EE 569 Homework 3 Write Up

Armin Bazarjani

February 22th, 2021

1 Geometric Image Modification

1.1 Motivation

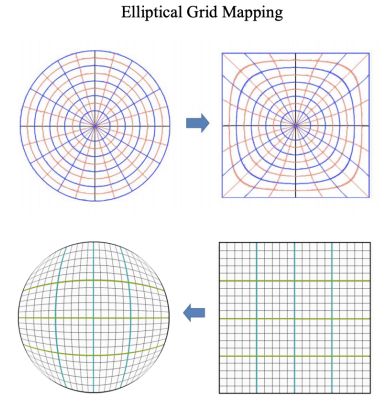
Geometric image modification is one of the most fundamental and frequently used image processing techniques. This includes spatially translating, scaling in size, rotating, non linearly warping, or viewing an image from a different perspective. Geometric image modification algorithms can be applied in both the 2D domain as well as the 3D domain. The use case can be from as simple as constructing an interesting and entertaining image filter, to transforming an image from one perspective into another entirely so as to match images in the second perspective.

1.2 Approach and Procedures

1.2.1 Unit Circle to Unit Square and Vice Versa

One important thing to note is that there is a plethora of online resources that document how to transform a unit square into a unit circle and similarly how to transform a unit circle into a unit square. In this situation, the unit circle will be inscribed inside of the square. Luckily, many of these mappings were documented in a paper published by Chamberlain Fong in 2014. It explores many different closed-form and invertible equations. The beauty of invertibility is that we can perform the mapping back and forth from circle to square. Additionally, it is worth noting that the canonical mapping, and the ones used in the paper, are both presented for a unit circle and a unit square. Meaning, the values are all between 0 and 1.

Among the many mappings that the paper described, I decided to use the elliptical grid mapping. Below you can see the radial grid inside of the circle converted to a square, and the square grid converted to a disc.



(a) Mapping

Figure 1: Image Showing Mapping Liens

The equation for square to disc mapping is (Note that (x,y) are the cartesian coordinates of our square image and (u,v) are the cartesian coordinates of our circle image):

$$x = \frac{1}{2}\sqrt{2 + u^2 - v^2 + 2\sqrt{2}u} - \frac{1}{2}\sqrt{2 + u^2 - v^2 - 2\sqrt{2}u}$$

$$y = \frac{1}{2}\sqrt{2 + u^2 - v^2 + 2\sqrt{2}v} - \frac{1}{2}\sqrt{2 + u^2 - v^2 - 2\sqrt{2}v}$$

The equation for disc to square mapping is:

$$u = x\sqrt{1 - \frac{y^2}{2}}$$

$$v = y\sqrt{1 - \frac{x^2}{2}}$$

1.2.2 Warping to a Larger Disk-Shaped Image

For the purposes of this homework, we were tasked with converting a few different images into a larger disk-shaped image. Thus, we want to transform our original square image to a circle that is large enough where the original square image is inscribed inside of it. So, the radius of our larger disk-shaped image is

$$r = \sqrt{2}L$$

Where L is the length of one of the sides of our square.

For my implementation, I decided that the easiest and most structured way would be for me to use one of the defined disc to square transformations from

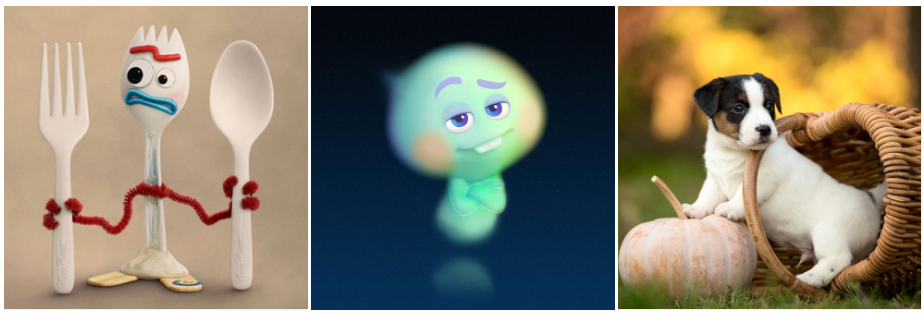
the paper earlier, and then upsample them.

