Jigsaw Puzzle Solver Write Up

Name: Jennifer Liu, Aaron Cote

Abstract

Solving jigsaw puzzles have traditionally been done as a fun exercise to challenge our brains. Depending on the level of difficulty of the jigsaw puzzles, it can take a while to solve. Our goal is to understand how the human mind goes about solving these puzzles, so we program a computer image processing system to do it faster. A computer can compute repeatable process quicker than a human can, but it does not have the flexibility and analytical skills of a human brain. For instance, finding corners of a piece against different backgrounds and at different light exposures may seem trivial for us, but it is difficult for a computer to sort through this kind of noise. In this paper, we present an approach to estimate whether a puzzle piece is present in the image, the location of a puzzle piece in the image, and the type of puzzle piece it is. We implement multiple image processing techniques including morphology, Harris corner detection, Sobal edge detection, and some fundamental linear algebra techniques. With this collected data, we can start our way to designing a computer image processing program to solve jigsaw puzzles much faster than we can. The system present here is designed to be efficient and resilient under a variety of conditions.

Introduction

In this project, we set out to take a collection of several images of jigsaw pieces identify useful data about them that could help solve how they all fit back together. We used image processing techniques that we learned in class over the semester and new ones that we discovered while researching the problem.

Traditionally, most Computer Vision problems are broken into multiple stages, data collection, noise removal, feature selection, modeling, classification and confirmation. The data collection process was done for us by Dr. Kinsman; however, it understanding the collected data before proceeding. While the data collection process was designed with controlled variable in mind, there is still artificially introduced noise to make the problem more realistic.

The original images of the jigsaw pieces were captured at 3072x2048 pixels. They were then down sampled to 900x900 pixels. The down-sampling eliminates the need to compensate for “dark edges” along the border of the images. The images do not have any motion blur. Three exposures of each piece was provided to enable us to choose the best exposure for our processing. An image can either have auto exposure, be 1/3rd underexposed, 2/3rds underexposed. The puzzle pieces are not guaranteed to be aligned with the edges of the image and can be rotated 90 degrees or 180 degrees in either direction. It is also possible that an image may not have a puzzle piece.

Figure 1: These are some examples of the images that we need to handle for this project.

The difficulty of this problem drastically increase due to introduced noise. Through various experiments and sleepless nights, we have settled on tackling the problem by breaking the harder problem into 5 easier sub-problems.

Five sub-problems:

1. **Image Segmentation –** We separate the puzzle piece from its background using multiple cleaning and detection processes and binarizing the final results as to create a piece segment mask.
2. **Determining the existence of a puzzle piece –** Through the use of colour histogram and image segmentation results, we estimate whether there is a puzzle piece or not in the image.
3. **Finding exact location of the puzzle piece –** Using the segmented image, and its bounding boxes, we estimate the cartesian location of the puzzle, and we crop the image in such a way that it eliminates the background as much as possible.
4. **Finding the 4 Corners –** We locate the four corners of the puzzle piece to allow us to better understand how the image of the piece is oriented and where it's edges ought to be.
5. **Determining Edge Type –** Using the 4 corners, we estimate where we would expect straight edges in the piece and analyze how the actual edges protrude and intrude differently than the straight edge model.

We use these collected data results to create an in depth profile of what the piece is, where it is, how it might fit together in the final puzzle.

In this paper, we will elaborate on our methods used to achieve our desired results, what we have learned along the way, how our final results turned out and what some of the challenges we faced were.

Methodology

Image Segmentation

In the image segmentation process, we took the original image of the piece and applied several preprocessing layers to remove noise and segment out the unwanted parts while keeping the parts we wanted.

First we converted the image into a double precision grayscale image and then applied a small median filter across the entire image to clean out background noise. Next, we used a Sobel edge detection filter to find the magnitude of all of the changes in the image and then binarize the image into black and white using a low threshold. This allowed us to find the edges of the piece regardless of how faint the edges were due to exposure or color similarity with the background, but it also found a lot of remaining noise in the background of the image, see figure 2.

