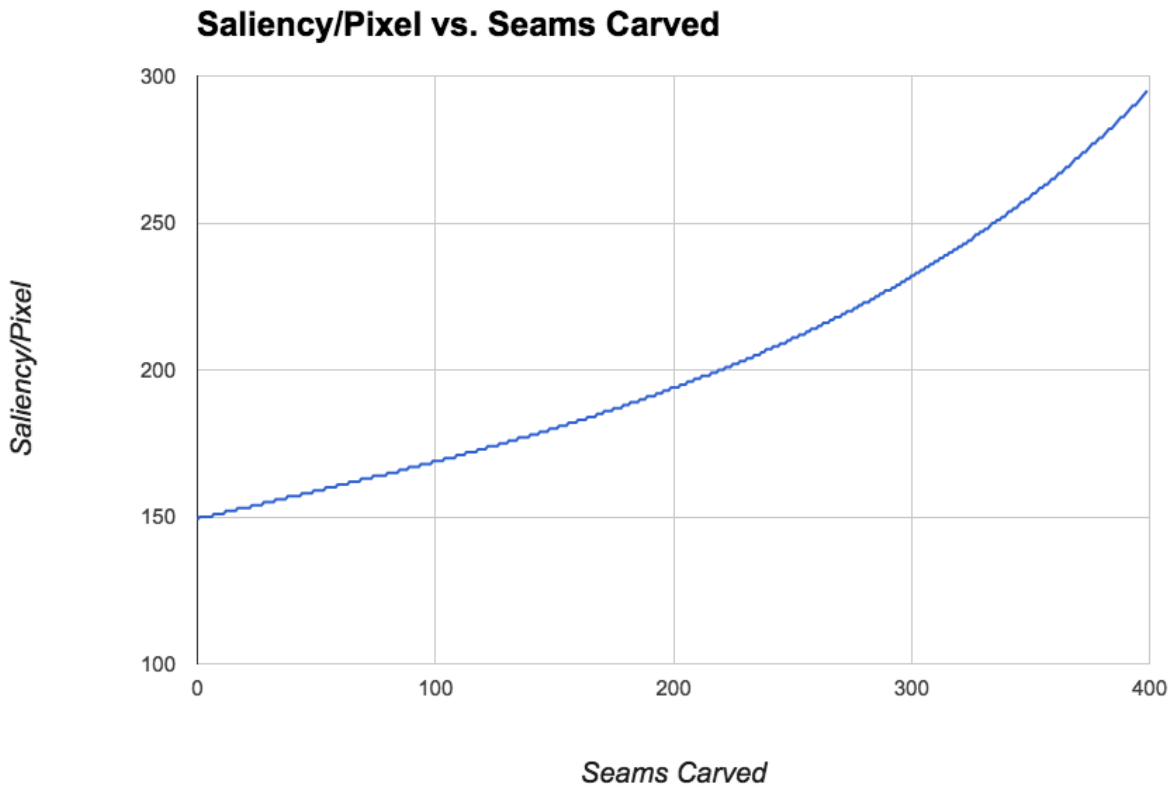


**Fig 3.1.7: Saliency per pixel vs. Number of seams carved for "Tower.jpg"**

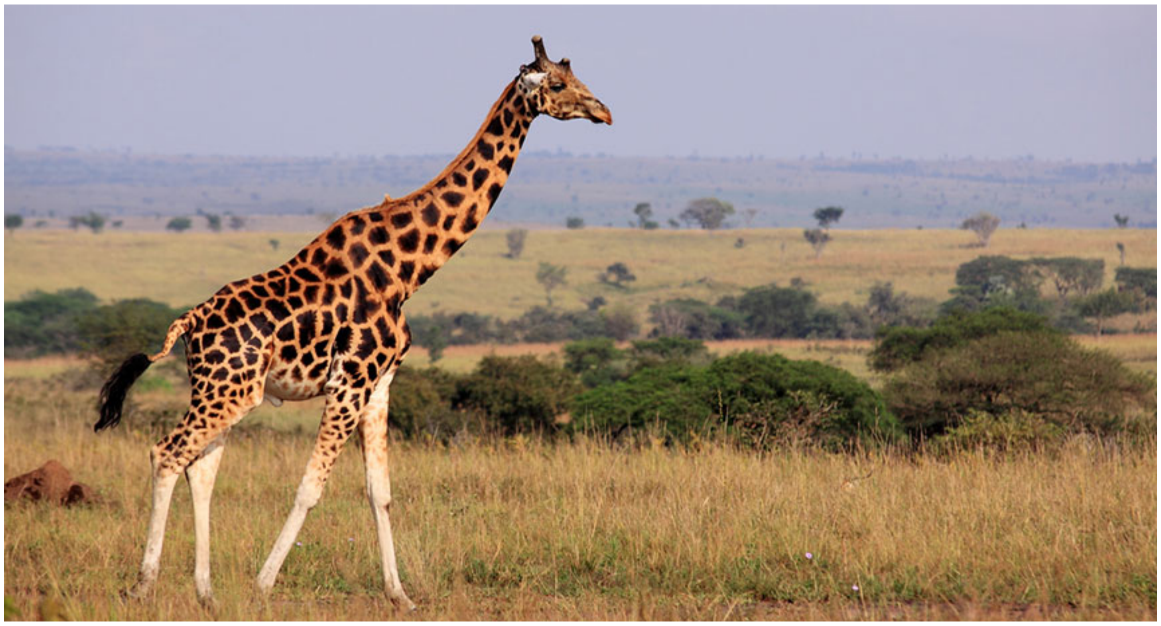


**Fig 3.2.1: Original image “Giraffe.jpg” — Dimensions: 860x460 pixels**

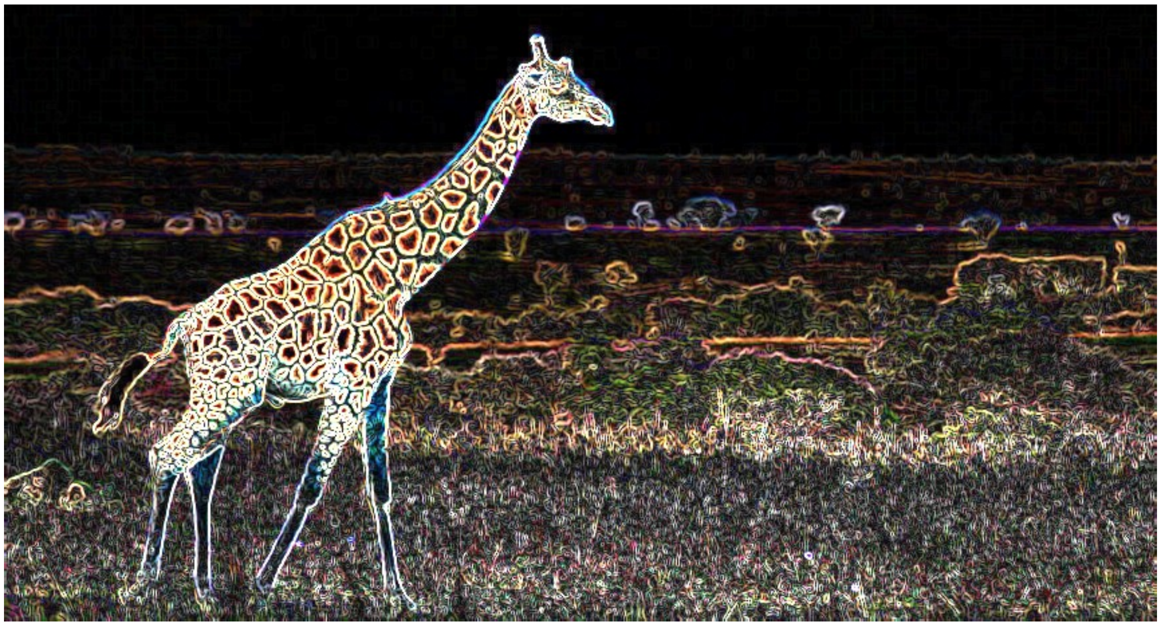


**Fig 3.2.1: Color Saliency Map**

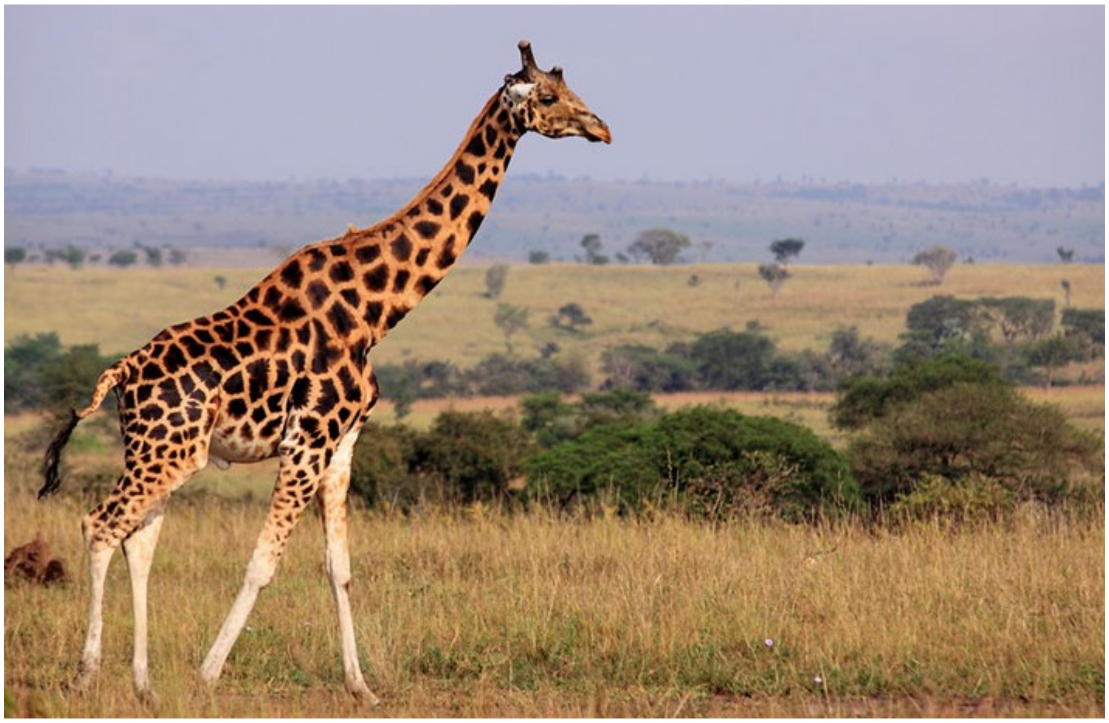


**Fig 3.2.3: Seams Carved: 150, vSeam Saliency: 36882, Saliency/Pixel: 204**

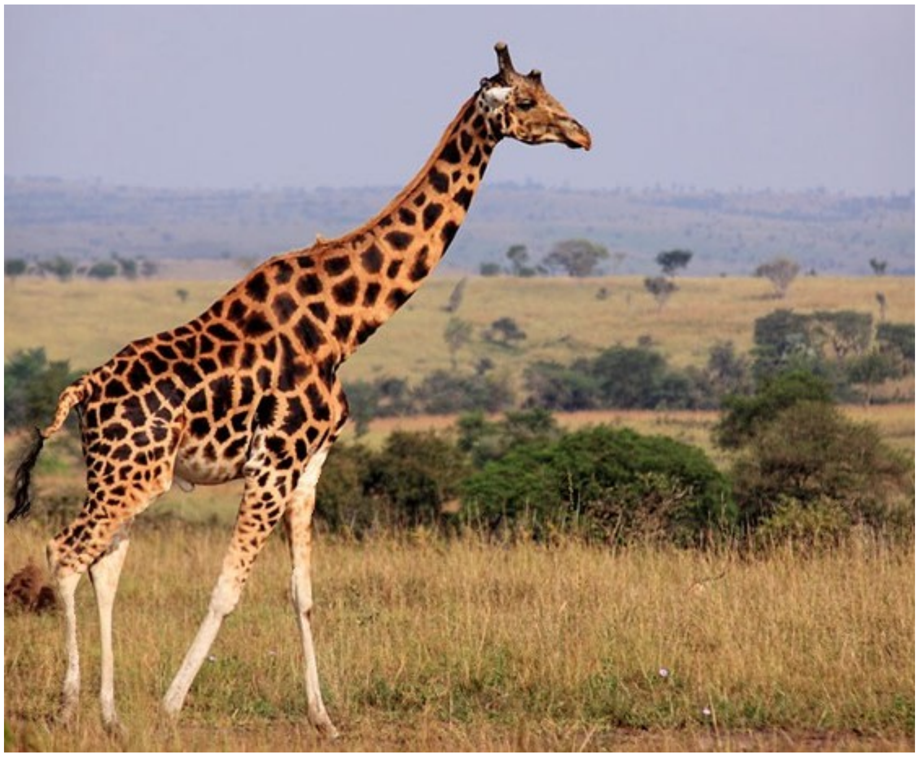


**Fig 3.2.4: Seams Carved: 300, vSeam Saliency: 41106, Saliency/Pixel: 228**

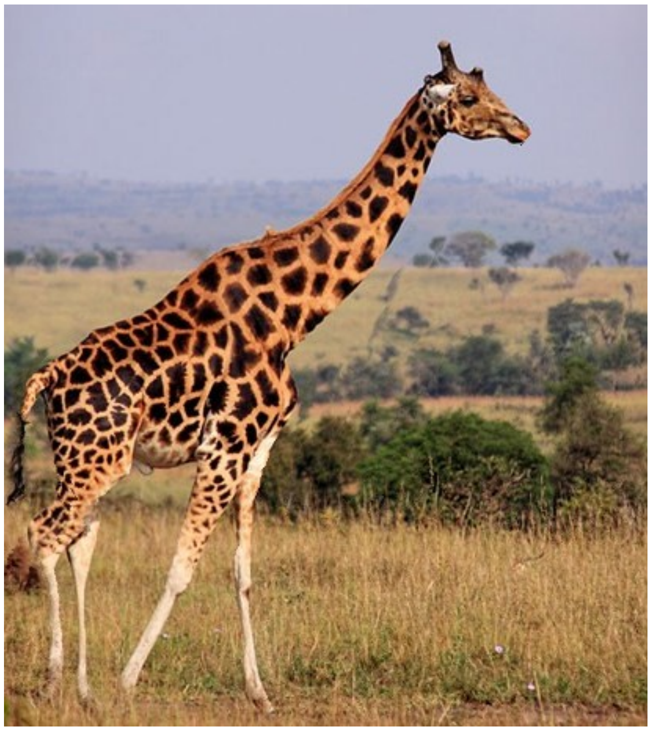


**Fig 3.2.5: Seams Carved: 450, vSeam Saliency: 46946, Saliency/Pixel: 263**

| **Seams Carved** | **vSeam Saliency** |
|---|---|
| 0 | 25145 |
| 50 | 32461 |
| 100 | 33978 |
| 150 | 36882 |
| 200 | 38373 |
| 250 | 39341 |
| 300 | 41106 |
| 350 | 43188 |
| 400 | 45111 |
| 450 | 46946 |

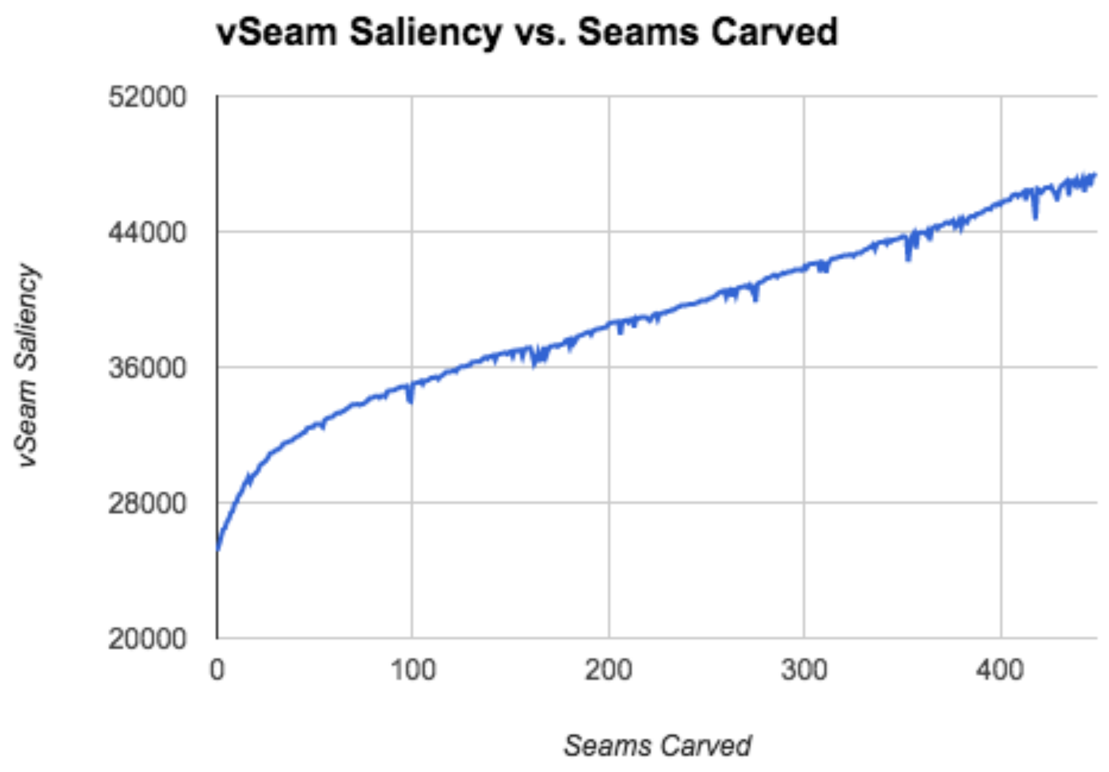


**Fig 3.2.6: Total saliency of pixels in vertical seam vs. Number of seams carved for "Giraffe.jpg"**

| vSeams Carved | Saliency/Pixel |
|---|---|
| 0 | 186 |
| 50 | 192 |
| 100 | 198 |
| 150 | 204 |
| 200 | 212 |
| 250 | 220 |
| 300 | 228 |
| 350 | 238 |
| 400 | 250 |
| 450 | 263 |