First, I looped through the original square image, converted the (i,j) pixel location to an (x,y) cartesian location where the origin is in the center of the image. Next, I normalized the (x,y) cartesian location to be between 0 and 1. The reason I normalized was because the equations were defined for a unit circle and a unit square. I'm sure they could have worked without this normalization step, but I didn't want to take chances and I didn't have enough time to test it out. Next, after normalizing the cartesian (x,y) values, I used the elliptical grid mapping equation to convert to (u,v) cartesian locations on a circle.
Consequently, I unnormalized the (u,v) values to get them back into pixel values (p',q').
Finally, I multiplied the the intermediate pixel values with the ratio between the circle size I wanted to get, and the circle size they were in. $p = ratio * p'$, $q = ratio * q'$

Now, because we are trying to construct a larger image out of a smaller one, there are going to be some values that don't have a direct match. Thus, there are going to be some missing values (or black dots) in our image that we need to interpolate. One of the most common interpolation methods, and the one that I used for this problem in particular was bilinear interpolation.

1.3 Experimental Results

First, I will show the original images.



(a) Forky Original (b) 22 Original (c) Dog Original

Figure 2: Original Images of Forky, Dog, and 22

Next, I will show the disk-shaped images after applying forward mapping.

(a) Forky Forward

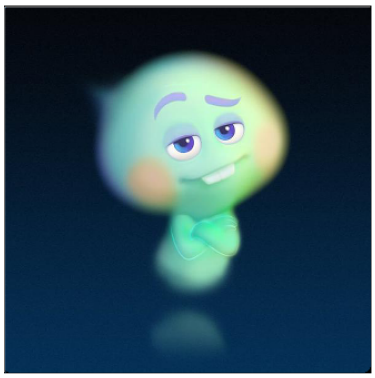
(b) 22 Forward

(c) Dog Forward

Figure 3: Forward mapped images of Forky, Dog, and 22

Finally, I will show the reverse mapped images back to a square.

(a) Forky Reversed



(b) 22 Reversed

(c) Dog Reversed

Figure 4: Reverse mapped images of Forky, Dog, and 22

1.4 Discussion

Question 1 - I described the method in detail above in section "1.2 Approach and Procedures". Also, please note that although the forward mapped images appear to be the same size, that is because of the document editor I am using. They are in fact larger.

Question 2 - You can see the recovered images in section "1.4 Experimental Results".

Question 3 - As you can see, there are very little, if any, differences between the reverse mapped image and the original image. I believe the reason for this is because I used simple and well defined mathematical equations that were proved to result in a one to one mapping. The few differences that I did see where in the corners of the images. My suspicion for this is that because I had to first downsample and then apply the transformation, the extrema sometimes weren't

able to map back to certain values as I had to round.

2 Homographic Transformation and Image Stitching

2.1 Motivation

The homography matrix essentially defines a transformation between two different planes. When we apply the homography matrix to some points in a cartesian space, we will get a projective mapping from one plane to another. The usefulness of such an application is plentiful. For example, it can be used for camera pose estimation from coplanar points in augmented reality, perspective removal and/or correction, image warping, and for our situation panorama stitching.

Panorama stitching is when we synthesize and stitch images that were taken from different camera viewpoints, combining all of them into a single image from a single perspective.

2.2 Approach and Procedures

The approach, although very theoretical, was relatively straightforward as we only had to solve for the given homography matrix. The general equation for applying the homography matrix to two points (x_1,y_1) to get a new set of points (x_2,y_2) is as follows.

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} * \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x_2^{'} \\ y_2^{'} \\ w_2^{'} \end{bmatrix}$$

Where

$$x_2 = \frac{x_2^{'}}{w_2^{'}}$$

$$y_2 = \frac{y_2^{'}}{w_2^{'}}$$

From a single pair of matching points (x_1-¿x_2 and y_1-¿y_2) we can get two equations. Because there are eight unknowns in our homography matrix ($h_{33} = 1$) then we only need to find 4 pairs of matching points.

$$h_{11}x_1 + h_{12}y_1 + h_{13} = x_2(h_{31}x_1 + h_{32}y_1 + h_{33})$$

$$h_{21}x_1 + h_{22}y_1 + h_{23} = y_2(h_{31}x_1 + h_{32}y_1 + h_{33})$$

Thus, after you select 4 viable points. The job ahead of you is as easy as solving a system of linear equations.

