Powerful Puzzling

A Jigsaw-Puzzle Solver That Works with Island Pieces

Jean Charle Yaacoub
j.yaacoub@queensu.ca
Queen's University
Kingston, ON, Canada

James Gleave
17jhg5@queensu.ca
Queen's University
Kingston, ON, Canada

Christian Muise
christian.muise@queensu.ca
Queen's University
Kingston, ON, Canada

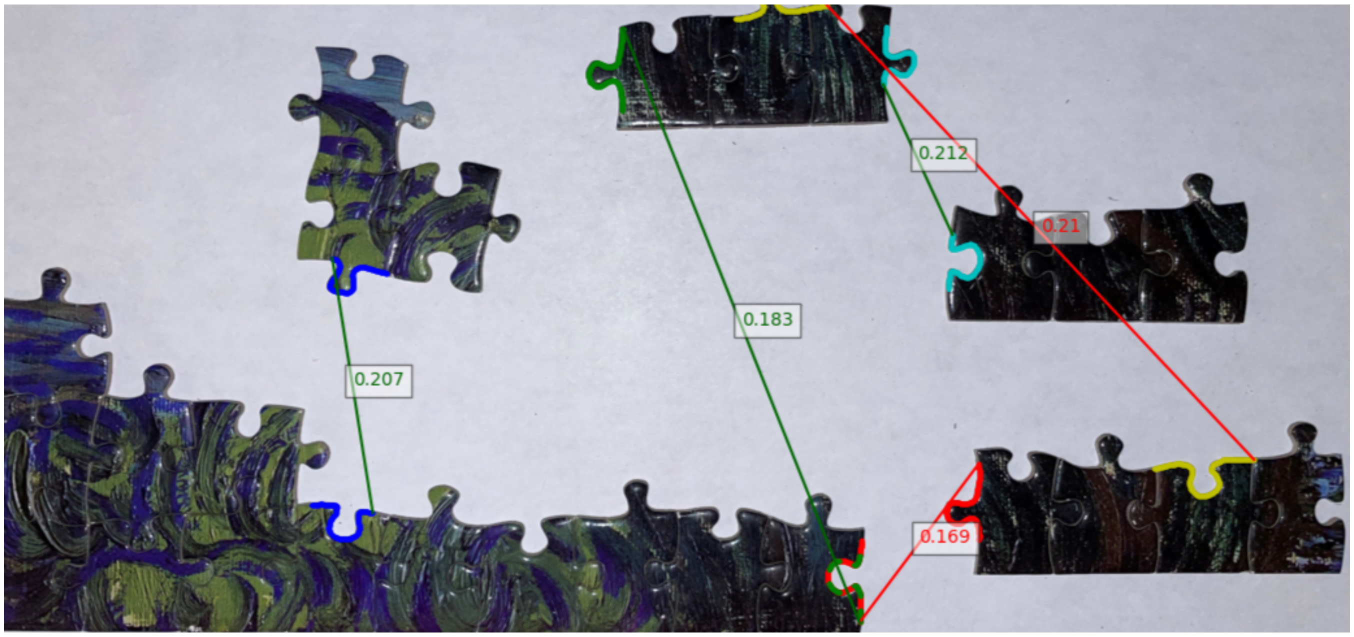

Figure 1: Top five matches found from Powerful Puzzling on a puzzle with island pieces. Lines in green indicate correctly identified matches and the numbers are their corresponding match scores (lower is better).

ABSTRACT

Jigsaw puzzles are a staple of at-home entertainment and have exploded in demand due to COVID-19 forcing people to stay at home [5], and with that comes the demand for jigsaw puzzle solvers. However, currently available solutions require a fully disassembled puzzle in order to work [9, 12], robbing the player from the satisfaction of completing a puzzle.

Here we explore a solution to this with our "Powerful Puzzling" algorithm. A jigsaw puzzle solver that can work with island pieces (a group of two or more connected pieces), making for a more practical approach to puzzle-solvers allowing for its use at any stage in the puzzle-solving journey. The backbone of our algorithm is the unrolling of border contours into 1D strips which can be segmented and compared with other strips to find matches. By working in 1 dimension we do not have to worry about making rotations and translations to get pieces to match.

Furthermore, when comparing strips, we use Dynamic Time Warping (DTW) for both shape and color matching combining the results to get a final match score [13]. We apply a weighting of (2, 1) for the shape and color score, respectively, to get the best performance with three out of the top five matches correctly identified. And with the use of high-level filters to eliminate unlikely matches, we can avoid performing costly DTW comparisons allowing the entire program to be run in under 3 seconds.