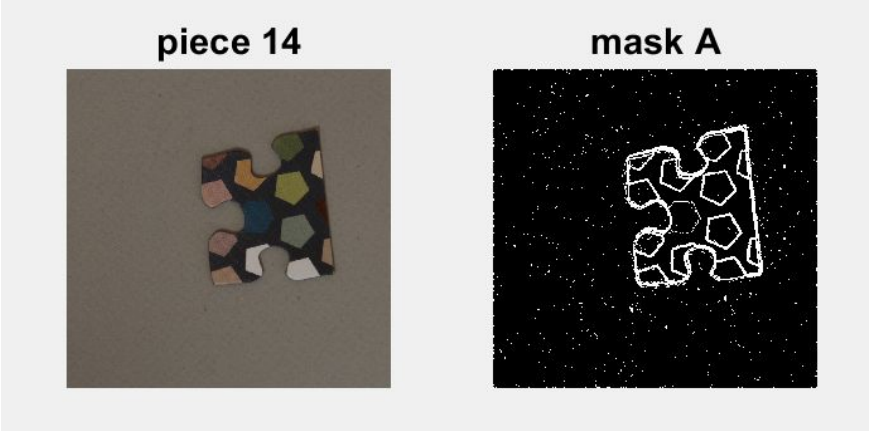

Figure 2: Binary image of edge detection with noise

The type of nose seen in figure two is commonly referred to as “salt and pepper” noise, and lucky for us, it is easily cleaned away with a median filter. We used a small median filter to clean up some of the noise without losing the edges of the piece. The resulting image is hollow, however, we wanted to segment out the entire piece. To do this we applied imfill to the image to fill all of the solid shapes, i.e. the edges of the piece; however, this was not perfect because the edges contained gaps. We resolved the gaps by using a dilation before filling followed by erosion to clean up after the piece was filled. Now that have a solid piece, we are able to use a larger median filter to eliminate persisting large speckle noise and obtain out final result, see figure 3.

Figure 3: Resulting piece segmentation after filling and cleaning

Determine whether a puzzle piece is present or not

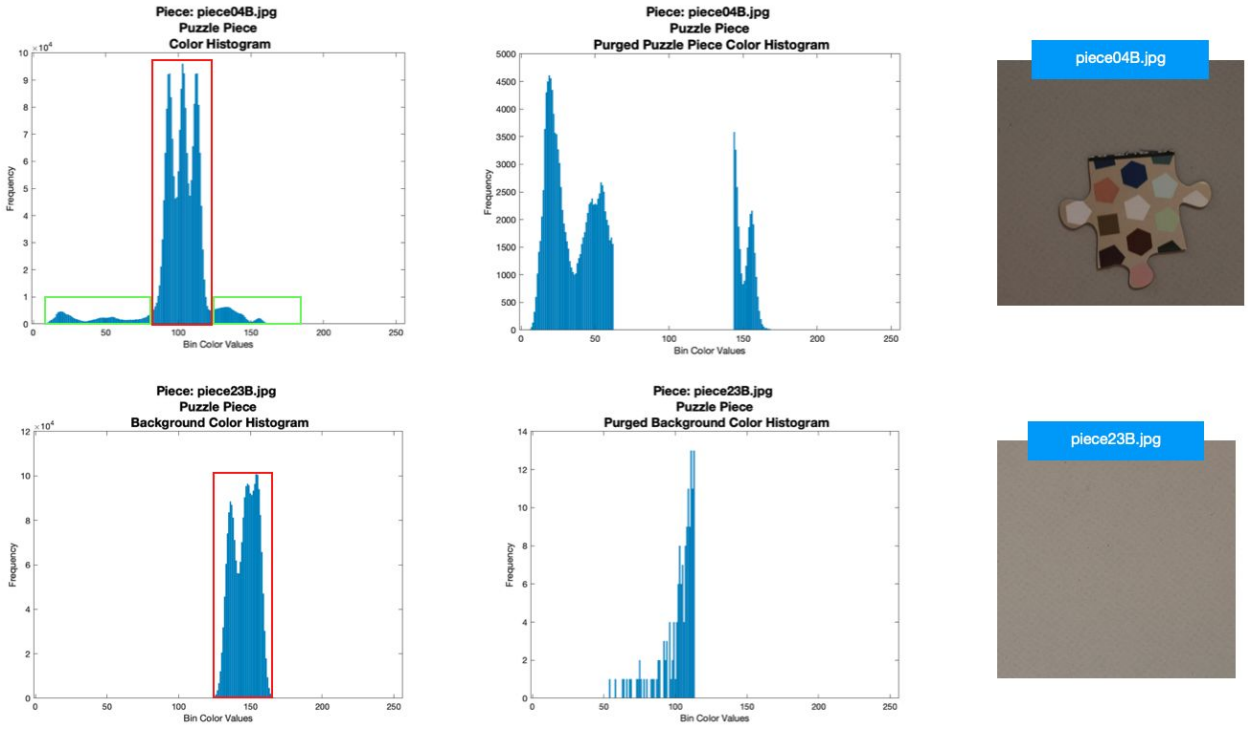

Figure 4: A comparison between the color histogram of an image that contains a puzzle piece and one that does not. The red bounding box on each of the color histograms on the left of the image is representative of the background. The green bounding box on the color histogram is representative of the foreground color. In this case, the foreground is the puzzle piece.