My implementation was as such:
First - I ran SURF feature detector on all three images (left, middle, and right).
Second - I matched all the features to get a set of points with all of the most descriptive features that matched one another in each image.
Third - I manually inspected each matched feature to make sure (1) they actually matched and (2) in order to hand pick points that were far from one another.
Fourth - I used the two sets of matched points to solve for two different homography matrices. One for the left side that will transform the left image to the middle, and one for the right side that will transform the right image to the middle.
Fifth - I applied the left and right homography matrices in a forward mapping fashion to get a new set of points.
Finally - I interpolated the missing points in the final image.

2.3 Experimental Results

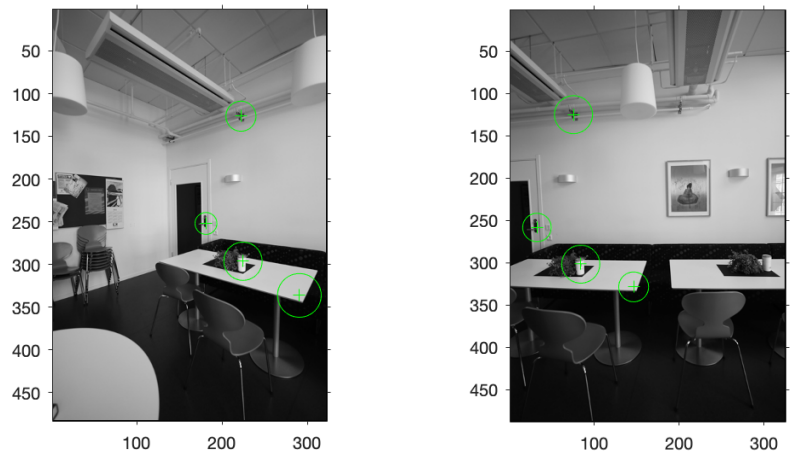
The final resulting panoramic photo

(a) Classroom Panorama

Figure 5: Final Stitched Image

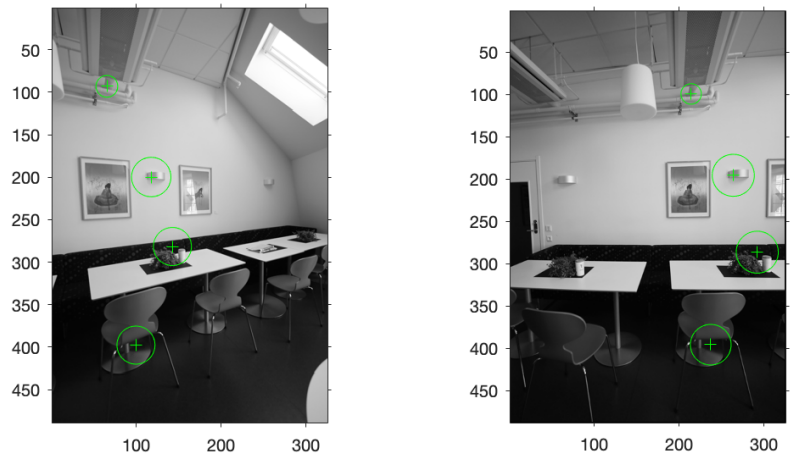
2.4 Discussion

Question 1 - I used 4 control points. Please find the images for the left and left-middle control points as well as the right and right-middle control points in the following images.



(a) Left (b) Left-Middle

Figure 6: Left and Left-Middle Control Points



(a) Right (b) Right-Middle

Figure 7: Right and Right-Middle Control Points

Question 2 - I described a bit of the process above in section "2.2 Approach and Procedures", but I will give more detail here. First, I transformed the im-

ages to black and white. Then, I used detectSURFFeatures() on all three black and white images. Next, I used extractFeatures() to get the features as well as their locations. Finally, I used matchFeatures() to return a a list of index pairs of matched features.

After getting the index pairs of matched features, I pulled out the relevant and matched features and then I plotted all of them on top of the original images. I used the plotted features to determine two things. First, I could visually identify good and descriptive features that weren't too close to one another. Second, I could make sure that the features actually matched one another as some of the features were not direct matches when you look at them visually.