From the results, we observe that the program still has room to improve. For instance, the technique for image segmentation is not dynamic to all environment conditions and requires even lighting and a plain background to accurately extract the borders of each puzzle piece. The results could be improved with better segmentation techniques/more time spent training and tuning the Mask-RCNN we had attempted to use. In addition, the color matcher lags far behind the shape matcher in terms of providing accurate match scores. However, the Powerful Puzzling algorithm still shows promise and more research is needed to improve the sampling of the colors to get a better representation of the border, and/or the DTW algorithm used.

KEYWORDS

Computer Vision, Mask-RCNN, Puzzle-solver, Puzzle Matching

1 INTRODUCTION

Current jigsaw puzzle solvers are simple algorithms that require ideal conditions to extract the borders of individual puzzle pieces. They can completely solve a puzzle from start to finish but fail to work with island pieces and thus cannot pick up from where a player left off. This failure is due to their reliance on brute force matching, performing rotations and translations with each pair of puzzle pieces until a match is found [9, 12]. Herin we define a match as the pair of puzzle pieces that have the lowest match score, representing the distance between border pairs in terms of their color values and shape. The lower the distance, the lower the match score.

Using this idea of a match score, we explore the problem of the jigsaw puzzle solver and present a solution that works with island pieces. We approach the problem from the perspective of a tool to help players in their puzzle-solving journey. Or, in other words, to provide hints as they solve the puzzle without requiring disassembly.

We begin with a background on the puzzle solver with a short dive into the current day solutions and other related works that were useful to the inception of the Powerful Puzzling algorithm. Next, we explain our approach to solving the problem from a high-level perspective. And follow up with a detailed dive into how the problem was broken up into smaller subproblems and how we arrived at their corresponding solutions. Finally, we end the report with an evaluation of the work done, a summary that includes an analysis of the results, and our ideas for future work/improvements.

2 BACKGROUND

In our approach, we consider how a puzzle player could use such a tool. We do not want to rob the player of the enjoyment of solving a puzzle, and so our solution aims to solve puzzles that range from disassembled to partly completed with island pieces. This allows us to provide help to players at any step in their puzzle solving journey.

Examples of current puzzle solvers include Maxim Terleev's from his article in Towards Data Science and Abto Software's solver on their own site. These solutions are straightforward with Terleev's being done in around 300 lines of code, and Abto following a similar structure but without any code provided [9, 12].

Both solutions follow the same general process. Starting with image processing, the goal is to extract the borders/contours of the puzzle pieces from the image. Here, both solutions use a simple thresholding technique to create a mask that separates the background from the individual puzzle pieces. Then this mask is used to extract the borders using tools like OpenCV's *findContours()* to do so [2]. Then they identify the location of the locks on each (the jigsaw nodes that connect with other pieces) and segment them for use in matching. For matching, there are two metrics that are used: the shape of the lock and the color values of the points along said lock.

For the shape matching Terleev uses OpenCV's *matchShape()* function with each lock to determine similarity using Hu-moment values [2]. The team at Abto Software gets a match score by calculating the max exclusive (XOR) of adjacent areas after performing 10-pixel length translations left, right, up, and down [12]. For color matching, Terleev uses Dynamic Time Warping (DTW) on the HSV values along each segment. And Abto samples an average of the RGB values for pixels that are perpendicular to the border and uses that to compare two pieces by calculating their difference. Finally, with these metrics calculated they can identify the best matches for the puzzle by combining the two into a final score.

3 APPROACH

For our approach, we started with a solution similar to Terleev's and made a couple notable differences to improve performance and include island piece matching. This approach can be summed up with five steps:

(1) **Image segmentation:** extracting the border of each island piece.
(2) **Border unrolling:** convert the border into a 1D strip.
(3) **Lock identification:** identify and segment out locks on the strips.
(4) **High-level filter:** filter out unlikely matches based on concavity.
(5) **Shape and color matching:** using DTW to get a final match score using shape and color.

3.1 Image Segmentation

For image segmentation we tested four different methods to extracting borders: adaptive thresholding, clustering, superpixelization, and using Mask-RCNNs (a deep learning approach).

3.1.1 Thresholding. Adaptive thresholding is the fastest and simplest approach. It separates desirable objects in the foreground from the background using pixel intensities and a threshold value as a cutoff. For this we used OpenCV's *adaptiveThreshold()* method, and to extract the actual borders from the segmented pieces we used *findContours()* [2]. The downside to this method is that it only works under ideal conditions. Differences in lighting can cause extreme fluctuations in border accuracy and would require re-tuning of parameters.

3.1.2 Clustering. With clustering the concept is similar to thresholding, but instead of having to manually adjust parameters to each lighting condition we can use unsupervised learning to do so. For this we tried three different methods (all available with the SciKit-Learn library): K-means, spectral, and fuzzy-c-means and found that K-means worked the best in terms of speed and accuracy. However, K-means was considerably slower than adaptive thresholding and was not able to handle noisy backgrounds like a carpet [3, 16].