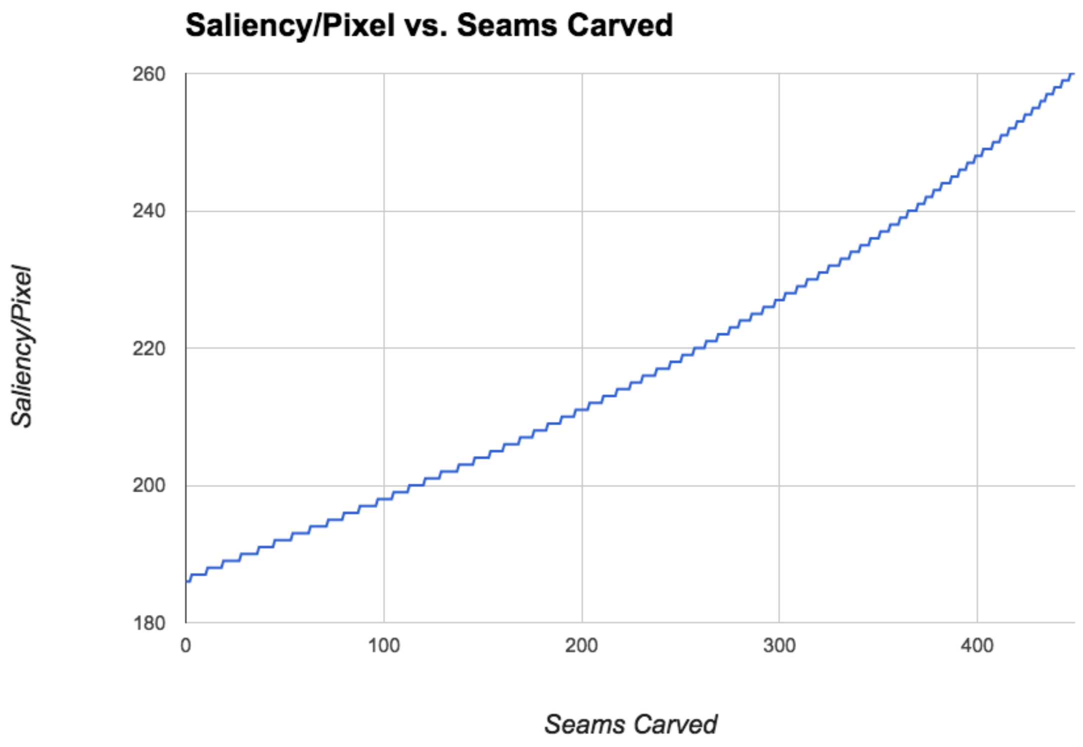


**Fig 3.2.7: Saliency per pixel vs. Number of seams carved for "Giraffe.jpg"**

## *3.2 Horizontal Seam Carving*

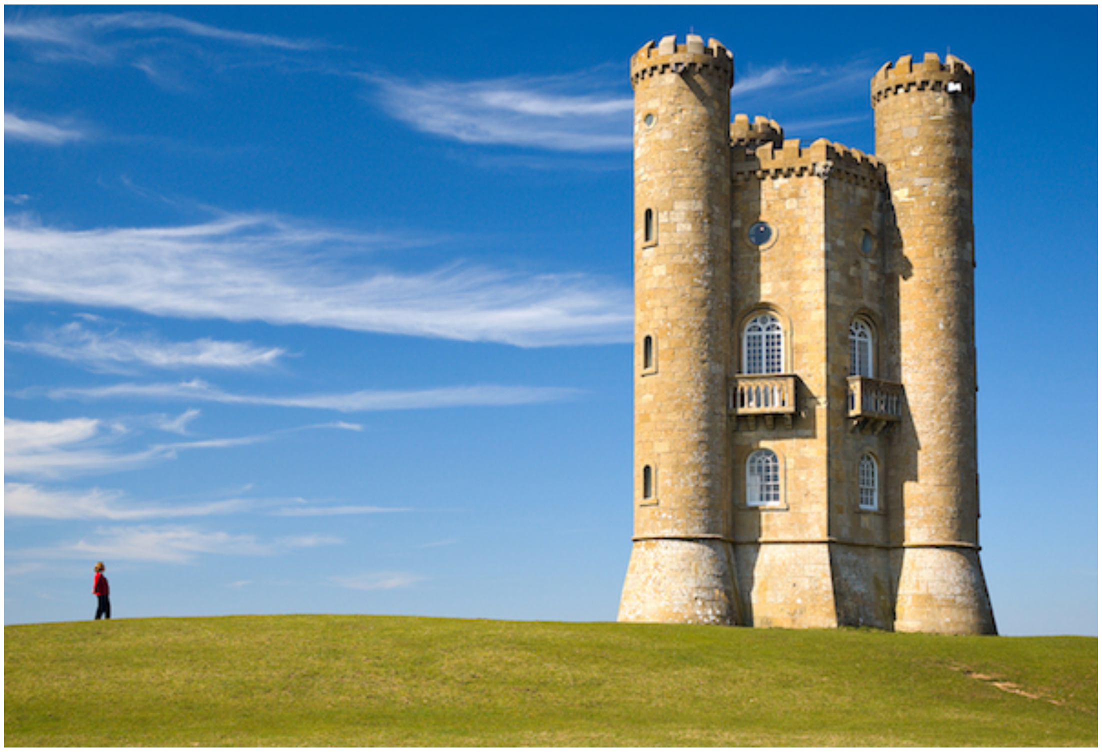


**Fig 3.3.1: Original Image “Tower.jpg” — Dimensions: 600x407 pixels**

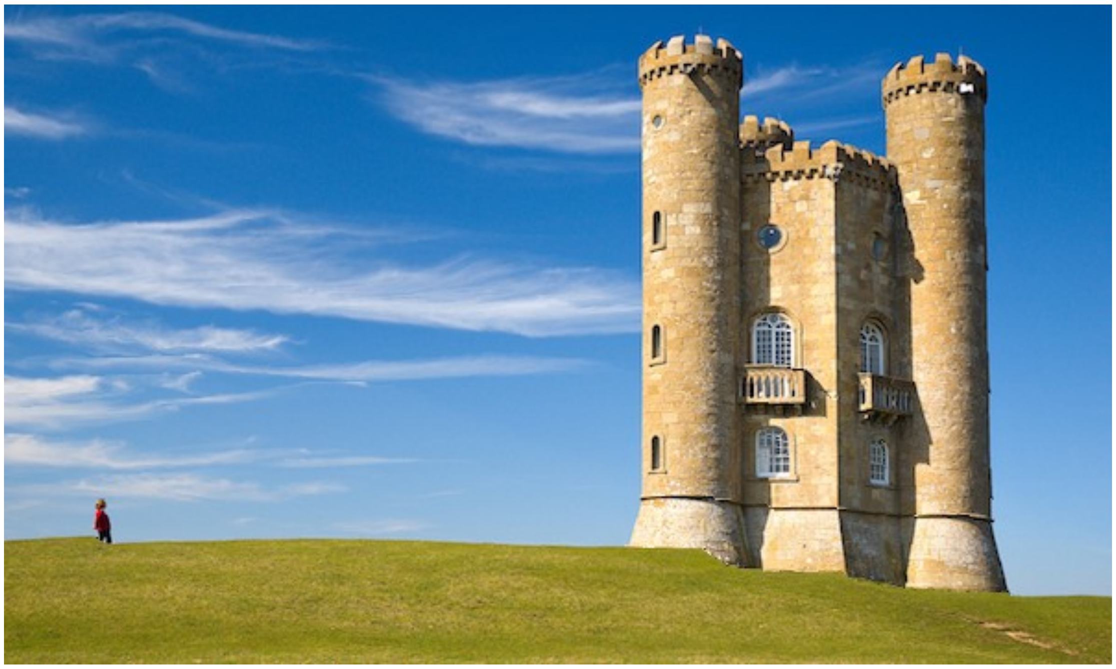


**Fig 3.3.2: Seams Carved: 50, hSeam Saliency: 52145, Saliency/Pixel: 150**

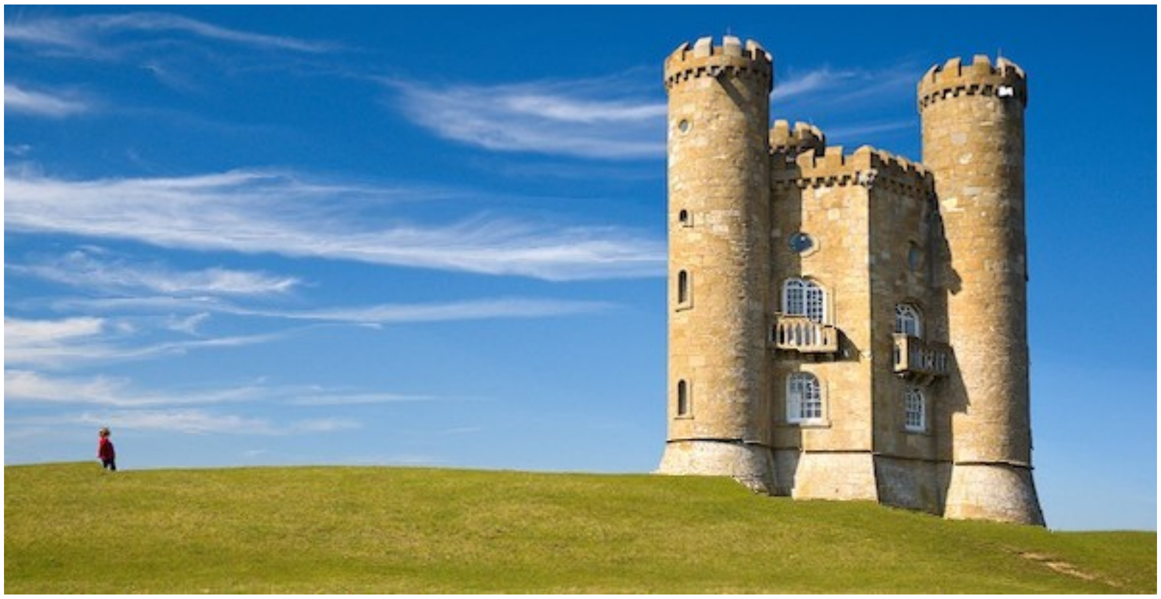


**Fig 3.3.3: Seams Carved: 100, hSeam Saliency: 54242, Saliency/Pixel: 160**

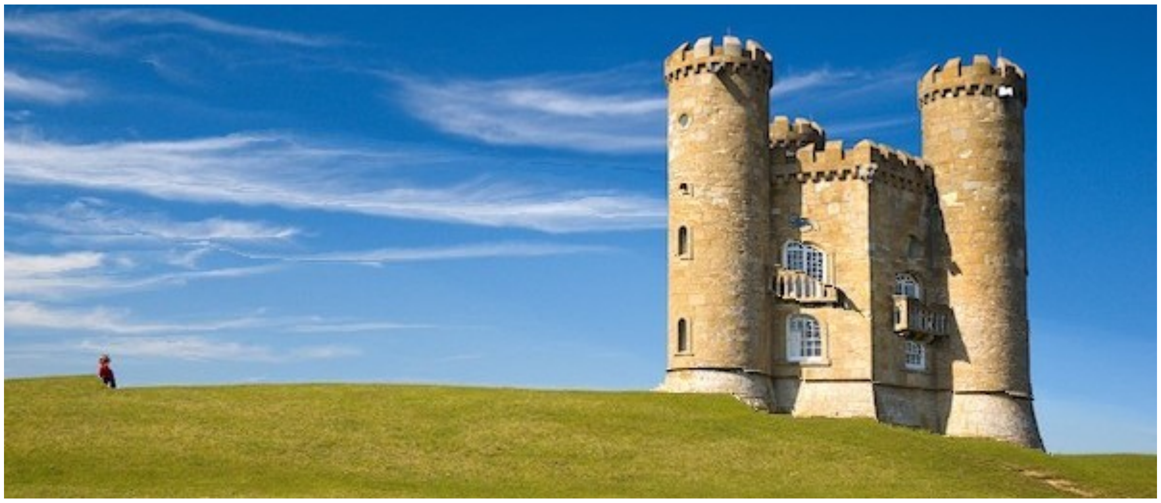


**Fig 3.3.4: Seams Carved: 150, hSeam Saliency: 53794, Saliency/Pixel: 165**

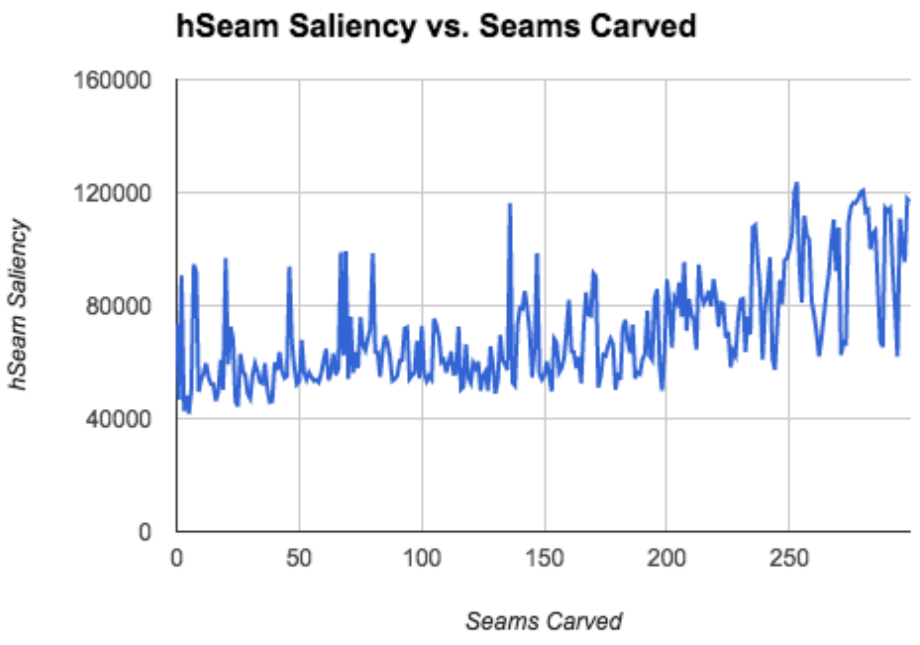


**Fig 3.3.5: Total saliency of pixels in horizontal seam vs. Number of seams carved for "Tower.jpg"**

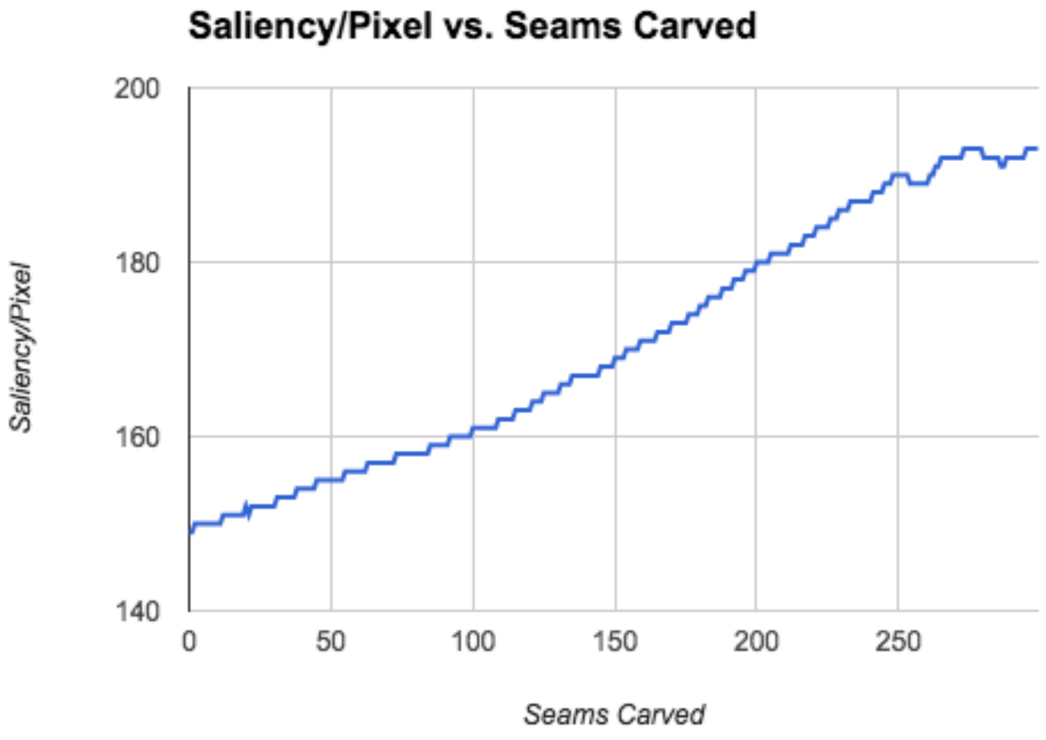


**Fig 3.3.6: Saliency per pixel vs. Number of seams carved for "Tower.jpg"**

*3.3 Seam Insertion*

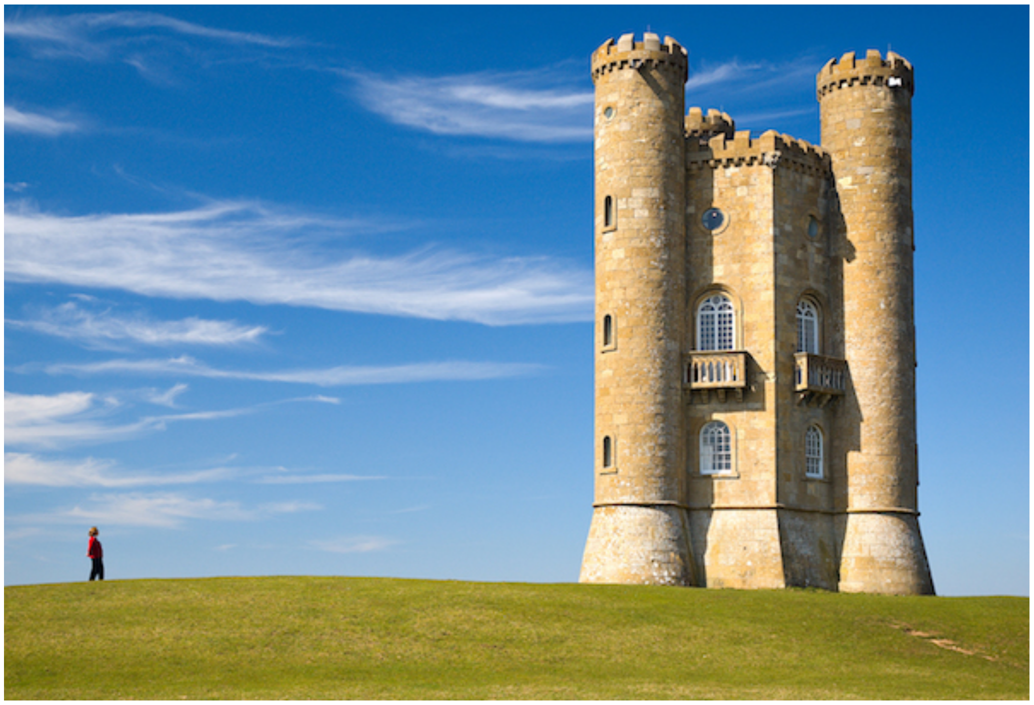


**Fig 3.4.1: Original Image to be expanded — Dimensions: 600x407 pixels**

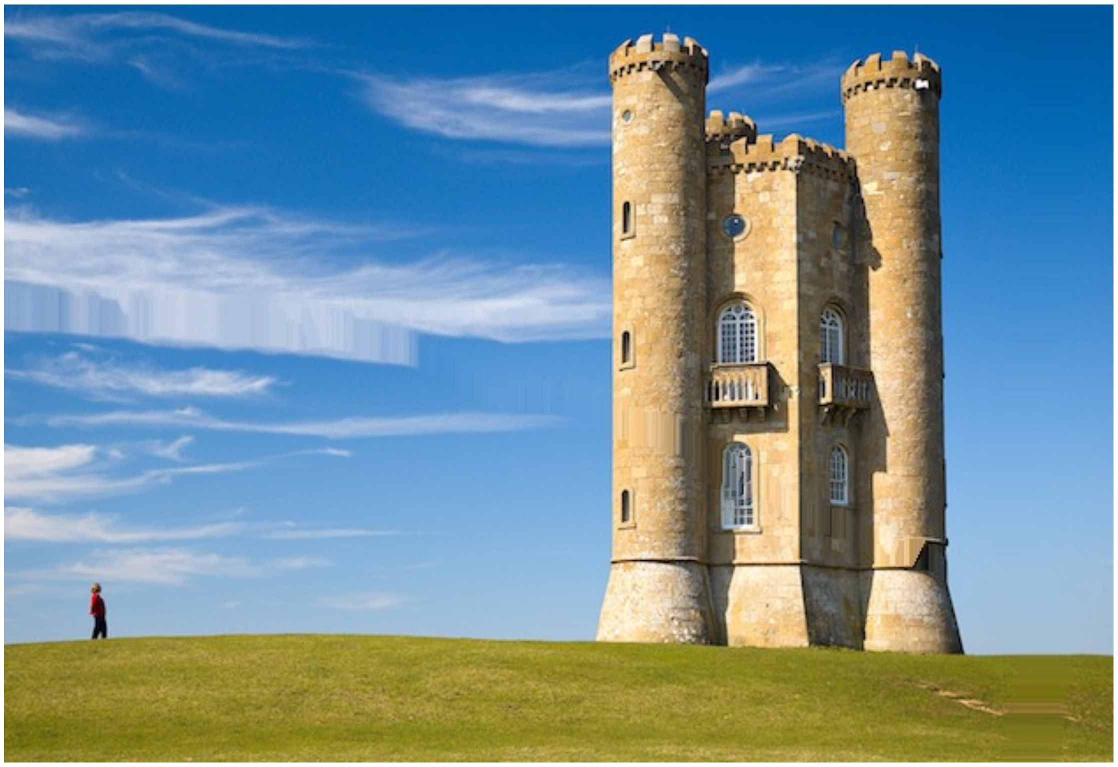


**Fig 3.4.2: Seams inserted — New dimensions: 630x430 pixels**

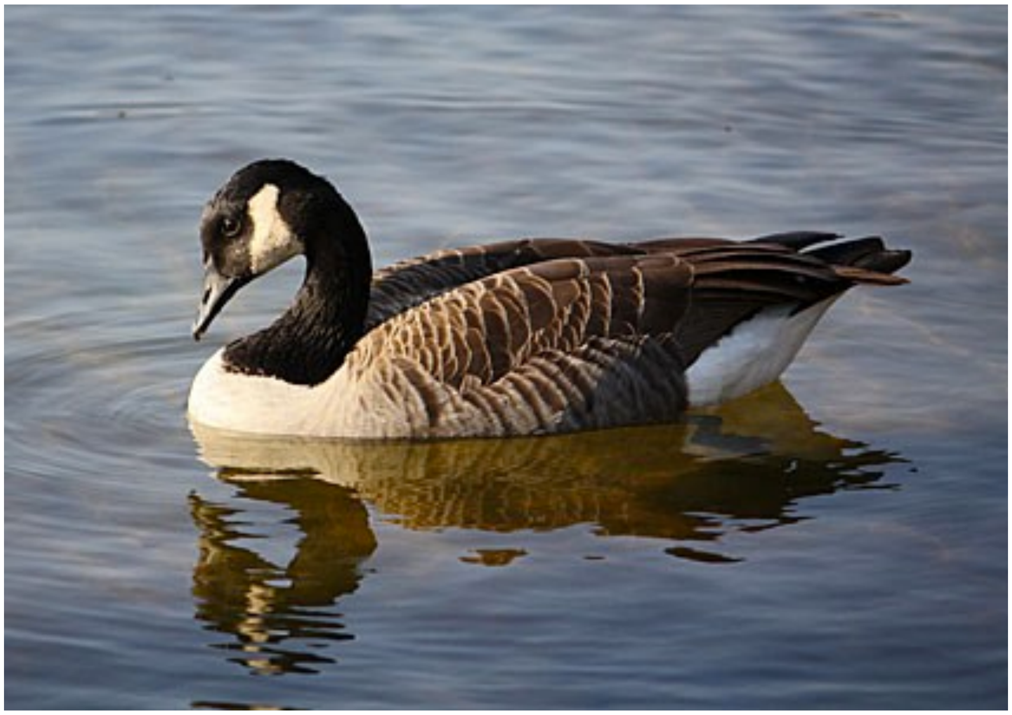


**Fig 3.4.1: Original Image to be expanded — Dimensions: 425x299 pixels**

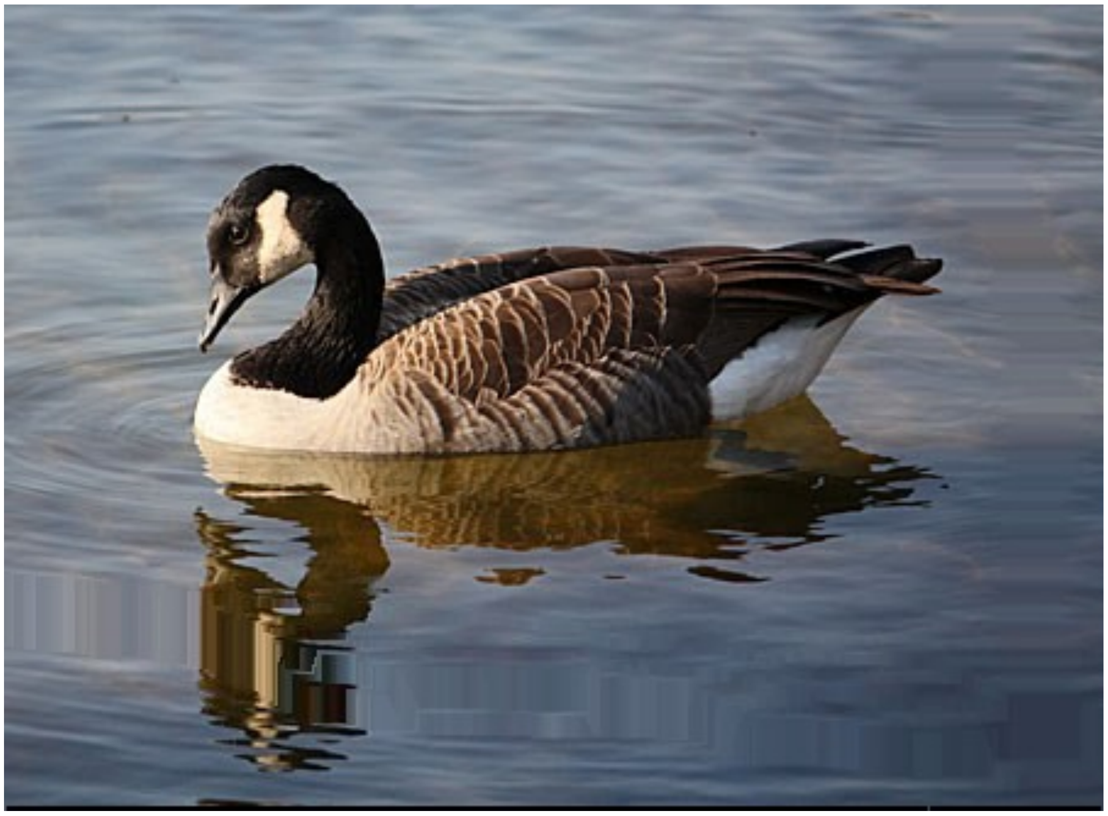


**Fig 3.4.2: Seams inserted — New dimensions: 450x330 pixels**

## *3.4 Failure Cases*

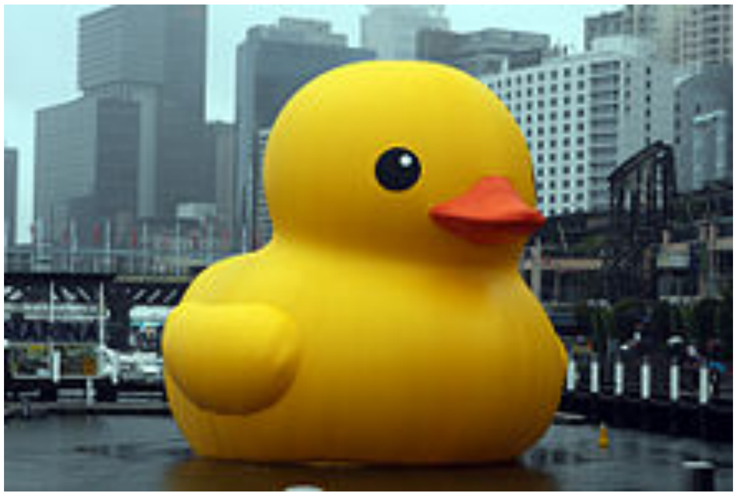


**Fig 3.5.1: Original Image to be carved — Dimensions: 220x147 pixels**

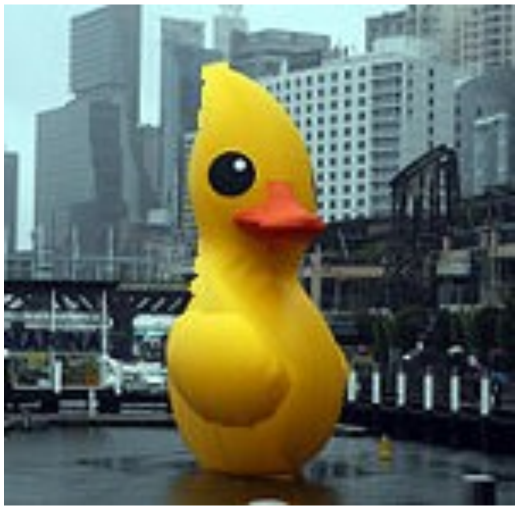


**Fig 3.5.2: Carving failure — New dimensions: 150x147 pixels**

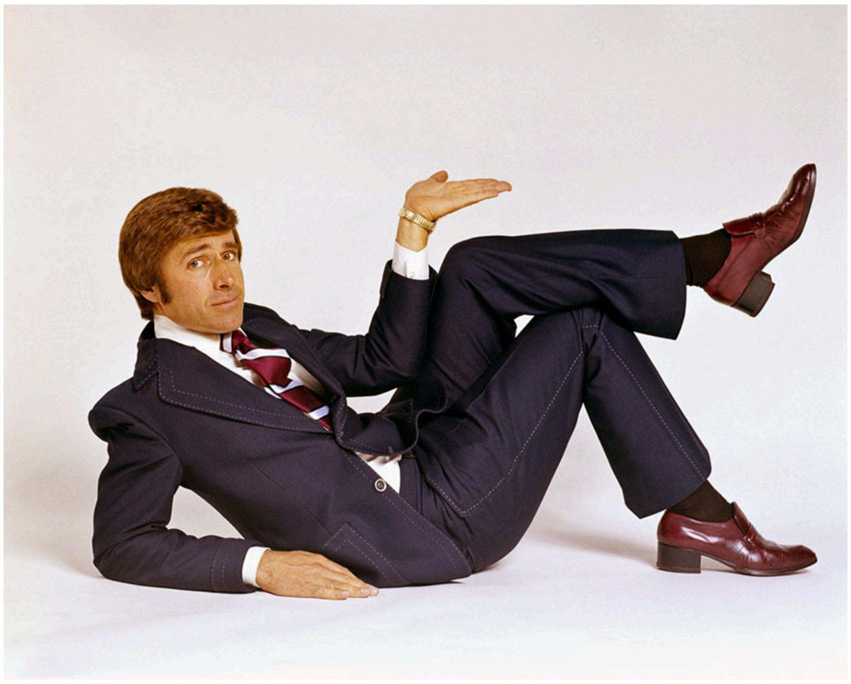


**Fig 3.6.1: Original Image to be carved — Dimensions: 880x704 pixels**

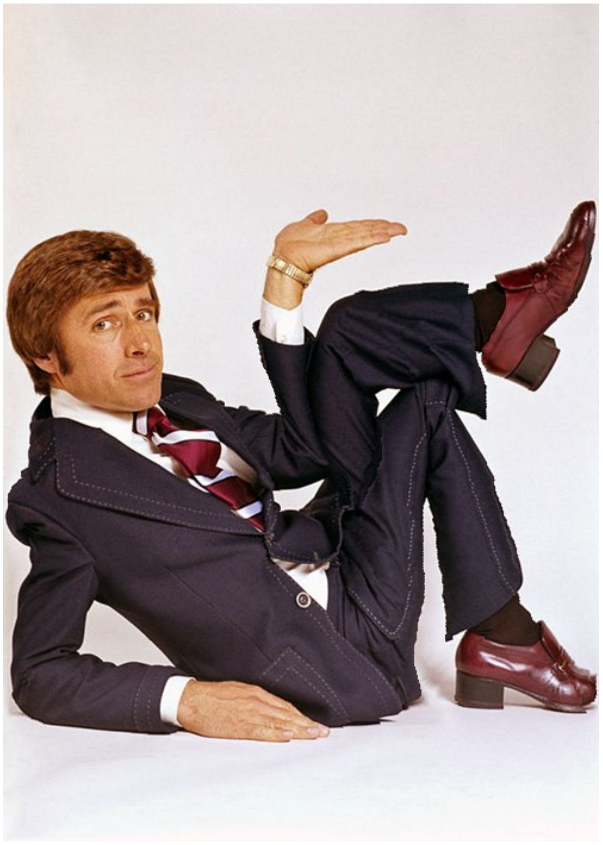


**Fig 3.6.2: Carving failure — New dimensions: 600x704 pixels**

# 4
# Analysis

## *4.1 The "10% Rule"*

While measuring the saliency per pixel in the vertical seam carving images, I noticed a trend. The first 150 or so seams that were carved seemed to increase the average pixel saliency by a regular amount. In **Fig 3.1.7**, we can see that from 0 seams carved to 150 seams carved, every 50 seams carved reduces the average pixel saliency by about 10. Once 200 seams had been carved, this number jumped to 15, and began to steadily increase. Carving seams 250 to 300 resulted in an increase of 21 in the average pixel saliency, and carving seams 400 to 450 increased the average saliency by a whopping 37.

In **Fig 3.2.7**, we see a similar trend. From 0 to 150 seams carved, the average pixel saliency increased by 6 for every 50 seams carved. From 150 to 300 seams carved that average increase rose to 8 per 50 seams carved. After 300 seams carved, this rate of increase only became greater.

So what's so special about the 150-seam mark? Time for some math…