We were able to identify which region in the image is the background by observing their color histogram. The background color would have the most frequency as it takes up the majority of the image. We also noticed that images that do contain a puzzle piece will still have frequency for color values for the foreground color (The puzzle piece represented as the green bounding boxes in Figure 4). However, a background image would not have any other color but the background color. Hence, all other color values would have 0 frequency. This proves to be true in Figure 4. This means that if we extract all the color values that make up the background color from the color histogram, for images that have no puzzle piece will theoretically be zero. On the other hand, for images that do have a puzzle piece, there will still be many color values left from the puzzle piece. In Figure 4, while the purged color histogram of a background image was not completely clean, the frequency for the color values were extremely low. To give us an accurate estimate, we obtain the maximum frequency from the purged color histogram, and if the maximum frequency is greater than 30, it is considered a puzzle piece, and if not, it is considered a background image.

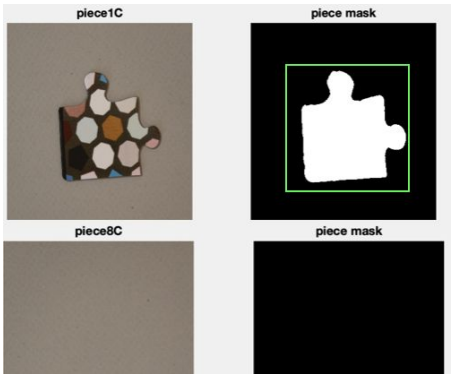

Figure 5: Drawing a bounding box around a puzzle piece that was segmented.

This worked for most of the images except for a few. To help boost the accuracy of our puzzle detection algorithm. We obtain the bounding box of an image, and if there is a bounding box, there must be a puzzle piece. If the image is a background image, for a perfectly segmented image, there would be no white pixels in the binarized image. Therefore, there could not be a bounding box. If there is no bounding box and the color histogram is satisfied, we know for sure there is a puzzle piece. With this algorithm, we achieved a 100% accuracy for detecting the existence of a puzzle piece.

Finding the exact location of the puzzle piece

With a well segmented image, this task was extremely easy. We merely had to draw a bounding box on the white pixels. As shown in Figure 5, we were able to draw a bounding box around the puzzle piece. With the bounding box, we could crop away all the background as much as possible, and focus the attention on the puzzle piece. The cropping will become necessary when finding the 4 corners.

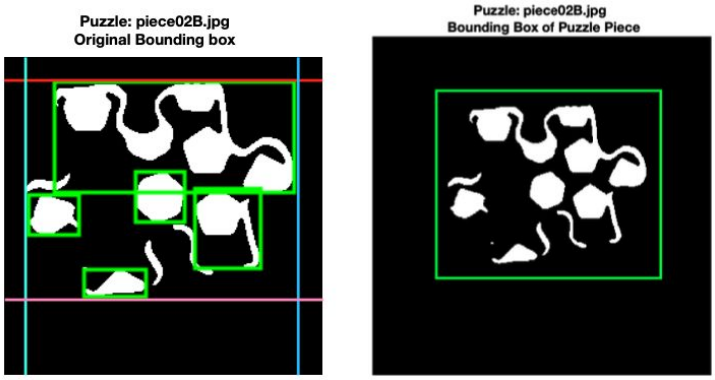

Figure 6: Left Image: An image demonstrating why we cannot just take the bounding box of a poorly segmented image. For each of the individual white pieces, it will create its own bounding box. The red line = minimum y of all bounding boxes, blue line = maximum x, turquoise line = minimum x, pink line = maximum y. Right Image: A bounding box correctly drawn on a poorly segmented image using the minimum and maximum x, y's with a 10 pixel padding on the right, left, top, and bottom of the image.

For images that are not well segmented, which are rare, we cannot just take the bounding box. Because each individual piece will end up with its own bounding box. To combat this issue, we obtain all of the bounding boxes

that have an area greater than 300, and we find the minimum x and y, and maximum x and y values from all of the bounding box. The 4 points of the bounding box are calculated like so; Top left corner = [min_x, min_y], Top Right Corner = [max_x, min_y], Bottom left corner = [max_x, max_y], Bottom right corner = [min_x, max_y].