3.1.3 Superpixelization. A superpixel is a group of nearby pixels which share common characteristics like pixel intensity. And using the SciKit-image library we found that in ideal conditions with relatively simple islands, Superpixelization managed to work well. However, a significant problem arose under harder conditions with artifacts forming due to overlapping/missing superpixels. Since islands sometimes contained complex printed patterns, the superpixels would often bleed from the island's border to the background, ruining the edge [14]. In short, Superpixelization provided no more benefit than thresholding and we still needed a method that worked with noisy backgrounds.

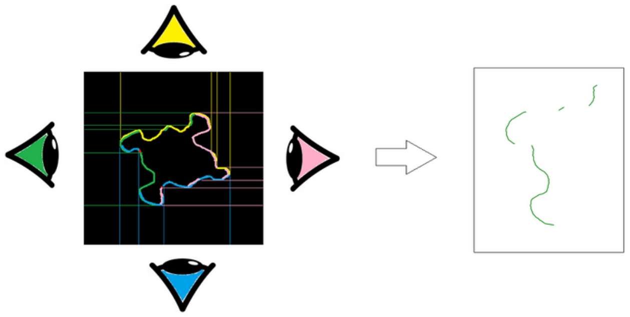

Figure 2: A visual representation of the approached used by Terleev, where rays are cast out from 4 different sides of the puzzle piece to get a 1D representation of the piece [10].

3.1.4 Mask-RCNN. A Mask-RCNN is an image segmentation deep learning model and is built-into the TensorFlow library [1]. It uses an image classification model as a backbone and convolutional layers to generate a mask of an instance [7].

The downside to this approach is that a lot of data is needed for it to work. Using the VIA image annotator to generate the data [4], we had to hand-labeled all the images. This was time-consuming but crucial step. So, to reduce the necessary training set size needed we used transfer learning - using the pretrained COCO weights as a starting point [8]. We trained for twenty epochs with a small learning rate of 0.001. The resulting trained model seemed to show the most promise out of the attempts we have tried in terms of dealing with noisy backgrounds. However, it struggles with large islands which is due to the resolution of the masks. And due to RAM limitations, this could not be increased.

We believe for this solution to work we would need to put a lot more effort into creating a better dataset, training/tuning the model, and acquiring all the computational resources needed to do so. Considering that the focus of this paper is the matching algorithm and not around segmenting puzzle pieces, we decided it was not worth the effort. So, we decided to use the simplest approach of adaptive thresholding as it was the fastest and best at extracting puzzle pieces.

3.2 Border Unrolling

The goal of border unrolling is to convert our 2D representation of the border (list of xy points) into a 1D representation that we can use for matching by sliding across each border strip looking for matches. This is important for matching large islands where simple translations and rotations would fail.

We had originally thought to follow a similar technique to Terleev's where we "observe" the border from four sides as our 1D representation, illustrated in figure 2 [9].

However, in Terleev's own words "this does not guarantee the 100% cover of the borderline". Which is especially true for U-shaped islands, and large islands in general. We would be losing a lot of information if we went with this method.

So, instead we choose to convert the 2D list of xy points to a 1D array by sampling the deviation angle as we move clockwise around the puzzle piece. Then we just store the deviation angles in

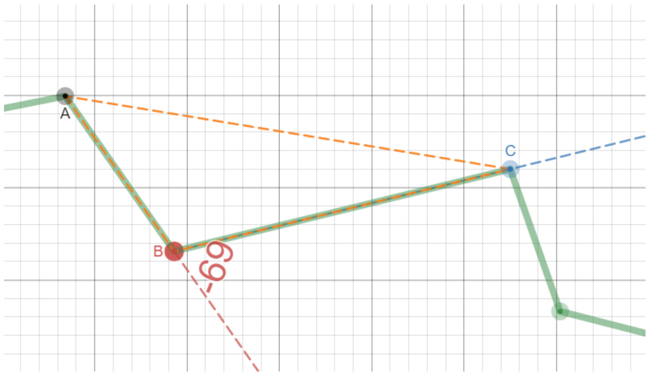

Figure 3: A visual representation of what the border unrolling algorithm extracts from the border to represent it as a 1D array. This was done using the Desmos online graphing calculator (https://www.desmos.com/calculator/04ojjjnfzt).

a list as our "unrolled" border. This deviation angle is represented in figure 3 as the angle between the red and blue dotted lines.