**Fig 3.1.1** started with dimensions 600x407 pixels and an average pixel saliency of 149. This would make the total saliency of the image 36,385,800. After 150 seams carved, the image is now 450x407 pixels with an average pixel saliency of 180. This makes the total saliency 32,967,000, which is roughly 90.6% of the original saliency.

**Fig 3.2.1** started with dimensions 860x460 pixels and an average pixel saliency of 186. The total saliency of **Fig 3.2.1** would then be 73,581,600. After carving 150 seams, the image is now 710x460 pixels with an average pixel saliency of 204. The total saliency is then 66,626,400, or roughly 90.4% of the original saliency.

It seems as though removing a seam from the image will increase the average pixel saliency by about the same amount for all seams removed before 10% of the image's original saliency has been carved. After that, removing seams will begin to increase the average saliency by more and more for each seam removed.

But don't take my word for it. Let's look at some data…

This time, let's use a much wider image, so that it will take the removal of more than 150 seams to reduce the image to 90% original saliency.

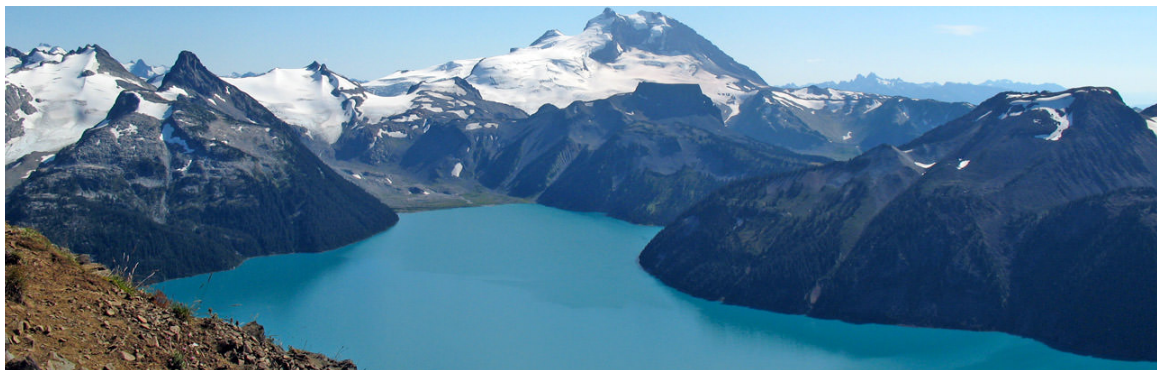


**Fig 4.1: Original Image "Landscape.jpg" — Dimensions: 1280x404 pixels**

| vSeams Carved | Saliency/Pixel |
|---|---|
| 0 | 132 |
| 75 | 138 |
| 150 | 144 |
| 225 | 150 |
| 300 | 155 |
| 375 | 164 |
| 450 | 173 |
| 525 | 184 |
| 600 | 196 |
| 675 | 210 |
| 750 | 228 |

**Fig 4.2: Saliency per pixel vs. Number of seams carved for "Landscape.jpg"**

For this image, the original size is 1280x404 pixels with an average pixel saliency of 132, making the total saliency 68,259,840. By 300 seams removed, the size is 980x404 pixels with an average pixel saliency of 155. The total saliency is then 61,367,600, or 89.9% of the original saliency.

So let's find out if our hypothesis holds — that after 10% of the original saliency has been removed, the rate at which removing seams increases the average pixel saliency begins to grow.

From 0 seams carved to 225 seams carved, every 75 seams removed increases the average pixel saliency by 6. From 255 to 300, it only increases the average by 5, but that's still not an increase. From 300 to 525 seams removed (after 10% of the original saliency is gone) the average saliency goes up to an increase of 9 per 75 seams. Then it grows to 12 per 75 seams, then 14, then 18…

We can see that the increase in average pixel saliency growth per seam removed isn't tied to the 150 seam mark, but rather to the removal of 10% of the original saliency.

Whatever is happening at the 10% mark must be a characteristic of these images, not of seam carving itself. Seam carving can take any rectangular image as an input, including images that have previously been carved. Since this is the case, one could therefore take an image that has already had 10% of the original saliency carved out of it and define that as an original image. Seam carving can't tell whether the image is original or not.

This "10% rule" therefore cannot possibly apply to all images, and must have something to do with the image content. Whether this "10% rule" is a figment of my imagination or there is some explanation involving the distribution of saliency in landscape images, I don't know. Perhaps someone else will write a senior project involving saliency and solve this mystery for me.