3 Morphological processing

3.1 Motivation

Morphological image processing is a process of non-linear operations that are related to the shape or the morphology of an image. With morphological processing, we are working directly in the images spatial domain without taking any color channel information into account. Generally, the images that are used are binary or binarized. The most basic morphological processing applications are erosion, dilation, opening, and closing. For our purposes, we focused on a slightly more advanced set of operations. Specifically shrinking, thinning, and skeletonizing.

3.2 Approach and Procedures

3.2.1 Basic morphological process implementation

For each operation, we first compare the image to a conditional mark pattern and then to an unconditional mark pattern. The idea is as such. When we first have our image, F(i,j), we compare it to all of the conditional mark patterns associated with our operation (in our situation these will be shrinking, thinning, and skeletonizing). If there is a hit, ie. if the conditional mark pattern centered on our current pixel, (i,j), matches one of the patterns of interest, we mark that location as a 1 on our mask, M(i,j), else, we mark it as a 0. I believe it is worth noting that the mask is initialized to be the same size as our image.

Next, we use the mask we constructed in step (1) and do the same operation with the unconditional mark patterns. If the mask matches, then we keep the original value in our image, else we set it to 0.

One important thing that I believe is worth noting is that in step (1) our conditional mark patterns only mark pixels that we are interested with keeping or removing. Thus, once we go into step (2), we are only focused on actually keeping or removing the marked pixels. If a pixel is not marked, then step (2)

should not modify it in that iteration.

Now, we iteratively apply step (1) and step (2), keeping and removing pixels, until we reach a point where no pixels have been removed. Once we reach this point, we have converged and we can stop iterating.

My implementation was to hand code all of the conditional and unconditional mark patterns in as binary values (0b010011010). Then when iterating through the image, I would construct a binary pattern value centered around my current pixel location. I believe this process made it a lot simpler for two reasons. First, I only have to apply one checker to see if the binary values are equal to one another. Second, when we are faced with the unconditional mark patterns that can have either a 0 or a 1 in certain locations, I simply created a mask binary pattern applied a bit-wise AND operation to my current pattern and then checked the ANDed value with the unconditional mask.

3.2.2 Solution to the maze

Solving the maze was relatively simple. After correctly being able to apply morphological filters to the images in part (A), you essentially completed the problem. To solve the maze you simply need to apply the shrinking operation to the maze, the converged image is the correct path through the maze.

The reason for this is because the entire maze image is white, except for the boundaries being black. When we apply shrinking, we shrink all of the boundaries around an image to single point. However, because the maze is open on two ends, it cannot converge on a single point. Thus, we will shrink away all of the interconnected walls, leaving only the true path through the maze.

3.2.3 Defect detection and count

Detecting and counting the defects as was also somewhat straightforward after part (A) because, again, we are really only using the shrinking method here.

In order to count the number of defects, I first inverse binarized the image. Then I applied the shrinking operation so that all of the defects merged into single points. Afterwards, I looped through the image and counted all of the single points, those being the defects. Because of some border issues when shrinking the image, I also had to add another statement saying that if the current pixel is white and NONE of it's 1 hop neighbors are white, then it is the location of a defect.

To correct the defects, I stored all of the locations that I found after shrinking and applying my custom neighbor operation in a vector. I then looped through the vector of defect locations and applied a Breadth-First Search algorithm from the location on the original image to clean up the pixel values.

To get the sizes of the defects, I counted inside of my BFS algorithm. I thought this would be a better and more exact approach than counting the iterations of shrinking as suggested in discussion.

Implementing the CCL algorithm, I followed the procedure as discussed in the discussion section exactly. I looped through the image, got the neighboring values, if there was no neighboring values (ie. all of them are 0) then I set the current location to the current ccl count value, else, I set the current location to the smallest value of the neighbor. If we were in the else statement, I then updated equivalence table to say that all the seen values are in the same neighborhood. On the second pass I used the equivalence table to update all the neighbor values so each defect had its own value.

To count using CCL, I simple counted how many times a specific value appeared and stored them all in a .txt file.