Finding the 4 Corners

To obtain the 4 corners. We make a few assumptions and guesses. We are making the estimation that a corner exist if it is approximately 45°, 135°, -45°, or -135° away from the centroid, and it is most likely to have the furthest euclidean distance from the centroid. Additionally, the puzzle piece was divided into four quadrants based on the centroid. Theoretically, there should only be 1 corner point at each of the four regions. Nonetheless, because the edges of a puzzle piece is not guaranteed to be aligned with the image, the likelihood of a corner at the exact angles described are really low. We added a 20° leniency. At each quadrant, we pick the the point that lies within (25° to 55° for Q2, Q3) or (115° to 155° for Q1, Q4) and has the maximum euclidean distance to the centroid. These are the points we estimate is most likely to be a corner point. We took the absolute values of all the degrees so that we do not have to create another set of "if statements" to check the negative degrees. In Figure 7, we illustrate how a segmented puzzle piece is divided into 4 quadrants, and the blue shaded regions of each of the four quadrants are where we estimate a corner point is most likely to be at. It is important to note that cropping the image so that the puzzle piece is centered is essential for this to work. As we are able to see from Figure 1, a puzzle piece may not always be at the center of the image in the original image. By cropping it based on the puzzle piece bounding box, we can center the puzzle piece to the best of our ability. But, it still cannot be perfectly centered because the piece may be rotated.

The centroid of the puzzle piece is calculated using the centroid value returned from the regionprop of the segmented puzzle piece. For puzzle pieces that were not segmented well, we estimate the centroid by dividing the height and width of the cropped image by half.

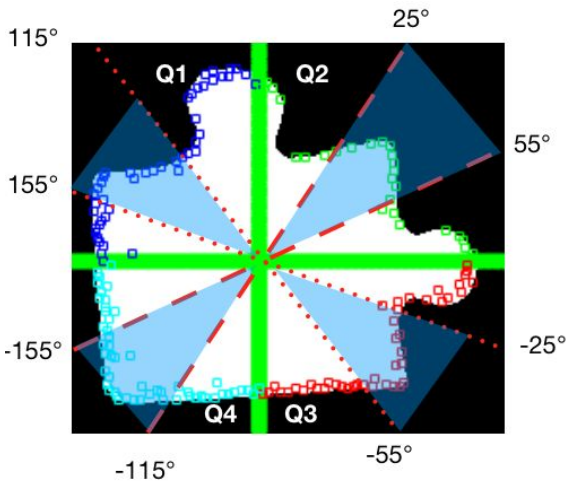

Figure 7: A segmented image that is divided into 4 quadrants. The blue shaded areas are the regions where we predict is mostly to have a corner point.

In order for this to work, we need to obtain the points that make up the outline of the puzzle piece, and we also need to know the distance and degree of every outline point with respect to the centroid. To get this information, we converted all of the puzzle piece points (i.e the white pixels) from cartesian space into polar coordinate space with respect to the centroid. We tried two approaches. We first tried finding the outline using the harris features. We then

tried using the “tips” that make the theta versus rho graph to get the outline of the puzzle piece. Both results were equally as good for well segmented puzzle pieces. In the end, we decided to use the results from the polar coordinates because harris features started picking corner points that are not part of the outline for poorly segmented images. (Refer to Figure 9)

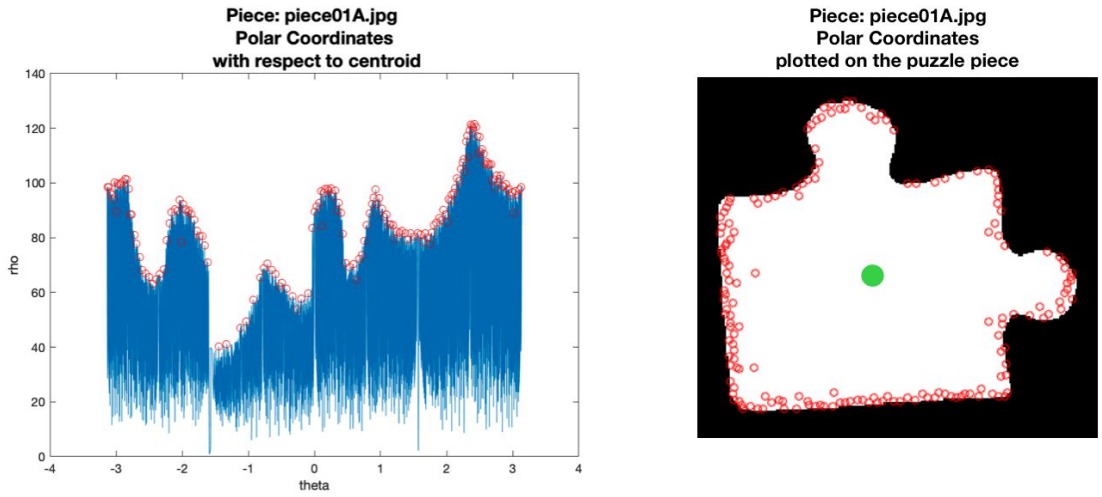

Figure 8: Left: The polar coordinate graph after converting the white pixels from cartesian space into polar coordinate space with respect to the centroid, which is the green dot on the right figure. The red dots in the graph are the outline points. Right: The outline obtained using the polar coordinates.