## *4.2 Vertical Seam Carving Analysis*

When examining the saliency of seams that are being carved, it became apparent that the more seams that had been carved before the seam in question, the higher the saliency of that seam was likely to be. This does not, however, mean that every seam will have a higher saliency than the one carved before it. Despite the saliency of the seam having an upward trend as more of them are carved, it appears quite common for seams to drop in saliency.

When a seam is removed, pixels on opposite sides of the seam that were previously not in contact with each other may now touch. This opens up new pathways for the seam-finding algorithm described in Section 2.2, and can potentially lead to the discovery of new low-saliency seams. This could also be the result of the saliency map changing. Since the saliency map must be recalculated after every seam is carved, it is certainly possible that areas of the map that previously had high saliency could have reduced saliency as the image around them changes.

## *4.3 Horizontal Seam Carving Analysis*

Carving the horizontal seams showed a much less consistent trend in seam saliency vs. seams carved. While there was still a definite upward trend, the value of the seam’s total saliency tended to jump around quite a bit more than the vertical seam. This is probably due to the fact that it is much harder to draw a horizontal path across the image that doesn’t intersect with the content of the image. Vertical seams can simply go between objects, but horizontal seams have to try to go above or below them. When you have a tall object like the tower in the example used, it is quite difficult to carve horizontal seams without going through the tower. This also means that horizontal seam carving can tend to be less "clean" than vertical seam carving as it is more likely to disrupt salient content.

## *4.4 Seam Insertion Analysis*

While the idea of using the seam-finding algorithm to insert seams is quite promising, in practice it has issues. When inserting a seam, the colors of the pixels inserted are chosen by finding the average of their neighbor's colors. This is to make the new pixels blend into the image well. However, they seem to blend *too* well for their own good. The fact that these pixels are very similar to their neighbors means they will naturally have very low saliency. Thus, after a seam is inserted, the next seam that is chosen is very likely to be the same as the first. The lowest-saliency seam is not removed — its saliency is lowered even further by the introduction of the new seam of pixels that strongly resembles it.

When inserting multiple seams, this can result in seams of the image being repeated over and over. If this seam only passes through a clear blue sky it may not be noticeable, but if there is salient content involved it can create a disruption in the image.

