JIGSAW PUZZLE SOLVER
USING IMAGE PROCESSING TECHNIQUES AND NEURAL NETWORKS

Jason Dang, Ravi Patel, Julia Braun

ABSTRACT

We present an automatic jigsaw puzzle solver program that, given an input image of a scrambled jigsaw puzzle under certain constraints, outputs an image of the solved puzzle. In the process, each puzzle piece is separated from the background and isolated using Sobel and Canny edge detection and identifying the contours. Corner detection is achieved by traversing the borders of the pieces and finding potential points that create angles within a certain range, then choosing the set of points that create the biggest area. Some issues in corner detection occurred, requiring manual adjusting of corner detection rmse values. Each piece is divided into four edge images classified with a label of "hole", "head" or "flat" obtained by a multiclass linear regression model. To match edges, transfer learning was achieved by categorizing each piece as a corner, edge or neither, and matching and combining edges with highest probability. This program is able to successfully solve puzzles with 20 or fewer puzzle pieces, at a speed significantly faster than a human manually solving the same puzzle.

1. INTRODUCTION

In this project our goal is to take input of a non-overlapping scrambled jigsaw puzzle and to output a solved picture of the puzzle. This will be done using various pixel level manipulation techniques, online libraries and PyTorch neural networks.

2. RELATED WORK

Our inspiration for developing an automatic puzzle solver originated from a video on YouTube by the channel "Stuff Made Here" [1]. Their puzzle machine solved puzzles of all white puzzle pieces, which gives it a different set of problems due to the fact it ignores one of the most important characteristics of human vision, which is color. Color plays a crucial role in puzzle-solving due to its ability to convey information, create distinctions, and facilitate pattern recognition [2]. The incorporation of color enhances the visual complexity of puzzles, allowing people to discern and categorize puzzle features more efficiently.

3. APPROACH TO PUZZLE SOLVER

Creating a puzzle solver requires various operations to simplify the task of putting the pieces together. In this section we will explain our thought process and stages to build our current puzzle solver version. In each stage, we will describe why the stage is important, the challenges we faced, and the method we have decided on as our solution to the problem. Additionally, we will talk about other methods we attempted to use.

3.1. Requirements

We laid out some requirements for the input image that would simplify the workload towards designing the functionality of the puzzle solver: the input image's puzzle pieces must be rectangular shaped, the puzzle pieces must not be overlapping, the input image must be captured "top-down" with minimal distortion, the background of the image must be a solid color dissimilar to the main color of the puzzle pieces, each side of a puzzle piece must be either a flat side, a "head", or a "hole", the puzzle in the image must consist only of pieces from the same puzzle with no missing or extra pieces, and the pieces must be laid out such that each piece can have a rectangle drawn around it without touching any other piece's rectangle. Figure 1 shows an example of an input image that would satisfy these requirements.

Figure 1: Image of puzzle to solve

3.2. Background removal

Removing the background of the puzzle image is crucial to isolate and independently analyze and manipulate each puzzle piece. Our biggest challenge we encountered came from background noise, defined by subtle variations in color tones, and from the background removal process that

occasionally removed distinct boundaries of the puzzle pieces. Our first method of background removal was to find the most common color within the image and remove it. To achieve this, we identified the highest occurring pixel value in each RGB channel within the image. Then, we designated all pixels deviating a specific standard deviation from this predominant value to be set to 0, and everything else be set to 1. This performed well in removing the background, however the puzzle pieces potentially lost their defining boundary.

We then transitioned to using Sobel and Canny edge detection to detect the edges of the puzzle pieces that we could then extract by filling in the edge images. This gave us much better quality of the extracted puzzle pieces, with less distortion in the edges of the puzzle. Sobel edge detection looked for the largest changes in the x and y gradients. Afterwards, Canny edge detection looked for the gradient changes in the Sobel edge detected image, but also performed non-maximum suppression and hysteresis thresholding, which helps to smooth and remove unnecessary edges detected by the background (Figure 2). We further used different thresholding (gaussian, binary and Otsu) and erosion and dilation to further sharpen the image (Figure 3).