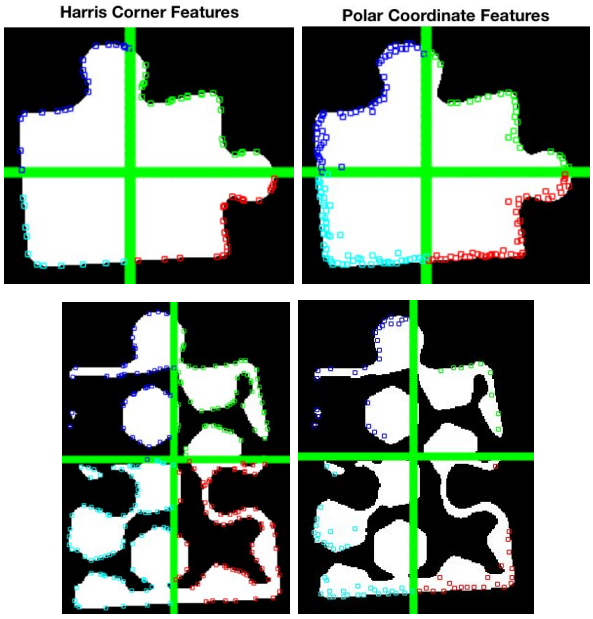

Figure 9: Left 2 Vertical Figures: The outline of the puzzle piece obtained using harris features. **Right 2 Vertical Figures:** The outline of the puzzle piece obtained using polar coordinates.

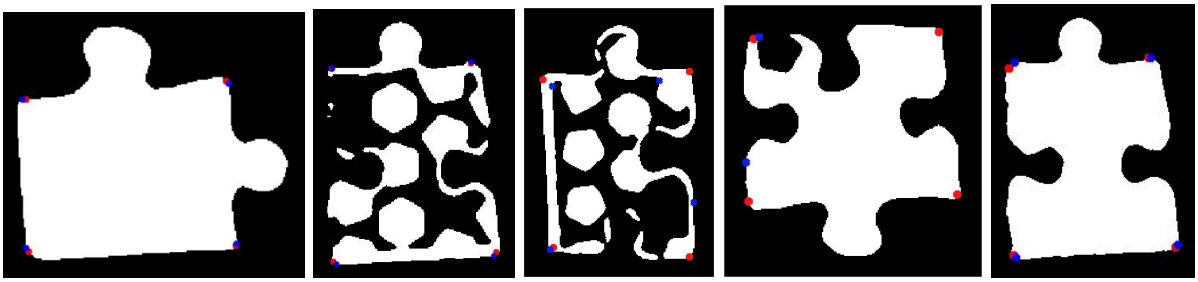

Figure 10: The result of corner detection. The red dots are the corners obtained using polar coordinates. The blue dots are the corners obtained using harris features.

As noticed in Figure 10, in general, polar coordinates did better for all the images. Therefore, in the end, we chose to use polar coordinates to help us find the 4 corner points.

Determining Edge Type

To determine edge type of the pieces, we employed the for corners found earlier to approximate the straight edge lines along each side of the piece, see figure 16. We obtained these edge approximation using the built in polyfit function in matlab on a 1 degree polynomial (i.e. a line). It approximates the slope and constant for a line that best fits two points. We then use that data to generate a set of points along that line using the slope-intercept formula ($y = mx + b$) and fit those points to the pixels in the image. We apply this method with all combinations of adjacent corners to obtain the 4 line segments representing the edges.

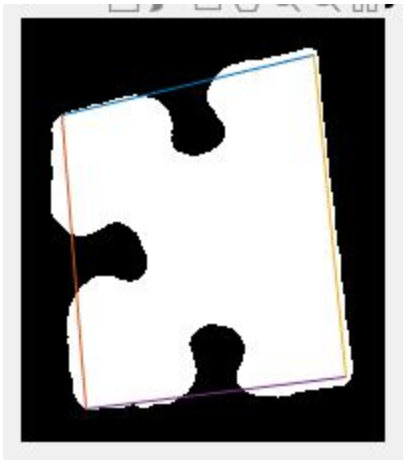

Figure 16: Piece with approximate straight edge lines

Using the straight edges approximations, we analyze the segments of the binary piece image on the outside and inside of these edges to look for protrusions and intrusions in the flat edge approximations. The logic is that if a portion of the foreground protrudes outside the straight edge, we have detected an outie, and if a portion of the background intrudes into the straight edge, we have detected an innie. The case where neither are detected indicates a flat edge on the puzzle piece. We analyze the image by placing a parallel line segment to the outside and inside of the straight edge approximation lines, these can be seen in figure 17. If the line segments intersect an intrusion or protrusion then than a positive is found for that edge to contain either in innie or an outie respectively.