Take for example **Fig 3.4.2** from **Section 3.3**…

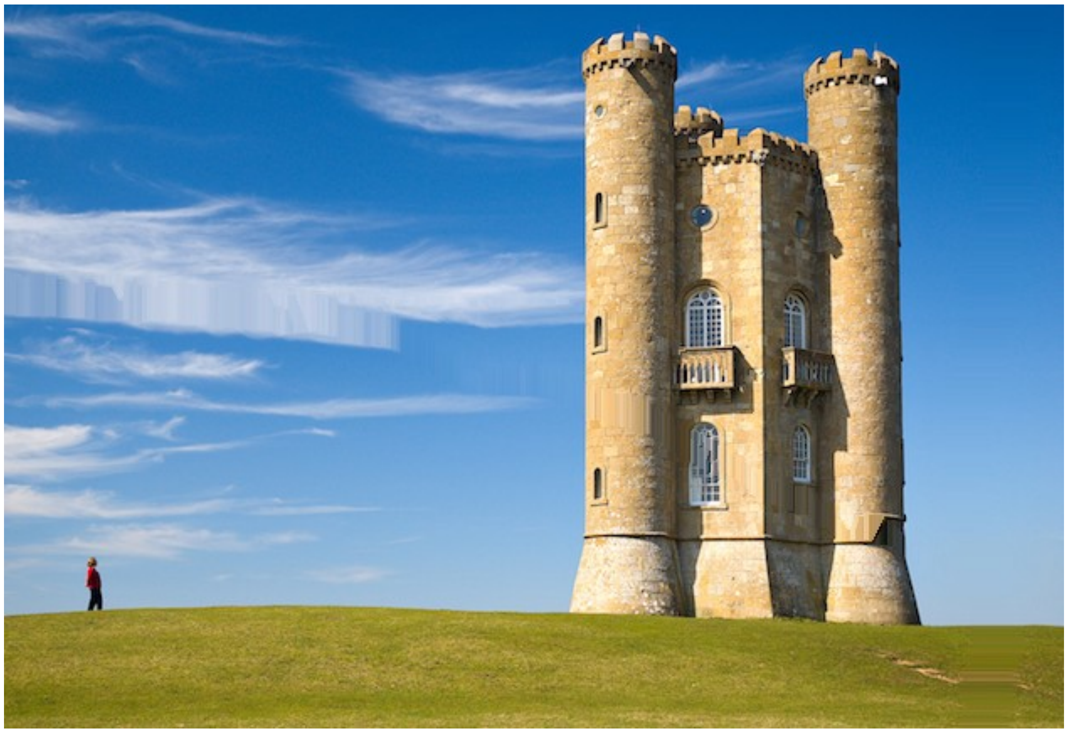


**Fig 3.4.2: Seams inserted — New dimensions: 630x430 pixels**

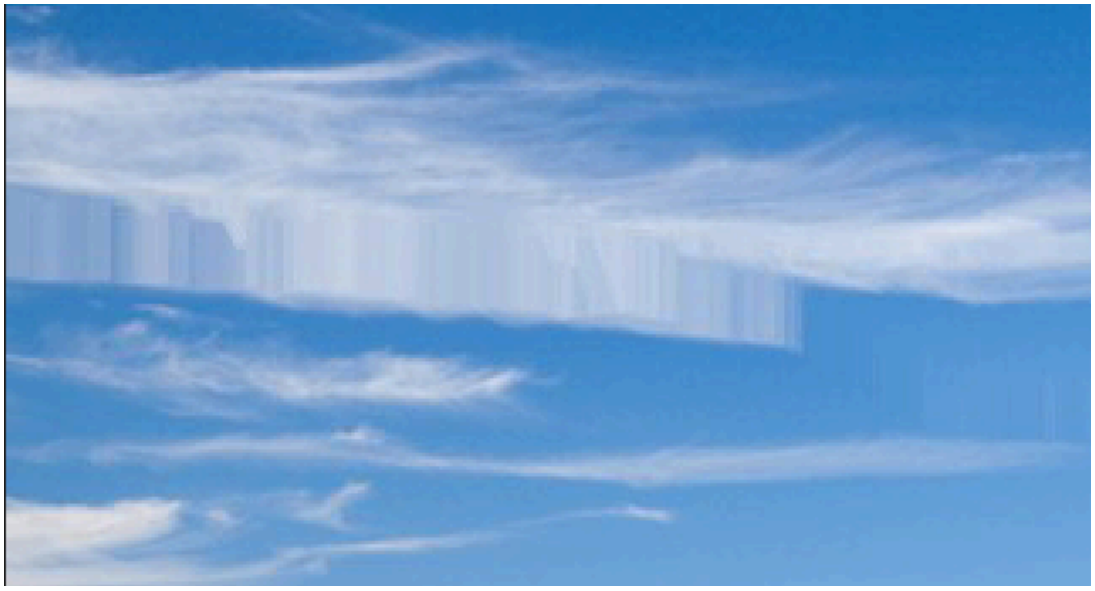


**Fig 4.3: Clouds being disrupted by inserted horizontal seams**

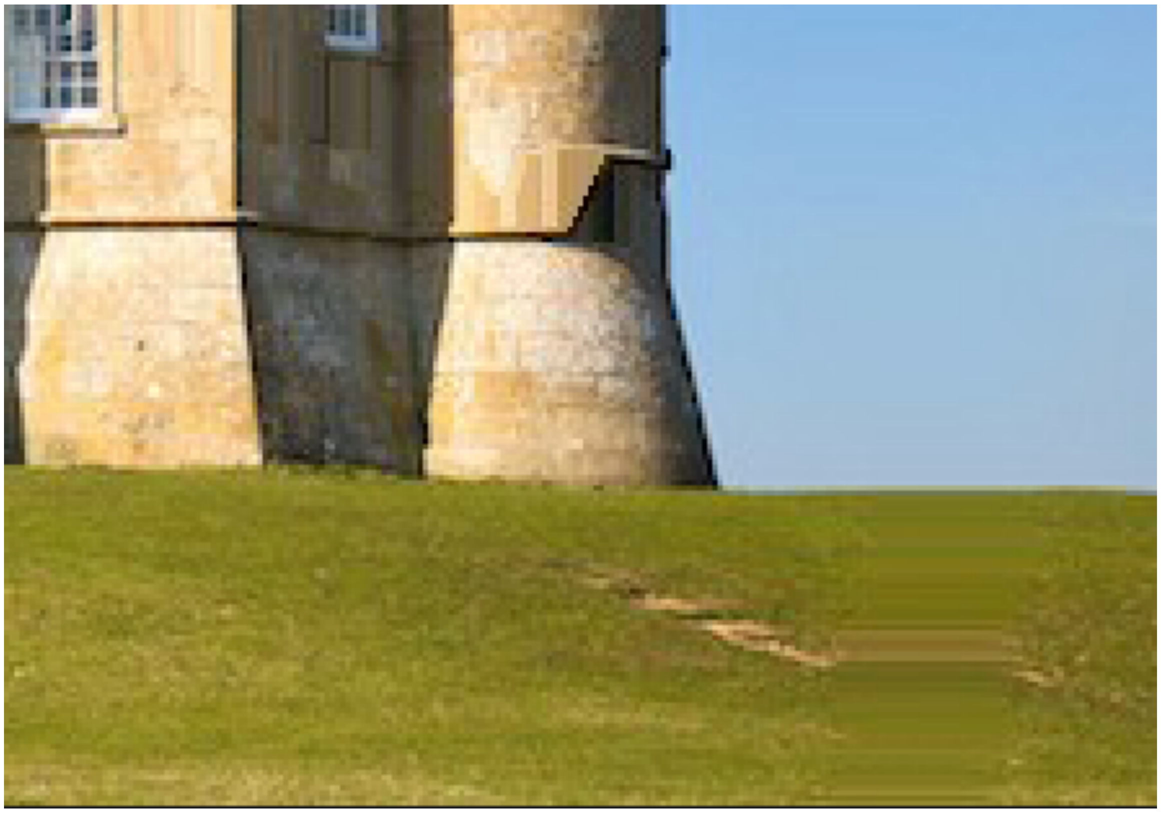


**Fig 4.4: Grass disturbed by inserted vertical seams**

While in some cases these disruptions in image coherency are not noticeable, for many images this is a major distraction. This problem makes seam insertion less desirable than other texture synthesis methods.

I would like to see someone take an approach on this that would prevent the seam-finding algorithm from selecting the same seam over and over again, perhaps by artificially increasing the saliency of that seam after insertion. The seam insertion idea is sound if the seams are not bunched together.

## *4.5 Failure Case Analysis*

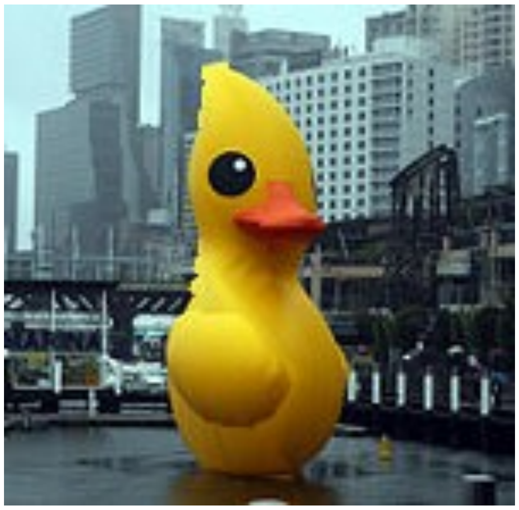


**Fig 3.5.2: Carving failure — New dimensions: 150x147 pixels**

In this case, the algorithm fails to realize that the giant rubber duck is actually the focus of the image. Because the majority of the duck is made up of one uniform color, the saliency map gives that whole area a low saliency value, allowing seams to be carved through the poor, defenseless duck.

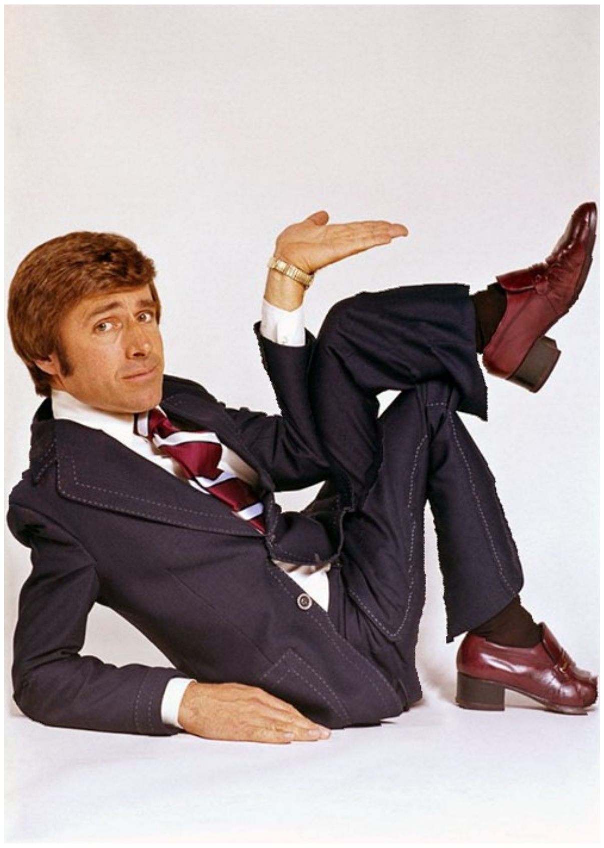


**Fig 3.6.2: Carving failure — New dimensions: 600x704 pixels**

This is just a case of trying to condense an image further than it should go. If you really wanted to reduce the width of this image this much, you should probably just crop out the man’s feet. But because seam carving is trying to preserve them, it ends up squishing his body/limbs and messing with his proportions. I would stress, however, that unlike the previous failure case, this image is relatively clean. Yes, his proportions are a bit off, but for a “failure case” it looks pretty god.

## *4.6 Conclusion*

On the whole I am highly impressed with seam carving. If I had more time to work on this project, I would be trying to improve the runtime of my seam carving implementation. At present it can take quite a while to work on a larger images (1000 pixels or more pixels per seam). I am certain that the version I wrote can be improved upon and optimized to streamline the algorithm's efficiency.

I would also have liked to implement a version of seam insertion that doesn't choose the same seam repeatedly. It may be possible to restructure the code so that the saliency map for the next seam is actually generated as a part of each seam insertion. This would mean that I would be able to artificially increase the saliency of both the seam selected for that iteration of insertion and the seam it inserts, forcing the seam-finding algorithm to find a new place to insert pixels. This would theoretically make seam insertion far less disruptive to the image.

As for the actual results, seam carving works quite consistently, though it is definitely a tool for a very specific job. For some images it can seem like nothing more than a glorified cropping tool, but it still gets the job done. Where this algorithm really shines is in images with salient content on both sides of the image. Those images are quite difficult to resize using traditional methods, but seam carving will almost always give you a clean result. Sure, there are some cases in which it fails to meet the goal of leaving salient content undisturbed, but I had to go through countless images that were carved properly in order to find just a few that weren't. For most images this tool will spit out a beautiful result that is, for lack of a better word, seamless.

# Bibliography

[1] Shai Avidan and Ariel Shamir. 2007. Seam carving for content-aware image resizing. *ACM Trans. Graph.* 26, 3, Article 10 (July 2007).

[2] Alexei A. Efros and William T. Freeman. 2001. Image Quilting for Texture Synthesis and Transfer. *Proceedings of the 28th annual conference on computer graphics and interactive techniques.* Pages 341 - 346 (August 2001).