The angle is found by first calculating the length of each side on the triangle formed by the three points using Euclidean distance. Then we find the inner angle at point B by using the law of cosines and subtract by 180° (π radians) to get the deviation angle:

$$d = \pi - cos^{-1}(\frac{a^2 + b^2 - c^2}{2ab})$$

Now we need to get which direction the angle is deviating (the sign of the angle) to give us information on the concavity (convex or concave). By the nature of cos^{-1}, if the input angle is greater than 90° ($\pi/2$ rad) then the sign will flip so to stop this from happening we define an adjustment term to be the following:

$$adj = \frac{\pi}{2} - d$$

And to determine if we are going left (-) or right (+) we can use the slope of the lines AB (red line) and BC (blue line) from figure 3 with the adjustment term to get the following equation, where m_1 and m_2 are the slopes of AB and BC, respectively:

$$s = sign(\frac{m_1 - m_2}{1 + m_1 * m_2} * adj)$$

Finally, by multiplying d with s we get our final angle that includes the direction of the deviation. One important thing to note is that if we sampled every three points to get our angles our unrolled border would be made up of mostly zero-degree angles and noise would have a much larger impact (see figure 4). So, to solve this we make sure to pick points that are a good distance away from each other (we choose to sample points that are at least 25 indices away).

3.3 Lock Identification

The lock of a puzzle piece is its jigsaw segments that form the connections with other pieces. We need to be able to find and segment these locks no matter how many there are on the island using the unrolled border from the previous step.

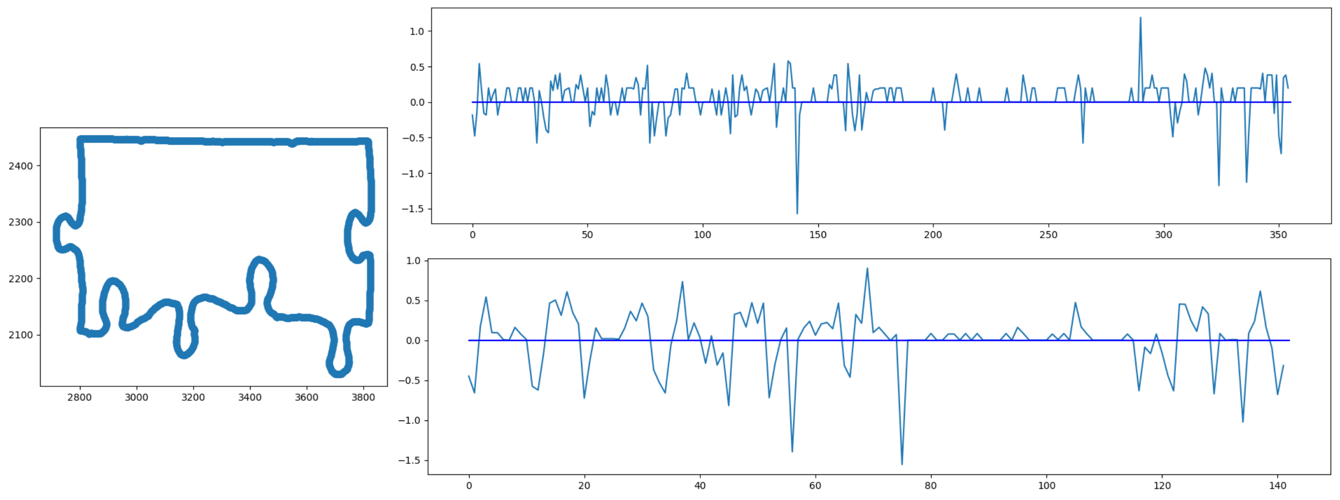

Figure 4: Plots showing how the sampling rate impacts the final unrolled border strip created. Left: the original border. Top-right: the unrolled border with a sampling rate of 25. Bottom-right: the unrolled border with a sampling rate of 10.

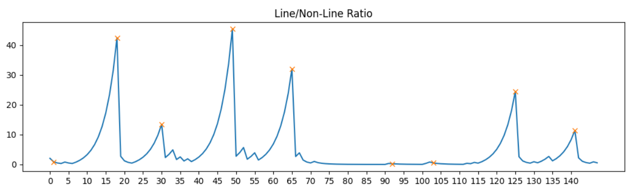

Figure 5: A plot of the ratios of line points to non-line points with x markers for the peaks that are identified by SciPy's *find_peaks_cwt()* function.

To do this we start by iterating through the unrolled border and classifying points as "line points" and "non-line points" depending on if their deviation angles are within a certain threshold. The threshold basically determines if they are close enough to 0 to be considered as a straight line (essentially zero deviation angle).