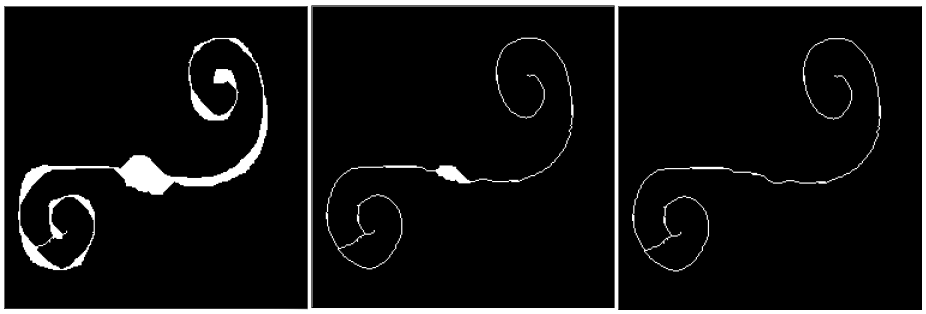
3.3 Experimental Results

3.3.1 Basic morphological process

First, I will show all the original images. Then, I will show each morphological process (thinning, shrinking, skeletonizing) on each image. For each process I will also show 3 images showing the final processed image, the image 1/3 of the way processed, and the image 2/3 of the way processed.

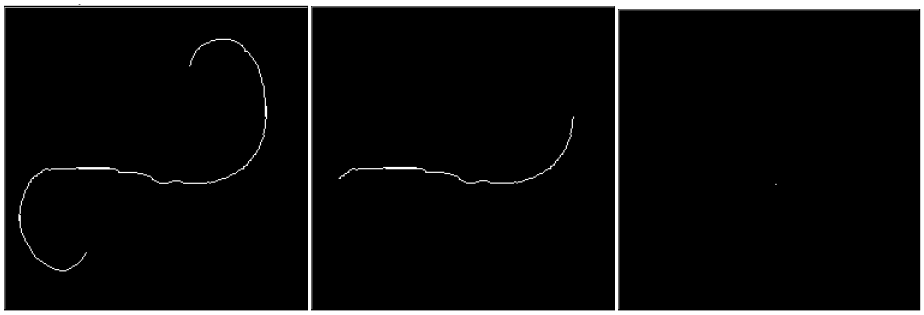
(a) Spring Original (b) Flower Original (c) Jar Original

Figure 8: Original Images of Spring, Flower, and Jar



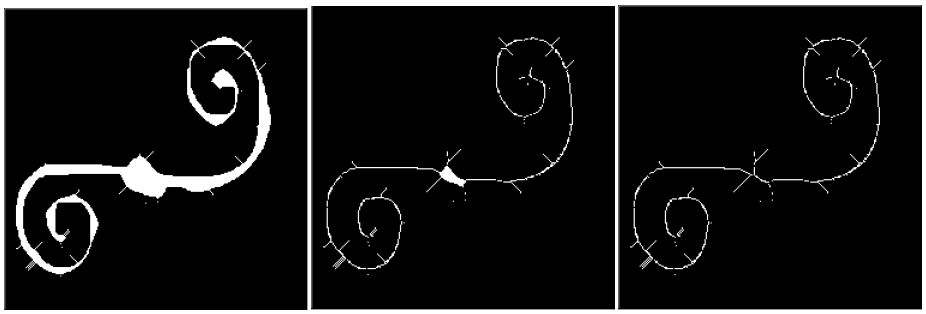
(a) Spring Thin 1/3 (b) Spring Thin 2/3 (c) Spring Thin final

Figure 9: Different iterations of thinning Spring



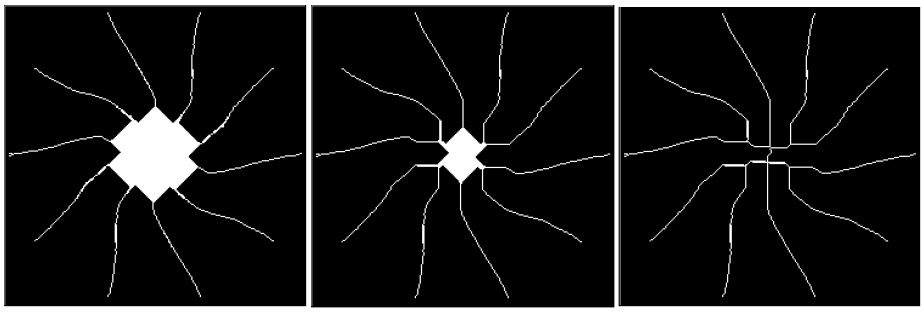
(a) Spring Shrink 1/3 (b) Spring Shrink 2/3 (c) Spring Shrink Final