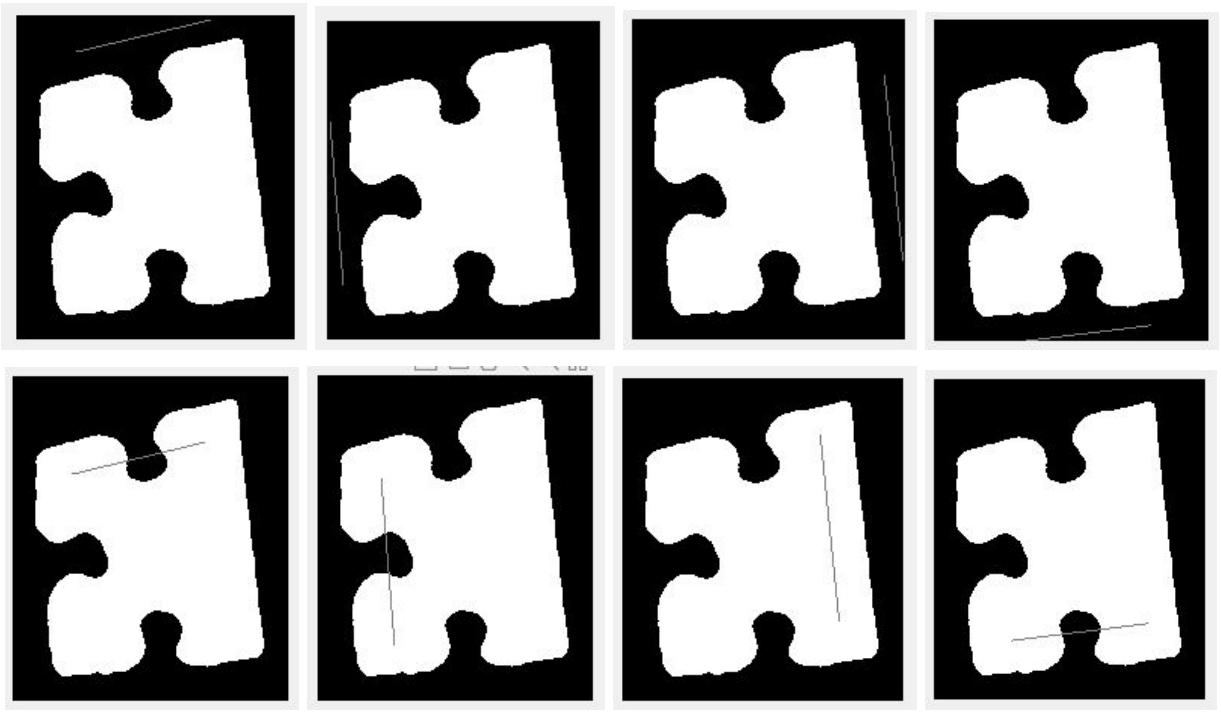

Figure 17: Images of detection bars for each edge

Learnings

We have learned a lot about the image processing methodologies through trial and error in our developments on this project. We have highlighted some of our learning below.

Corner Detection Trial and Errors

Initially, we tried to obtain the strongest 4 corners from all corners obtained using harris corner. But, this approach absolutely did not work.

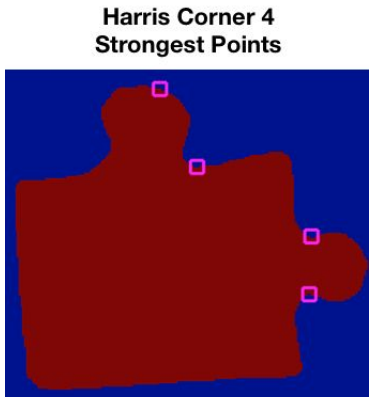

Figure 11: Strongest 4 corners obtained using the harris corner on a really well segmented puzzle piece.

As we are able to see from Figure 11, these four corners that were considered the “strongest” were very far away from the four corners that we are truly interested in. Because of this failure, we did not use harris corners for determining the 4 corners.

