Edge Piece Detection

Michael Smith

April 23, 2022

Abstract

Puzzle solving is an activity that is visually intensive and requires a fine attention to detail. One part of puzzle solving that can be cumbersome is sorting each piece into a category of whether or not it is an edge piece or not. This is the typical order of procedures for most puzzles as the pieces must be taken out of the box, flipped over, and separated into like categories.

In an attempt to remove this fickle task that gets in the way of enjoyable puzzle solving, a method for detecting edge puzzle pieces has been developed using computer vision. The method has been implemented in python using Open-CV and linear algebra applications to determine whether or not a puzzle piece is an edge piece or not an edge piece.

1 Introduction

Solutions for edge detection in computer vision are widely used to determine the locations of lines. Straight lines, in the case of determining the location of edge pieces, was the obvious target of detection for this method. Because of this, the Hough transform implemented in Open-CV was used to detect the edges of the puzzle pieces. This implementation however did not accurately detect the locations of straight edges in the image O.-C. Developers n.d.

Because of the lack of results from the Hough transform, a different approach was used to detect the location of edge pieces. Rather than detecting straight edges on the image, which can be challenging given the resolution and quality of the image, the distance from a bounding box edge of a particular puzzle piece was taken. If the maximum distance for one of these sides on a puzzle piece was below a certain threshold, then the piece was highlighted as an edge piece. This general approach was inspired by the following stackoverflow reference rikyeah n.d.

2 Image Manipulation

From the input image, some manipulation must be performed on the original image to acquire more pertinent data about the image.

2.1 Thresholding

The first step in finding the location of edge pieces was to find the location of puzzle pieces in the image. This required a filter that forced everything below a certain gray-scale value to be black and everything above that value to be white. This function was used from the Open-CV library to make the locations of puzzle pieces clear and set up the image for the next step in image manipulation

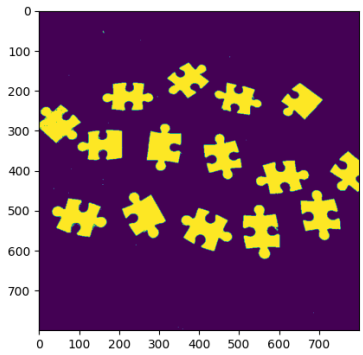

Figure 1: Threshold of input image M. Developers n.d.

2.2 Contours

After the threshold image was created, the black and white image was passed to the Open-CV contours function that outlined each puzzle piece with coordinates marking the perimeter of each puzzle piece. Once this operation was performed, a list of coordinates for each piece was obtained and used for functions in the rest of the method

One issue that arose was the additional contours that were detected by the contours function call even though those contours did not indicated puzzle pieces. These were typically caused by par-

ticles on the white background that the picture was taken. To prevent these contours from being counted as puzzle pieces, an area check was performed on each piece to ensure that the are of the contour was above a certain margin.

3 Segmentation

Once the contours of each puzzle piece is obtained, it is important to understand the relative direction of each puzzle piece and how each puzzle piece must be segmented.

3.1 Bounding Box

To get some idea of the direction and additional useful data of each puzzle piece, the minimum rectangle that can be created around each puzzle piece was created to bound each piece with a rectangle that encompasses the minimum area that the contours of the puzzle piece could take up. The coordinates of these boxes proved beneficial when testing each side for its feature.

3.2 Contour Division

Once the box was created, the puzzle piece had to be divided into sides in some way. One method that was originally approached was the Harris corner detector implemented in Open-CV. Like the issue with the Hough transform, the Harris corner detector implemented in Open-CV was too dependent on the quality of the image and struggled with detecting the correct corners. Because of this issue, the problem was generalized by drawing lines that connect the corners of each bounding box. These intersecting lines were used in segmenting the existing contour on the basis of each point's orientation to the lines created.

The cross product of each contour was taken with respect to the two vectors defining the diagonal line created in the previous step. This calculation was given by the following.

$$(p \vec{-} a) \times (b \vec{-} a) < 0 \tag{1}$$

p in this case, represented by $\vec{p}$, is the point to be tested against the lines that are defined by $\vec{a}$ and $\vec{b}$. When this computes to a value less than 0, and the same segment satisfies the relationship for the other diagonal, a new segmented contour is created. This is repeated by inverting the above relationship for the pair of diagonals manjeet$_0$4 n.d. Once all four combinations were used, the contour was properly segmented nikhiltanna33 n.d.