As we iterate through the unrolled border, we also keep track of the ratio of line points to non-line points that we encounter. The ratio also includes a γ forgetting factor so that as we travel along the unrolled border points that are further away don't impact the ratio as much. In other words, the lower the γ the less we care about previous points. The formula for this ratio is as follows, where N and M are the numbers of line and non-line points, respectively.

$$ratio = \frac{L_{new} + L_{old} * \gamma}{N_{new} + N_{old} * \gamma}$$

Plotting these ratios in 5 for the entire unrolled border it becomes immediately clear where the locks/jigsaw nodes are. Note how the peaks in figure 5 all match up with the locations of locks on the original border (figure 4). Also, note that we do loop over the end of the unrolled border to account for locks that are along the end of the strip. We can see this with the peak at index 140 in figure 5.

Using SciPy's *find_peaks_cwt()* we can quickly target these peaks and make cuts to the left and right of the peak (where we encounter a line-point again) [15]. And with some extra padding we have successfully identified the locks (see figure 6).

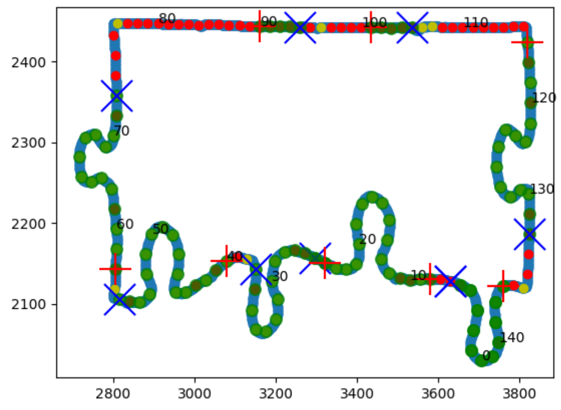

Figure 6: A visualization of the final output for lock identification. Red and yellow points indicate the "line" and "non-line" points of the border. Lock segments are highlighted in green with a red + sign indicating its start, and a blue X indicating where it ends.

3.4 High-Level Filter

Looking back at the result from lock identification (figure 6) we note that some segments are not actually locks and are just straight-lines detected due to random noise. So, here we will filter out these unwanted segments and classify locks as concave (into the island) or convex (sticking out of the island). Then we can filter out all matches that are not convex locks matching with concave locks, thereby improving performance.

Starting with the first filter we can identify line segments by performing a linear regression (on the original xy border positions) and returning the Mean Squared Error of the regression. This MSE will be low for line segments (0.0 up to 2.0) and large to extremely large for any segment with curves on it (300 up to 1500). So, by using a threshold of 5.0 we can easily filter out straight line segments. However, one limitation to this is that we must perform regression twice, alternating which axis is the horizontal axis and taking the min value. This is to prevent vertical line segments from resulting in a much higher MSE due to its near infinite slope.

Finally, for our second filter we can quickly classify concave from convex locks by fitting a 2nd degree polynomial on the unrolled border segment using Numpy's *polyfit()* function [6]. Then we can simply take the sign of the coefficient for the highest power as a high-level indication of its lock shape (negative for convex, positive for concave). Now we can check to see if a pair of segments are shaped compatible before diving into the more computationally expensive shape and color matching.

3.5 Shape and Color Matching

3.5.1 Shape Matching. For shape matching we originally tried to use OpenCV's *matchShape()* function with the original xy border contours. This would calculate a distance value using Hu-moments for each segment pair, and the segment pair with the lowest distance would be the best match [2]. However, this failed due to the curved nature of each segment and how unevenly distributed the points were (curved sections were more densely populated).

So instead, we chose to use Dynamic Time Warping (DTW) with the unrolled borders that we had segmented earlier. DTW is a popular technique commonly used for comparing time-series-like data to get a distance measure and local stretch or compressions that can be applied to the time-series to map them on top of each other [13]. This makes DTW a good candidate for comparing two unrolled border segments.

In Tarleev's approach they use an approximate DTW algorithm from the *fastDTW* python package for color matching [9, 11]. However, in this case where we have a short series length and a narrow warping factor (how much the pieces need to warp to overlay) we decided against *fastDTW* in favor of the standard *dtw-python* package. This is supported by how in our specific case FastDTW is generally slower despite being an approximation [17].

So, using DTW we get distance values for lock segments of the 1D strips for shape matching. This would tell us crucial information about how well two locks match up with each other without needing to perform any translations for a more accurate measure.

3.5.2 Color Matching. For color matching, more work is required to get a proper measure of how similar the two colors are. First, we need to ensure that we sample points along the border that are guaranteed to be part of the puzzle and not the background. This is done by calculating orthogonal points to the border on the original puzzle piece and sampling from those points instead. We also apply a median blur to the image to reduce the impact of noise.