[3] Irwin Sobel. 2014. History and Definition of the so-called "Sobel Operator", more appropriately named the Sobel-Feldman Operator. (2014). https://www.researchgate.net/publication/239398674_An_Isotropic_3_3_Image_Gradient_Operator

## Resources

*If a source is not listed for a figure, then was either derived from a previous figure that does have a source, or was created by the author of this paper.*

Fig 1.1: http://www.jqueryscript.net/images/jQuery-Plugin-for-Image-Cropping-Functionality-imgAreaSelect.jpg

Fig 1.2: https://www.wildgratitude.com/wp-content/uploads/2015/07/ladybug-spirit-animal.jpg

Fig 1.3: http://www.acfe.com/uploadedImages/ACFE_Website/Content/images/membership-certification/graduate-member.jpg

Fig 1.4: https://en.wikipedia.org/wiki/Seam_carving

Fig 2.2: http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm

Fig 2.5 - 2.7: https://en.wikipedia.org/wiki/Seam_carving

Fig 2.13: http://www.costafarms.com/CostaFarms/Costa-Farms-Cactus-Notocactus-Leninghausii.jpg?height=257&width=256&scale=both&crop=auto

Fig 3.2: https://www.sciencenews.org/sites/default/files/main/blogposts/wt_giraffeneck_free.jpg

Fig 3.4: https://www.allaboutbirds.org/guide/PHOTO/LARGE/canada_goose_3.jpg

Fig 3.5:

https://upload.wikimedia.org/wikipedia/commons/thumb/1/14/Rubber_Duck_%288374802487%29.jpg/220px-Rubber_Duck_%288374802487%29.jpg

Fig 3.6: http://static.boredpanda.com/blog/wp-content/uploads/2016/06/1970-stock-photos-robert-armstrong-14-575ea42b5e336__880.jpg