Breaking Hard Problem to Easier Problem

Because we broke the the larger problem down into smaller problems, it became significantly more manageable to work with. Further, with this approach, we were able to divide the task up between the two of us much easier. However, one of the challenges about this project is that each sub task is dependent on each other. For instance, suppose that corner detection does not do so well, it is likely that edge type detection will not work as well. But, we have soon learnt that if one stage of the processing pipeline fail for some images, it is better to keep moving on and try to fix these issues in the next stage. For instance, when trying to find the exact location of the puzzle piece, we provide two ways to crop the image.

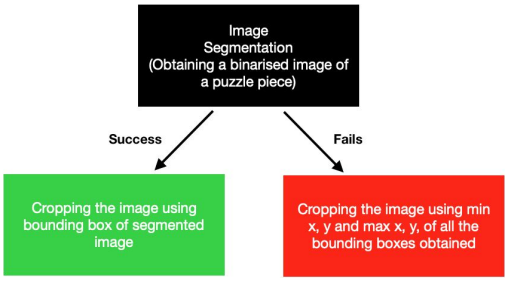

Figure 12: If image segmentation succeeds, which is 90% of the time, we can crop the image by taking the bounding box of the segmented puzzle piece. However, if the segmentation fails, we provide an alternative route.

As shown in Figure 12, we recover the problems introduced in stage 1 during stage 2 of the processing pipeline. This helped us move things along faster and not get stumped at 1 stage for too long.

Sub Problem Test Suits

Image Name	Top Edge Type	Bottom Edge Type	Left Edge Type	Right Edge Type
piece01A.jpg	Outtie	Flat	Flat	Outtie
piece02B.jpg	Innie	Innie	Outtie	Outtie
piece04A.jpg	Flat	Outtie	Outtie	Outtie
piece07B.jpg	Outtie	Outtie	Innie	Outtie

Figure 13 - A labeled vector array with image name and their correct edge type.

Initially, we wanted to create a Vector array of labeled data containing the image name and their corresponding edge type. However, we were running short on time, and thought that it is probably more important to get the project working rather than spending time to label the data and writing a test suit to check whether our program works.

In the future, though, we could automate the testing process by creating a validation data set like the one in Figure 13. With this validation data, we can write a program to output the accuracy of our model on the training data by

looping through each image and checking to see how much our prediction differs from the labeled data. This way, we would not have to run through 198 images each time, and manually check whether the predicted output is correct or not. Ultimately, making the testing process more efficient.

Results

Determine whether a puzzle piece is present or not

We were able to achieve 100% accuracy on the training data set for this sub problem.

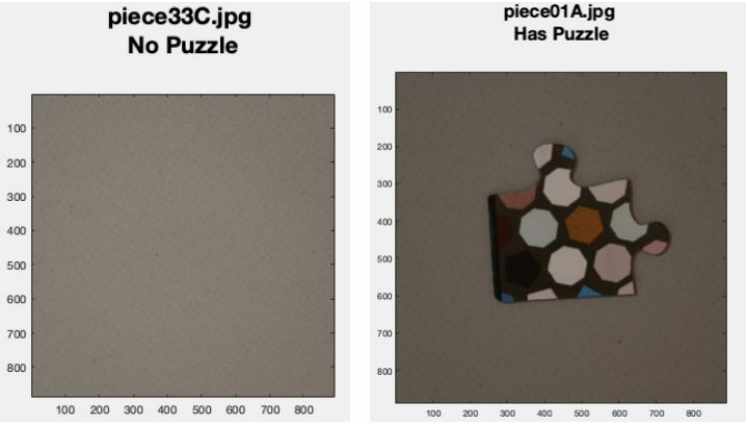

Figure 14: An example result from our algorithm for determining whether there is a puzzle piece or not.

Image Segmentation

Image segmentation was very successful with the approach taken. We were able to segment out the piece from the background with 97.5% accuracy. Only a couple of outliers resulted in flawed renderings as seen in mask A of figure 18. These flawed pieces are due to a bad image exposure where enough edge data could not be found. On the up side, these failed segmentations never occur more than once for each set of images for a given piece. Therefore, each and every piece can still be accurately segmented with the combined set of images.