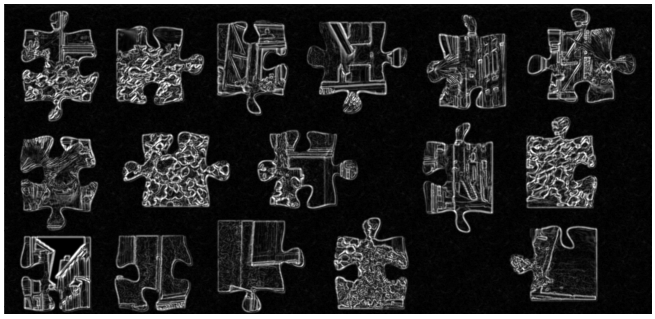

Figure 2: Combination of Sobel X and Y

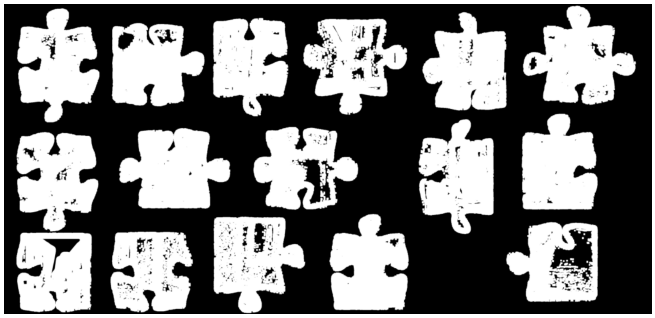

Figure 3: Combination of Sobel X and Y, Thresholding, Erosion and Dilation

3.3. Puzzle piece separation edge detection

The next step for us was to detect each puzzle piece in the removed-background image. With the previous step giving us the filled-in binary scale images of the puzzle pieces, all we had to do was find contours for each piece. For this purpose, we used cv2 library findContours function to get bounding boxes. However, as seen in Figure 4, even with filtering of very small sized bounding boxes, other bounding boxes were detected within the image were overlapping, and others were completely inside bigger bounding boxes. To solve this, we wrote another algorithm to combine overlapping bounding boxes (Figure 5).

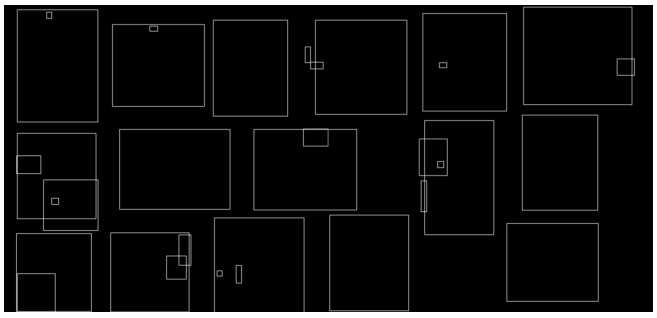

Figure 4: Contour detection of image shown on black screen

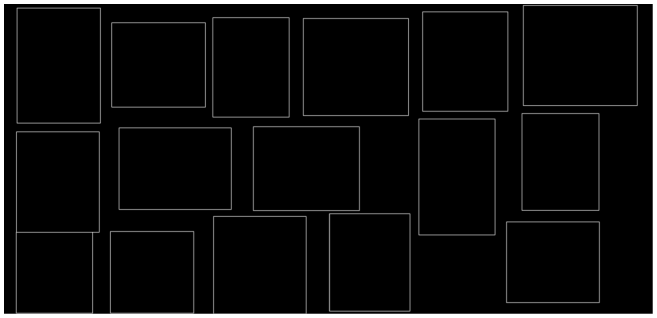

Figure 5: Contour detection with additional algorithm for filtering and combining on black screen

After detecting each puzzle piece, we divided them into separate images. However, to convert them from edge detected images into the puzzle shape we needed to process them as seen in Figure 6. First, we started by flood filling the image after which we inversed this flood filled image. Finally, operating logical OR with the original image and eroding it gave us the puzzle piece. We could then use this as a mask to get the colored piece (Figure 7).