Figure 10: Different iterations of shrinking Spring



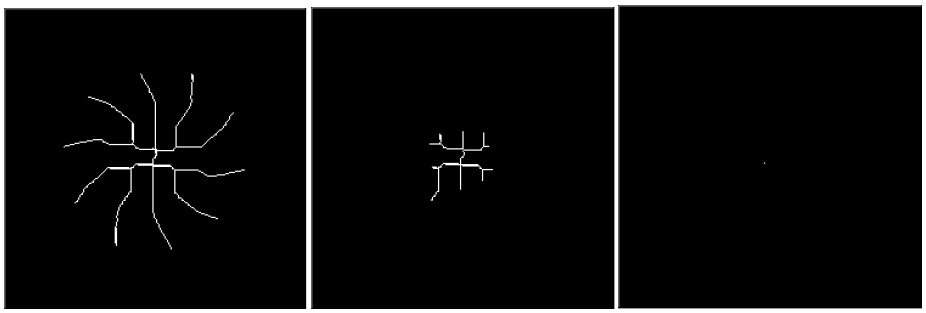
(a) Spring Skeleton 1/3 (b) Spring Skeleton 2/3 (c) Spring Skeleton Final

Figure 11: Different iterations of skeletonizing Spring



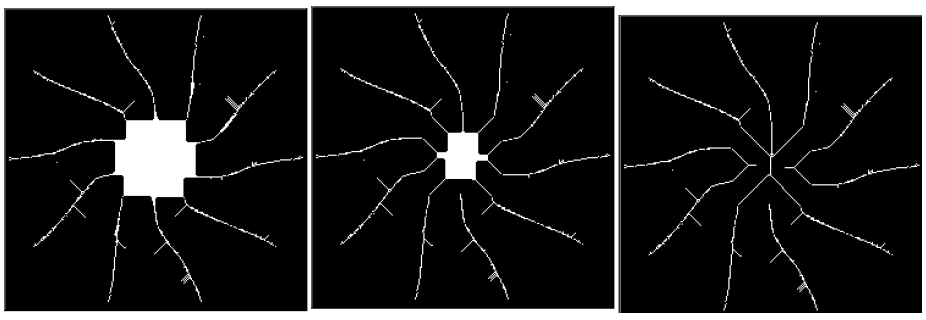
(a) Flower Thin 1/3 (b) Flower Thin 2/3 (c) Flower Thin final

Figure 12: Different iterations of thinning Flower



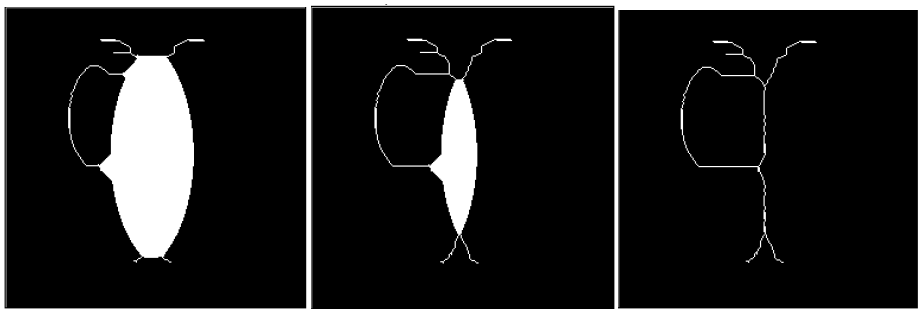
(a) Flower Shrink 1/3 (b) Flower Shrink 2/3 (c) Flower Shrink Final