The calculation of the orthogonal point requires sampling two points from the original border. We treat the line connecting them as the hypotenuse to a right-angled triangle. Then we can calculate the length of all the sides of the triangle and use that to calculate

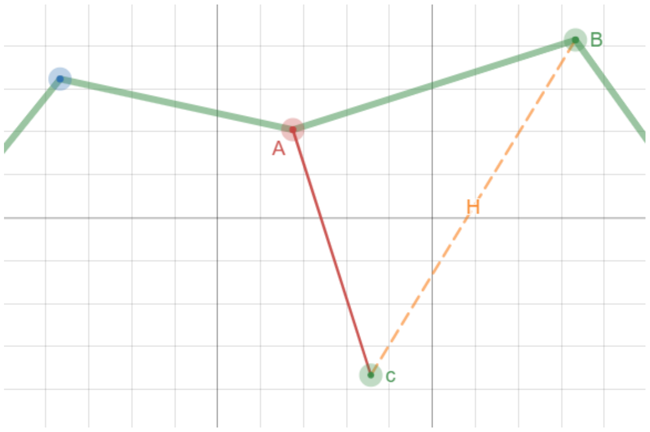

Figure 7: Illustration of how two sampled points from the border (points A and B) can be used to find a third point (c) that is orthogonal from the border and at a fixed distance d_{ist}. This was done using the Desmos online graphing calculator (https://www.desmos.com/calculator/4puznv4r7h).

the position of an orthogonal point with the following equations:

$$c_x = A_x - d_{ist} * \frac{h}{H}$$

$$c_y = A_y + d_{ist} * \frac{w}{H}$$

Where c is the orthogonal point, d_{ist} is the distance away from the border, and h, w, and H are the height, width, and hypotenuse of the right triangle formed by the sampled points A and B respectively (illustrated in figure 7).

3.5.3 Putting it all together. Now that we have a way to measure the color and shape distance of two segments, we can take their normalized score and combine them in a weighted sum for our final distance/match score. The weighted sum allows us to favor one score over the other, depending on which is a better metric for match compatibility.

4 EVALUATION

Looking at figure 8 we can see that the Powerful Puzzling algorithm successfully identifies three connections out of the top five matches when we apply the right weighting to the match scores. We found that a weighting of (2,1) for shape and color, respectively, to be the best. Taking a look at incorrect matches we note that there are no instances of catastrophic failures where we have matches with line segments, or matches of two completely incompatible shapes (e.g. concave with concave). This largely thanks to the effectiveness of the high level filters. However, when we weight color matching higher than shape matching, we tend to see some catastrophic failures with matches containing two convex or two concave locks (see top-left image in figure 8).

We believe the culprit for the majority of failed matches is due to the impact of noise on color matching. This is evident when we compare results with different match score weightings. We can see that with only shape matching (bottom-left plot in figure 8) we still manage to get at least one match, but on the other hand, using