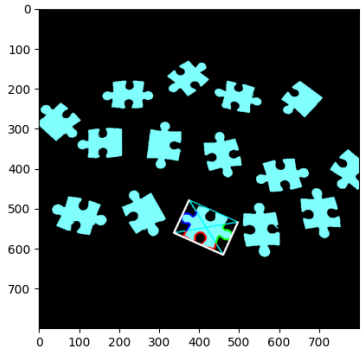

Figure 2: Boxed puzzle piece M. Developers n.d.

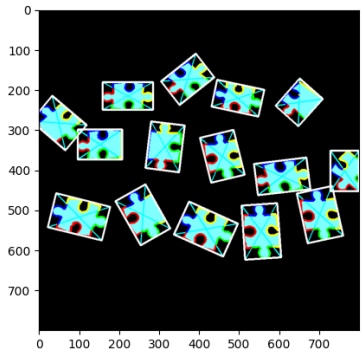

Figure 3: Fully boxed puzzle pieces M. Developers n.d.

4 Normalized Distance

Once the contour was divided into segments of four, the feature of each side had to be determined. Specifically, the challenge arose as to how to determine one side of a puzzle piece was an edge, a hole, or a knob. One way to determine this was with each coordinates normalized distance away from a given line.

4.1 Distance Calculation

To calculate the distance between each coordinate and a given line, the line had to first be defined. From the bounding box created in the setup phase of this method, a line could be used from the coordinates of the box. The edges of this box were then used to obtain the normal distance of each coordinate to this line. The line chosen for each segment was the adjacent line of the given contour segment.

The calculation was performed by the following.

$$D = \frac{||\vec{AP} \times \vec{d}||}{||\vec{d}||} \tag{2}$$

Where $\vec{d}$ is the direction vector that defines the box line segment and P defines the point being tested. The output D is the perpendicular distance that the point is away from the line segment used for the computation DotPi n.d. These values were first smoothed and then plotted.

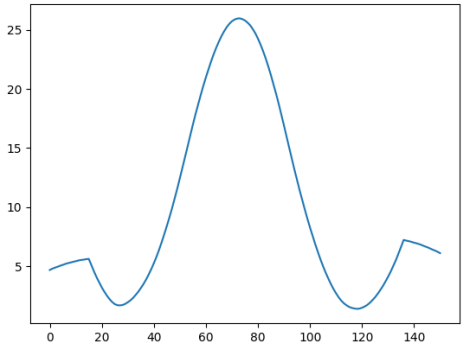

Figure 4: Distance plot of hole feature M. Developers n.d.

M. Developers n.d.

4.2 Rolling

An issue occurred when computing the distances from the segments created and the lines defined by the bounding boxes of the puzzle pieces. Based on the location that the contours function started in Open-CV, the segment was occasionally not created as one continuous list of coordinates. Because of this, a rolling operation had to be performed on the contour to prevent the distance results from being tarnished by a lack of continuous data N. Developers n.d.(b).

5 Feature Plots

The smoothed list of distance values for each segment created a defining relationship for each of the respective puzzle features. The values needed to be smoothed by a convolution step, because the contours obtained represented discrete values. These discrete values created noise in the distance output that needed to be reduced to create a clear relationship. Based on the shape of the graph created, a clear decision of whether or not a segment was a hole, knob, or edge feature could be determined. This approach was from the numpy.convolve method N. Developers n.d.(a) The feature plot for a hole was given above in 4; the others are given below.

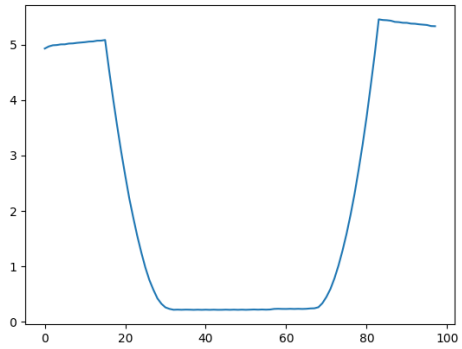

Figure 5: Distance plot of edge feature M. Developers n.d.