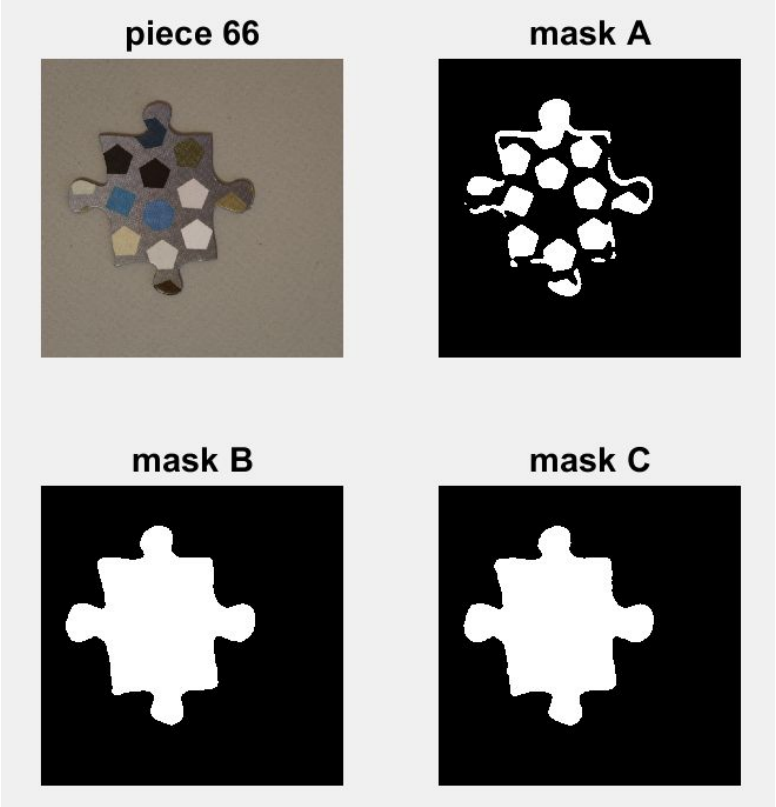

Figure 18: Sample of segmentation results with one failed segmentation

Finding the exact location of the puzzle piece

For this, we were also able to find the bounding box for each of the puzzle piece, and crop the image out to eliminate as much background as possible for every training image. For results, please refer to Figure 7.0.

Finding the 4 Corners

The corner finding worked for the majority of the images, but there are definitely some where it could do better. Refer to Figure 7.0 for corner detection results that worked. However, the precision here did not really matter, because we were able to recover from this failure in the next stage.

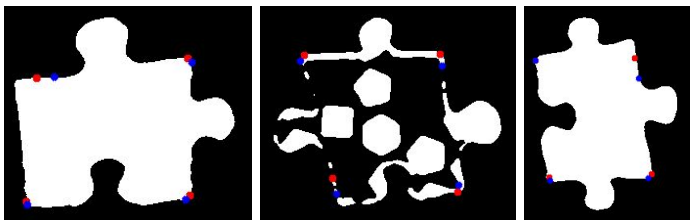

Figure 15: Cases where corner finding did not work so well.

Determining Edge Type

The results for edge type detection were promising, detecting edges with 97.5% accuracy, only failing on the pieces that had a failed segmentation. As you can see in figure 19, the results are accurately displayed for each side.

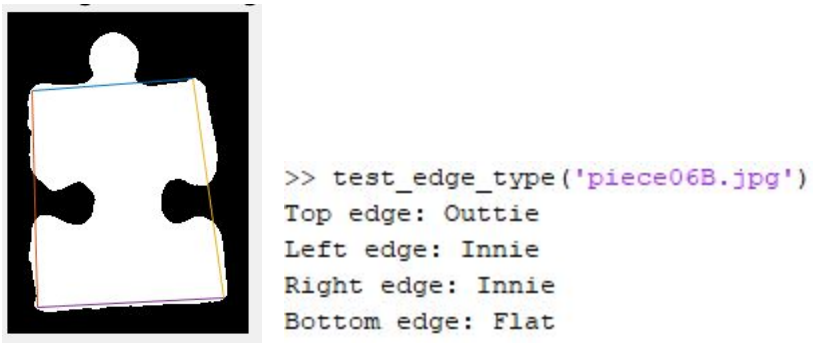

Figure 19: Edge type detection results

Conclusion:

Building a system to solve a jigsaw puzzle may not have as much of an impact as building autonomous driving car, but it is an opportunity for students like us to apply and learn some of the key techniques that are used in many modern day computer vision systems. In the future, we may also use similar techniques to solve some more useful problems such as lane finding in autonomous vehicles.

The problem of identifying and classifying puzzle pieces was a tricky one, and to get it to work, we had to spend a lot of time breaking down the problem, trying some stuff, learning what didn't work, talking to other *researchers* in the field, trying some more stuff, and eventually finding what did work. It took a lot of dedication.

Overall this project taught us a lot and we are both proud of the results we were able to obtain for segmenting the images, detecting puzzle pieces, finding the piece, finding the corners and identifying the edges. We may not have been able to solve the larger problem set out from the start of putting the puzzle back together, but our work most certainly brings us closer to that goal. In the future our processes could be used in a more complex program to finally match and assemble all of the pieces of the jigsaw puzzle.

Citations

https://stackoverflow.com/questions/41462091/how-to-find-coordinates-of-corners-in-matlab