Figure 13: Different iterations of shrinking Flower



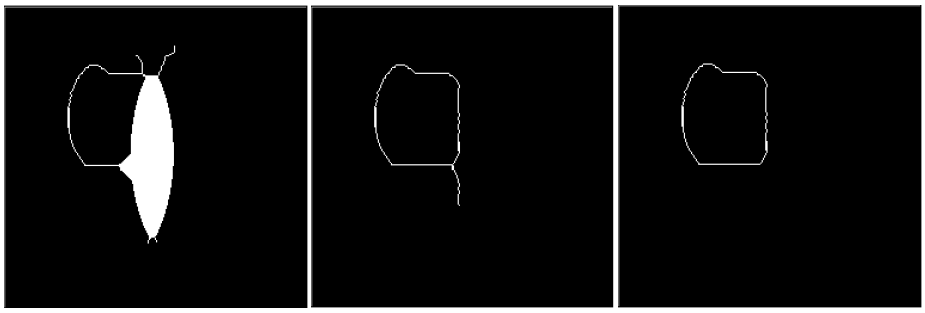
(a) Flower Skeleton 1/3 (b) Flower Skeleton 2/3 (c) Flower Skeleton Final

Figure 14: Different iterations of skeletonizing Flower



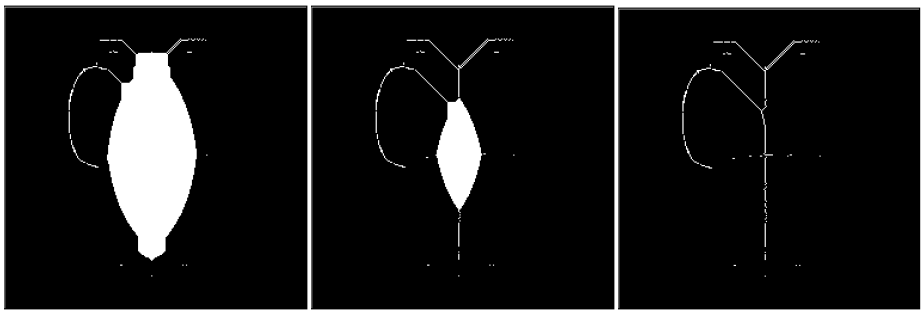
(a) Jar Thin 1/3 (b) Jar Thin 2/3 (c) Jar Thin final

Figure 15: Different iterations of thinning Jar



(a) Jar Shrink 1/3 (b) Jar Shrink 2/3 (c) Jar Shrink Final

Figure 16: Different iterations of shrinking Jar

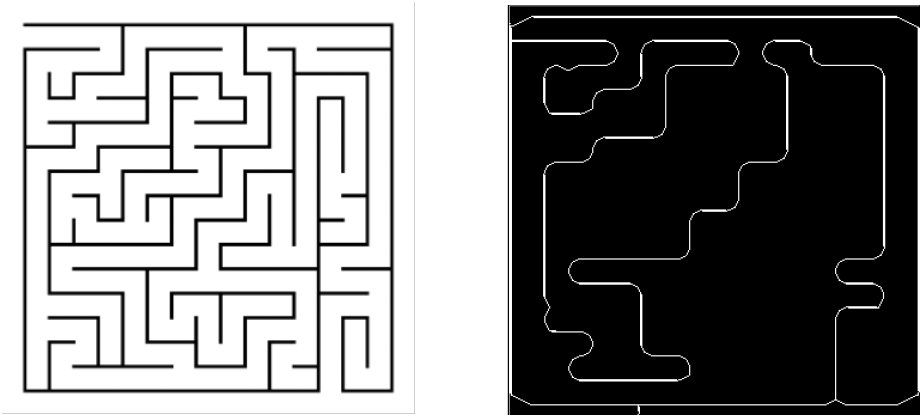


(a) Jar Skeleton 1/3 (b) Jar Skeleton 2/3 (c) Jar Skeleton Final

Figure 17: Different iterations of skeletonizing Jar

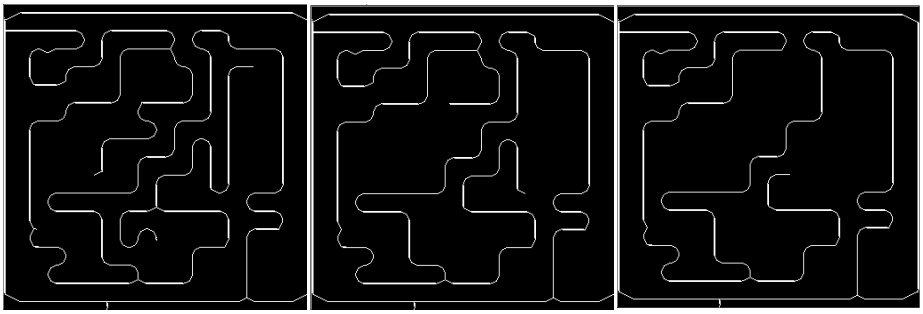
Now, I will show the original maze image along with the maze solution after

applying shrinking, as well as a few iterations of the process before it converges.