Fig 4.2: https://www.vancouvertrails.com/images/hikes/panorama-ridge.jpg

## Appendix

## Full Code:

```
// Seam Carving

// as implemented by Race D. Morel

double[][] kernelx = {{ -1, +0, +1},
                      { -2, +0, +2},
                      { -1, +0, +1}};

double[][] kernely = {{ +1, +2, +1},
                      { +0, +0, +0},
                      { -1, -2, -1}};

PImage img;
int targetwidth = 600; //set target dimensions
int targetheight = 407;

void setup(){
  img = loadImage("tower.jpg");
  size(targetwidth, targetheight);
  img.loadPixels();
  carve(img);
}

void carve(PImage img){

  while (targetwidth < img.width){
     img = vertseam(img, sobel(img));
  }

  while (targetheight < img.height){
     img = horizseam(img, sobel(img));
  }
  while (targetwidth > img.width){
     img = addvert(img, sobel(img));
```

```
    }
    while (targetheight > img.height){
        img = addhoriz(img, sobel(img));
    }

    img.save("carved.jpg");
    System.out.println("carved");
  }

  PImage vertseam(PImage original, PImage filtered){
    img = filtered; // image that has been put through sobel filter
    img.loadPixels();

    VNode[] nodeArray = new VNode[img.pixels.length];
                          // create array of Nodes
     for (int i = 0; i<img.pixels.length; i++){// for each pixel,
                                           create corresponding Node
       int ypos = i/img.width; // calculating x and y coordinates
                          based on position in array
       int xpos = i - (ypos * img.width);
        nodeArray[i] = new VNode (xpos, ypos, nodeArray, img, i);
        nodeArray[i].setY();
        nodeArray[i].setX();
     }
     for (int i = 0; i<nodeArray.length; i++){
        nodeArray[i].setParent();// set parent for each node
     }
     int[] bottomRow = new int[img.width];// array containing
                          positions of bottom nodes in nodeArray
     int num = 0;
     for (int i = 0; i<nodeArray.length; i++){
       if (nodeArray[i].getY() == img.height - 1){
             bottomRow[num] = i;num++;
       }
     }
     int root = bottomRow[0]; // root = index of bottom node in the
                                     lowest energy seam
     double lowest = nodeArray[root].getSum();
     for (int i = 1; i < bottomRow.length; i++){
       double summ = nodeArray[bottomRow[i]].getSum();
       if (summ < lowest){
             lowest = summ;
             root = bottomRow[i];
       }
     }
     VNode[] vertSeam = new VNode[img.height]; // contains all
```

```
                                nodes in lowest energy seam
   int[] seamIndex = new int[vertSeam.length];// array containing
                                indexes of all pixels in seam
   VNode pixel = nodeArray[root];
   for (int i = 0; i<vertSeam.length; i++){
      vertSeam[i] = pixel;
      seamIndex[i] = pixel.getIndex();
      pixel = pixel.getparent();
   }

   img.updatePixels();

   PImage post = createImage(img.width - 1 , img.height,RGB);
                    //final image
   post.loadPixels();
   int j = 0;
   boolean inSeam = false;
   for (int i = 0; i < img.pixels.length; i++){
     for (int x = 0; x < seamIndex.length; x++){
       if (i == seamIndex[x]){ // check to see if pixel is in the
                                seam
           inSeam = true;
       }
     }
     if (inSeam == false) { // if pixel is not in seam, copy it
                           into new image
           post.pixels[j] = original.pixels[i];
           j++;
     }
     else { inSeam = false; } // if pixel is in seam, skip it and
                                reset inSeam variable
   }
   post.updatePixels();
   image(post,0,0);
   return(post);
}

PImage horizseam(PImage original, PImage filtered){
  img = filtered; // image that has been put through sobel filter
  img.loadPixels();
  HNode[] nodeArray = new HNode[img.pixels.length];// create
                                        array of Nodes
  for (int i = 0; i<img.pixels.length; i++){// for each pixel,
                                create corresponding Node
    int ypos = i/img.width; // calculating x and y coordinates
                           based on position in array
```

```
 int xpos = i - (ypos * img.width);
 nodeArray[i] = new HNode (xpos, ypos, nodeArray, img, i);
 nodeArray[i].setY();
 nodeArray[i].setX();
}
for (int i = 0; i<nodeArray.length; i++){
 nodeArray[i].setParent();// set parent for each node
}
int[] leftSide = new int[img.height];// array containing
              positions of leftmost nodes in nodeArray
int num = 0;
for (int i = 0; i<nodeArray.length; i++){
  if (nodeArray[i].getX() == 0){
    leftSide[num] = i;
    num++;
  }
}
int root = leftSide[0]; // root = index of leftmost node in the
                         lowest energy seam
double lowest = nodeArray[root].getSum();
for (int i = 1; i < leftSide.length; i++){
  double summ = nodeArray[leftSide[i]].getSum();
  if (summ < lowest){
    lowest = summ;
    root = leftSide[i];
  }
}
HNode[] horizSeam = new HNode[img.width]; // contains all nodes
                                in lowest energy seam
int[] seamIndex = new int[horizSeam.length];// array containing
                           indexes of all pixels in seam
HNode pixel = nodeArray[root];
for (int i = 0; i<horizSeam.length; i++){
 horizSeam[i] = pixel;
 seamIndex[i] = pixel.getIndex();
 pixel = pixel.getparent();
}
img.updatePixels();

PImage post = createImage(img.width, img.height -1, RGB);
                                             //final image
post.loadPixels();
for (int x = 0; x < original.width; x++){ //for each x
 boolean carved = false;
 for (int y = 0; y < original.height-1; y++){ //iterate through
                                             the column
```

```
      if (carved == false){
         for (int i = 0; i < seamIndex.length; i++){
            if (y*img.width + x == seamIndex[i]){ // check to see
                                                   if pixel is in seam
                carved = true;
                int y2 = y;
                while (y2 < post.height){ // copy pixels into
                     space above original location
                     post.pixels[y2*original.width + x] =
                     original.pixels[(y2+1)*img.width + x];
                     y2++;
                }
           }
           else{ // if above seam, just copy pixel normally
                post.pixels[y*original.width + x] =
                original.pixels[y*original.width + x];
           }
        }
     }
   }
  }

  post.updatePixels();
  image(post,0,0);
  return(post);
}

PImage addvert(PImage original, PImage filtered){
  img = filtered; // image that has been put through sobel filter
  img.loadPixels();

  VNode[] nodeArray = new VNode[img.pixels.length];// create
                                                   array of Nodes
  for (int i = 0; i<img.pixels.length; i++){// for each pixel,
                                          create corresponding Node
   int ypos = i/img.width; // calculating x and y coordinates
                       based on position in array
   int xpos = i - (ypos * img.width);
   nodeArray[i] = new VNode (xpos, ypos, nodeArray, img, i);
   nodeArray[i].setY();
   nodeArray[i].setX();
  }
  for (int i = 0; i<nodeArray.length; i++){
   nodeArray[i].setParent();// set parent for each node
  }
```

```
int[] bottomRow = new int[img.width];// array containing
              positions of bottom nodes in nodeArray
int num = 0;
for (int i = 0; i<nodeArray.length; i++){
  if (nodeArray[i].getY() == img.height - 1){
    bottomRow[num] = i;
    num++;
  }
}
int root = bottomRow[0]; // root = index of bottom node in the
                         lowest energy seam
double lowest = nodeArray[root].getSum();
for (int i = 1; i < bottomRow.length; i++){
  double summ = nodeArray[bottomRow[i]].getSum();
  if (summ < lowest){
    lowest = summ;
    root = bottomRow[i];
  }
}
VNode[] vertSeam = new VNode[img.height]; // contains all nodes
                                    in lowest energy seam
int[] seamIndex = new int[vertSeam.length];// array containing
                             indexes of all pixels in seam
VNode pixel = nodeArray[root];
for (int i = 0; i<vertSeam.length; i++){
 vertSeam[i] = pixel;
 seamIndex[i] = pixel.getIndex();
 pixel = pixel.getparent();
}

img.updatePixels();

PImage post = createImage(img.width + 1 , img.height, RGB);
                                              //final image
post.loadPixels();
for (int y = 0; y < original.height; y++){
  for (int x = 0; x < original.width; x++){
   for (int i = 0; i < seamIndex.length; i++){
    if (y*original.width + x == seamIndex[i]){
         post.pixels[y*post.width + x] =
         original.pixels[y*img.width + x];
         color first = original.pixels[y*img.width + x - 1];
         color second = original.pixels[y*img.width + x + 1];
         color avg =
         color ((red(first) + red(second))/2, (green(first) +
         green(second))/2, (blue(first) + blue(second))/2);
```

```
            post.pixels[y*post.width + x + 1] =
            avg; //original.pixels[y*img.width + x];
            x++;
            while (x < original.width){
               post.pixels[y*post.width + x + 1] =
               original.pixels[y*img.width + x];
               x++;
            }
            break;
        }
        else{
           post.pixels[y*post.width + x] =
           original.pixels[y*original.width + x];
        }
      }
     }
   }
   post.updatePixels();
   image(post,0,0);
   return(post);
 }

 PImage addhoriz(PImage original, PImage filtered){
   img = filtered; // image that has been put through sobel filter
   img.loadPixels();

   HNode[] nodeArray = new HNode[img.pixels.length];// create
                              Array of Nodes
   for (int i = 0; i<img.pixels.length; i++){// for each pixel,
                                         create corresponding Node
    int ypos = i/img.width; // calculating x and y coordinates
                              based on position in array
    int xpos = i - (ypos * img.width);
    nodeArray[i] = new HNode (xpos, ypos, nodeArray, img, i);
    nodeArray[i].setY();
    nodeArray[i].setX();
   }
   for (int i = 0; i<nodeArray.length; i++){
    nodeArray[i].setParent();// set parent for each node
   }
   int[] leftSide = new int[img.height];// array containing
                 positions of leftmost nodes in nodeArray
   int num = 0;
   for (int i = 0; i<nodeArray.length; i++){
    if (nodeArray[i].getX() == 0){
      leftSide[num] = i;
    num++;
  }
}
int root = leftSide[0]; // root = index of leftmost node in the
                           lowest energy seam
double lowest = nodeArray[root].getSum();
for (int i = 1; i < leftSide.length; i++){
  double summ = nodeArray[leftSide[i]].getSum();
  if (summ < lowest){
    lowest = summ;
    root = leftSide[i];
  }
}
HNode[] horizSeam = new HNode[img.width]; // contains all nodes
                                      in lowest energy seam
int[] seamIndex = new int[horizSeam.length];// array containing
                           indexes of all pixels in seam
HNode pixel = nodeArray[root];
for (int i = 0; i<horizSeam.length; i++){
 horizSeam[i] = pixel;
 seamIndex[i] = pixel.getIndex();
 pixel = pixel.getparent();
}