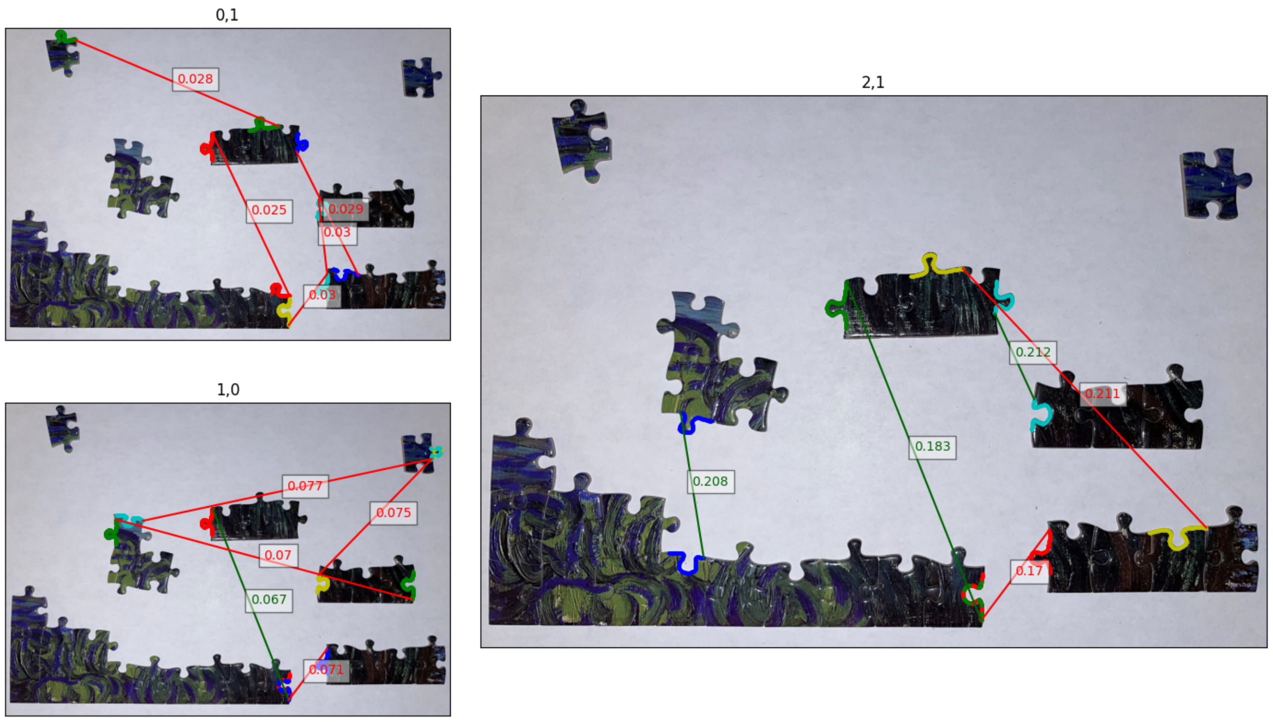

Figure 8: The final output of the Powerful Puzzling algorithm under different match score weightings for shape and color. Displayed are the top 5 matches it found along with their match score (the lower the better). Lines in green indicate correctly identified matches. Top-left: a weighting of (0,1) for shape and color respectively. Bottom-left: a weighting of (1,0). Right: a weighting of (2,1).

only color matching (top-left image in figure 8) we get no correct matches. One way to overcome the noisiness of color matching would be to (on top of applying a median blur) take an averaging of pixels along the orthogonal line to the border instead of a single pixel.

Despite this, we know that color matching is still effective by how much better the performance is when we have a weighting of (2,1) compared to just shape matching alone (right plot in figure 8).

We also observe, from figure 9, that image segmentation using Mask-RCNN does a overall good job at identifying islands from a textured/fuzzy background. However, upon closer inspection we notice that it fails to capture a crisp and clean border that is crucial to the accuracy of the shape matcher and the high-level filters, this occurs even under ideal conditions where we have a plain background with even lighting. On the other hand, adaptive thresholding thrives under ideal conditions and from figure 10 we can see that it produces a superior border outline that is almost perfect compared to the Mask-RCNN results.

Finally, in terms of performance observations the matching algorithm performs exceptionally well taking a just over 2 seconds to run (see table 1). We note that the main bottleneck is the extraction of the border contours themselves taking around 1.3 seconds to run.

5 CONCLUSION

Based on the results from evaluation we believe the Powerful Puzzling algorithm performed as we expected. It manages to be a successful tool for players in spite of some incorrect match-ups which can be easily ruled out by the player.

Aside from improving the color matcher to be more resilient to noise, another addition to improve the accuracy of the matcher is to prevent duplicate matches with a single segment from occurring. This means for any segment where we have two pieces that are contested on a single segment (e.g. the red and green segment in the right plot of figure 8) the match with the lower score would be forced to pick its next best candidate instead. This would be a difficult problem to solve as we would need to keep track of all the previous segments that each segment matches with and continuously check if a better match has overtaken them to pick the next candidate. This would also need to work in reverse to restore segments when a match that overtook a previous match is overtaken on its other segment.

Improving the border extraction process during image segmentation could also improve the capabilities of the Powerful Puzzling algorithm to work dynamically under various conditions. It would be compelling to increase the resolution of the mask for the Mask-RCNN to do so, as increasing the masks' resolution would result

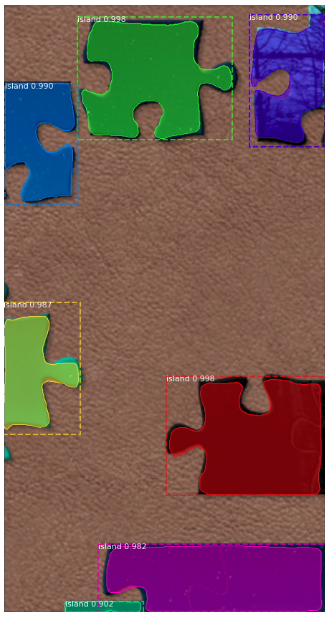

Figure 9: The output from Mask-RCNN border extraction for a image of puzzle pieces that have a textured/fuzzy background. The image shows what the model thinks are island pieces, along with the confidence value for each.

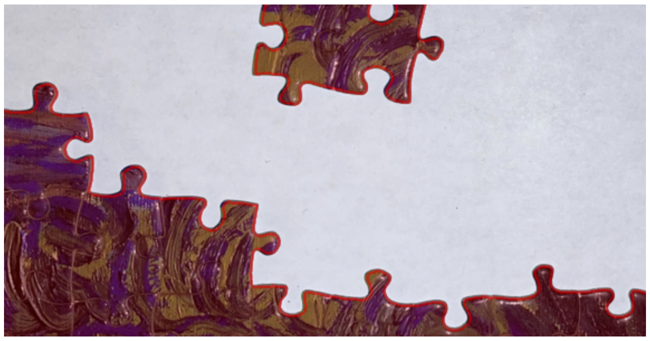

Figure 10: The output from adaptive thresholding border extraction for a image of puzzle pieces under ideal conditions. Here we see the extracted border as a red 'transparent overlay on top of the original image with an opaque border outline.