(a) Original Maze 1/3 (b) Maze Solution 2/3

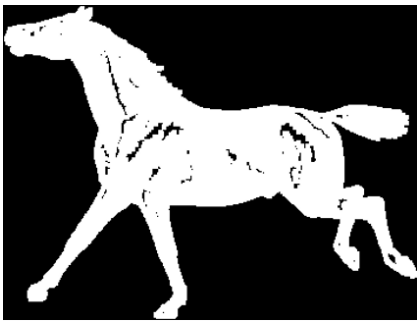
Figure 18: Original Maze and Solution after shrinking



(a) Maze Solution 1/4 (b) Maze Solution 2/4 (c) Maze Solution 3/4

Figure 19: Different iterations of shrinking the maze to find the solution

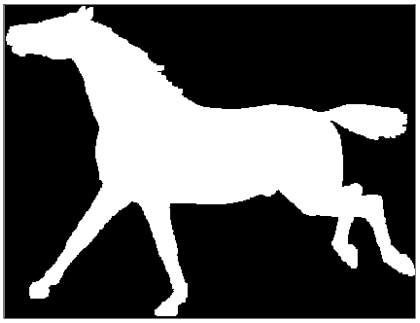
Now, I will show the results from defect detection and filling. I will show the results of shrinking the inverse horse image to find the defects, the results after clearing those defects, and the results from using the CCL algorithm to find the defects.



(a) Original Horse

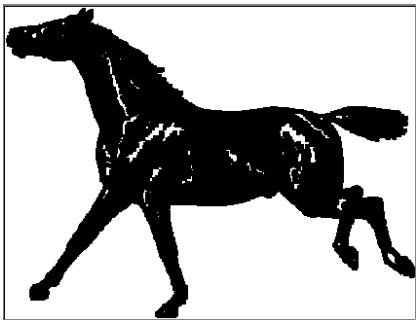
Figure 20: Original Horse Image

(a) Horse Shrinking



(b) Horse Defect Cleared

Figure 21: Horse After Srhinking to show defects and then after applying BFS to clear the defects



(a) CCL Detection

Figure 22: CCL defect detection image

3.4 Discussion

3.4.1 Part (A) Basic Morphological Process

There weren't any questions that I could see here, so please find the results above in section "3.3 Experimental Results".

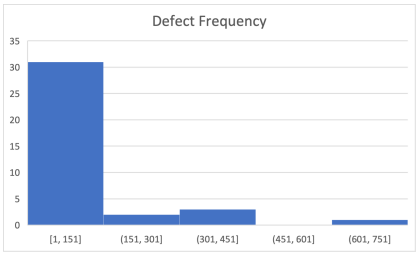
3.4.2 Part (B) Solution to Maze

The method to get the maze solution was very straightforward. I simply applied the shrinking method to the maze until it converged. Please see the results in section "3.3 Experimental Results". I also went into more detail as to why the solution worked in section "3.2.2 Solution to the maze".

3.4.3 Defect Detection and Count

Question 1 - I found the total number of defects to be 70 after I ran my program

Question 2 - There are 37 different defect sizes present in the image. The frequency histogram is as follows:



(a) Defect Histogram

Figure 23: Frequency Histogram of Defect Sizes

Question 3 - The results of clearing can be seen above in section "3.3 Experimental Results". The details of clearing can be found in section "3.2.3 Defect detection and count". I decided to use a Breadth-First Search algorithm to clear the defects after finding the locations using the shrinking method.

Question 4 - I implemented the CCL algorithm as described in discussion. The details can be found in section "3.2.3 Defect detection and count", the results can be seen in section "3.3 Experimental Results".