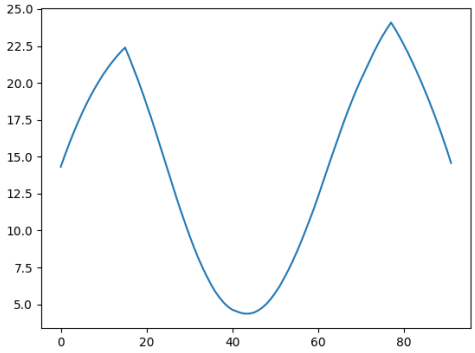

Figure 6: Distance plot of knob feature M. Developers n.d.

The plots are inverted because the distance is being calculated from a line outside of contour. This means that knobs are closer to this line than holes for example. This gives a clear indication for the feature detected on each segment.

6 Classification

Despite the fact that holes and knobs could be detected, the main objective of this method was to determine the presence of edges. To differentiate between distance data-sets, a defining characteristic of the edge plot had to be used. The maximum value of the edge plot is much smaller than the maximum value from all other plots. This

is because the edge features lies completely flat against the minimum spanning rectangle. This minimizes the distance between the edge piece and the bounding box line segment.

While there are far more elegant solutions to classify each side as an edge piece or not an edge piece, classifying each piece as an edge piece if it has a plot maximum below some constant arbitrary value that prevented knob and hole pieces from being detected if they were not edge pieces.

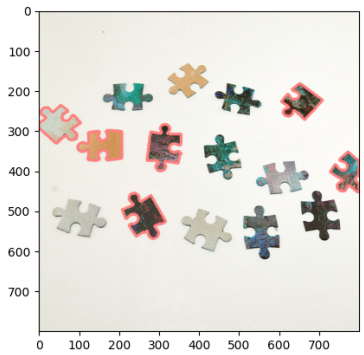

Figure 7: Final detected edge pieces M. Developers n.d.

The first assumption was to find a value that clearly would not allow the wrong feature to be classified, but then the potential for pictures taken with larger puzzle pieces needed to be considered as a larger puzzle piece might cause the distance plotted in a straight edge plot to have a higher maximum. The only reason that the edge plot has a maximum larger than 0 in the first place is because of how the puzzle piece was divided into segments. In most cases, the segment created included the other sides of the puzzle. The assumption made is that the effective limit to segregate edge pieces from non-edge pieces is some fraction of the adjacent puzzle edge length. A fraction of 0.1 was used for this implementation as it allowed for the smallest error.

7 Conclusion

The implementation of this method was challenging at first to identify the pieces and gather some sort of object that was associated with each puzzle, but it became easier when the method of thresholding was used in conjunction with the Open-CV contours function. This helped gain access to the geometry of each puzzle piece and allowed the performance of more linear algebra techniques to find the distance between each bounding side and the contours of each puzzle piece. Once it became clear that an approach of taking the max of each data set and determining if that number was less than some preset number based on the size of the input puzzle pieces, it was quick and easy to identify each puzzle piece as an edge piece. Now to find the edge pieces in a puzzle, all one needs to do is take a picture.

References

Developers, Matplotlib (n.d.). *Matplotlib: Visualization with Python*. URL: https://matplotlib.org/. (accessed: 04.22.2022).

Developers, Numpy (n.d.[a]). *numpy.convolve*. URL: https://numpy.org/doc/stable/reference/generated/numpy.convolve.html. (accessed: 04.22.2022).

— (n.d.[b]). *numpy.roll*. URL: https://numpy.org/doc/stable/reference/generated/numpy.roll.html. (accessed: 04.22.2022).

Developers, Open-CV (n.d.). *Hough Line Transform*. URL: https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html. (accessed: 04.22.2022).

DotPi (n.d.). *Distance between point and a line (from two points)*. URL: https://stackoverflow.com/questions/39840030/distance-between-point-and-a-line-from-two-points. (accessed: 04.22.2022).

manjeet$_0$4 (n.d.). *Python — Filter list by Boolean list*. URL: https://www.geeksforgeeks.org/python-filter-list-by-boolean-list/. (accessed: 04.22.2022).

nikhiltanna33 (n.d.). *How to invert the elements of a boolean array in Python?* URL: https://www.geeksforgeeks.org/how-to-invert-the-elements-of-a-boolean-array-in-python/. (accessed: 04.22.2022).

rikyeah (n.d.). *Want to detect edge and corner parts in a jigsaw puzzle, but can't find the 4 corners of each piece*. URL: https://stackoverflow.com/questions/36703964/want-to-detect-edge-and-corner-parts-in-a-jigsaw-puzzle-but-cant-find-the-4-co. (accessed: 04.22.2022).