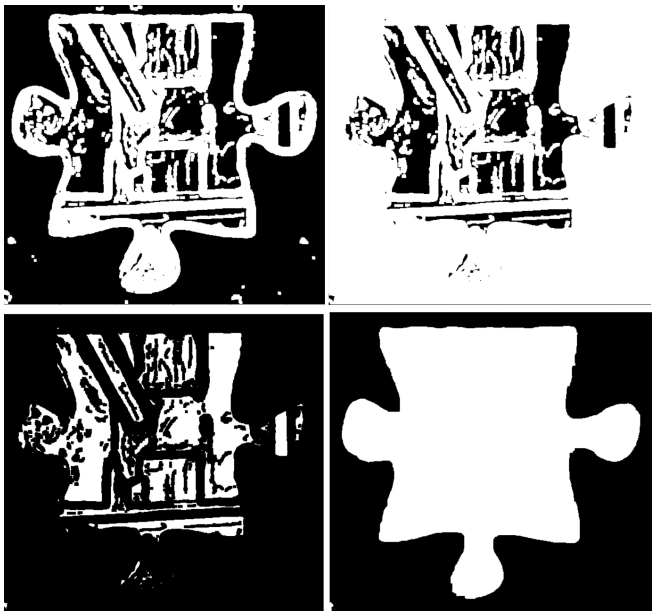

Figure 6: Top left image: Original image with edge detection, Top right image: Flood filled image, Bottom left image: Inverse image of the flood filled image, Bottom right image: Logical OR of original image and the inverse image

Figure 7: Individual piece removed from background

3.5. Corner detection

Identifying the corners of each piece turned out to be the most challenging part so far. This stage was important for realigning pieces and identifying edges for the next step. The challenges arose due to more than four "corners" being detected from the "head" and "holes" of the puzzle pieces.

We experimented with various approaches such as Harris Corner Detection and using a graph correlating angle from the center of the piece and radius from the center. The Harris Corner Detection from the cv2 library works by evaluating a window around a pixel and looking at the derivative of pixel intensities in all directions of the window. This was a problem because the "heads" and "holes" also have a distinct change in intensities, resulting in multiple false corners being detected. Singling out the four correct corners from this proved to be a challenging task. We then tried to use a graph correlating the polar coordinates of the edge, angle vs radius from the center. Once plotted, the sharp peaks in the graph would tell us where the corners would be. The problem with this was: how do we find the peaks? We could tell where the corners were, however, the image was noisy and small false peaks were being shown. The boundary of the puzzle pieces was not smooth and created small and jagged edges that made peaks in the graph.

The final solution we came up with was creating our own function that traverses the entire border and finds potential points that created an angle ranging between certain chosen degrees. This was done by saving the last n points and finding the angle the line created from the first n/2 points made with the second n/2 points. In this step we were able to reduce the points on the "heads" and "holes" of the image by removing the point as a corner if the standard deviation of the points from the line were high. This worked because the corner points were straighter and the points at the "heads" and holes" were curved. This provided us with multiple potential points. This was better than the previous two approaches because it was finding points in all four corners. Next, we clustered points in close proximity, considering them as repetitions of the same corner. This involved selecting the point with the lowest standard deviation within the cluster to serve as the representative point. This reduced the number of points we had to look at and allowed us to finalize the corner points by finding the set of points that created the biggest area, since the points creating the biggest area had to be the four corners.