Table 1: The average time (of three trials) it takes for each step of the Powerful Puzzling algorithm in seconds. For image segmentation we only show the thresholding technique, however, Mask-RCNN can perform faster if using GPU acceleration.

Step	Average (s)
Image Segmentation (Thresholding)	1.311
Border Unrolling and Lock Identification	0.611
Filtering and Matching	0.857
Total	**2.779**

in more accurate masks and allow for large islands to be more accurately segmented with noisy backgrounds. But due to RAM limitations this could not be tested and isn't realistic.

Another idea to improve border extraction is to use a mix of thresholding or grab-cut (like seen in Photoshop selecting) and Mask-RCNN, where this technique would use the non-deep learning techniques on the large islands [2]. Combining these two methods could solve the two most significant problems we encountered. The Mask-RCNN works exceptionally well for single pieces, even in a noisy background, while thresholding was poor with noisy backgrounds. A limitation of the grab cut method is that a region of interest must be defined to segment; however, if we combined grab cut with the Mask-RCNN, which produces regions of interest within their bounding boxes, this limitation can be overcome.

To conclude, the work we have done here is still in its infancy. Currently there are no readily available papers that discuss solutions for jigsaw puzzle solvers that work with island pieces. We believe that there is still a lot that can be improved with our Powerful Puzzling algorithm, and hope future papers will eventually lead to an improved version of the Powerful Puzzling algorithm with a higher matching accuracy that works under any conditions.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[3] Lars Buitinck, Gilles Louppe, Mathieu Blondel, and Others. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[4] Abhishek Dutta and Andrew Zisserman. 2019. The VIA Annotation Software for Images, Audio and Video. In *Proceedings of the 27th ACM International Conference on Multimedia* (Nice, France) *(MM '19)*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3343031.3350535

[5] Miller Hannah. 2020. Demand for jigsaw puzzles is surging as coronavirus keeps millions of Americans indoors. *CNBC* (April 2020). https://www.cnbc.com/2020/04/03/coronavirus-sends-demand-for-jigsaw-puzzles-surging.html

[6] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[7] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. 2017. Mask R-CNN. *CoRR* abs/1703.06870 (2017). arXiv:1703.06870 http://arxiv.org/abs/1703.06870

[8] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, et al. 2014. Microsoft COCO: Common Objects in Context. *CoRR* abs/1405.0312 (2014). arXiv:1405.0312 http://arxiv.org/abs/1405.0312

[9] Terleev Maxim. 2021. *Jigsaw Puzzle AI from A to Z*. Towards Data Science. Retrieved February 22, 2022 from https://towardsdatascience.com/jigsaw-puzzle-ai-from-a-to-z-b4bdb53d8686

[10] Terleev Maxim. 2021. *Jigsaw puzzle geometrical fit*. Retrieved February 22, 2022 from https://www.simple-ai.net/post/jigsaw-puzzle-geometrical-fit

[11] Stan Salvador and Philip Chan. 2007. Toward Accurate Dynamic Time Warping in Linear Time and Space. *Intell. Data Anal.* 11, 5 (oct 2007), 561–580.

[12] Abto Software. 2018. *Computer Vision Powers Automatic Jigsaw Puzzle Solver*. Abto Software. Retrieved February 22, 2022 from https://www.abtosoftware.com/blog/computer-vision-powers-automatic-jigsaw-puzzle-solver

[13] Giorgino Toni. 2009. Computing and Visualizing Dynamic Time Warping Alignments in R: The dtw Package. *Journal of Statistical Software* 31, 7 (Aug. 2009), 1–24. https://doi.org/10.18637/jss.v031.i07

[14] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, et al. 2014. scikit-image: image processing in Python. *PeerJ* 2 (6 2014), e453. https://doi.org/10.7717/peerj.453

[15] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, et al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[16] Josh Warner, Jason Sexauer, scikit fuzzy, et al. 2019. *JDWarner/scikit-fuzzy: Scikit-Fuzzy version 0.4.2.* https://doi.org/10.5281/zenodo.3541386

[17] Renjie Wu and Eamonn J. Keogh. 2020. FastDTW is approximate and Generally Slower than the Algorithm it Approximates. *CoRR* abs/2003.11246 (2020). arXiv:2003.11246 https://arxiv.org/abs/2003.11246