img.updatePixels();

PImage post = createImage(img.width, img.height + 1, RGB);
                                            //final image
post.loadPixels();
for (int x = 0; x < original.width; x++){
  boolean carved = false;
  for (int y = 0; y < original.height-1; y++){
    if (carved == false){
      for (int i = 0; i < seamIndex.length; i++){
        if (y*img.width + x == seamIndex[i]){
         carved = true;
         post.pixels[y*original.width + x] =
         original.pixels[y*original.width + x];
        y++;
        color first =
        original.pixels[(y-1)*img.width + x];
        color second = original.pixels[(y+1)*img.width + x];
        color avg =
        color ((red(first) + red(second))/2, (green(first) +
        green(second))/2, (blue(first) + blue(second))/2);
        post.pixels[y*original.width + x] = avg;
        int y2 = y;
```

```
          while (y2 < original.height){
             post.pixels[(y2+1)*original.width + x] =
             original.pixels[(y2)*img.width + x];
             y2++;
          }
       }
       else{
             post.pixels[y*original.width + x] =
             original.pixels[y*original.width + x];
         }
       }
     }
   }
  }

  post.updatePixels();
  image(post,0,0);
  return(post);
}

PImage sobel(PImage img){
  img.loadPixels();
  PImage edgeImg = createImage(img.width, img.height, RGB);
  // Loop through every pixel in the image.
  for (int y = 1; y < img.height-1; y++) { // Skip top and bottom
                                             edges
   for (int x = 1; x < img.width-1; x++) { // Skip left and right
                                             edges
      double sumxr = 0; // Kernel sum for this pixel - red,
                           horizontal
      double sumyr = 0; // red, vertical
      double sumxg = 0; // green, horizontal
      double sumyg = 0; // etc.
      double sumxb = 0;
      double sumyb = 0;
      double magnituder = 0; // magnitude red
      double magnitudeg = 0; // magnitude green
      double magnitudeb = 0; // magnitude blue
      int magintr = 0; // going to convert magnitudes to ints
      int magintg = 0;
      int magintb = 0;
      for (int ky = -1; ky <= 1; ky++) {
        for (int kx = -1; kx <= 1; kx++) {
          // Calculate the adjacent pixel for this kernel point
          int pos = (y + ky)*img.width + (x + kx);
          // find RGB values for pixel
          double valred = red(img.pixels[pos]);
          double valgreen = green(img.pixels[pos]);
          double valblue = blue(img.pixels[pos]);
          // Multiply RGB values for pixels based on the kernel
                values
           sumxr += kernelx[ky+1][kx+1] * valred;
           sumyr += kernely[ky+1][kx+1] * valred;
           sumxg += kernelx[ky+1][kx+1] * valgreen;
           sumyg += kernely[ky+1][kx+1] * valgreen;
           sumxb += kernelx[ky+1][kx+1] * valblue;
           sumyb += kernely[ky+1][kx+1] * valblue;
        }
      }
      // For this pixel in the new image, set the RGB values
      // based on the sums from the kernel
   magnituder = Math.sqrt((sumxr * sumxr) + (sumyr * sumyr));
   magintr = (int)Math.round(magnituder);
   magnitudeg = Math.sqrt((sumxg * sumxg) + (sumyg * sumyg));
   magintg = (int)Math.round(magnitudeg);
   magnitudeb = Math.sqrt((sumxb * sumxb) + (sumyb * sumyb));
   magintb = (int)Math.round(magnitudeb);
   edgeImg.pixels[y*img.width + x] = color(magintr, magintg,
     magintb);
   }
  }
  for (int i = 0; i < edgeImg.pixels.length; i++){
   int ypos = i/img.width;
   int xpos = i - (ypos * img.width);
   if (ypos == 0){
     edgeImg.pixels[i] = edgeImg.pixels[i+img.width];
    }
    else if (ypos == img.height - 1){
       edgeImg.pixels[i] = edgeImg.pixels[i-img.width];
    }
    if (xpos == 0){
       edgeImg.pixels[i] = edgeImg.pixels[i+1];
    }
    else if (xpos == img.width-1){
       edgeImg.pixels[i] = edgeImg.pixels[i-1];
    }
   }
   edgeImg.updatePixels();
   //edgeImg.save("filtered.jpg");
   return(edgeImg);
}
```

```
public class VNode {// each node corresponds to a pixel
  int x, y;
  double valred, valgreen, valblue, energy, sum;
  PImage image;
  VNode[] nodeArray;
  VNode parent;
  int index;

  public VNode(int xpos, int ypos, VNode[] arrayOfNodes,
                                    PImage imge, int ind){
    image = imge;
    x = xpos;
    y = ypos;
    valred = red(image.get(x,y));
    valgreen = green(image.get(x,y));
    valblue = blue(image.get(x,y));
    energy = valred + valgreen + valblue;
                  // energy = sum of RGB values
    nodeArray = arrayOfNodes;
    sum = energy; // for now - will be updated in setParent()
    index = ind;
  }

  // basic getter/setter functions for Node attributes
  public int getX() { return this.x; }
  public int getY() { return this.y; }
  public double getEnergy() { return this.energy; }
  public double getSum() { return this.sum; }
  public VNode getparent() { return this.parent; }
  public int getIndex() { return this.index; }

  public void setX() { this.x = this.getIndex() -
 (this.getY() * img.width); }
  public void setY() { this.y = this.getIndex()/img.width; }
  public void setEnergy(double en) { this.energy = en; }
  public void setSum(double sm) { this.sum = sm; }
  public void setIndex(int ind) { this.index = ind; }

  public void setParent(){
     // finds parent with least sum and sets sum for this node
    double sum1;
    double sum2;
    double sum3;
    VNode n1;
    VNode n2;
    VNode n3;
```

```
this.setY();
this.setX();
if (this.y == 0){ // top row
  this.parent = this; // sum is already set to energy
}
else if (this.x == 0){ // left column
  sum1 = this.getSum() + nodeArray[(this.getIndex() -
  img.width + 1)].getSum(); // node above & right
  n1 = nodeArray[(this.getIndex() - img.width + 1)];
  sum2 = this.getSum() + nodeArray[(this.getIndex() -
  img.width)].getSum(); // node above
  n2 = nodeArray[(this.getIndex() - img.width)];
  if (sum1 <= sum2){// compare sums & assign sum and parent
    this.sum = sum1;
    this.parent = n1;
  }
  else{
    this.sum = sum2;
    this.parent = n2;
  }
}
else if (this.getX() == this.image.width - 1){ // right
                                                column
   sum1 = this.getSum() + nodeArray[(this.getIndex() -
   img.width - 1)].getSum(); // node above & left
   n1 = nodeArray[(this.getIndex() - img.width - 1)];
   sum2 = this.getSum() + nodeArray[(this.getIndex() -
   img.width)].getSum(); // node above
   n2 = nodeArray[(this.getIndex() - img.width)];
  if (sum1 <= sum2){// compare sums & assign sum and parent
    this.sum = sum1;
    this.parent = n1;
  }
  else{
    this.sum = sum2;
    this.parent = n2;
  }
}
else{ // not on edge
  sum1 = this.getSum() + nodeArray[(this.getIndex() -
  img.width - 1)].getSum(); // node above & left
  n1 = nodeArray[(this.getIndex() - img.width - 1)];
  sum2 = this.getSum() + nodeArray[(this.getIndex() -
  img.width)].getSum(); // node above
  n2 = nodeArray[(this.getIndex() - img.width)];
```

```
      sum3 = this.getSum() + nodeArray[(this.getIndex() -
      img.width + 1)].getSum(); // node above & right
      n3 = nodeArray[(this.getIndex() - img.width + 1)];
       if (sum1 <= sum2 && sum1 <= sum3){// compare sums & assign
                                                 parent
         this.sum = sum1;
         this.parent = n1;
       }
       else if (sum2 <= sum1 && sum2 <= sum3){
         this.sum = sum2;
         this.parent = n2;
       }
       else if (sum3 <= sum1 && sum3 <= sum2){
         this.sum = sum3;
         this.parent = n3;
       }
     }
   }
 }

 public class HNode {// each node corresponds to a pixel
   int x, y;
   double valred, valgreen, valblue, energy, sum;
   PImage image;
   HNode[] nodeArray;
   HNode parent;
   int index;

   public HNode(int xpos, int ypos, HNode[] arrayOfNodes,
                                     PImage imge, int ind){
     image = imge;
     x = xpos;
     y = ypos;
     valred = red(image.get(x,y));
     valgreen = green(image.get(x,y));
     valblue = blue(image.get(x,y));
     energy = valred + valgreen + valblue;
                      // energy = sum of RGB values
     nodeArray = arrayOfNodes;
     sum = energy; // for now - will be updated in setParent()
     index = ind;
   }

   // basic getter/setter functions for Node attributes
   public int getX() { return this.x; }
   public int getY() { return this.y; }
```

```
public double getEnergy() { return this.energy; }
public double getSum() { return this.sum; }
public HNode getparent() { return this.parent; }
public int getIndex() { return this.index; }

public void setX() { this.x = this.getIndex() -
(this.getY() * img.width); }
public void setY() { this.y = this.getIndex()/img.width; }
public void setEnergy(double en) { this.energy = en; }
public void setSum(double sm) { this.sum = sm; }
public void setIndex(int ind) { this.index = ind; }

public void setParent(){// finds parent with least sum and sets
                         sum for this node
double sum1;
double sum2;
double sum3;
HNode n1;
HNode n2;
HNode n3;
this.setY();
this.setX();
if (this.x == img.width - 1){ // right column
  this.parent = this; // sum is already set to energy
}
else if (this.getY() == 0){ // top row
    sum1 = this.getSum() + nodeArray[(this.getIndex() + img.widt
    h + 1)].getSum(); // node right & below
    n1 = nodeArray[(this.getIndex() + img.width + 1)];
    sum2 = this.getSum() +
    nodeArray[(this.getIndex() + 1)].getSum(); // node right
    n2 = nodeArray[(this.getIndex() + 1)];
     if (sum1 <= sum2){// compare sums & assign sum and parent
       this.sum = sum1;
       this.parent = n1;
     }
     else{
       this.sum = sum2;
       this.parent = n2;
     }
  }
  else if (this.getY() == this.image.height - 1){ // bottom row
    sum1 = this.getSum() + nodeArray[(this.getIndex() -
    img.width + 1)].getSum(); // node right & above
    n1 = nodeArray[(this.getIndex() - img.width + 1)];
    sum2 = this.getSum() +
        nodeArray[(this.getIndex() + 1)].getSum(); // node right
        n2 = nodeArray[(this.getIndex() + 1)];
         if (sum1 <= sum2){// compare sums & assign sum and parent
           this.sum = sum1;
           this.parent = n1;
         }
         else{
           this.sum = sum2;
           this.parent = n2;
         }
       }
       else{ // not on edge
        sum1 = this.getSum() + nodeArray[(this.getIndex() + img.widt
        h + 1)].getSum(); // node right & below
        n1 = nodeArray[(this.getIndex() + img.width + 1)];
        sum2 = this.getSum() +
        nodeArray[(this.getIndex() + 1)].getSum(); // node right
        n2 = nodeArray[(this.getIndex() + 1)];
        sum3 = this.getSum() + nodeArray[(this.getIndex() -
        img.width + 1)].getSum(); // node right & above
        n3 = nodeArray[(this.getIndex() - img.width + 1)];
         if (sum1 <= sum2 && sum1 <= sum3){// compare sums & assign
                                                  parent
           this.sum = sum1;
           this.parent = n1;
         }
         else if (sum2 <= sum1 && sum2 <= sum3){
           this.sum = sum2;
           this.parent = n2;
         }
         else if (sum3 <= sum1 && sum3 <= sum2){
           this.sum = sum3;
           this.parent = n3;
         }
       }
     }
   }
```