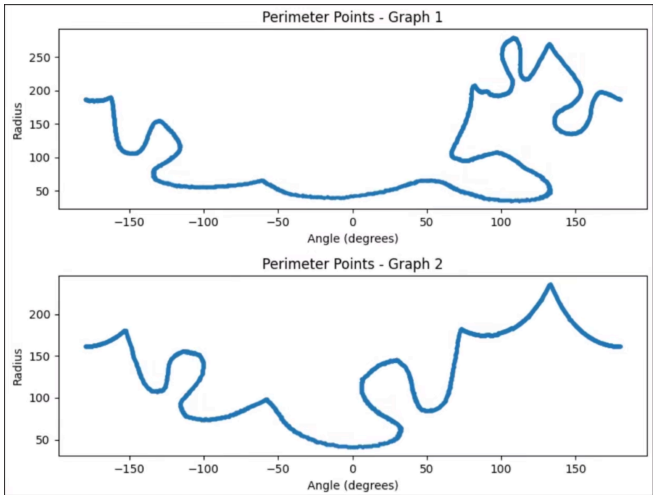

Figure 8: Polar graph of the border

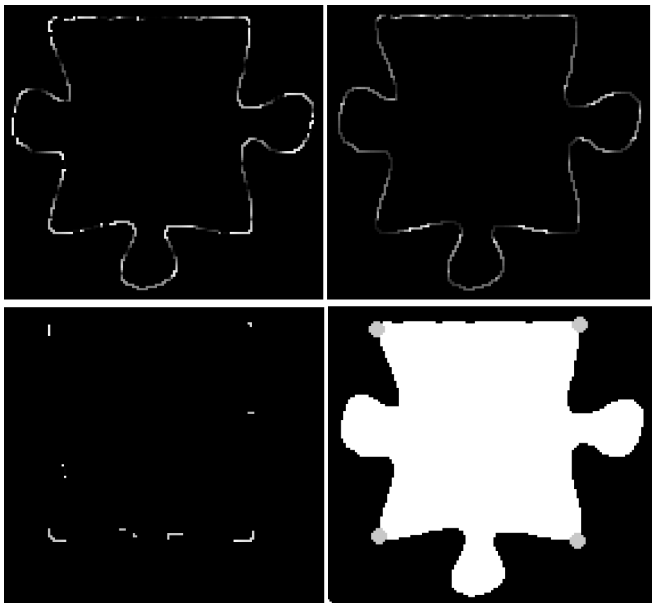

Figure 9: Top left: Boundary with angle at each point(as the color goes from black to white the angle increases from 0 to 90), Top right: Boundary with rsme value at each point(perfect black is 0 rsme value and it increases towards white), Bottom left: boundary with points left are filtered based on angle and rsme, Bottom right: corners detected

3.6. Edge extraction and detection

After detecting each corner of a piece, we divided each piece into four edge images for further processing. We found each of these edge images by connecting two corners with a line and moving that line parallel outwards till there were no more image pixels on that line. We then drew another parallel line at the center of that piece. This gave us the boundary of the edge image. Out next step was to detect if the edge was a "hole", "head" or a "flat". Our first solution to this was by measuring the area in that image which was not covered by the image. If it was above a certain threshold, it was a head, if below certain threshold, a flat, and in the middle, a hole. However, this caused us some problems with different size images. So we decided to train a multiclass logistic regression to determine if the image was a head, hole, or flat. To train this classifier, we used the relative change in height as you moved along the edge of the image. We gathered data on thousands of pieces and hand labeled them with their correct classification. We then trained the model and fine-tuned it to get 99.8% accuracy. We avoided overfitting by using cross validation. After the logistic regression model had figured out if the edge was a head, hole, or flat, we needed to process the images so that they were easy universally easy to combine with each other. To do this we oriented all the heads facing downwards and the holes facing upwards. Furthermore, we cropped and padded them to the same size such that the bottom of the hole shape touched the bottom of image, and the top of the head shape touched the bottom of the image as well. After this, we could combine the heads and holes by directly overlapping one image over the other, with the combination of logical XORs, ORs, and ANDs.

Figure 10: Head and Hole edge images cropped and padded to the same size

3.7. Edge match detecting

For detecting if two pieces are a match or not, we decided to use transfer learning. We fine-tuned the ResNet-18 model to fit our data using the binary cross entropy loss function. For collection of the data, we combined all the heads and holes together and manually labelled if each combination was a match or not. However, this data was highly skewed since the match images were significantly less than the non-match images. To solve this, we used data augmentation, that is flipping orientation and adding noise and other modifications in the images. We also reduced the number of non matches to further improve the balance in the dataset. After collecting all our data, we pre-processed it using resizing and normalization. We then trained the model upon this data, and to prevent overfitting we used early stopping.

Figure 11: Non-match combination

Figure 12: Match combination

3.8. Edge comparison algorithm

Finally, after training our model to predict how well pieces match with each other we can finally move onto solving the puzzle. To start systematically reconstructing the puzzle the puzzle pieces are first categorized into corners, edges, or neither. We start with the frame, assuming that this will prevent us from comparing every single piece and reduce the number of pieces we will have to compare in the future. Taking the first corner found as the top left corner piece, we move around the frame counterclockwise. The first step is to grab every unused piece and combine the edges if they are of opposite types; heads to holes and holes to heads. The combined edges are passed through our fine-tuned ResNet-18 model, which spits out probabilities representing the compatibility of the pieces. The highest probability out of all pieces is chosen to be the edges match. This process is repeated until the frame is created; however, it is important to keep track of orientation throughout the process for reconstruction. The edges that are matched are labeled with which side the edges should be facing. The important part of this step is to make that we logically rotate the "bottom" of the pieces as we hit the corners of the frame. After the frame has been formed, the process now starts in the next inner later, moving counterclockwise. This step uses the same idea as constructing the frame, however, there is more complexity due to the possibility of multiple edges touching an inner piece. Therefore, to compare every edge that the piece is adjacent to. This process is repeated with each consecutive layer until all pieces have been used.

Figure 13: Top left: Original image, Top right: Rotated Image, Bottom: Padded and cropped the rotated image for final combination

3.9. Completing the puzzle

After finding the location of each puzzle piece in the completed puzzle grid, we need to reorient the puzzle pieces. Earlier when comparing edges, we found which edge would be at the bottom in the final layout; therefore, with the help of this, we reorient all the puzzle piece in the right orientation. After this, we need to pad and crop each puzzle piece to combine it into a single big, completed puzzle. We do this by considering each puzzle piece's width and height and the height of heads and the depth of holes. After completing the padding of each piece, we can OR them together to display the final puzzle image.

Figure 14: Completed puzzle

4. RESULTS OF PUZZLE SOLVER

We tested our puzzle solver on random online jigsaw puzzles that we had not seen before. The puzzle solver ran on two computers with different hardware: a Dell XPS 15 (16 GB RAM, Intel Core i7 Processor, and Intel UHD Graphics) and a M2 MacBook Air (16 GB RAM and Apple M2 Processor). For a successful solve of a 16-piece puzzle, the Dell XPS 15 finished the puzzle within 21 seconds, meanwhile the M2 MacBook Air finished within 13 seconds. Almost half the time with a faster processor. From self-testing, this was much faster than manually solving the puzzle.

The solver worked on most random puzzles with fewer than 16 pieces, however there were issues with finding corners on puzzle pieces. This caused errors for the successfulness of the puzzles since there are invalid edges. Therefore, various puzzles required additional manual fine-tuning to the corner detection rmse value to successfully

solve the puzzle. This is because the heads have almost well-defined angles that can be identified as a corner. We found that generally two rmse values, 6 or 8, work for almost all puzzles.

Another problem we encountered is with the edge comparison algorithm used to reconstruct the puzzle. This step assumes that it will always choose the correct piece to be successful on the first attempt by selecting the attaching piece with the highest probability to work. A way to solve this problem in the future is to backtrack when solving the puzzle if it deems that it cannot be completed.

5. CONCLUSION

The automatic puzzle solver has been completed well enough for use with the help of image processing techniques, custom algorithms, and machine learning models. This program can successfully solve puzzles with 20 or fewer puzzle pieces. With more time, it is possible to rewrite the algorithm to implement backtracking, add more data for the model to predict with higher accuracy, and adjust the corner algorithm to choose corners with higher accuracy. An addition that could be made for the project is to add an interface for users to increase accessibility.

This project has allowed us to apply the lessons taught in ECE 371Q: Digital Image Processing to create an automatic puzzle solver inspired by a video on the internet.

6. REFERENCES

[1] Stuff Made Here. (2022, Nov 30). *Worlds hardest jigsaw vs. puzzle machine (all white). Youtube. https://www.youtube.com/watch?v=WsPHBD5NsS0&t=1158s&ab_channel=StuffMadeHere*

[2] Bovik, A. (2023). Course Introduction, Imaging Geometry, Perception, Pixels, Perceptrons [PowerPoint Presentation]. https://utexas.instructure.com/.